

# Dal linguaggio macchina al C



62	Algoritmi e notazioni .....	15
62.1	Notazione BNF .....	17
62.2	Pseudocodifica .....	20
62.3	Problemi elementari di programmazione .....	22
62.4	Scansione di array .....	34
62.5	Problemi classici di programmazione .....	37
62.6	Gestione dei file .....	53
62.7	Trasformazione in lettere .....	66
62.8	Algoritmi elementari con la shell POSIX .....	79
62.9	Riferimenti .....	94
63	Linguaggio macchina .....	97
63.1	Organizzazione della memoria .....	99
63.2	Architettura, linguaggio, contesto virtuale, terminologia 124	
63.3	Rappresentazione di valori numerici .....	133
63.4	Calcoli con i valori binari rappresentati nella forma usata negli elaboratori .....	147
63.5	Scorrimenti, rotazioni, operazioni logiche .....	163
63.6	Confronti attraverso la sottrazione .....	174
63.7	Riferimenti .....	177
64	Microprocessori x86-32 .....	179

64.1	Terminologia impropria .....	183
64.2	Registri principali fino ai 32 bit 183	
64.3	Sintesi delle istruzioni principali .....	186
64.4	Primo approccio al linguaggio assembler per x86 213	
64.5	Esempi con le «quattro operazioni» .....	244
64.6	Esempi con gli «spostamenti» .....	275
64.7	Esempi con i confronti .....	290
64.8	Le istruzioni di salto .....	298
64.9	Esempi di programmi con strutture di controllo .....	305
64.10	Funzioni .....	322
64.11	Esempi di funzioni ricorsive .....	350
64.12	Indirizzamento dei dati .....	358
64.13	Rappresentazione dei dati in memoria attraverso un esempio .....	368
64.14	Esempi con gli array .....	375
64.15	Calcoli con gli indirizzi in fase di compilazione ...	391
64.16	Interazione con il sistema operativo .....	395
64.17	Riferimenti .....	405
65	Compilazione e formato binario eseguibile .....	407
65.1	Compilazione di programmi composti da più file sorgenti 410	
65.2	Librerie dinamiche e librerie statiche .....	419
65.3	Dal sorgente all'immagine in memoria .....	432

65.4	Formato ELF .....	453
65.5	Programmi completamente autonomi .....	470
65.6	Compilazione C dal basso in alto .....	495
65.7	Compilazione C dall'alto in basso .....	509
65.8	Compilazione guidata con Make .....	523
65.9	Riferimenti .....	552
66	Introduzione al linguaggio C .....	557
66.1	Nozioni minime .....	563
66.2	Istruzioni del precompilatore .....	619
66.3	Dal campo di azione alla compilazione .....	650
66.4	Annotazioni sulla terminologia .....	670
66.5	Puntatori, array, stringhe e allocazione dinamica della memoria .....	677
66.6	Le funzioni .....	743
66.7	Struttura, unione, campo, enumerazione, costante composta .....	755
66.8	Tipi di dati speciali, di uso comune .....	777
66.9	Configurazione locale .....	789
66.10	Organizzazione dei file sorgenti .....	800
66.11	K&R .....	808
66.12	Riferimenti .....	816
67	Gestione dei flussi di file in C .....	821
67.1	Concetti generali .....	822
67.2	Utilizzo comune dei file .....	832

67.3	Conversione di input e output .....	850
67.4	Riferimenti .....	871
Esempi di programmazione in C .....		873
Problemi elementari .....		873
Scansione di array .....		890
Algoritmi tradizionali .....		896
68	Introduzione alle estensioni POSIX .....	909
68.1	Dal C a POSIX .....	912
68.2	Espressioni regolari POSIX .....	916
68.3	Avvio e conclusione dei processi .....	940
68.4	Nozioni sui thread POSIX .....	953
68.5	I file secondo i sistemi POSIX .....	979
68.6	Il file system Unix e la sua gestione tipica .....	996
68.7	Il file system Minix 1 .....	1008
68.8	Creazione ed eliminazione di file di qualunque tipo	1022
68.9	Condotti .....	1034
68.10	Lettura delle directory .....	1047
68.11	Riferimenti .....	1051
69	Libreria C, con qualche estensione POSIX .....	1055
69.1	Funzionalità di libreria non dichiarate .....	1069
69.2	File «assert.h» .....	1069
69.3	File «limits.h» .....	1071
69.4	File «stdint.h» .....	1077
69.5	File «errno.h» .....	1087

69.6	File «locale.h»	1092
69.7	File «ctype.h»	1101
69.8	File «stdarg.h»	1123
69.9	File «stdlib.h»	1127
69.10	File «inttypes.h»	1156
69.11	File «iso646.h»	1167
69.12	File «stdbool.h»	1168
69.13	File «stddef.h»	1169
69.14	File «string.h»	1170
69.15	File «signal.h»	1223
69.16	File «time.h»	1239
69.17	File «stdio.h»	1254
69.18	Riferimenti	1302
70	Libreria POSIX	1307
70.1	File «sys/types.h»	1312
70.2	File «sys/stat.h»	1315
70.3	File «strings.h»	1321
70.4	File «fcntl.h»	1322
70.5	File «unistd.h»	1329
70.6	File «dirent.h»	1342
70.7	File «termios.h»	1345
70.8	Riferimenti	1350
	Tabelle riepilogative della libreria C e POSIX	1353
	File «stdarg.h»	1359

File «limits.h»	.....	1360
File «stdint.h»		1361
File «inttypes.h»		1365
File «ctype.h»	.....	1375
File «stdlib.h»		1379
File «string.h»	.....	1390
File «time.h»	.....	1399
File «stdio.h» per la gestione dei file e degli errori		1406
File «stdio.h» per la composizione dell'output		1421
File «stdio.h» per l'interpretazione dell'input	...	1428
File «assert.h»	.....	1435
File «stddef.h»	.....	1435
File «locale.h»	.....	1436
File «regex.h»	.....	1437
File «sys/stat.h»		1442
71 Gettext	.....	1449
71.1 Principio di funzionamento	.....	1449
71.2 Fasi di preparazione	.....	1450
71.3 Abbinamento a un «pacchetto»	.....	1454
71.4 Creazione e mantenimento dei file PO	.....	1455
71.5 Gettext con i programmi Perl	.....	1457

# Algoritmi e notazioni



62.1	Notazione BNF .....	17
62.1.1	BNF essenziale .....	17
62.1.2	Estensioni usuali .....	18
62.2	Pseudocodifica .....	20
62.3	Problemi elementari di programmazione .....	22
62.3.1	Somma tra due numeri positivi .....	22
62.3.2	Moltiplicazione di due numeri positivi attraverso la somma .....	24
62.3.3	Divisione intera tra due numeri positivi .....	25
62.3.4	Elevamento a potenza .....	26
62.3.5	Radice quadrata .....	28
62.3.6	Fattoriale .....	29
62.3.7	Massimo comune divisore .....	30
62.3.8	Numero primo .....	31
62.3.9	Successione di Fibonacci .....	32
62.4	Scansione di array .....	34
62.4.1	Ricerca sequenziale .....	34
62.4.2	Ricerca binaria .....	36
62.5	Problemi classici di programmazione .....	37
62.5.1	Bubblesort .....	37
62.5.2	Fusione tra due array ordinati .....	39
62.5.3	Torre di Hanoi .....	41

62.5.4	Quicksort (ordinamento non decrescente) .....	44
62.5.5	Permutazioni .....	51
62.6	Gestione dei file .....	53
62.6.1	Fusione tra due file ordinati .....	53
62.6.2	Riordino attraverso la fusione .....	56
62.7	Trasformazione in lettere .....	66
62.7.1	Da numero a sequenza alfabetica pura .....	66
62.7.2	Da numero a numero romano .....	69
62.7.3	Da numero a lettere, nel senso verbale .....	75
62.8	Algoritmi elementari con la shell POSIX .....	79
62.8.1	ARCS0: ricerca del valore più grande tra tre numeri interi .....	80
62.8.2	ARCS1: moltiplicazione di due numeri interi .....	81
62.8.3	ARCS2: valore assoluto della differenza tra due valori 82	
62.8.4	ARCS3: somma tra due numeri .....	83
62.8.5	ARCS4: prodotto tra due numeri .....	83
62.8.6	ARCS5: quoziente .....	85
62.8.7	ARCS6: verifica della parità di un numero .....	86
62.8.8	ARCS7: fattoriale .....	87
62.8.9	ARCS8: coefficiente binomiale .....	87
62.8.10	ARCS10: massimo comune divisore .....	89
62.8.11	ARCS11: massimo comune divisore .....	90
62.8.12	ARCS12: radice quadrata intera .....	91



62.8.13	ARCS13: numero primo .....	92
62.8.14	ARCS14: numero primo .....	93
62.9	Riferimenti .....	94

## 62.1 Notazione BNF

In molti documenti si usa la «notazione BNF» per mostrare la sintassi di qualcosa, particolarmente quando si tratta della descrizione formale dei linguaggi di programmazione. La sigla BNF sta per *Bac-  
kus Naur form*, a ricordare che si tratta di una notazione introdotta da John Backus e Peter Naur, tra il 1959 e il 1960.

### 62.1.1 BNF essenziale

La notazione BNF utilizza pochi simboli per attribuire un significato a ciò che descrive:

Simbolo	Significato
$::=$	Si legge come: «è definito da». Sta a indicare che l'oggetto alla sinistra di tale simbolo viene definito come ciò che si trova alla destra di questo.
	Si legge come: «oppure». Sta a indicare che può essere usato l'oggetto che sta a sinistra del simbolo, oppure quello a destra, indifferentemente.
$\langle \textit{nome} \rangle$	Indica il nome di una categoria. Il nome può essere scritto senza vincoli particolari.
$x$	Qualunque cosa sia scritta al di fuori delle parentesi angolari ('<', '>'), escludendo altri simboli di cui sia stato dichiarato il significato, va interpretata letteralmente (come parola chiave).

A titolo di esempio, viene mostrata la definizione di una lettera dell'alfabeto latino, suddividendo il problema, definendo cosa sono le lettere latine minuscole e cosa sono le lettere latine maiuscole:

```
<lettera_alfabeto_latino> ::=
    <lettera_alfabeto_latino_maiuscola>
  | <lettera_alfabeto_latino_minuscola>
```

```
<lettera_alfabeto_latino_maiuscola> ::=
    A | B | C | D | E | F | G | H | I | J | K | L
  | M | N | O | P | Q | R | S | T | U | V | W | X
  | Y | Z
```

```
<lettera_alfabeto_latino_minuscola> ::=
    a | b | c | d | e | f | g | h | i | j | k | l
  | m | n | o | p | q | r | s | t | u | v | w | x
  | y | z
```

## 62.1.2 Estensioni usuali

«

Di fatto, la notazione BNF viene usata estendendo la simbologia, per semplificarne la lettura ed evitare ambiguità, ma a volte la simbologia viene anche cambiata leggermente.

Simbolo	Significato
[...]	Le parentesi quadre vengono usate normalmente per delimitare una porzione facoltativa della dichiarazione.

Simbolo	Significato
{...}	Le parentesi graffe vengono usate normalmente per delimitare una porzione necessaria della dichiarazione, che però, per qualche ragione, va intesa come un blocco unito. A volte, l'inclusione tra parentesi graffe va intesa come la possibilità di ripetere indefinitamente il suo contenuto, ma per questo, di solito si aggiungono tre puntini di sospensione in coda.
...	Tre puntini di sospensione vengono usati per indicare una porzione della dichiarazione che può essere ripetuta.
"..." '...'	Una coppia di apici doppi o singoli, può essere usata per delimitare qualcosa che va inteso letteralmente e non va confuso con la simbologia usata per la notazione BNF.
<i>nome</i>	I nomi di qualcosa potrebbero essere annotati senza le parentesi angolari, se si usa una forma tipografica particolare per evidenziarli (per esempio in corsivo o in nero, rispetto a un testo normale per ciò che va inteso letteralmente).

Segue un esempio molto semplice, dove si vede l'uso delle parentesi quadre, graffe e dei puntini di sospensione, per descrivere un'istruzione condizionale di un certo linguaggio, senza però entrare troppo nel dettaglio:

```

<istruzione_condizionale> ::=
    IF <espressione_logica>
        THEN
            <sequenza_di_istruzioni>
        [ ELSE
            <sequenza_di_istruzioni> ]
    END FI

<sequenza_di_istruzioni> ::=
    { <istruzione> | <commento> | <riga_bianca> }...

```

Segue lo stesso esempio, modificato in modo da evidenziare i nomi di categoria, evitando così l'uso delle parentesi angolari:

```

istruzione_condizionale ::= IF espressione_logica
                                THEN
                                    sequenza_di_istruzioni
                                [ ELSE
                                    sequenza_di_istruzioni ]
                                END FI

sequenza_di_istruzioni ::= { istruzione | commento | riga_bianca }...

```

## 62.2 Pseudocodifica



Un tempo la programmazione avveniva attraverso lunghe fasi di studio a tavolino. Prima di iniziare il lavoro di scrittura del programma (su moduli cartacei che venivano trasferiti successivamente nella

macchina) si passava per la realizzazione di un diagramma di flusso, o *flow chart*.

Il diagramma di flusso andava bene fino a quando si utilizzavano linguaggi di programmazione procedurali, come il COBOL. Quando si sono introdotti concetti nuovi che rendevano tale sistema di rappresentazione più complicato del linguaggio stesso, si è preferito schematizzare gli algoritmi attraverso righe di codice vero e proprio o attraverso una pseudocodifica più o meno adatta al concetto che si vuole rappresentare di volta in volta.

Nelle sezioni successive appaiono esempi realizzati attraverso una pseudocodifica. Tali esempi non sono ottimizzati perché si intende puntare sulla chiarezza piuttosto che sull'eventuale velocità di esecuzione. La pseudocodifica si rifà a termini e concetti comuni a molti linguaggi di programmazione recenti. Vale la pena di chiarire solo alcuni dettagli:

- le variabili di scambio di una subroutine (una procedura o una funzione) vengono semplicemente nominate a fianco del nome della procedura, tra parentesi, cosa che corrisponde a una dichiarazione implicita di quelle variabili con un campo di azione locale e con caratteristiche identiche a quelle usate nelle chiamate relative;
- il trasferimento dei parametri di una chiamata alla subroutine avviene per valore, impedendo l'alterazione delle variabili originali;
- per trasferire una variabile per riferimento, in modo che il suo valore venga aggiornato al termine dell'esecuzione di una sub-

routine, occorre aggiungere il simbolo '@' di fronte al nome della variabile utilizzata nella chiamata;

- il simbolo '#' rappresenta l'inizio di un commento;
- il simbolo ':=' rappresenta l'assegnamento;
- il simbolo ':==:' rappresenta lo scambio tra due operandi.

La pseudocodifica in questione non gestisce i puntatori e l'uso dell'operatore «@» è solo un modo per affermare che le modifiche apportate alla variabile devono essere mantenute alla conclusione della funzione.

## 62.3 Problemi elementari di programmazione

«

Nelle sezioni seguenti sono descritti alcuni problemi elementari attraverso cui si insegnano le tecniche di programmazione ai principianti. Assieme ai problemi vengono proposte le soluzioni in forma di pseudocodifica.

### 62.3.1 Somma tra due numeri positivi

«

La somma di due numeri positivi può essere espressa attraverso il concetto dell'incremento unitario:  $n+m$  equivale a incrementare  $m$ , di un'unità, per  $n$  volte, oppure incrementare  $n$  per  $m$  volte. L'algoritmo risolutivo è banale, ma utile per apprendere il funzionamento dei cicli:

```
SOMMA (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := X
    FOR I := 1; I <= Y; I++
        Z++
    END FOR

    RETURN Z

END SOMMA
```

In questo caso viene mostrata una soluzione per mezzo di un ciclo enumerativo, **FOR**. Il ciclo viene ripetuto **Y** volte, incrementando la variabile **Z** di un'unità. Alla fine, **Z** contiene il risultato della somma di **X** per **Y**. La pseudocodifica seguente mostra invece la traduzione del ciclo **FOR** in un ciclo **WHILE**:

```
SOMMA (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := X
    I := 1
    WHILE I <= Y
        Z++
        I++
    END WHILE

    RETURN Z

END SOMMA
```

## 62.3.2 Moltiplicazione di due numeri positivi attraverso la somma



La moltiplicazione di due numeri positivi, può essere espressa attraverso il concetto della somma:  $n*m$  equivale a sommare  $m$  volte  $n$ , oppure  $n$  volte  $m$ . L'algoritmo risolutivo è banale, ma utile per apprendere il funzionamento dei cicli:

```
MOLTIPLICA (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := 0
    FOR I := 1; I <= Y; I++
        Z := Z + X
    END FOR

    RETURN Z

END MOLTIPLICA
```

In questo caso viene mostrata una soluzione per mezzo di un ciclo **FOR**. Il ciclo viene ripetuto **Y** volte, incrementando la variabile **Z** del valore di **X**. Alla fine, **Z** contiene il risultato del prodotto di **X** per **Y**. La pseudocodifica seguente mostra invece la traduzione del ciclo **FOR** in un ciclo **WHILE**:



```
MOLTIPLICA (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := 0
    I := 1
    WHILE I <= Y
        Z := Z + X
        I++
    END WHILE

    RETURN Z

END MOLTIPLICA
```

### 62.3.3 Divisione intera tra due numeri positivi

La divisione di due numeri positivi, può essere espressa attraverso la sottrazione:  $n:m$  equivale a sottrarre  $m$  da  $n$  fino a quando  $n$  diventa inferiore di  $m$ . Il numero di volte in cui tale sottrazione ha luogo, è il risultato della divisione.



```
DIVIDI (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := 0
    I := X
    WHILE I >= Y
        I := I - Y
        Z++
    END WHILE

    RETURN Z

END DIVIDI
```

### 62.3.4 Elevamento a potenza



L'elevamento a potenza, utilizzando numeri positivi, può essere espresso attraverso il concetto della moltiplicazione:  $n^{**}m$  equivale a moltiplicare  $m$  volte  $n$  per se stesso.

```
EXP (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := 1
    FOR I := 1; I <= Y; I++
        Z := Z * X
    END FOR

    RETURN Z

END EXP
```

In questo caso viene mostrata una soluzione per mezzo di un ciclo **FOR**. Il ciclo viene ripetuto **Y** volte; ogni volta la variabile **Z** viene moltiplicata per il valore di **X**, a partire da uno. Alla fine, **Z** contiene il risultato dell'elevamento di **X** a **Y**. La pseudocodifica seguente mostra invece la traduzione del ciclo **FOR** in un ciclo **WHILE**:

```
EXP (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := 1
    I := 1
    WHILE I <= Y
        Z := Z * X
        I++
    END WHILE

    RETURN Z

END EXP
```

La pseudocodifica seguente mostra una soluzione ricorsiva:

```
EXP (X, Y)

    IF X = 0
        THEN
            RETURN 0
        ELSE
            IF Y = 0
                THEN
                    RETURN 1
                ELSE
                    RETURN X * EXP (X, Y-1)
            END IF
        END IF
    END IF

END EXP
```

### 62.3.5 Radice quadrata



Il calcolo della parte intera della radice quadrata di un numero si può fare per tentativi, partendo da 1, eseguendo il quadrato fino a quando il risultato è minore o uguale al valore di partenza di cui si calcola la radice.

```
RADICE (X)

    LOCAL Z INTEGER
    LOCAL T INTEGER

    Z := 0
    T := 0

    WHILE TRUE

        T := Z * Z
```

```
        IF T > X
            THEN
                # È stato superato il valore massimo.
                Z--
                RETURN Z
            END IF

            Z++

        END WHILE

    END RADICE
```

### 62.3.6 Fattoriale

Il fattoriale è un valore che si calcola a partire da un numero positivo. Può essere espresso come il prodotto di  $n$  per il fattoriale di  $n-1$ , quando  $n$  è maggiore di 1, mentre equivale a 1 quando  $n$  è uguale a 1. In pratica,  $n! = n * (n-1) * (n-2) \dots * 1$ .

```
FATTORIALE (X)

    LOCAL I INTEGER

    I := X - 1

    WHILE I > 0
        X := X * I
        I--
    END WHILE

    RETURN X

END FATTORIALE
```

La soluzione appena mostrata fa uso di un ciclo **‘WHILE’** in cui l'indice **‘I’**, che inizialmente contiene il valore di **‘X-1’**, viene usato per essere moltiplicato al valore di **‘X’**, riducendolo ogni volta di un'unità. Quando **‘I’** raggiunge lo zero, il ciclo termina e **‘X’** contiene il valore del fattoriale. L'esempio seguente mostra invece una soluzione ricorsiva che dovrebbe risultare più intuitiva:

```
FATTORIALE (X)

    IF X == 1
        THEN
            RETURN 1
        ELSE
            RETURN X * FATTORIALE (X - 1)
    END IF

END FATTORIALE
```

### 62.3.7 Massimo comune divisore



Il massimo comune divisore tra due numeri può essere ottenuto sottraendo a quello maggiore il valore di quello minore, fino a quando i due valori sono uguali. Quel valore è il massimo comune divisore.

```
MCD (X, Y)

    WHILE X != Y

        IF X > Y
            THEN
                X := X - Y
            ELSE
                Y := Y - X
            END IF

    END WHILE

    RETURN X

END MCD
```

### 62.3.8 Numero primo

Un numero intero è numero primo quando non può essere diviso per un altro intero diverso dal numero stesso e da 1, generando un risultato intero. <<

```
PRIMO (X)

    LOCAL PRIMO BOOLEAN
    LOCAL I INTEGER
    LOCAL J INTEGER

    PRIMO := TRUE
    I := 2

    WHILE (I < X) AND PRIMO

        J := X / I
        J := X - (J * I)
```

```
        IF J == 0
            THEN
                PRIMO := FALSE
            ELSE
                I++
            END IF
        END WHILE

        RETURN PRIMO

END PRIMO
```

### 62.3.9 Successione di Fibonacci

«

La successione di Fibonacci è una sequenza di numeri interi positivi che hanno la proprietà di essere costituiti dalla somma dei due numeri precedenti nella sequenza stessa. Pertanto, l' $n$ -esimo elemento di questa successione, indicato solitamente come  $F_n$ , è dato dalla somma di  $F_{n-1}$  e  $F_{n-2}$ .

La successione di Fibonacci parte storicamente dal presupposto che  $F_1$  e  $F_2$  siano entrambi pari a uno, ma attualmente si indica anche  $F_0$  pari a zero, cosa che consente di calcolare correttamente  $F_2$ .

Per il calcolo della successione di Fibonacci, dall'elemento zero, fino all'elemento  $n$ -esimo, vengono proposte due modalità di calcolo, la prima in forma ricorsiva, la seconda in forma iterativa.



```
FIBONACCI (N)
  IF N == 0
  THEN
    RETURN 0
  ELSE
    IF N == 1
    THEN
      RETURN 1
    ELSE
      RETURN (FIBONACCI (N - 1) + FIBONACCI (N - 2))
    END IF
  END IF
END FIBONACCI
```

```
FIBONACCI (N)
  LOCAL F1 := 1
  LOCAL F0 := 0
  LOCAL FN := N
  LOCAL I

  FOR I := 2; I <= N; I++
    FN := F1 + F0
    F0 := F1
    F1 := FN
  END FOR

  RETURN FN

END FIBONACCI
```

La successione di Fibonacci, per cui  $F_0$  è pari a zero e  $F_1$  è pari a uno, è: 0, 1, 1, 2, 3, 5, 8, 13,...

## 62.4 Scansione di array



Nelle sezioni seguenti sono descritti alcuni problemi legati alla scansione di array. Assieme ai problemi vengono proposte le soluzioni in forma di pseudocodifica.

### 62.4.1 Ricerca sequenziale



La ricerca di un elemento all'interno di un array disordinato può avvenire solo in modo sequenziale, cioè controllando uno per uno tutti gli elementi, fino a quando si trova la corrispondenza cercata. Segue la descrizione delle variabili più importanti che appaiono nella pseudocodifica successiva:

Variabile	Descrizione
LISTA	È l'array su cui effettuare la ricerca.
X	È il valore cercato all'interno dell'array.
A	È l'indice inferiore dell'intervallo di array su cui si vuole effettuare la ricerca.
Z	È l'indice superiore dell'intervallo di array su cui si vuole effettuare la ricerca.

Ecco un esempio di pseudocodifica che risolve il problema in modo iterativo:

```
RICERCASEQ (LISTA, X, A, Z)

    LOCAL I INTEGER

    FOR I := A; I <= Z; I++
        IF X == LISTA[I]
            THEN
                RETURN I
            END IF
    END FOR

    # La corrispondenza non è stata trovata.
    RETURN -1

END RICERCASEQ
```

**Solo a scopo didattico, viene proposta una soluzione ricorsiva:**

```
RICERCASEQ (LISTA, X, A, Z)

    IF A > Z
        THEN
            RETURN -1
        ELSE
            IF X == LISTA[A]
                THEN
                    RETURN A
                ELSE
                    RETURN RICERCASEQ (@LISTA, X, A+1, Z)
            END IF
        END IF

    END IF

END RICERCASEQ
```

## 62.4.2 Ricerca binaria

&lt;&lt;

La ricerca di un elemento all'interno di un array ordinato può avvenire individuando un elemento centrale: se questo corrisponde all'elemento cercato, la ricerca è terminata, altrimenti si ripete nella parte di array precedente o successiva all'elemento, a seconda del suo valore e del tipo di ordinamento esistente.

Il problema posto in questi termini è ricorsivo. La pseudocodifica mostrata utilizza le stesse variabili già descritte per la ricerca sequenziale.

```
RICERCABIN (LISTA, X, A, Z)

LOCAL M INTEGER

# Determina l'elemento centrale dell'array.
M := (A + Z) / 2

IF M < A
  THEN
    # Non restano elementi da controllare:
    # l'elemento cercato non c'è.
    RETURN -1
  ELSE
    IF X < LISTA[M]
      THEN
        # Si ripete la ricerca nella parte
        # inferiore.
        RETURN RICERCABIN (@LISTA, X, A, M-1)
      ELSE
        IF X > LISTA[M]
          THEN
            # Si ripete la ricerca nella
            # parte superiore.
```

```

        RETURN RICERCABIN (@LISTA, X, M+1, Z)
    ELSE
        # M rappresenta l'indice
        # dell'elemento cercato.
        RETURN M
    END IF
END IF
END IF
END RICERCABIN

```

## 62.5 Problemi classici di programmazione

Nelle sezioni seguenti sono descritti alcuni problemi classici attraverso cui si insegnano le tecniche di programmazione. Assieme ai problemi vengono proposte le soluzioni in forma di pseudocodifica.

### 62.5.1 Bubblesort

Il Bubblesort è un algoritmo relativamente semplice per l'ordinamento di un array, in cui ogni scansione trova il valore giusto per l'elemento iniziale dell'array stesso. Una volta trovata la collocazione di un elemento, si ripete la scansione per il segmento rimanente di array, in modo da collocare un altro valore. La pseudocodifica dovrebbe chiarire il meccanismo.

Variabile	Descrizione
LISTA	È l'array da ordinare.
A	È l'indice inferiore del segmento di array da ordinare.
Z	È l'indice superiore del segmento di array da ordinare.

**Viene mostrata una soluzione iterativa:**

```
BSORT (LISTA, A, Z)

    LOCAL J INTEGER
    LOCAL K INTEGER

    # Scandisce l'array attraverso l'indice J in modo da
    # collocare ogni volta il valore corretto all'inizio
    # dell'array stesso.
    FOR J := A; J < Z; J++

        # Scandisce l'array attraverso l'indice K scambiando
        # i valori quando sono inferiori a quello di
        # riferimento.
        FOR K := J+1; K <= Z; K++

            IF LISTA[K] < LISTA[J]
                THEN
                    # I valori vengono scambiati.
                    LISTA[K] ::= LISTA[J]
                END IF
            END FOR
        END FOR
    END FOR

END BSORT
```

**Segue una soluzione ricorsiva:**

```
BSORT (LISTA, A, Z)

    LOCAL K INTEGER

    # L'elaborazione termina quando l'indice inferiore è
```

```
# maggiore o uguale a quello superiore.
IF A < Z
  THEN

    # Scandisce l'array attraverso l'indice K
    # scambiando i valori quando sono inferiori a
    # quello iniziale.
    FOR K := A+1; K <= Z; K++

      IF LISTA[K] < LISTA[A]
        THEN
          # I valori vengono scambiati.
          LISTA[K] ::= LISTA[A]
        END IF

      END FOR

    # L'elemento LISTA[A] è collocato correttamente,
    # adesso si ripete la chiamata della funzione in
    # modo da riordinare la parte restante
    # dell'array.
    BSORT (@LISTA, A+1, Z)

  END IF

END BSORT
```

## 62.5.2 Fusione tra due array ordinati

Due array a una dimensione, con la stessa struttura, ordinati secondo qualche criterio, possono essere fusi in un array singolo, che mantiene l'ordinamento. <<

Variabile	Descrizione
A	È il primo array.
B	È il secondo array.
C	È l'array da generare con la fusione di 'A' e 'B'.
I	È l'indice usato per scandire 'A'.
J	È l'indice usato per scandire 'B'.
K	È l'indice usato per scandire 'C'.
N	È la dimensione di 'A' (l'indice dell'ultimo elemento dell'array).
M	È la dimensione di 'B' (l'indice dell'ultimo elemento dell'array).

Viene mostrata una soluzione iterativa, dove si presume che gli array siano ordinati in modo non decrescente:

```
MERGE (A, N, B, M, C)

LOCAL I INTEGER
LOCAL J INTEGER
LOCAL K INTEGER

# Si presume che l'indice del primo elemento degli
# array sia pari a uno.

I := 1
J := 1
K := 1
```



```
UNTIL I >= N AND J >= M

    IF A(I) <= B(J)
        THEN
            C(K) := A(I)
            I++
        ELSE
            C(K) := B(J)
            J++
        END IF
    K++

END UNTIL

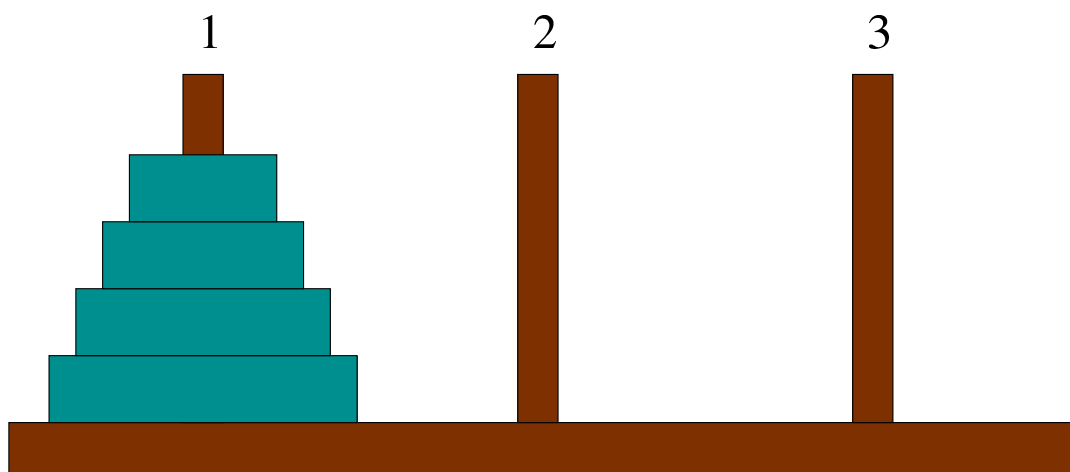
END MERGE
```

### 62.5.3 Torre di Hanoi

La torre di Hanoi è un gioco antico: si compone di tre pioli identici conficcati verticalmente su una tavola e di una serie di anelli di larghezze differenti. Gli anelli sono più precisamente dei dischi con un foro centrale che permette loro di essere infilati nei pioli.

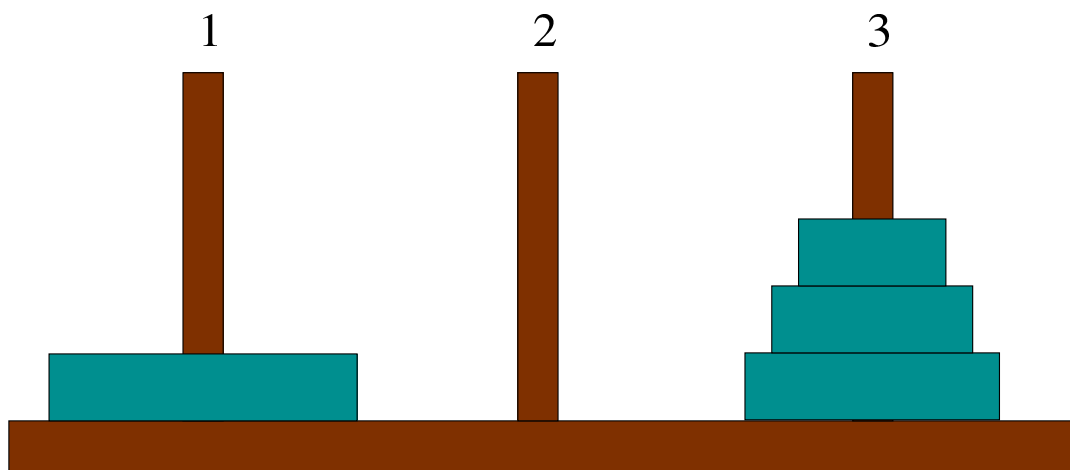
Il gioco inizia con tutti gli anelli collocati in un solo piolo, in ordine, in modo che in basso ci sia l'anello più largo e in alto quello più stretto. Si deve riuscire a spostare tutta la pila di anelli in un dato piolo muovendo un anello alla volta e senza mai collocare un anello più grande sopra uno più piccolo.

Figura 62.27. Situazione iniziale della torre di Hanoi all'inizio del gioco.



Nella figura 62.27 gli anelli appaiono inseriti sul piolo 1; si supponga che questi debbano essere spostati sul piolo 2. Si può immaginare che tutti gli anelli, meno l'ultimo, possano essere spostati in qualche modo corretto, dal piolo 1 al piolo 3, come nella situazione della figura 62.28.

Figura 62.28. Situazione dopo avere spostato  $n-1$  anelli.



A questo punto si può spostare l'ultimo anello rimasto (l' $n$ -esimo), dal piolo 1 al piolo 2; quindi, come prima, si può spostare in qualche modo il gruppo di anelli posizionati attualmente nel piolo 3, in modo che finiscano nel piolo 2 sopra l'anello più grande.

Pensando in questo modo, l'algoritmo risolutivo del problema deve essere ricorsivo e potrebbe essere gestito da un'unica subroutine che può essere chiamata opportunamente 'HANOI'.

Variabile	Descrizione
N	È la dimensione della torre espressa in numero di anelli: gli anelli sono numerati da 1 a 'N'.
P1	È il numero del piolo su cui si trova inizialmente la pila di 'N' anelli.
P2	È il numero del piolo su cui deve essere spostata la pila di anelli.
6-P1-P2	È il numero dell'altro piolo. Funziona così se i pioli sono numerati da 1 a 3.

Segue la pseudocodifica ricorsiva per la soluzione del problema:

```

HANOI (N, P1, P2)

  IF N > 0
    THEN
      HANOI (N-1, P1, 6-P1-P2)
      scrivi: "Muovi l'anello" N "dal piolo" P1 "al piolo" P2
      HANOI (N-1, 6-P1-P2, P2)
    END IF
  END HANOI

```

Se 'N', il numero degli anelli da spostare, è minore di 1, non si deve compiere alcuna azione. Se 'N' è uguale a 1, le istruzioni che dipendono dalla struttura IF-END IF vengono eseguite, ma nessuna delle chiamate ricorsive fa alcunché, dato che 'N-1' è pari a zero. In questo caso, supponendo che 'N' sia uguale a 1, che 'P1' sia pari a 1 e 'P2' pari a 2, il risultato è semplicemente:

```
Muovi l'anello 1 dal piolo 1 al piolo 2
```

Il risultato è quindi corretto per una pila iniziale consistente di un solo anello.

Se 'N' è uguale a 2, la prima chiamata ricorsiva sposta un anello ('N-1' = 1) dal piolo 1 al piolo 3 (ancora assumendo che i due anelli debbano essere spostati dal primo al terzo piolo) e si sa che questa è la mossa corretta. Quindi viene stampato il messaggio che dichiara lo spostamento del secondo piolo (l' 'N'-esimo) dalla posizione 1 alla posizione 2. Infine, la seconda chiamata ricorsiva si occupa di spostare l'anello collocato precedentemente nel terzo piolo, nel secondo, sopra a quello che si trova già nella posizione finale corretta.

In pratica, nel caso di due anelli che devono essere spostati dal primo al secondo piolo, appaiono i tre messaggi seguenti.

```
Muovi l'anello 1 dal piolo 1 al piolo 3  
Muovi l'anello 2 dal piolo 1 al piolo 2  
Muovi l'anello 1 dal piolo 3 al piolo 2
```

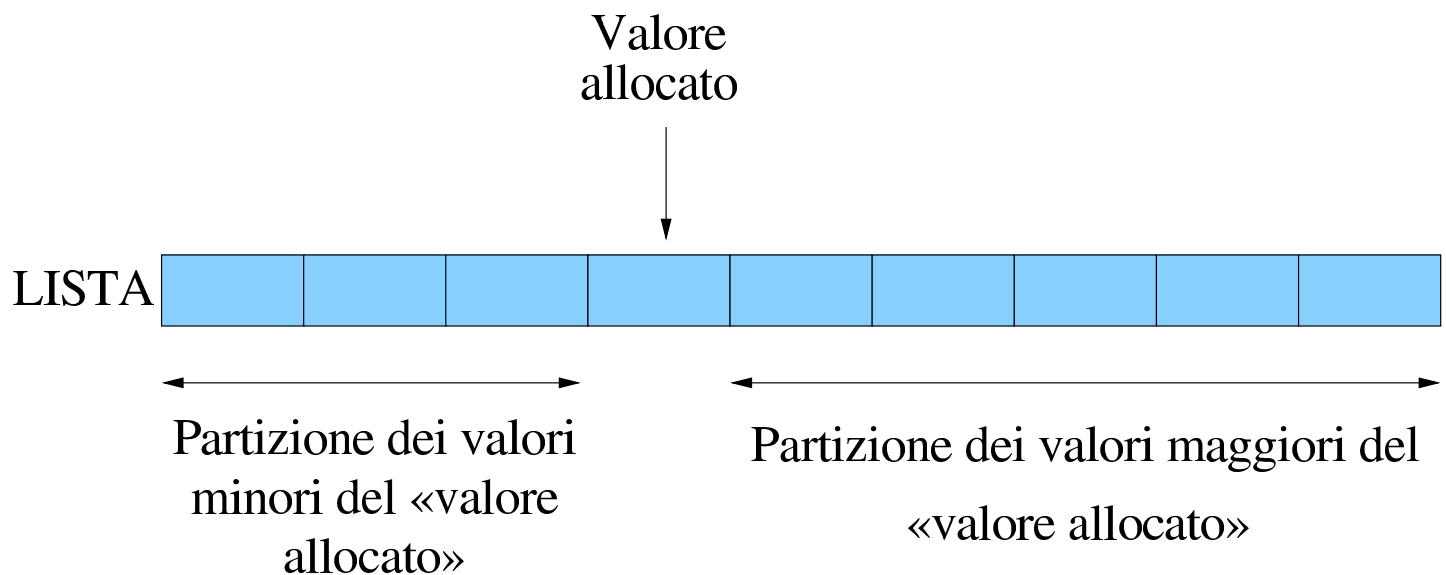
Nello stesso modo si potrebbe dimostrare il funzionamento per un numero maggiore di anelli.

#### 62.5.4 Quicksort (ordinamento non decrescente)

«

L'ordinamento degli elementi di un array è un problema tipico che si può risolvere in tanti modi. Il Quicksort è un algoritmo sofisticato, ottimo per lo studio della gestione degli array, oltre che per quello della ricorsione. Il concetto fondamentale di questo tipo di algoritmo è rappresentato dalla figura 62.33.

Figura 62.33. Il concetto base dell'algoritmo del Quicksort: suddivisione dell'array in due gruppi disordinati, separati da un valore piazzato correttamente nel suo posto rispetto all'ordinamento.



Una sola scansione dell'array è sufficiente per collocare definitivamente un elemento (per esempio il primo) nella sua destinazione finale e allo stesso tempo per lasciare tutti gli elementi con un valore inferiore a quello da una parte, anche se disordinati, e tutti quelli con un valore maggiore, dall'altra.

In questo modo, attraverso delle chiamate ricorsive, è possibile elaborare i due segmenti dell'array rimasti da riordinare.

L'algoritmo può essere descritto grossolanamente come:

1. localizzazione della collocazione finale del primo valore, separando in questo modo i valori;
2. ordinamento del segmento precedente all'elemento collocato definitivamente;
3. ordinamento del segmento successivo all'elemento collocato

definitivamente.

Viene qui indicato con '**PART**' la subroutine che esegue la scansione dell'array, o di un suo segmento, per determinare la collocazione finale (indice '**CF**') del primo elemento (dell'array o del segmento in questione).

Sia '**LISTA**' l'array da ordinare. Il primo elemento da collocare corrisponde inizialmente a '**LISTA[A]**' e il segmento di array su cui intervenire corrisponde a '**LISTA[A:Z]**' (cioè a tutti gli elementi che vanno dall'indice '**A**' all'indice '**Z**').

Alla fine della prima scansione, l'indice '**CF**' rappresenta la posizione in cui occorre spostare il primo elemento, cioè '**LISTA[A]**'. In pratica, '**LISTA[A]**' e '**LISTA[CF]**' vengono scambiati.

Durante la scansione che serve a determinare la collocazione finale del primo elemento, '**PART**' deve occuparsi di spostare gli elementi prima o dopo quella posizione, in funzione del loro valore, in modo che alla fine quelli inferiori o uguali a quello dell'elemento da collocare si trovino nella parte inferiore e gli altri dall'altra. In pratica, alla fine della prima scansione, gli elementi contenuti in '**LISTA[A: (CF-1)]**' devono contenere valori inferiori o uguali a '**LISTA[CF]**', mentre quelli contenuti in '**LISTA[(CF+1):Z]**' devono contenere valori superiori.

Indichiamo con '**QSORT**' la subroutine che esegue il compito complessivo di ordinare l'array. Il suo lavoro consisterebbe nel chiamare '**PART**' per collocare il primo elemento, continuando poi con la chiamata ricorsiva di se stessa per la parte di array precedente all'elemento collocato e infine alla chiamata ricorsiva per la parte restante di array.

Assumendo che **'PART'** e le chiamate ricorsive di **'QSORT'** svolgano il loro compito correttamente, si potrebbe fare un'analisi informale dicendo che se l'indice **'z'** **non** è maggiore di **'A'**, allora c'è un elemento (o nessuno) all'interno di **'LISTA[A:Z]'** e inoltre, **'LISTA[A:Z]'** è già nel suo stato finale. Se **'z'** è maggiore di **'A'**, allora (per assunzione) **'PART'** ripartisce correttamente **'LISTA[A:Z]'**. L'ordinamento separato dei due segmenti (per assunzione eseguito correttamente dalle chiamate ricorsive) completa l'ordinamento di **'LISTA[A:Z]'**.

Le figure 62.34 e 62.35 mostrano due fasi della scansione effettuata da **'PART'** all'interno dell'array o del segmento che gli viene fornito.

Figura 62.34. La scansione dell'array da parte di **'PART'** avviene portando in avanti l'indice **'I'** e portando indietro l'indice **'CF'**. Quando l'indice **'I'** localizza un elemento che contiene un valore maggiore di **'LISTA[A]'** e l'indice **'CF'** localizza un elemento che contiene un valore inferiore o uguale a **'LISTA[A]'**, gli elementi cui questi indici fanno riferimento vengono scambiati, quindi il processo di avvicinamento tra **'I'** e **'CF'** continua.

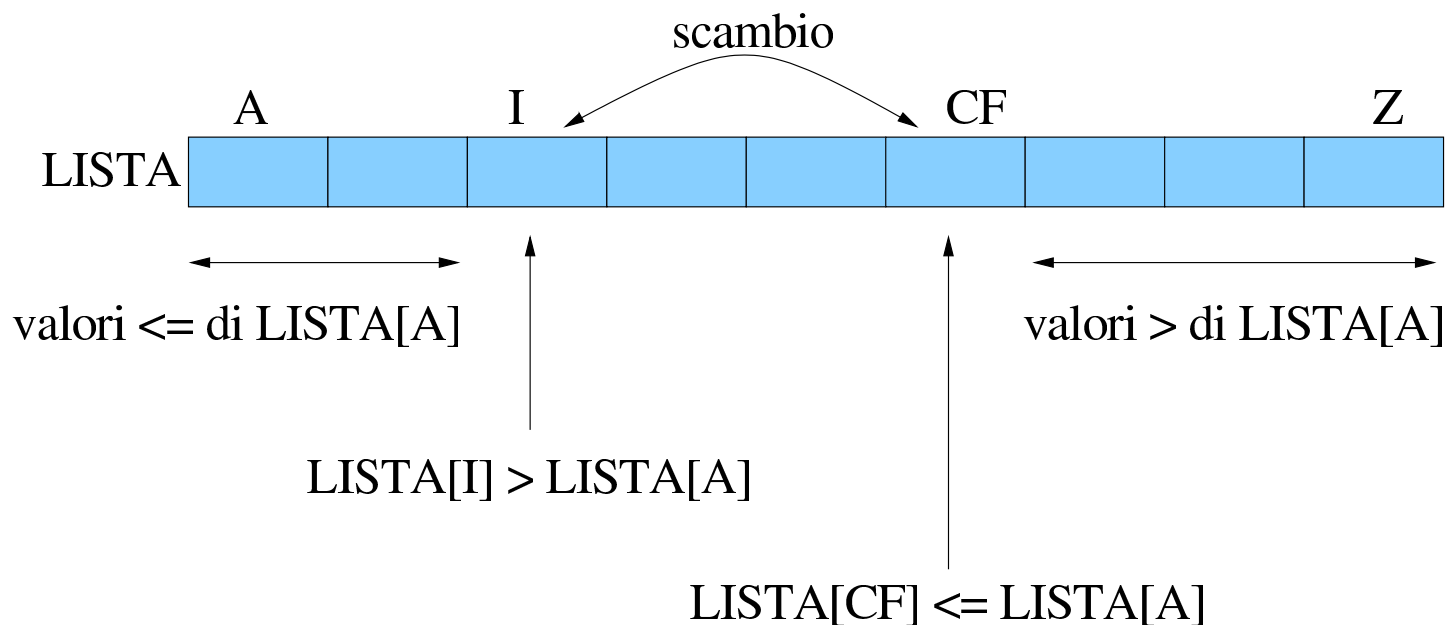
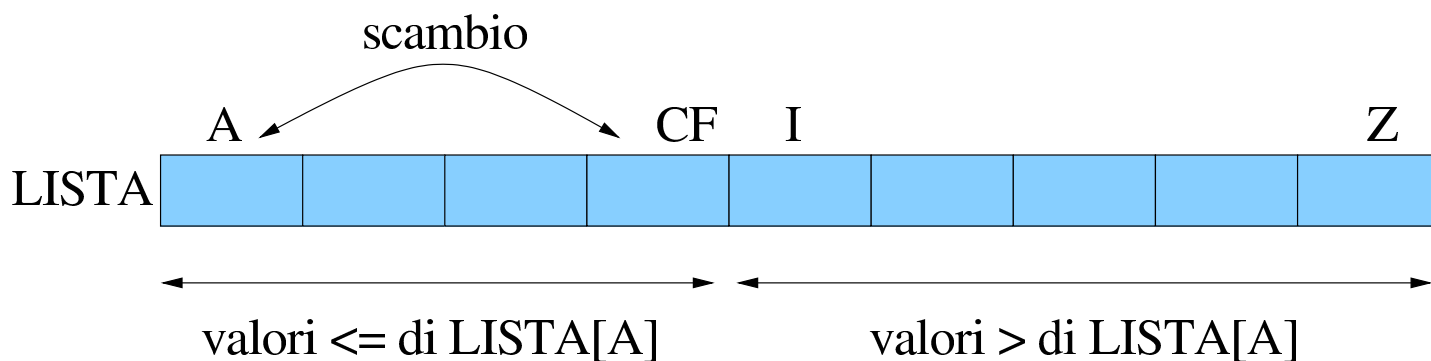


Figura 62.35. Quando la scansione è giunta al termine, quello che resta da fare è scambiare l'elemento '**LISTA[A]**' con '**LISTA[CF]**'.



In pratica, l'indice '**I**', iniziando dal valore '**A+1**', viene spostato verso destra fino a che viene trovato un elemento maggiore di '**LISTA[A]**', quindi è l'indice '**CF**' a essere spostato verso sinistra, iniziando dalla stessa posizione di '**Z**', fino a che viene incontrato un elemento minore o uguale a '**LISTA[A]**'. Questi elementi vengono scambiati e lo spostamento di '**I**' e '**CF**' riprende. Ciò prosegue fino a che '**I**' e '**CF**' si incontrano, momento in cui '**LISTA[A:Z]**' è stata ripartita e '**CF**' rappresenta l'indice di un elemento che si trova nella sua collocazione finale.

Variabile	Descrizione
<b>LISTA</b>	L'array da ordinare in modo crescente.
<b>A</b>	L'indice inferiore del segmento di array da ordinare.
<b>Z</b>	L'indice superiore del segmento di array da ordinare.
<b>CF</b>	Sta per «collocazione finale» ed è l'indice che cerca e trova la posizione giusta di un elemento nell'array.
<b>I</b>	È l'indice che insieme a ' <b>CF</b> ' serve a ripartire l'array.



## Segue la pseudocodifica delle due subroutine:

```
PART (LISTA, A, Z)

LOCAL I INTEGER
LOCAL CF INTEGER

# si assume che A < Z

I := A + 1
CF := Z

WHILE TRUE # ciclo senza fine.

    WHILE TRUE

        # sposta I a destra

        IF ((LISTA[I] > LISTA[A]) OR (I >= CF))
            THEN
                BREAK
            ELSE
                I := I + 1
            END IF

    END WHILE

    WHILE TRUE

        # sposta CF a sinistra

        IF (LISTA[CF] <= LISTA[A])
            THEN
                BREAK
            ELSE
                CF := CF - 1
            END IF

    END WHILE

END PART
```

```
        END IF

    END WHILE

    IF CF <= I
        THEN
            # è avvenuto l'incontro tra I e CF
            BREAK
        ELSE
            # vengono scambiati i valori
            LISTA[CF] ::= LISTA[I]
            I := I + 1
            CF := CF - 1
        END IF

    END WHILE

    # a questo punto LISTA[A:Z] è stata ripartita e CF è la
    # collocazione di LISTA[A]

    LISTA[CF] ::= LISTA[A]

    # a questo punto, LISTA[CF] è un elemento (un valore)
    # nella giusta posizione

    RETURN CF

END PART
```

```
QSORT (LISTA, A, Z)

    LOCAL CF INTEGER

    IF Z > A
        THEN
            CF := PART (@LISTA, A, Z)
            QSORT (@LISTA, A, CF-1)
            QSORT (@LISTA, CF+1, Z)
        END IF
    END QSORT
```

Vale la pena di osservare che l'array viene indicato nelle chiamate in modo che alla subroutine sia inviato un riferimento a quello originale, perché le variazioni fatte all'interno delle subroutine devono riflettersi sull'array originale.

### 62.5.5 Permutazioni

La permutazione è lo scambio di un gruppo di elementi posti in sequenza. Il problema che si vuole analizzare è la ricerca di tutte le permutazioni possibili di un dato gruppo di elementi.

Se ci sono  $n$  elementi in un array, allora alcune delle permutazioni si possono ottenere bloccando l' $n$ -esimo elemento e generando tutte le permutazioni dei primi  $n-1$  elementi. Quindi l' $n$ -esimo elemento può essere scambiato con uno dei primi  $n-1$ , ripetendo poi la fase precedente. Questa operazione deve essere ripetuta finché ognuno degli  $n$  elementi originali è stato usato nell' $n$ -esima posizione.

Variabile	Descrizione
LISTA	L'array da permutare.
A	L'indice inferiore del segmento di array da permutare.
Z	L'indice superiore del segmento di array da permutare.
K	È l'indice che serve a scambiare gli elementi.

Segue la pseudocodifica:

```
PERMUTA (LISTA, A, Z)

    LOCAL K INTEGER
    LOCAL N INTEGER

    IF (Z - A) >= 1
        # Ci sono almeno due elementi nel segmento di array.
        THEN
            FOR K := Z; K >= A; K--

                LISTA[K] ::= LISTA[Z]

                PERMUTA (LISTA, A, Z-1)

                LISTA[K] ::= LISTA[Z]

            END FOR
        ELSE
            scrivi LISTA
        END IF

    END PERMUTA
```

## 62.6 Gestione dei file

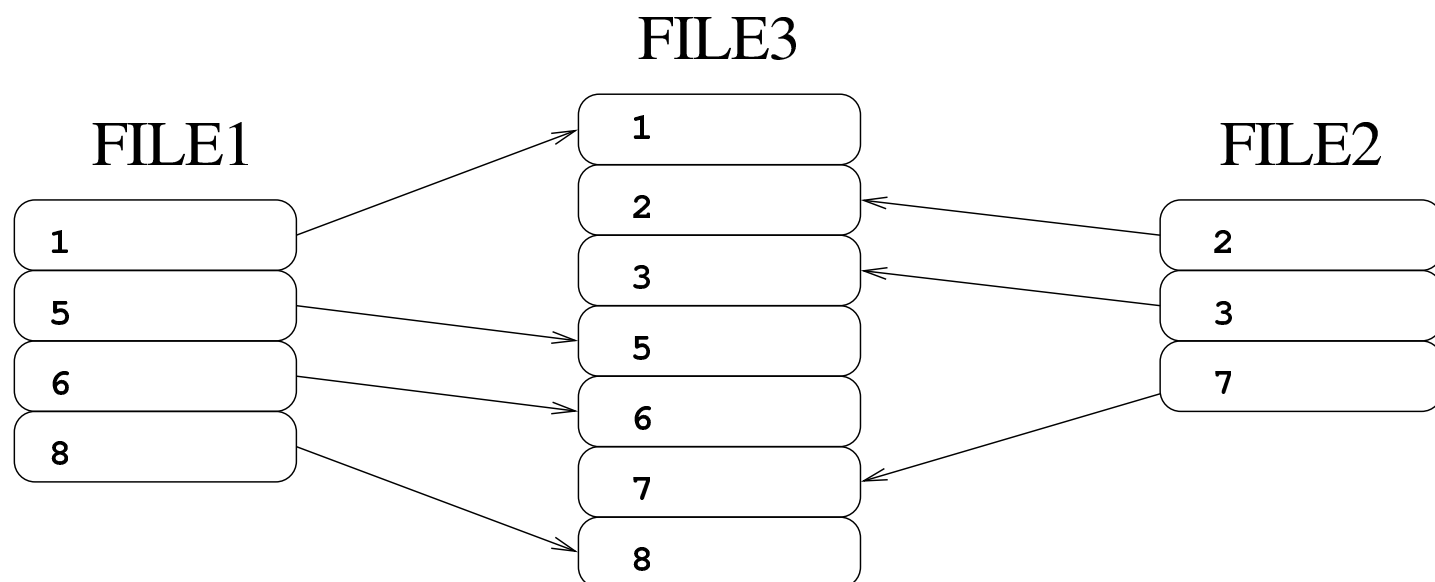
La gestione dei file è sempre la questione più complessa nello studio degli algoritmi, quanto si esclude la possibilità di gestire semplicemente tutto nella memoria centrale. In questo tipo di situazione, il file deve essere inteso come un insieme di record, che spesso sono di dimensione uniforme.

Nelle sezioni seguenti sono descritti alcuni problemi legati alla gestione dei file, con la soluzione in forma di pseudocodifica.

### 62.6.1 Fusione tra due file ordinati

La fusione di due file ordinati, aventi la stessa struttura, avviene leggendo un record da entrambi i file, confrontando la chiave di ordinamento e scrivendo nel file da ottenere il record con chiave più bassa. Successivamente, viene letto solo record successivo del file che conteneva la chiave più bassa e si ripete il confronto.

Figura 62.41. Il file 'FILE1' e 'FILE2' vengono fusi per generare il file 'FILE3'.



Segue un esempio di pseudocodifica per la soluzione del problema della fusione tra due file. La funzione riceve il riferimento ai file da elaborare e si può osservare l'utilizzo di variabili strutturate per accogliere i record dei file da elaborare. Come si può intuire, le variabili booleane il cui nome inizia per 'EOF', rappresentano l'avverarsi della condizione di «fine del file»; in pratica, quando contengono il valore *Vero*, indicano che la lettura del file a cui si riferiscono è andata oltre la conclusione del file.

```
FUSIONE_DUE_FILE (FILE_IN_1, FILE_IN_2, FILE_OUT)
```

```
LOCAL RECORD_1:
```

```
    CHIAVE_1      CHARACTER (8)
```

```
    DATI_1       CHARACTER (72)
```

```
LOCAL EOF_1 := FALSE
```

```
LOCAL RECORD_2:
```

```
    CHIAVE_2      CHARACTER (8)
```

```
    DATI_2       CHARACTER (72)
```

```
LOCAL EOF_2 := FALSE
```

```
LOCAL RECORD_3      CHARACTER (80)
```

```
OPEN INPUT  FILE_IN_1
```

```
OPEN INPUT  FILE_IN_2
```

```
OPEN OUTPUT FILE_OUT
```

```
READ FILE_IN_1 NEXT RECORD INTO RECORD_1
```

```
IF END OF FILE
```

```
    THEN
```

```
        EOF_1 := TRUE
```

```
    END IF
```

```
READ FILE_IN_2 NEXT RECORD INTO RECORD_2
```

```
IF END OF FILE
  THEN
    EOF_2 := TRUE
  END IF

UNTIL EOF_1 AND EOF_2

  IF EOF_1
    THEN
      RECORD_3 := RECORD_2
      READ FILE_IN_2 NEXT RECORD INTO RECORD_2
      IF END OF FILE
        THEN
          EOF_2 := TRUE
        END IF
    ELSE
      IF EOF_2
        THEN
          RECORD_3 := RECORD_1
          READ FILE_IN_1 NEXT RECORD INTO RECORD_1
          IF END OF FILE
            THEN
              EOF_1 := TRUE
            END IF
        ELSE
          IF CHIAVE_1 < CHIAVE_2
            THEN
              RECORD_3 := RECORD_1
              READ FILE_IN_1 NEXT RECORD INTO RECORD_1
              IF END OF FILE
                THEN
                  EOF_1 := TRUE
                END IF
            ELSE
```

```
        RECORD_3 := RECORD_2
        READ FILE_IN_2 NEXT RECORD INTO RECORD_2
        IF END OF FILE
            THEN
                EOF_2 := TRUE
            END IF
        END IF
    END IF
END IF

WRITE FILE_OUT RECORD FROM RECORD_3

END UNTIL

CLOSE FILE_IN_1
CLOSE FILE_IN_2
CLOSE FILE_OUT

END FUSIONE_DUE_FILE
```

## 62.6.2 Riordino attraverso la fusione



Un file non ordinato può essere ordinato, attraverso una serie di passaggi, che prevedono la divisione in due parti del file (ovvero **biforcazione**), contenenti le raccolte dei blocchi di record che risultano essere nella sequenza corretta, per poi fondere queste due parti e ripetere il procedimento. Le figure successive mostrano le due fasi: la separazione in due file, la fusione dei due file. Se il file risultante non è ordinato completamente, occorre procedere con una nuova fase di separazione e fusione. Si osservi, in particolare, che nelle figure, il file iniziale contiene solo tre blocchi di record in sequenza, pertanto,



l'ultimo di questi blocchi viene collocato nel file finale senza una fusione con un blocco corrispondente.

Figura 62.43. Biforcazione del file.

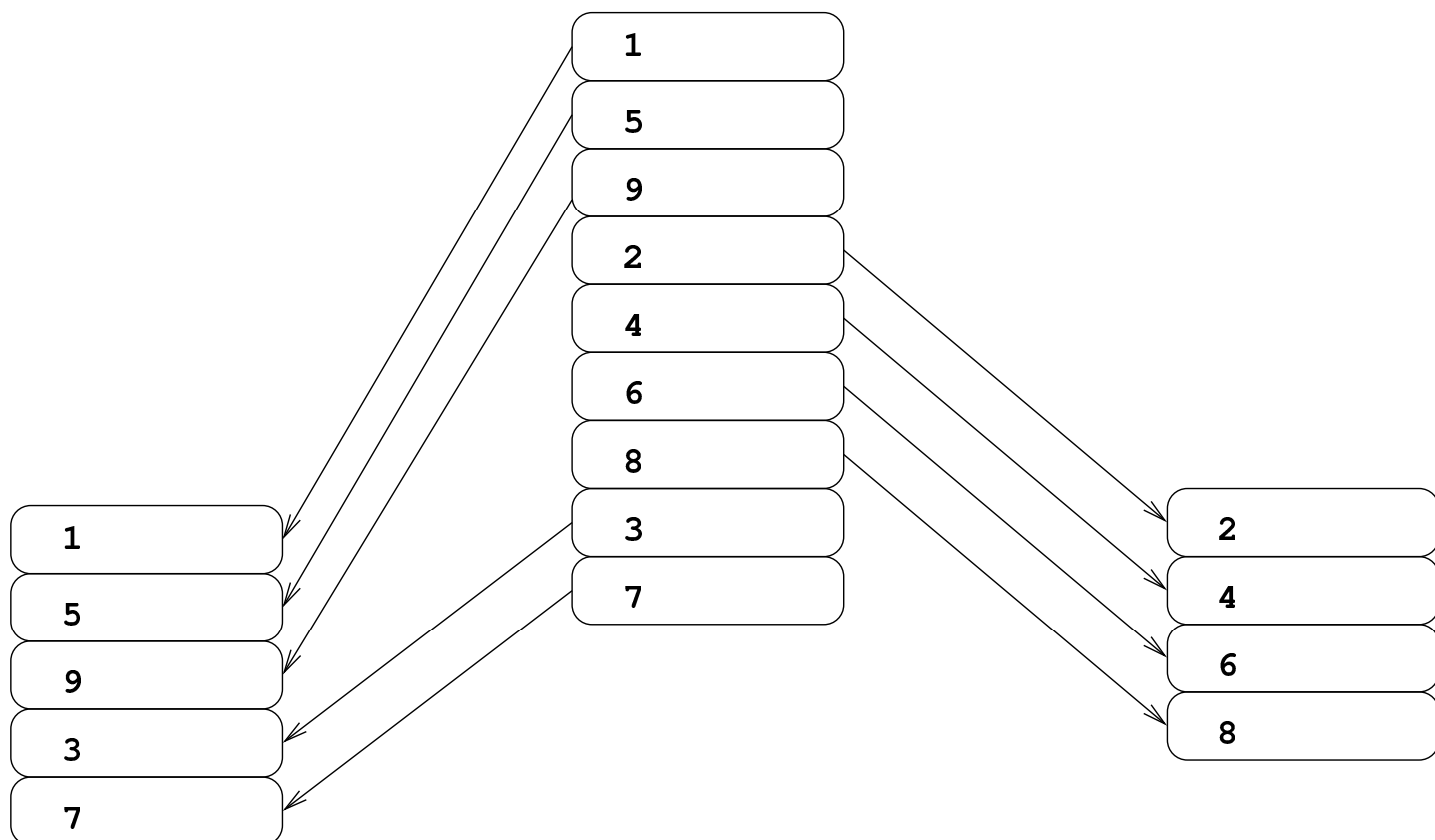
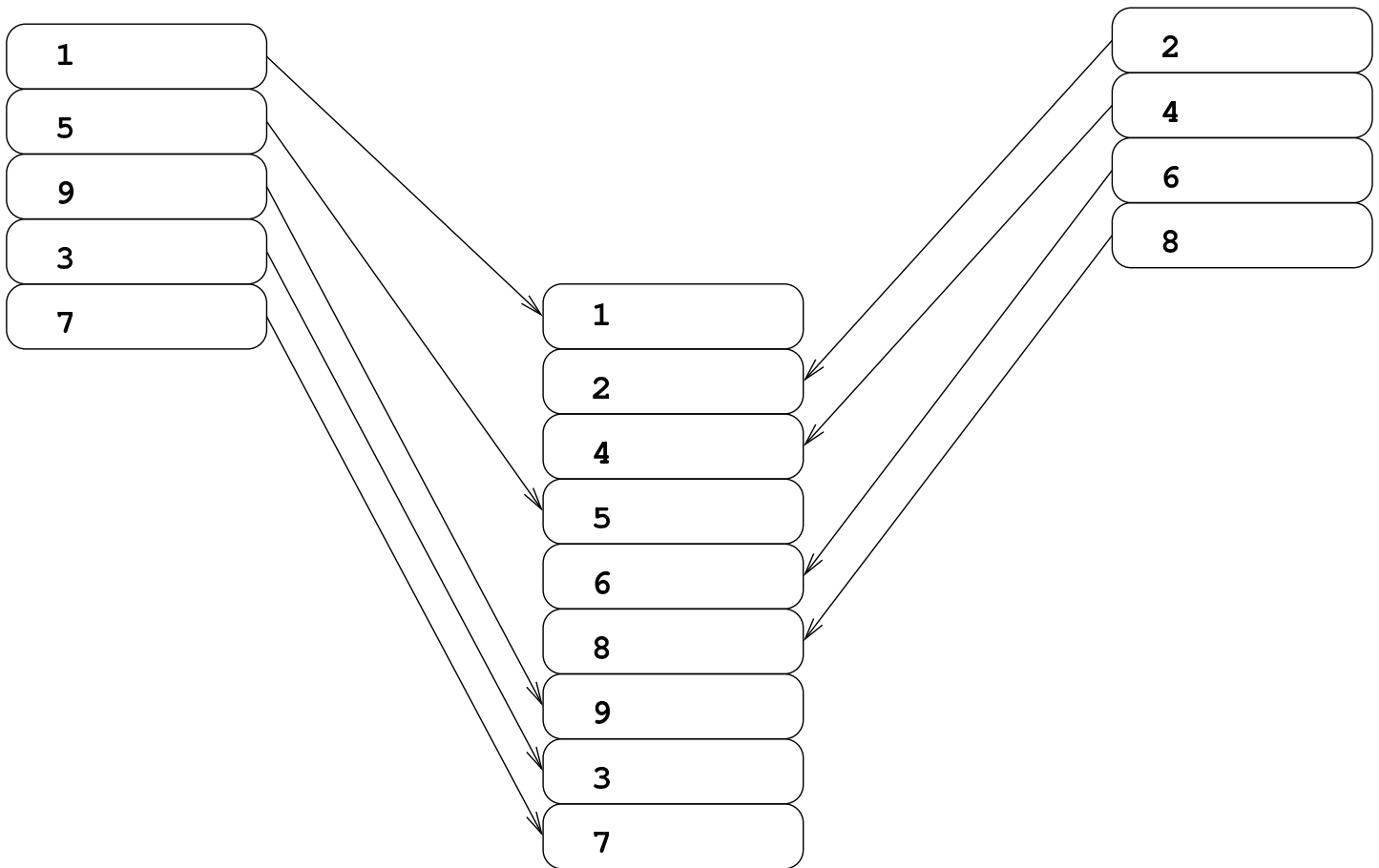


Figura 62.44. Fusione.



Per costruire un programma che utilizza questa tecnica di ordinamento, si può considerare che sia avvenuta l'ultima fase di biforcazione quando si contano un massimo di due blocchi; pertanto, la fase successiva di fusione produce sicuramente il file ordinato finale.

Segue un esempio di pseudocodifica per eseguire la biforcazione. Si osservi che i nomi dei file vengono passati in qualche modo, così che dopo la chiamata di questa procedura sia possibile riaprire tali file per la fusione; inoltre, l'informazione contenuta nella variabile **BIFORCAZIONI**, viene restituita come valore della chiamata della funzione, in modo da poter conoscere, dopo la chiamata, quante separazioni sono state eseguite nel file di partenza.

```
BIFORCA (FILE_IN_1, FILE_OUT_1, FILE_OUT_2)
```

```
LOCAL RECORD_IN_1:
    CHIAVE          CHARACTER (8)
    DATI            CHARACTER (72)
LOCAL CHIAVE_ORIG  CHARACTER (8)
LOCAL EOF_1 := FALSE

LOCAL RECORD_OUT_1 CHARACTER (80)
LOCAL RECORD_OUT_2 CHARACTER (80)

LOCAL SCAMBIO := 1
LOCAL BIFORCAZIONI := 0

OPEN INPUT  FILE_IN_1
OPEN OUTPUT FILE_OUT_1
OPEN OUTPUT FILE_OUT_2

READ FILE_IN_1 NEXT RECORD INTO RECORD_IN_1
IF END OF FILE
    THEN
        EOF_1 := TRUE
    ELSE
        BIFORCAZIONI++
        WRITE FILE_OUT_1 RECORD FROM RECORD_IN_1
        CHIAVE_ORIG := CHIAVE
        READ FILE_IN_1 NEXT RECORD INTO RECORD_IN_1
        IF END OF FILE
            THEN
                EOF_1 := TRUE
            END IF
        END IF
    END IF

UNTIL EOF_1
```

```
IF CHIAVE >= CHIAVE_ORIG
  THEN
    IF SCAMBIO == 1
      THEN
        WRITE FILE_OUT_1 RECORD FROM RECORD_IN_1
        CHIAVE_ORIG := CHIAVE
        READ FILE_IN_1 NEXT RECORD INTO RECORD_IN_1
        IF END OF FILE
          THEN
            EOF_1 := TRUE
          END IF
        ELSE
          WRITE FILE_OUT_2 RECORD FROM RECORD_IN_1
          CHIAVE_ORIG := CHIAVE
          READ FILE_IN_1 NEXT RECORD INTO RECORD_IN_1
          IF END OF FILE
            THEN
              EOF_1 := TRUE
            END IF
        END IF
      ELSE
        BIFORCAZIONI++
        CHIAVE_ORIG := CHIAVE
        IF SCAMBIO == 1
          THEN
            SCAMBIO := 2
          ELSE
            SCAMBIO := 1
          END IF
        END IF
      END IF
    END UNTIL

  CLOSE FILE_IN_1
  CLOSE FILE_OUT_1
```

```
CLOSE FILE_OUT_2

RETURN BIFORCAZIONI

END BIFORCA
```

Segue un esempio di pseudocodifica per eseguire la fusione a blocchi. Si osservi che i nomi dei file vengono passati in qualche modo, così che dopo la chiamata di questa procedura sia possibile riaprire tali file per la biforcazione e di nuovo per la fusione. Come si può intuire, le variabili booleane il cui nome inizia per 'EOB', rappresentano l'avverarsi della condizione di «fine del blocco» non decrescente; in pratica, quando contengono il valore *Vero*, indicano che la lettura del file a cui si riferiscono ha prodotto un record che ha una chiave inferiore rispetto a quello letto precedentemente, oppure che non sono disponibili altri record.

```
FUSIONE (FILE_IN_1, FILE_IN_2, FILE_OUT)

LOCAL RECORD_1:
    CHIAVE_1      CHARACTER (8)
    DATI_1        CHARACTER (72)
LOCAL CHIAVE_1_ORIG CHARACTER (8)
LOCAL EOF_1 := FALSE
LOCAL EOB_1 := FALSE

LOCAL RECORD_2:
    CHIAVE_2      CHARACTER (8)
    DATI_2        CHARACTER (72)
LOCAL CHIAVE_2_ORIG CHARACTER (8)
LOCAL EOF_2 := FALSE
LOCAL EOB_2 := FALSE
```

```
LOCAL RECORD_3          CHARACTER (80)

OPEN INPUT  FILE_IN_1
OPEN INPUT  FILE_IN_2
OPEN OUTPUT FILE_OUT

READ FILE_IN_1 NEXT RECORD INTO RECORD_1
IF END OF FILE
  THEN
    EOF_1 := TRUE
    EOB_1 := TRUE
  ELSE
    CHIAVE_1_ORIG := CHIAVE_1
END IF

READ FILE_IN_2 NEXT RECORD INTO RECORD_2
IF END OF FILE
  THEN
    EOF_2 := TRUE
    EOB_2 := TRUE
  ELSE
    CHIAVE_2_ORIG := CHIAVE_2
END IF

UNTIL EOF_1 AND EOF_2
  UNTIL EOB_1 AND EOB_2

  IF EOB_1
    THEN
      RECORD_3 := RECORD_2
      READ FILE_IN_2 NEXT RECORD INTO RECORD_2
      IF END OF FILE
        THEN
          EOF_2 := TRUE
```

```
        EOB_2 := TRUE
    END IF
ELSE
    IF EOB_2
    THEN
        RECORD_3 := RECORD_1
        READ FILE_IN_1 NEXT RECORD INTO RECORD_1
        IF END OF FILE
        THEN
            EOF_1 := TRUE
            EOB_1 := TRUE
        END IF
    ELSE
        IF CHIAVE_1 < CHIAVE_2
        THEN
            RECORD_3 := RECORD_1
            READ FILE_IN_1 NEXT RECORD INTO RECORD_1
            IF END OF FILE
            THEN
                EOF_1 := TRUE
                EOB_1 := TRUE
            ELSE
                IF CHIAVE_1 >= CHIAVE_1_ORIG
                THEN
                    CHIAVE_1_ORIG := CHIAVE_1
                ELSE
                    EOB_1 := TRUE
                END IF
            END IF
        END IF
    ELSE
        RECORD_3 := RECORD_2
        READ FILE_IN_2 NEXT RECORD INTO RECORD_2
        IF END OF FILE
        THEN
```

```
        EOF_2 := TRUE
        EOB_2 := TRUE
    ELSE
        IF CHIAVE_2 >= CHIAVE_2_ORIG
            THEN
                CHIAVE_2_ORIG := CHIAVE_2
            ELSE
                EOB_2 := TRUE
            END IF
        END IF
    END IF
END IF
END IF
END IF

WRITE FILE_OUT RECORD FROM RECORD_3

END UNTIL

IF NOT EOF_1
THEN
    EOB_1 := FALSE
END IF

IF NOT EOF_2
THEN
    EOB_2 := FALSE
END IF

END UNTIL

CLOSE FILE_IN_1
CLOSE FILE_IN_2
CLOSE FILE_OUT
```



```
END FUSIONE
```

Per poter riordinare effettivamente un file, utilizzando le procedure descritte, si può utilizzare la pseudocodifica seguente, che si avvale di quanto già descritto. Per non dover mostrare nella pseudocodifica come si dichiarano i file, si suppone che questo sia compito di un'altra porzione di codice assente, nel quale si chiama la procedura sottostante, indicando i riferimenti ai file da utilizzare. Si osservi che il file originale non viene modificato, producendo eventualmente un altro file ordinato.

```
ORDINAMENTO (FILE_IN, FILE_TMP_1, FILE_TMP_2, FILE_OUT)

LOCAL BIFORCAZIONI

BIFORCAZIONI := BIFORCA (@FILE_IN_1, @FILE_TMP_1,
                        @FILE_TMP_2)

#
# se la variabile BIFORCAZIONI contiene zero, significa
# che il file è vuoto.
#
IF BIFORCAZIONI > 0
  THEN
    FUSIONE (@FILE_TMP_1, @FILE_TMP_2, @FILE_OUT)
    WHILE BIFORCAZIONI > 2
      BIFORCAZIONI := BIFORCA (@FILE_OUT, @FILE_TMP_1,
                              @FILE_TMP_2)
      FUSIONE (@FILE_TMP_1, @FILE_TMP_2, @FILE_OUT)
    END WHILE
  END IF

END ORDINAMENTO
```

## 62.7 Trasformazione in lettere

« Nell'ambito della realizzazione di applicativi gestionali, capitano frequentemente problemi di conversione di numeri interi in una qualche forma alfabetica. In queste sezioni vengono mostrati degli algoritmi molto semplici per risolvere questo tipo di problemi.

### 62.7.1 Da numero a sequenza alfabetica pura

« Esiste un tipo di numerazione in cui si utilizzano solo le lettere dell'alfabeto, dalla «a» alla «z», senza accenti o altri simboli speciali, senza distinguere tra maiuscole e minuscole. In generale, i simboli da «a» a «z» consentono di rappresentare valori da 1 a 26, dove lo zero è escluso. Per rappresentare valori superiori, si possono accoppiare più lettere, ma il calcolo non è banale, proprio perché manca lo zero.

Attraverso la pseudocodifica introdotta nella sezione [62.2](#), si può descrivere una funzione che calcoli la stringa corrispondente a un numero intero positivo, maggiore di zero. Per prima cosa, occorre definire una sotto funzione che sia in grado di trasformare un numero intero, compreso tra 1 e 26 nella lettera alfabetica corrispondente; ciò si ottiene abbastanza facilmente, attraverso la verifica di più condizioni in cascata. Il vero problema, purtroppo, sta nel costruire una stringa composta da più lettere, quando si vuole rappresentare un valore superiore a 26. Non essendoci lo zero, diventa difficile fare i calcoli. Se si parte dal presupposto che il numero da convertire non possa essere superiore a 702, si sa con certezza che servono al massimo due lettere alfabetiche (perché la stringa «ZZ» corrisponderebbe proprio al numero 702); in tal caso, è sufficiente dividere il numero per 26, dove la parte intera rappresenta la prima lettera, mentre il re-

sto rappresenta la seconda. Tuttavia, se la divisione non dà resto, la stringa corretta è quella precedente. Per esempio, il numero 53 corrisponde alla stringa «BA», perché  $53/26 = 2$  con un resto di 1. Nello stesso modo, però, 52 si traduce nella stringa «AZ», perché  $52/26 = 2$ , ma non c'è resto, pertanto, «B\_» diventa «AZ».

La pseudocodifica seguente riepiloga il concetto in modo semplificato, dove, a seconda delle esigenze, la conversione è sempre limitata a un valore massimo. Le omissioni sono parti di codice facilmente intuibili.

```
INTEGER_TO_ALPHABET (N)

    ALPHABET_DIGIT (DIGIT)
        IF DIGIT = 0
            THEN
                RETURN ""
        ELSE IF DIGIT = 1
            THEN
                RETURN "A"
        ELSE IF DIGIT = 2
            THEN
                RETURN "B"
        ...
        ...
        ELSE IF DIGIT = 26
            THEN
                RETURN "Z"
        ELSE
            RETURN "##ERROR##"
        FI
    END ALPHABET_DIGIT

IF N <= 0
```

```
        THEN
            RETURN "##ERROR##";
ELSE IF N <= 26
    THEN
        RETURN ALPHABET_DIGIT (N)
ELSE IF N <= 52
    THEN
        N := N - 52
        RETURN "A" ALPHABET_DIGIT (N)
ELSE IF N <= 78
    THEN
        N := N - 78
        RETURN "B" ALPHABET_DIGIT (N)
...
...
ELSE IF N <= 702
    THEN
        N := N - 702
        RETURN "Z" ALPHABET_DIGIT (N)
ELSE IF N <= 728
    THEN
        N := N - 728
        RETURN "AA" ALPHABET_DIGIT (N)
ELSE IF N <= 754
    THEN
        N := N - 754
        RETURN "AB" ALPHABET_DIGIT (N)
...
...
ELSE
    RETURN "##ERROR##"
END IF
END INTEGER_TO_ALPHABET
```

## 62.7.2 Da numero a numero romano

La conversione di un numero intero positivo in una stringa che rappresenta un numero romano, ha un discreto livello di difficoltà, perché la numerazione romana non prevede lo zero, perché la tecnica prevede la somma e la sottrazione di simboli (a seconda della posizione) e poi perché diventa difficile indicare valori multipli delle migliaia.

Per prima cosa è necessario conoscere il valore associato ai simboli elementari:

Simbolo	Valore corrispondente
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Un simbolo posto alla destra di un altro simbolo con un valore maggiore o uguale di questo, viene sommato; al contrario, un simbolo posto alla sinistra di un altro simbolo con un valore maggiore di questo, viene sottratto. Per esempio, «VI» equivale a  $5+1$ , mentre «IV» equivale a  $5-1$ . Esistono comunque anche altri vincoli, per evitare di creare numeri difficili da interpretare a causa di una complessità di calcolo eccessiva.

Per risolvere il problema con un algoritmo relativamente semplice, si può scomporre il valore di partenza in fasce: unità, decine, centinaia e migliaia (la conversione di valori superiori genererebbe sol-

tanto una serie lunghissima di «M» che risulta poi troppo difficile da leggere).

```
INTEGER_TO_ROMAN (N)

    LOCAL DIGIT_1 INTEGER
    LOCAL DIGIT_2 INTEGER
    LOCAL DIGIT_3 INTEGER
    LOCAL DIGIT_4 INTEGER

    DIGIT_1 := 0
    DIGIT_2 := 0
    DIGIT_3 := 0
    DIGIT_4 := 0

    DIGIT_1_TO_ROMAN (DIGIT)
        IF DIGIT = 0
            THEN
                RETURN ""
            ELSE IF DIGIT = 1
                THEN
                    RETURN "I"
            ELSE IF DIGIT = 2
                THEN
                    RETURN "II"
            ELSE IF DIGIT = 3
                THEN
                    RETURN "III"
            ELSE IF DIGIT = 4
                THEN
                    RETURN "IV"
            ELSE IF DIGIT = 5
                THEN
                    RETURN "V"
            ELSE IF DIGIT = 6
```

```
        THEN
            RETURN "VI"
    ELSE IF DIGIT = 7
        THEN
            RETURN "VII"
    ELSE IF DIGIT = 8
        THEN
            RETURN "VIII"
    ELSE IF DIGIT = 9
        THEN
            RETURN "IX"
    END IF
END DIGIT_1_TO_ROMAN

DIGIT_2_TO_ROMAN (DIGIT)
    IF DIGIT = 0
        THEN
            RETURN ""
    ELSE IF DIGIT = 1
        THEN
            RETURN "X"
    ELSE IF DIGIT = 2
        THEN
            RETURN "XX"
    ELSE IF DIGIT = 3
        THEN
            RETURN "XXX"
    ELSE IF DIGIT = 4
        THEN
            RETURN "XL"
    ELSE IF DIGIT = 5
        THEN
            RETURN "L"
    ELSE IF DIGIT = 6
```

```
        THEN
            RETURN "LX"
    ELSE IF DIGIT = 6
        THEN
            RETURN "LXX"
    ELSE IF DIGIT = 8
        THEN
            RETURN "LXXX"
    ELSE IF DIGIT = 9
        THEN
            RETURN "XC"
    END IF
END DIGIT_2_TO_ROMAN

DIGIT_3_TO_ROMAN (DIGIT)
    IF DIGIT = 0
        THEN
            RETURN ""
    ELSE IF DIGIT = 1
        THEN
            RETURN "C"
    ELSE IF DIGIT = 2
        THEN
            RETURN "CC"
    ELSE IF DIGIT = 3
        THEN
            RETURN "CCC"
    ELSE IF DIGIT = 4
        THEN
            RETURN "CD"
    ELSE IF DIGIT = 5
        THEN
            RETURN "D"
    ELSE IF DIGIT = 6
```



```
        THEN
            RETURN "DC"
    ELSE IF DIGIT = 7
        THEN
            RETURN "DCC"
    ELSE IF DIGIT = 8
        THEN
            RETURN "DCCC"
    ELSE IF DIGIT = 9
        THEN
            RETURN "CM"
    END IF
END DIGIT_3_TO_ROMAN

DIGIT_4_TO_ROMAN (DIGIT)
    IF DIGIT = 0
        THEN
            RETURN ""
    ELSE IF DIGIT = 1
        THEN
            RETURN "M"
    ELSE IF DIGIT = 2
        THEN
            RETURN "MM"
    ELSE IF DIGIT = 3
        THEN
            RETURN "MMM"
    ELSE IF DIGIT = 4
        THEN
            RETURN "MMMM"
    ELSE IF DIGIT = 5
        THEN
            RETURN "MMMMM"
    ELSE IF DIGIT = 6
```

```
        THEN
            RETURN "MMMMMM"
        ELSE IF DIGIT = 7
            THEN
                RETURN "MMMMMMM"
        ELSE IF DIGIT = 8
            THEN
                RETURN "MMMMMMMM"
        ELSE IF DIGIT = 9
            THEN
                RETURN "MMMMMMMMM"
        END IF
    END DIGIT_4_TO_ROMAN

    DIGIT_4 := int (N/1000)
    N := N - (DIGIT_4 * 1000)

    DIGIT_3 := int (N/100)
    N := N - (DIGIT_3 * 100)

    DIGIT_2 := int (N/10)
    N := N - (DIGIT_2 * 10)

    DIGIT_1 := N

    RETURN DIGIT_4_TO_ROMAN (DIGIT_4)
           DIGIT_3_TO_ROMAN (DIGIT_3)
           DIGIT_2_TO_ROMAN (DIGIT_2)
           DIGIT_1_TO_ROMAN (DIGIT_1)

END INTEGER_TO_ROMAN
```

Come si vede, dopo aver scomposto il valore in quattro fasce, si utilizzano quattro funzioni distinte per ottenere la porzione di stringa

che traduce il valore relativo. L'istruzione '**RETURN**' finale intende concatenare tutte le stringhe risultanti.

### 62.7.3 Da numero a lettere, nel senso verbale

Quando si trasforma un numero in lettere, per esempio quando si vuole trasformare 123 in «centoventitre», l'algoritmo di conversione deve tenere conto delle convenzioni linguistiche e non esiste una soluzione generale per tutte le lingue.

Per quanto riguarda la lingua italiana, esistono nomi diversi fino al 19, poi ci sono delle particolarità per i plurali o i singolari. La pseudocodifica seguente risolve il problema in una sola funzione ricorsiva. Le omissioni dovrebbero essere sufficientemente intuitive.

```
INTEGER_TO_ITALIAN (N)

LOCAL X INTEGER
LOCAL Y INTEGER

IF N = 0
  THEN
    RETURN ""
ELSE IF N = 1
  THEN
    RETURN "UNO"
ELSE IF N = 2
  THEN
    RETURN "DUE"
ELSE IF N = 3
  THEN
    RETURN "TRE"
ELSE IF N = 4
  THEN
    RETURN "QUATTRO"
```

```
ELSE IF N = 5
  THEN
    RETURN "CINQUE"
ELSE IF N = 6
  THEN
    RETURN "SEI"
ELSE IF N = 7
  THEN
    RETURN "SETTE"
ELSE IF N = 8
  THEN
    RETURN "OTTO"
ELSE IF N = 9
  THEN
    RETURN "NOVE"
ELSE IF N = 10
  THEN
    RETURN "DIECI"
ELSE IF N = 11
  THEN
    RETURN "UNDICI"
ELSE IF N = 12
  THEN
    RETURN "DODICI"
ELSE IF N = 13
  THEN
    RETURN "TREDICI"
ELSE IF N = 14
  THEN
    RETURN "QUATTORDICI"
ELSE IF N = 15
  THEN
    RETURN "QUINDICI"
ELSE IF N = 16
```

```
    THEN
        RETURN "SEDICI"
ELSE IF N = 17
    THEN
        RETURN "DICIASSETTE"
ELSE IF N = 18
    THEN
        RETURN "DICIOOTTO"
ELSE IF N = 19
    THEN
        RETURN "DICIANNOVE"
ELSE IF N = 20
    THEN
        RETURN "VENTI"
ELSE IF N = 21
    THEN
        RETURN "VENTUNO"
ELSE IF (N >= 22) AND (N <= 29)
    THEN
        RETURN "VENTI" INTEGER_TO_ITALIAN (N-20)
ELSE IF N = 30
    THEN
        RETURN "TRENTA"
ELSE IF N = 31
    THEN
        RETURN "TRENTUNO"
ELSE IF (N >= 32) AND (N <= 39)
    THEN
        RETURN "TRENTA" INTEGER_TO_ITALIAN (N-30)

...
...
ELSE IF N = 90
    THEN
```

```
        RETURN "NOVANTA"  
ELSE IF N = 91  
    THEN  
        RETURN "NOVANTUNO"  
ELSE IF (N >= 92) AND (N <= 99)  
    THEN  
        RETURN "NOVANTA" INTEGER_TO_ITALIAN (N-90)  
ELSE IF (N >= 100) AND (N <= 199)  
    THEN  
        RETURN "CENTO" INTEGER_TO_ITALIAN (N-100)  
ELSE IF (N >= 200) AND (N <= 999)  
    THEN  
        X := int (N / 100)  
        Y := N - (X * 100)  
        RETURN INTEGER_TO_ITALIAN (X)  
            "CENTO"  
            INTEGER_TO_ITALIAN (Y)  
ELSE IF (N >= 1000) AND (N <= 1999)  
    THEN  
        RETURN "MILLE" INTEGER_TO_ITALIAN (N-1000)  
ELSE IF (N >= 2000) AND (N <= 999999)  
    THEN  
        X := int (N / 1000)  
        Y := N - (X * 1000)  
        RETURN INTEGER_TO_ITALIAN (X)  
            "MILA"  
            INTEGER_TO_ITALIAN (Y)  
ELSE IF (N >= 1000000) AND (N <= 1999999)  
    THEN  
        RETURN "UNMILIONE"  
            INTEGER_TO_ITALIAN (N-1000000)  
ELSE IF (N >= 2000000) AND (N <= 999999999)  
    THEN  
        X := int (N / 1000000)
```

```
    Y := N - (X * 1000000)
    RETURN INTEGER_TO_ITALIAN (X)
        "MILIONI"
        INTEGER_TO_ITALIAN (Y)
ELSE IF (N >= 1000000000) AND (N <= 1999999999)
    THEN
        RETURN "UNMILIARDO"
        INTEGER_TO_ITALIAN (N-1000000000)
ELSE IF (N >= 2000000000) AND (N <= 99999999999)
    THEN
        X := int (N / 1000000000)
        Y := N - (X * 1000000000)
        RETURN INTEGER_TO_ITALIAN (X)
            "MILIARDI"
            INTEGER_TO_ITALIAN (Y)
ELSE
    "##ERROR##"
END IF
END INTEGER_TO_ITALIAN
```

## 62.8 Algoritmi elementari con la shell POSIX

Questa sezione raccoglie degli esempi di programmazione per lo studio elementare degli algoritmi, realizzati in forma di script per una shell POSIX. Questi esempi sono ottenuti ricostruendo un lavoro didattico del 1983, realizzato allora con degli script ARCS, un linguaggio del sistema operativo CMS (*Computer management system*) Burroughs.



## 62.8.1 ARCS0: ricerca del valore più grande tra tre numeri interi

&lt;&lt;

Una copia di questo file dovrebbe essere disponibile presso [allegati/sh/ARCS0.sh](#) .

```
#!/bin/sh
##
## ARCS0 1983-07-02
##
## Trovare il più grande fra tre numeri interi, diversi tra
## loro.
##

printf "inserisci il primo numero  "
read a

printf "inserisci il secondo numero "
read b

printf "inserisci il terzo numero  "
read c

if [ $a -gt $b ]
then
    if [ $a -gt $c ]
    then
        echo "il numero maggiore è $a"
    else
        echo "il numero maggiore è $c"
    fi
else
    if [ $b -gt $c ]
    then
        echo "il numero maggiore è $b"
    else
```



```
        echo "il numero maggiore è $c"
    fi
fi
```

## 62.8.2 ARCS1: moltiplicazione di due numeri interi

Una copia di questo file dovrebbe essere disponibile presso [allegati/sh/ARCS1.sh](#) .

```
##
## ARCS1 1983-07-06
##
## Moltiplicazione di due numeri.
##

printf "inserisci il primo numero - 0 per finire "
read x
printf "inserisci il secondo numero - 0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    z=0
    while [ $y -ne 0 ]
    do
        z=$((z + $x))
        y=$((y - 1))
    done
    echo "il risultato è $z"
    printf "inserisci il primo numero - 0 per finire "
    read x
    printf "inserisci il secondo numero - 0 per finire "
    read y
done
```

### 62.8.3 ARCS2: valore assoluto della differenza tra due valori

&lt;&lt;

Una copia di questo file dovrebbe essere disponibile presso [allegati/sh/ARCS2.sh](#) .

```
#!/bin/sh
##
## ARCS2 1983-07-06
##
## Valore assoluto della differenza tra due numeri.
##

printf "inserisci x    0 per finire "
read x
printf "inserisci y    0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    if [ $x -lt $y ]
    then
        u=$x
        x=$y
        y=$u
    fi
    z=$(( $x - $y ))
    echo "|x-y| = $z"
    printf "inserisci x    0 per finire "
    read x
    printf "inserisci y    0 per finire "
    read y
done
```

## 62.8.4 ARCS3: somma tra due numeri

Una copia di questo file dovrebbe essere disponibile presso [allegati/sh/ARCS3.sh](#) .

```
#!/bin/sh
##
## ARCS3 1983-07-07
##
## Somma tra due numeri.
##

printf "inserisci x    0 per finire "
read x
printf "inserisci y    0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    while [ $y -ne 0 ]
    do
        x=$(( $x + 1 ))
        y=$(( $y - 1 ))
    done
    echo "x+y = $x"
    printf "inserisci x    0 per finire "
    read x
    printf "inserisci y    0 per finire "
    read y
done
```

## 62.8.5 ARCS4: prodotto tra due numeri

Una copia di questo file dovrebbe essere disponibile presso [allegati/sh/ARCS4.sh](#) .

```
#!/bin/sh
##
## ARCS4 1983-07-07
##
## Prodotto tra due numeri.
##

printf "inserisci x    0 per finire "
read x
printf "inserisci y    0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    z=0
    t=$y
    while [ $t -ne 0 ]
    do
        u=$z
        v=$x
        while [ $v -ne 0 ]
        do
            u=$(( $u + 1 ))
            v=$(( $v - 1 ))
        done
        z=$u
        t=$(( $t - 1 ))
    done
    echo "x*y = $z"
    printf "inserisci x    0 per finire "
    read x
    printf "inserisci y    0 per finire "
    read y
done
```

## 62.8.6 ARCS5: quoziente

Una copia di questo file dovrebbe essere disponibile presso [allegati/sh/ARCS5.sh](#) .

```
#!/bin/sh
##
## ARCS5 1983-07-07
##
## Quoziente.
##

printf "inserisci x    0 per finire "
read x
printf "inserisci y    0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    if [ $y -eq 0 ]
    then
        echo "x / 0 = indefinito"
    else
        z=0
        if [ $x -ge $y ]
        then
            u=$x
            until [ $u -le 0 ]
            do
                u=$(( $u - $y ))
                if [ $u -ge 0 ]
                then
                    z=$(( $z + 1 ))
                fi
            done
            echo "x / y = $z"
        else
```

```
        echo "x / y = $z"
    fi
fi
printf "inserisci x    0 per finire "
read x
printf "inserisci y    0 per finire "
read y
done
```

### 62.8.7 ARCS6: verifica della parità di un numero

«

Una copia di questo file dovrebbe essere disponibile presso [allegati/sh/ARCS6.sh](#) .

```
#!/bin/sh
##
## ARCS6 1983-07-07
##
## Verifica della parità di un numero.
##

printf "inserisci x    0 per finire "
read x
until [ $x -eq 0 ]
do
    y=$((($x / 2) * 2))
    if [ $x -ne $y ]
    then
        echo "il numero $x è dispari"
    else
        echo "il numero $x è pari"
    fi
    printf "inserisci x    0 per finire "
    read x
done
```

## 62.8.8 ARCS7: fattoriale

Una copia di questo file dovrebbe essere disponibile presso [allegati/sh/ARCS7.sh](#) .

```
#!/bin/sh
##
## ARCS7 1983-07-08
##
## Fattoriale di un numero.
##

printf "inserisci il numero      99 per finire "
read n
until [ $n -eq 99 ]
do
    z=1
    k=0
    while [ $k -ne $n ]
    do
        k=$(( $k + 1 ))
        z=$(( $z * $k ))
    done
    echo "$n! = $z"
    printf "inserisci il numero      99 per finire "
    read n
done
```

## 62.8.9 ARCS8: coefficiente binomiale

Una copia di questo file dovrebbe essere disponibile presso [allegati/sh/ARCS8.sh](#) .

```
#!/bin/sh
##
## ARCS8 1983-07-08
```

```
##
## Coefficiente binomiale.
##
## /n\  =  n*(n-1)*...*(n-k+1) / k!
## \k/
##

printf "inserisci n      999 per finire "
read a
printf "inserisci k      999 per finire "
read b
until [ $a -eq 999 ] && [ $b = 999 ]
do
    if [ $b -eq 0 ]
    then
        echo "il coefficiente binomiale di n su 0 è 1"
    else
        k1=0
        z1=1
        while [ $k1 -ne $b ]
        do
            k1=$(( $k1 + 1 ))
            z1=$(( z1 * k1 ))
        done
        y=$z1
        k2=$(( $a - $b ))
        z2=1
        while [ $k2 -ne $a ]
        do
            k2=$(( $k2 + 1 ))
            z2=$(( $z2 * $k2 ))
        done
        x=$z2
        z=$(( $x / $y ))
    fi
done
```



```

    echo "il coefficiente binomiale è $x/$y"
    echo "che se viene calcolato con approssimazione"
    echo "di una unità, dà $z"
fi
printf "inserisci n      999 per finire "
read a
printf "inserisci k      999 per finire "
read b
done

```

## 62.8.10 ARCS10: massimo comune divisore

Una copia di questo file dovrebbe essere disponibile presso [allegati/sh/ARCS10.sh](#) .

```

#!/bin/sh
##
## ARCS10 1983-07-09
##
## M.C.D.
##

printf "inserisci x      0 per finire "
read x
printf "inserisci y      0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    z=$x
    w=$y
    while [ $z -ne $w ]
    do
        if [ $z -gt $w ]
        then
            z=$(( $z - $w ))

```

```
        else
            w=$(( $w - $z ))
        fi
    done
    echo "il M.C.D. di $x e $y è $z"
    printf "inserisci x      0 per finire "
    read x
    printf "inserisci y      0 per finire "
    read y
done
```

## 62.8.11 ARCS11: massimo comune divisore

«

Una copia di questo file dovrebbe essere disponibile presso [allegati/sh/ARCS11.sh](#).

```
#!/bin/sh
##
## ARCS11 1983-07-09
##
## M.C.D.
##

printf "inserisci x      0 per finire "
read x
printf "inserisci y      0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    z=$x
    w=$y
    while [ $z -ne $w ]
    do
        while [ $z -gt $w ]
```

```

        z=$(( $z - $w ))
    done
    while [ $w -gt $z ]
    do
        w=$(( $w - $z ))
    done
done
echo "il M.C.D. di $x e $y è $z"
printf "inserisci x      0 per finire "
read x
printf "inserisci y      0 per finire "
read y
done

```

## 62.8.12 ARCS12: radice quadrata intera

Una copia di questo file dovrebbe essere disponibile presso [allegati/sh/ARCS12.sh](#) .

```

#!/bin/sh
##
## ARCS12 1983-07-09
##
## radice quadrata intera
##

printf "inserisci il numero di cui vuoi la radice "
printf "      0 per finire "
read x
until [ $x -eq 0 ]
do
    z=0
    t=0
    until [ $t -ge $x ]
    do

```

```
        z=$(( $z + 1 ))
        t=$(( $z * $z ))
done
if [ $t -ne $x ]
then
    z=$(( $z - 1 ))
fi
echo "la radice intera di $x è $z"
printf "inserisci il numero di cui vuoi la radice "
printf "    0 per finire "
read x
done
```

### 62.8.13 ARCS13: numero primo



Una copia di questo file dovrebbe essere disponibile presso [allegati/sh/ARCS13.sh](#).

```
#!/bin/sh
##
## ARCS13 1983-07-09
##
## il numero è primo?
##

printf "inserisci il numero    9999 per finire "
read x
until [ $x -eq 9999 ]
do
    primo=1
    k=2
    while [ $k -lt $x ] && [ $primo -eq 1 ]
    do
        t=$(( $x / $k ))
        t=$(( $x - ($t * $k) ))
```

```
        if [ $t -eq 0 ]
        then
            primo=0
        else
            k=$(( $k + 1 ))
        fi
    done
    if [ $primo -eq 1 ]
    then
        echo "il numero $x è primo"
    else
        echo "il numero $x non è primo"
    fi
    printf "inserisci il numero          9999 per finire "
    read x
done
```

## 62.8.14 ARCS14: numero primo

Una copia di questo file dovrebbe essere disponibile presso [allegati/sh/ARCS14.sh](#) .

```
#!/bin/sh
##
## ARCS14 1983-07-09
##
## il numero è primo?
## questa versione non funziona correttamente con 0 e 1.
##

printf "inserisci il numero          9999 per finire "
read x
until [ $x -eq 9999 ]
do
    primo=1
```

```
z=0
t=0
until [ $t -ge $x ]
do
    z=$(( $z + 1 ))
    t=$(( $z * $z ))
done
if [ $t -ne $x ]
then
    z=$(( $z - 1 ))
else
    primo=0
fi
k=2
while [ $k -lt $x ] && [ $primo -eq 1 ]
do
    t=$(( $x / $k ))
    t=$(( $x - ( $t * $k )) )
    if [ $t -eq 0 ]
    then
        primo=0
    else
        k=$(( $k + 1 ))
    fi
done
if [ $primo -eq 1 ]
then
    echo "il numero $x è primo"
else
    echo "il numero $x non è primo"
fi
printf "inserisci il numero          9999 per finire "
read x
done
```

## 62.9 Riferimenti



- Th. Estier, *What is BNF notation?*, <http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html>





## Linguaggio macchina



63.1	Organizzazione della memoria .....	99
63.1.1	Pila per salvare i dati .....	99
63.1.2	Chiamate di funzioni .....	100
63.1.3	Funzioni attraverso le istruzioni di salto .....	107
63.1.4	Variabili e array .....	110
63.1.5	Gestione alternativa degli indici .....	115
63.1.6	Ordine dei byte .....	119
63.1.7	Stringhe, array e puntatori .....	120
63.1.8	Utilizzo della memoria .....	122
63.2	Architettura, linguaggio, contesto virtuale, terminologia 124	
63.2.1	Memoria e registri .....	124
63.2.2	Indicatori o «flag» .....	126
63.2.3	«Opcode» .....	127
63.2.4	Accesso alla memoria .....	127
63.2.5	Modello della memoria nei sistemi Unix .....	129
63.2.6	Sintassi «AT&T» e «Intel» .....	131
63.2.7	Macchina virtuale .....	131
63.2.8	Compilazione e collegamento .....	132
63.3	Rappresentazione di valori numerici .....	133
63.3.1	Codifica delle singole cifre .....	134
63.3.2	Rappresentazione binaria di numeri interi senza segno 136	

63.3.3	Rappresentazioni binarie superate di numeri interi con segno .....	136
63.3.4	Complemento a due .....	138
63.3.5	Rappresentazione binaria di numeri in virgola mobile 140	
63.3.6	Rappresentazione in virgola mobile IEEE 754 ..	142
63.3.7	Ordine dei byte .....	146
63.4	Calcoli con i valori binari rappresentati nella forma usata negli elaboratori .....	147
63.4.1	Modifica della quantità di cifre di un numero binario intero .....	147
63.4.2	Sommatorie con i valori interi con segno .....	149
63.4.3	Somme e sottrazioni con i valori interi senza segno 152	
63.4.4	Somme e sottrazioni in fasi successive .....	156
63.4.5	Indicatori .....	160
63.5	Scorrimenti, rotazioni, operazioni logiche .....	163
63.5.1	Scorrimento logico .....	164
63.5.2	Scorrimento aritmetico .....	164
63.5.3	Scorrimento e indicatori .....	165
63.5.4	Moltiplicazione .....	167
63.5.5	Divisione .....	167
63.5.6	Rotazione .....	168
63.5.7	Rotazione e indicatori .....	169
63.5.8	Operatori logici .....	170

63.5.9	Intervenire su bit singoli .....	172
63.5.10	Somme e sottrazioni abbinate agli operatori logici	173
63.6	Confronti attraverso la sottrazione .....	174
63.6.1	Confronto di valori senza segno .....	174
63.6.2	Confronto di valori con segno .....	175
63.7	Riferimenti .....	177

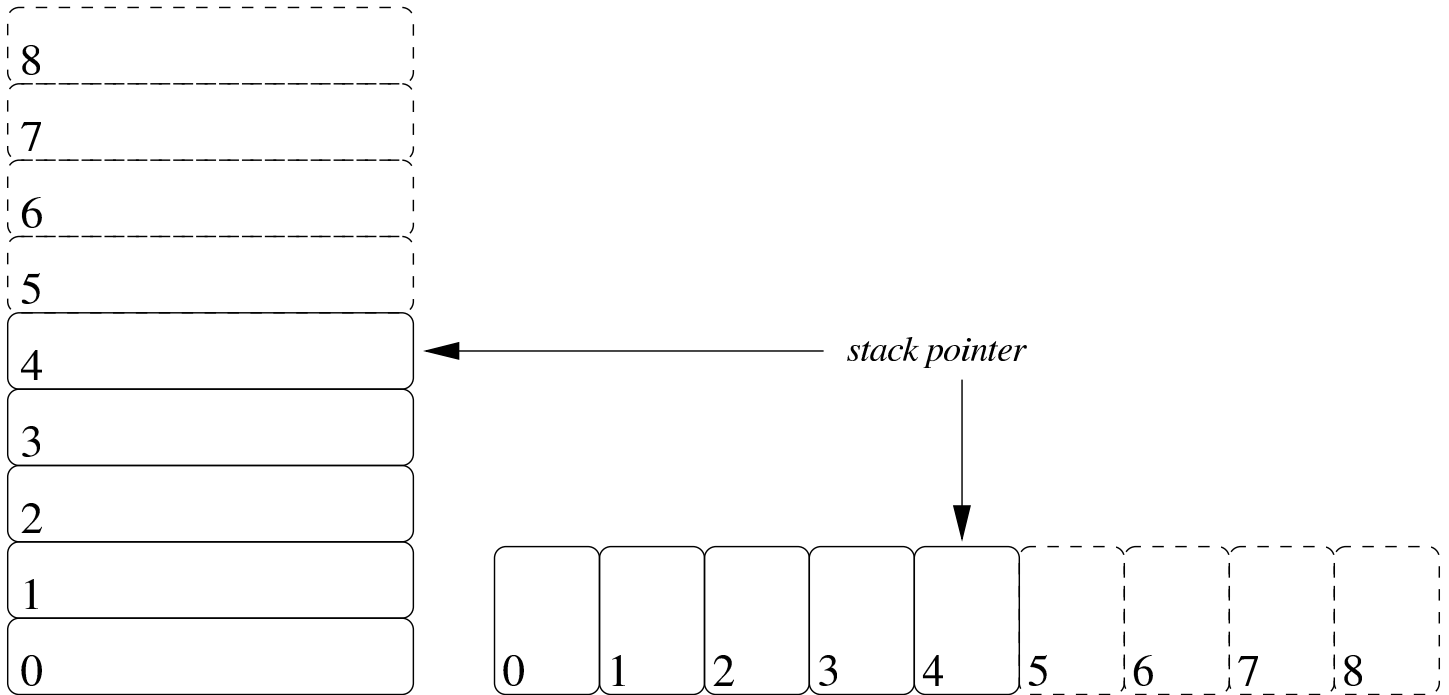
## 63.1 Organizzazione della memoria

Qui si introduce il problema dell'organizzazione della memoria, da un punto di vista molto vicino a quello della realtà fisica dell'elaboratore. In particolare, viene utilizzata la pseudocodifica, già descritta nella sezione 62.2, per dimostrare in parte il funzionamento e l'utilizzo della pila dei dati, di cui, normalmente, ogni programma dispone.

### 63.1.1 Pila per salvare i dati

Quando si scrive con un linguaggio di programmazione molto vicino a quello effettivo del microprocessore, si ha normalmente a disposizione una pila di elementi omogenei (*stack*), usata per accumulare temporaneamente delle informazioni, da espellere poi in senso inverso. Questa pila è gestita attraverso un vettore, dove l'ultimo elemento (quello superiore) è individuato attraverso un indice noto come *stack pointer* e tutti gli elementi della pila sono comunque accessibili, in lettura e in sovrascrittura, se si conosce la loro posizione relativa.

Figura 63.1. Una pila che può contenere al massimo nove elementi, rappresentata nel modo tradizionale, oppure distesa, come si fa per i vettori. Gli elementi che si trovano oltre l'indice (lo *stack pointer*) non sono più disponibili, mentre gli altri possono essere letti e modificati senza doverli estrarre dalla pila.



Per accumulare un dato nella pila (*push*) si incrementa di una unità l'indice e lo si inserisce in quel nuovo elemento. Per estrarre l'ultimo elemento dalla pila (*pop*) si legge il contenuto di quello corrispondente all'indice e si decrementa l'indice di una unità.

Tra le altre cose, la pila può servire quando si dispone di una quantità limitata di variabili e si devono accumulare temporaneamente dei valori.

### 63.1.2 Chiamate di funzioni

«

I linguaggi di programmazione più vicini alla realtà fisica della memoria di un elaboratore, possono disporre solo di variabili globali ed eventualmente di una pila, realizzata attraverso un vettore, come de-

scritto nella sezione precedente. In questa situazione, la chiamata di una funzione può avvenire solo passando i parametri in uno spazio di memoria condiviso da tutto il programma. Ma per poter generalizzare le funzioni e per consentire la ricorsione, ovvero per rendere le funzioni *rientranti*, il passaggio dei parametri deve avvenire attraverso la pila in questione.

Per mostrare un esempio iniziale che consenta di comprendere il meccanismo, si può supporre di poter utilizzare delle variabili locali nelle funzioni, mentre per il passaggio dei valori si deve usare la pila. Si vuole trasformare il codice della pseudocodifica seguente in modo da utilizzare tale pila. Si consideri che il programma inizia e finisce nella funzione *MAIN()*, all'interno della quale si fa la chiamata della funzione *SOMMA()*:

```
SOMMA (X, Y)
    LOCAL Z INTEGER
    Z := X + Y
    RETURN Z
END SOMMA

MAIN ()
    LOCAL A INTEGER
    LOCAL B INTEGER
    LOCAL C INTEGER
    A := 3
    B := 4
    C := SOMMA (A, B)
END MAIN
```

Il programma si trasforma in modo da accumulare nel vettore *PILA[]* i valori dei parametri della chiamata:

```
GLOBAL PILA[1000] INTEGER
GLOBAL SP          INTEGER
SP := -1

SOMMA ()
    LOCAL X := PILA[SP]           # Copia il valore del primo
                                  # parametro.
    LOCAL Y := PILA[SP - 1]       # Copia il valore del
                                  # secondo parametro.
    LOCAL Z INTEGER
    Z := X + Y

    SP++                           # Accumula il risultato
    PILA[SP] := Z                  # della somma.

END SOMMA

MAIN ()
    LOCAL A INTEGER
    LOCAL B INTEGER
    LOCAL C INTEGER
    A := 3
    B := 4

    SP++                           # Accumula il secondo parametro.
    PILA[SP] := B                  # nella pila.

    SP++                           # Accumula il primo parametro.
    PILA[SP] := A                  # nella pila.

    SOMMA ()                       # Chiama la funzione SOMMA().

    C := PILA[SP]                  #
    SP--                           # Estrae il risultato.
```

```

SP--          # Elimina il primo parametro
              # della chiamata.
SP--          # Elimina il secondo parametro
              # della chiamata.
END MAIN

```

Nella nuova versione della pseudocodifica, la chiamata della funzione *SOMMA()* è preceduta dall'accumulo nella pila dei parametri, quindi è seguita dall'estrazione del risultato della somma e dall'eliminazione dei due parametri usati nella chiamata (con la sola riduzione del valore dell'indice del vettore). All'interno della funzione *SOMMA()* si acquisiscono i parametri leggendo i dati corrispondenti dal vettore che li convoglia, sapendo che il primo è nella posizione dell'indice (in quanto è l'ultimo elemento inserito nella pila) e che il secondo è nella posizione precedente. Alla fine, dopo l'esecuzione della somma, il risultato viene inserito nella pila.

L'esempio seguente risolve il problema del calcolo del fattoriale, in modo ricorsivo, seguendo la modalità appena descritta:

```

GLOBAL PILA[1000] INTEGER
GLOBAL SP          INTEGER
SP := -1

FATTORIALE ( )
    LOCAL X := PILA[SP]
    LOCAL W INTEGER

    IF X == 1
        THEN
            SP++          # Accumula il risultato da
            PILA[SP] := 1 # restituire, pari a uno.

```

```
    ELSE
        SP++           # Accumula il parametro di
        PILA[SP] := X - 1 # chiamata della funzione con
                        # un valore pari a X - 1.

        FATTORIALE ()

        W := PILA[SP]  # Recupera il risultato
        SP--          # della chiamata ricorsiva
                        # con un valore pari a
                        # X - 1.

        SP--          # Scarica il parametro usato
                        # per la chiamata.

        SP++          # Accumula il risultato del
        PILA[SP] := X * W # fattoriale da restituire.
    END IF

END FATTORIALE

MAIN ()
    LOCAL F INTEGER
    F := 7

    SP++           # Accumula il valore di cui si
    PILA[SP] := F  # vuole calcolare il fattoriale.

    FATTORIALE () # Calcola il fattoriale.

    F := PILA[SP]  # Estrae il risultato del
    SP--          # fattoriale e scarica il
                        # valore dalla pila.

    SP--          # Scarica la pila del parametro
```



```
# usato nella chiamata.
```

```
END MAIN
```

Se non si possono gestire variabili locali, la pila va usata anche per salvare le variabili che altrimenti verrebbero sovrascritte con la chiamata ricorsiva:

```
GLOBAL PILA[1000] INTEGER
GLOBAL SP          INTEGER
SP := -1

GLOBAL X           INTEGER
GLOBAL W           INTEGER

FATTORIALE ( )
  X := PILA[SP]

  IF X == 1
    THEN
      SP++
      PILA[SP] := 1
      # Accumula il risultato da
      # restituire, pari a uno.
    ELSE
      SP++
      PILA[SP] := X
      # Salva il valore di X nella
      # pila.

      SP++
      PILA[SP] := X - 1
      # Accumula il parametro di
      # chiamata della funzione
      # con un valore pari a X - 1.

      FATTORIALE ( )

      W := PILA[SP]
      SP--
      # Recupera il risultato
      # della chiamata ricorsiva
      # con un valore pari a
```

```

# X - 1.

SP--      # Scarica il parametro usato
          # per la chiamata.

X := PILA[SP]  # Recupera il valore di X
SP--      # prima della chiamata
          # ricorsiva.

SP++      # Accumula il risultato del
PILA[SP] := X * W # fattoriale da restituire.
END IF

END FATTORIALE
...
...
```

Come si vede nel nuovo esempio, prima della chiamata ricorsiva viene salvata nella pila solo la variabile  $X$ , perché il valore di  $W$  non dipende dall'elaborazione e tale variabile riceve un valore utile solo dopo la chiamata in questione. Naturalmente, si comprende che in questo caso particolare, non sarebbe stato nemmeno necessario salvare la variabile  $X$ , in quanto il suo valore corretto, dopo la chiamata ricorsiva, lo si può determinare semplicemente reincrementandolo di una unità. Ma qui si è preferito mostrare un esempio molto semplice, risolvendolo in modo generalizzato, anche se ciò non sarebbe necessario.

### 63.1.3 Funzioni attraverso le istruzioni di salto



Con l'uso di linguaggi di programmazione ragionevolmente evoluti, i salti (*go to*), condizionati o meno, vanno evitati, dal momento che esistono delle strutture per il controllo del flusso e si può disporre di chiamate di funzioni o procedure. Tuttavia, quando si deve scrivere con un linguaggio molto vicino alla realtà fisica dell'elaboratore, non si dispone più di questi ausili, o comunque occorre fare i conti con un indice riferito alle istruzioni da eseguire.

In pratica, il programma viene a trovarsi disposto in un vettore, dove un indice serve a sapere qual è l'istruzione successiva da eseguire: *instruction pointer*. Nel momento in cui si esegue un'istruzione normale, l'indice viene incrementato automaticamente per posizionarsi all'inizio dell'istruzione successiva, mentre in presenza di un'istruzione di salto, l'esecuzione di tale istruzione sposta l'indice nella nuova destinazione.

In queste condizioni, per ottenere ciò che di solito si realizza con delle funzioni ricorsive, occorre gestire l'indice delle istruzioni direttamente. Per la precisione, prima di saltare all'inizio di una funzione, oltre che accumulare i parametri della chiamata nella pila già descritta, occorre accumulare l'indice delle istruzioni, in modo tale da poter riprendere dall'istruzione successiva alla chiamata dopo l'esecuzione di ciò che rappresenta tale funzione.

```
GLOBAL IP          INTEGER          # «Instruction pointer»
                                     # (sola lettura).
GLOBAL RETURN      INTEGER          # Destinazione per il
                                     # ritorno.
...
GLOBAL PILA[1000]  INTEGER
```

```
GLOBAL SP          INTEGER
SP := -1
...
GLOBAL X          INTEGER
GLOBAL W          INTEGER
...
FATTORIALE ()
    X := PILA[SP - 1] # Recupera il primo
                      # parametro. Si ricorda che
                      # "PILA[SP]" contiene
                      # l'indirizzo di ritorno.

    IF X == 1
        THEN
            RETURN := PILA[SP] # Recupera l'indirizzo di
                                # ritorno.

            SP++ # Accumula il risultato da
                # restituire, pari a uno.
            PILA[SP] := 1 # Torna all'istruzione
                # successiva alla chiamata.

            GO_TO RETURN

        ELSE
            SP++ # Salva il valore di X nella
                # pila.
            PILA[SP] := X

            SP++ # Accumula il parametro di
                # chiamata della funzione
            PILA[SP] := X - 1 # con un valore pari a X-1.

            SP++ # Accumula l'indirizzo
                # dell'istruzione successiva
            PILA[SP] := IP + 1 # alla prossima.

            GO_TO FATTORIALE () # Salta all'inizio della
                                # funzione.
```

```
        SP--          # Scarica l'indirizzo di
                        # ritorno della funzione
                        # appena conclusa.

        W := PILA[SP]  # Recupera il risultato
        SP--          # della chiamata ricorsiva
                        # con un valore pari a X-1.

        SP--          # Scarica il parametro usato
                        # per la chiamata.

        X := PILA[SP]  # Recupera il valore di X
        SP--          # prima della chiamata
                        # ricorsiva.

        RETURN := PILA[SP] # Recupera l'indirizzo di
                        # ritorno.

        SP++          # Accumula il risultato del
        PILA[SP] := X * W # fattoriale da restituire.
        GO_TO RETURN   # Torna all'istruzione
                        # successiva alla chiamata.

    END IF

END FATTORIALE

...

...
```

Nell'esempio mostrato si considera che la variabile **IP** sia accessibile in sola lettura e che contenga l'indice dell'istruzione successiva o di quella richiesta da un'istruzione di salto. Prima del salto all'inizio di una funzione, si accumula il valore di **IP** nella pila, ma incrementandolo di ciò che serve a raggiungere l'istruzione successiva al salto

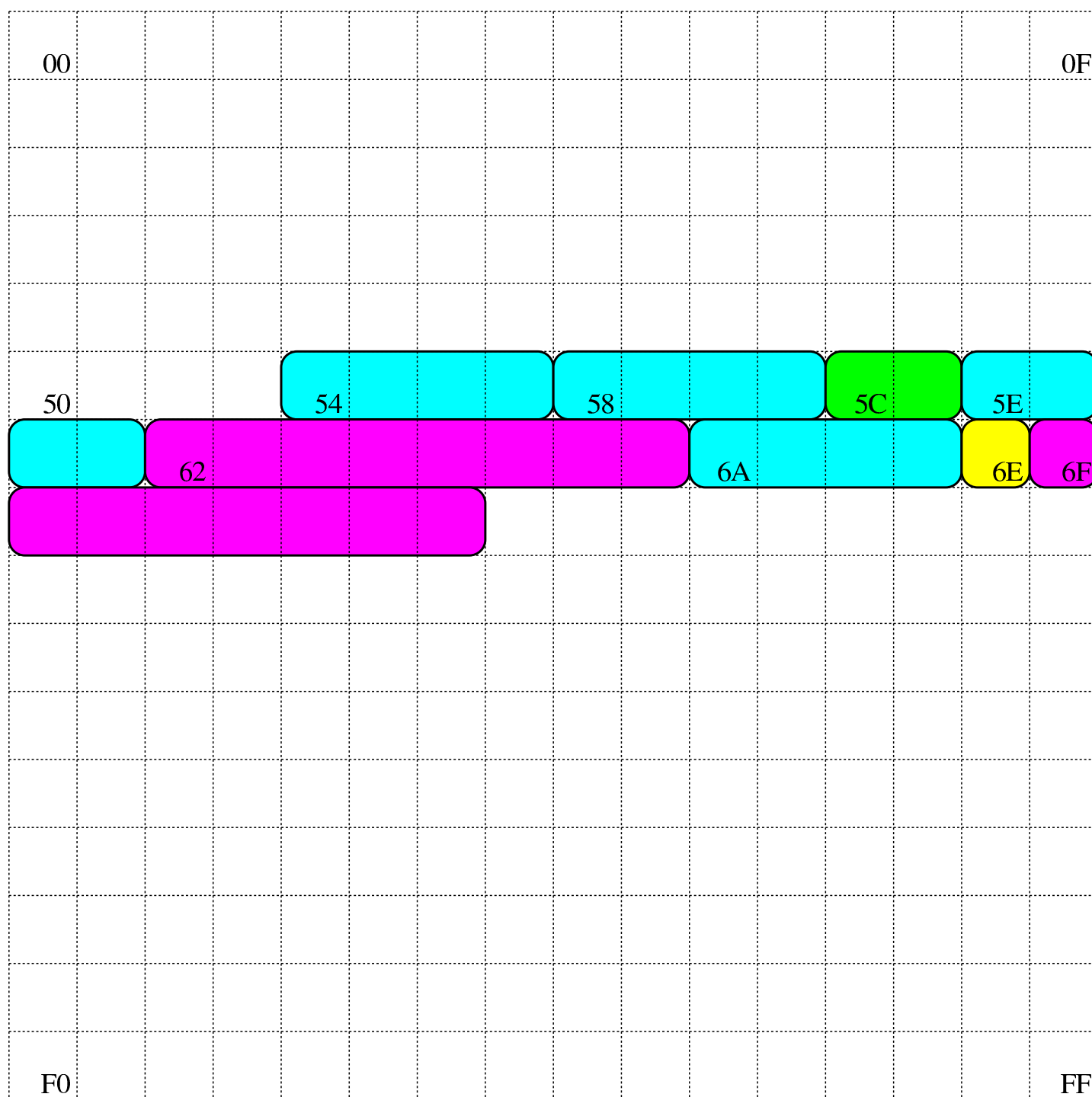
stesso (si suppone che l'incremento di una unità dia il risultato voluto). Nel momento appropriato, il valore dell'indice viene prelevato dalla pila e inserito in una variabile apposita, da usare per saltare alla posizione di ritorno.

#### 63.1.4 Variabili e array



Con un linguaggio di programmazione molto vicino alla realtà fisica dell'elaboratore, la memoria centrale viene vista come un vettore di celle uniformi, corrispondenti normalmente a un byte. All'interno di tale vettore si distendono tutti i dati gestiti, compresa la pila descritta nella sezione [63.1.1](#). In questo modo, le variabili in memoria si raggiungono attraverso un indirizzo che individua il primo byte che le compone ed è il programma che deve sapere di quanti byte sono composte complessivamente.

Figura 63.7. Esempio di mappa di una memoria di soli 256 byte, dove sono evidenziate alcune variabili. Gli indirizzi dei byte della memoria vanno da  $00_{16}$  a  $FF_{16}$ .

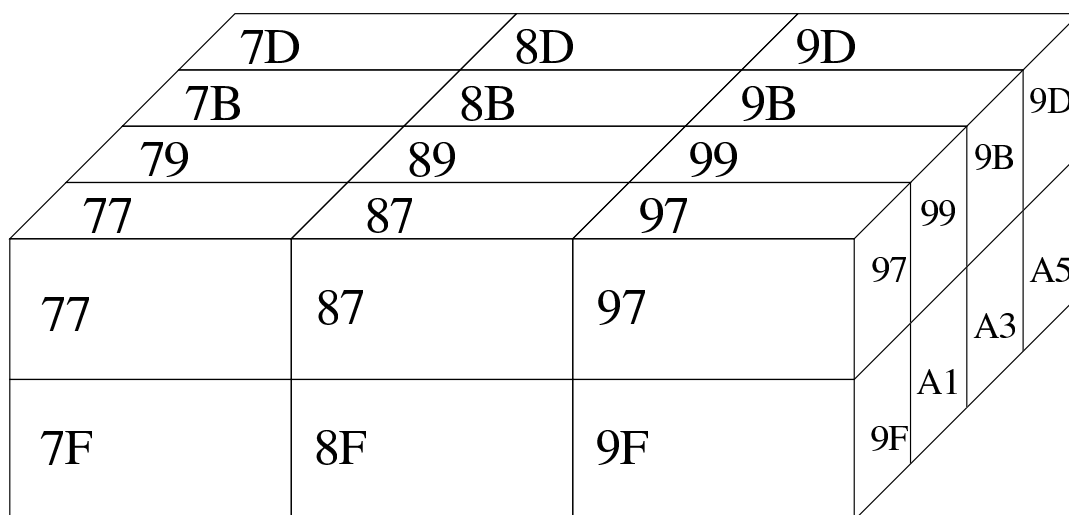


Nel disegno in cui si ipotizza una memoria complessiva di 256 byte, sono state evidenziate alcune aree di memoria:

Indirizzo	Dimensione	Indirizzo	Dimensione
54 <sub>16</sub>	4 byte	58 <sub>16</sub>	4 byte
5C <sub>16</sub>	2 byte	5E <sub>16</sub>	4 byte
62 <sub>16</sub>	8 byte	6A <sub>16</sub>	4 byte
6E <sub>16</sub>	1 byte	6F <sub>16</sub>	8 byte

Con una gestione di questo tipo della memoria, la rappresentazione degli array richiede un po' di impegno da parte del programmatore. Nella figura successiva si rappresenta una matrice a tre dimensioni; per ora si ignorino i codici numerici associati alle celle visibili.

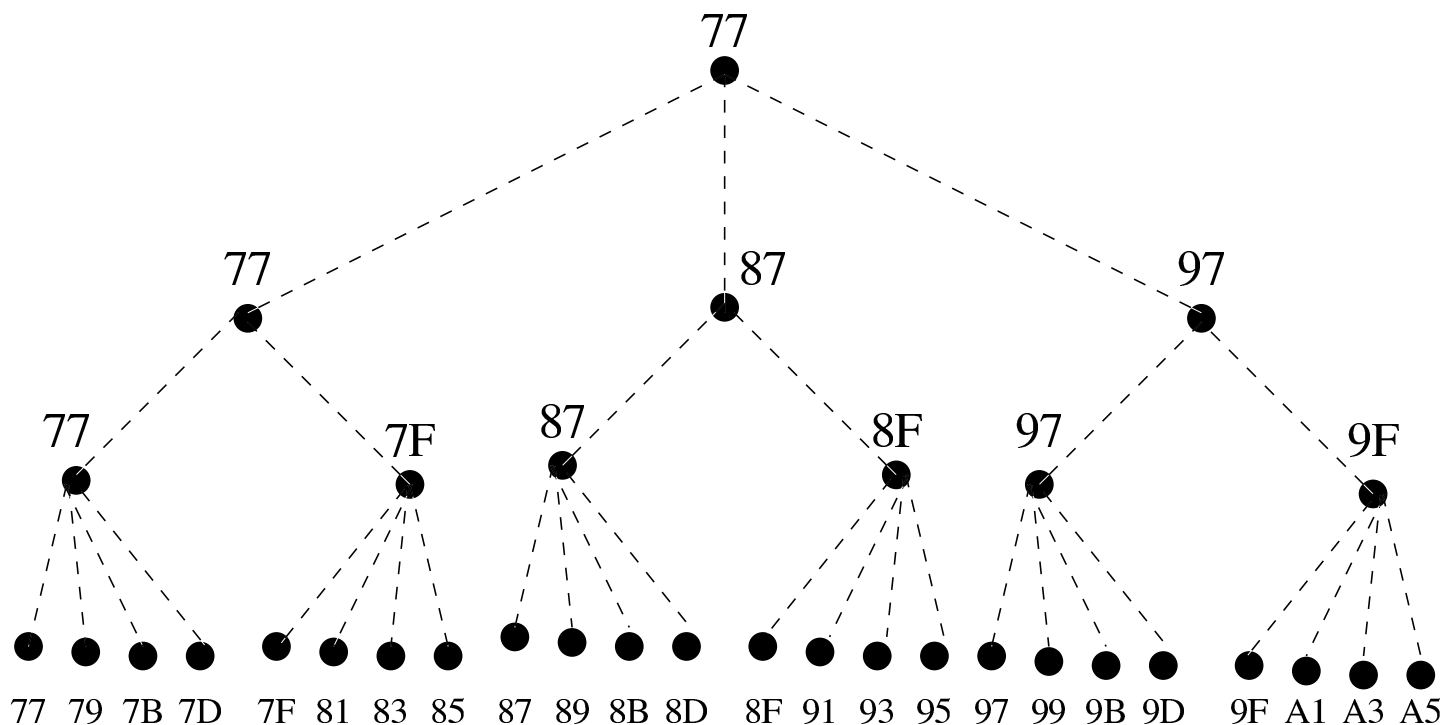
Figura 63.9. La matrice a tre dimensioni che si vuole rappresentare, secondo un modello spaziale. I numeri che appaiono servono a trovare successivamente l'abbinamento con le celle di memoria utilizzate.



Dal momento che la rappresentazione tridimensionale rischia di creare confusione, quando si devono rappresentare matrici che hanno più di due dimensioni, è più conveniente pensare a strutture ad albero. Nella figura successiva viene tradotta in forma di albero la matrice rappresentata precedentemente.

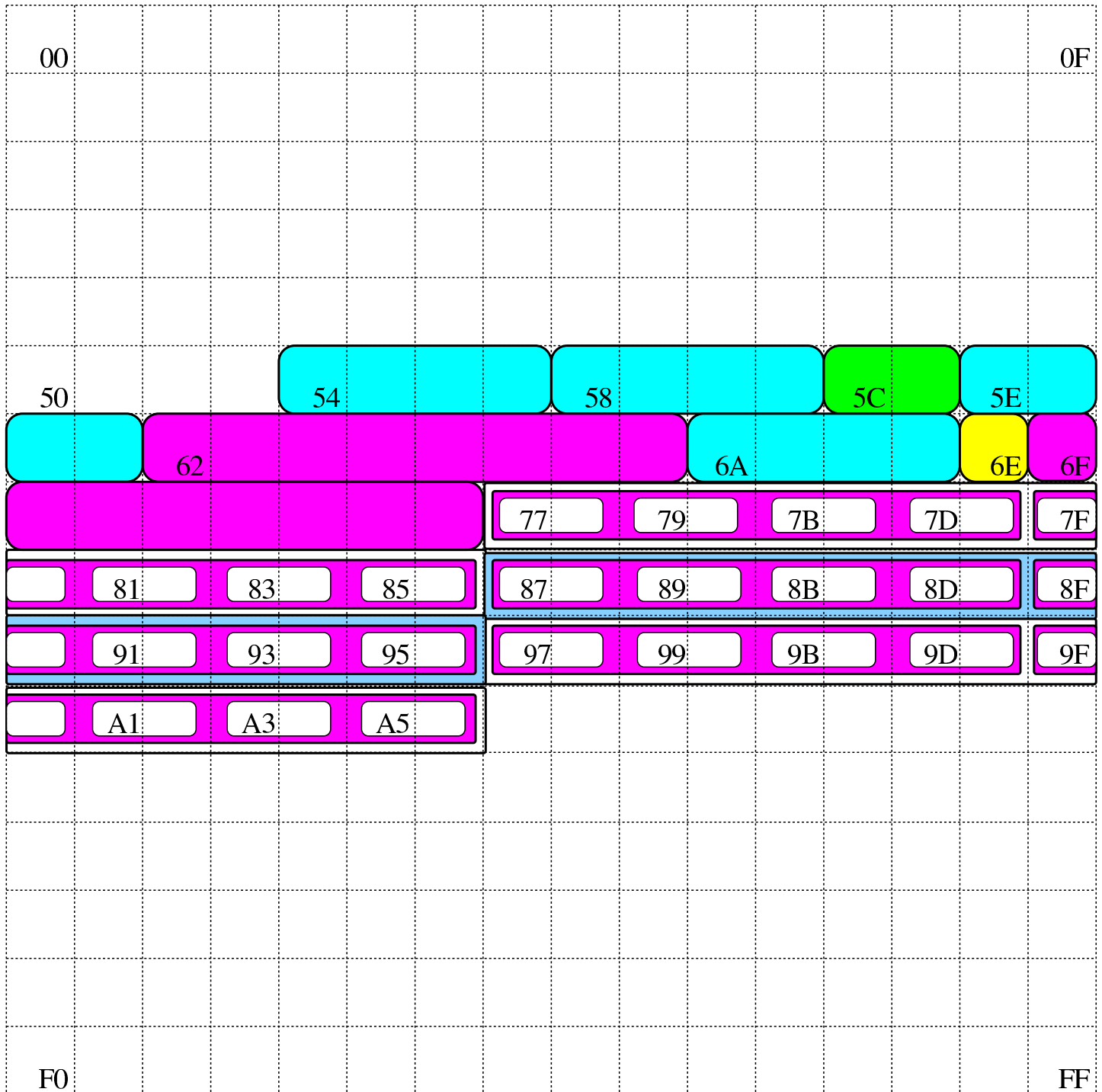


Figura 63.10. La matrice a tre dimensioni che si vuole rappresentare, tradotta in uno schema gerarchico (ad albero).



Si suppone di rappresentare la matrice in questione nella memoria dell'elaboratore, dove ogni elemento terminale contiene due byte. Supponendo di allocare l'array a partire dall'indirizzo  $77_{16}$  nella mappa di memoria già descritta, si potrebbe ottenere quanto si vede nella figura successiva. A questo punto, si può vedere la corrispondenza tra gli indirizzi dei vari componenti dell'array e le figure già mostrate.

Figura 63.11. Esempio di mappa di memoria in cui si distende un array che rappresenta una matrice a tre dimensioni con tre elementi contenenti ognuno due elementi che a loro volta contengono quattro elementi da due byte.



Si pone quindi il problema di scandire gli elementi dell'array. Con-

siderando che array ha dimensioni «3,2,4» e definendo che gli indici partano da zero, l'elemento [0,0,0] corrisponde alla coppia di byte che inizia all'indirizzo  $77_{16}$ , mentre l'elemento [2,1,3] corrisponde all'indirizzo  $A5_{16}$ . Per calcolare l'indirizzo corrispondente a un certo elemento occorre usare la formula seguente, dove: le variabili  $I$ ,  $J$ ,  $K$  rappresentano la dimensioni dei componenti; le variabili  $i$ ,  $j$ ,  $k$  rappresentano l'indice dell'elemento cercato; la variabile  $A$  rappresenta l'indirizzo iniziale dell'array; la variabile  $s$  rappresenta la dimensione in byte degli elementi terminali dell'array.

$$A + (i \cdot J \cdot K \cdot s + j \cdot K \cdot s + k \cdot s)$$

$$A + s \cdot (i \cdot J \cdot K + j \cdot K + k)$$

Si vuole calcolare la posizione dell'elemento 2,0,1. Per facilitare i conti a livello umano, si converte l'indirizzo iniziale dell'array in base dieci:  $77_{16} = 119_{10}$ :

$$119 + 2 \cdot (2 \cdot 2 \cdot 4 + 0 \cdot 4 + 1) = 153$$

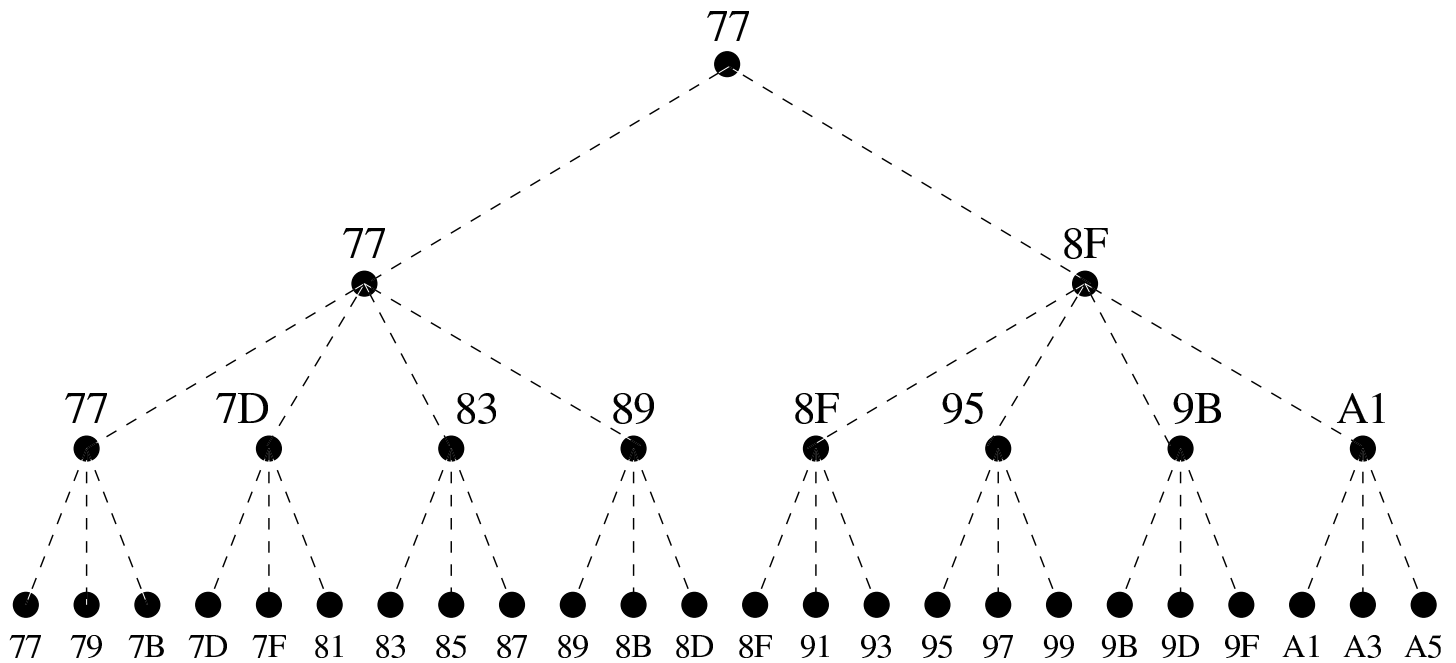
Il valore  $153_{10}$  si traduce in base sedici in  $99_{16}$ , che corrisponde effettivamente all'elemento cercato: terzo elemento principale, all'interno del quale si cerca il primo elemento, all'interno del quale si cerca il secondo elemento finale.

### 63.1.5 Gestione alternativa degli indici

Quando si vuole disporre un array nella memoria, se quello che conta è solo raggiungere gli elementi terminali che lo compongono, non fa molta differenza se la gerarchia con cui si organizza è diversa.

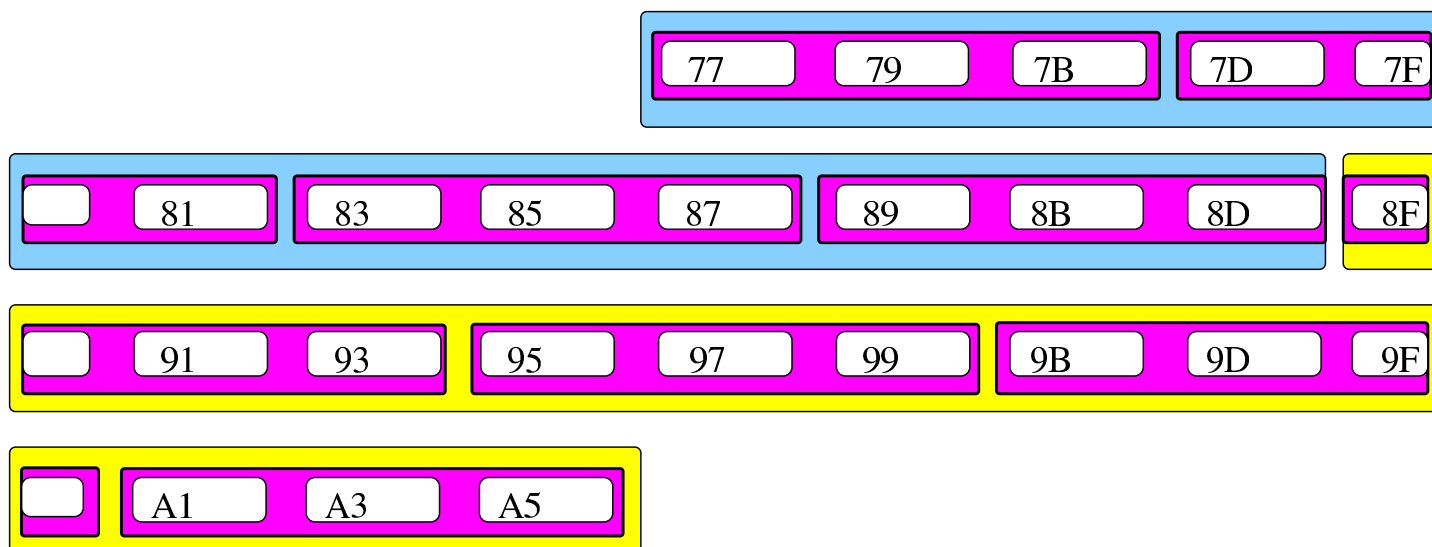
Per esempio, l'array che prima era strutturato in elementi di dimensione 3,2,4, potrebbe benissimo essere definito secondo la suddivisione 4,3,2, gestendo di conseguenza gli indici. Lo si può vedere nella figura successiva che riproduce la nuova gerarchia in forma di albero.

Figura 63.15. La stessa matrice, ma organizzata secondo una gerarchia differente.



Nella figura successiva si riprende l'esempio di mappa della memoria, dove l'array già apparso nella sezione precedente è disposto secondo la nuova suddivisione.

Figura 63.16. La nuova mappa della memoria.



Nella tabella successiva si mettono a confronto le coordinate calcolate per raggiungere gli elementi dell'array strutturato secondo le due gerarchie mostrate (quella della sezione precedente e quella attuale). Si può vedere che le celle di memoria vengono raggiunte nello stesso modo (nella tabella gli indirizzi sono annotati in base dieci). Viene anche mostrato cosa può accadere se si usano gli indici di accesso in modo non coincidente alla gerarchia prescelta.

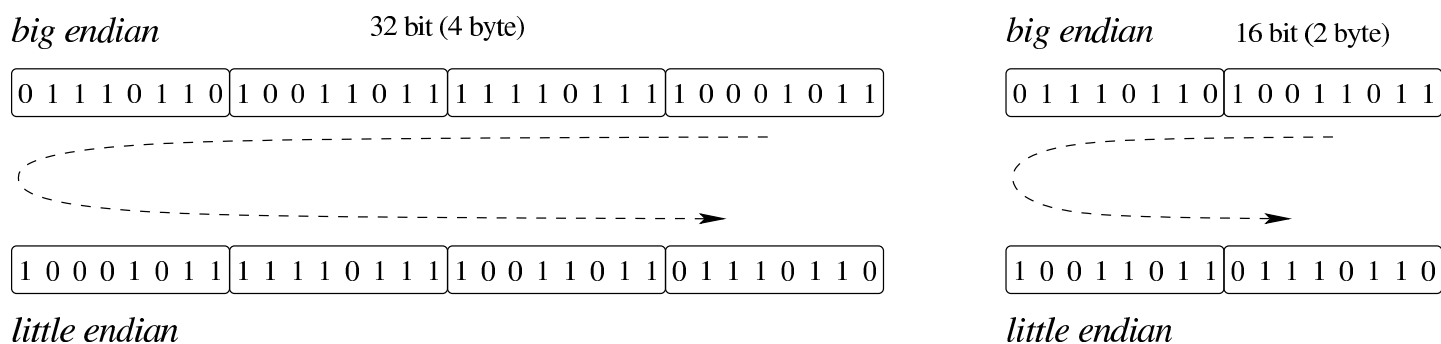
Tabella 63.17. Confronto dell'indirizzamento della memoria utilizzando due modi diversi di organizzare gli elementi dell'array, con un esempio di cosa accade quando gli indici non combaciano con la struttura scelta.

Array secondo la sua struttura prevista			Array con una suddivisione alternativa			Array con indici di accesso scambiati (ma il risultato è errato)											
I	J	K	indirizzo (in base dieci)			I	J	K	indirizzo (in base dieci)			indirizzo errato (in base dieci)					
3	2	4				2	4	3				3	2	4			
<i>i</i>	<i>j</i>	<i>k</i>				<i>i</i>	<i>j</i>	<i>k</i>				<i>j</i>	<i>k</i>	<i>i</i>			
0	0	0	119			0	0	0	119			0	0	0	119		
0	0	1	121			0	0	1	121			0	0	1	121		
0	0	2	123			0	0	2	123			0	0	2	123		
0	0	3	125			0	1	0	125			0	1	0	127		
0	1	0	127			0	1	1	127			0	1	1	129		
0	1	1	129			0	1	2	129			0	1	2	131		
0	1	2	131			0	2	0	131			0	2	0	135		
0	1	3	133			0	2	1	133			0	2	1	137		
1	0	0	135			0	2	2	135			0	2	2	139		
1	0	1	137			0	3	0	137			0	3	0	143		
1	0	2	139			0	3	1	139			0	3	1	145		
1	0	3	141			0	3	2	141			0	3	2	147		
1	1	0	143			1	0	0	143			1	0	0	135		
1	1	1	145			1	0	1	145			1	0	1	137		
1	1	2	147			1	0	2	147			1	0	2	139		
1	1	3	149			1	1	0	149			1	1	0	143		
2	0	0	151			1	1	1	151			1	1	1	145		
2	0	1	153			1	1	2	153			1	1	2	147		
2	0	2	155			1	2	0	155			1	2	0	151		
2	0	3	157			1	2	1	157			1	2	1	153		
2	1	0	159			1	2	2	159			1	2	2	155		
2	1	1	161			1	3	0	161			1	3	0	159		
2	1	2	163			1	3	1	163			1	3	1	161		
2	1	3	165			1	3	2	165			1	3	2	163		

## 63.1.6 Ordine dei byte

Come già descritto, normalmente l'accesso alla memoria avviene conoscendo l'indirizzo iniziale dell'informazione cercata, sapendo poi per quanti byte questa si estende. Il microprocessore, a seconda delle proprie caratteristiche e delle istruzioni ricevute, legge e scrive la memoria a gruppetti di byte, più o meno numerosi. Ma l'ordine dei byte che il microprocessore utilizza potrebbe essere diverso da quello che si immagina di solito.

Figura 63.18. Confronto tra *big endian* e *little endian*.



A questo proposito, per quanto riguarda la rappresentazione dei dati nella memoria, si distingue tra *big endian*, corrispondente a una rappresentazione «normale», dove il primo byte è quello più significativo (*big*), e *little endian*, dove la sequenza dei byte è invertita (ma i bit di ogni byte rimangono nella stessa sequenza standard) e il primo byte è quello meno significativo (*little*). La cosa importante da chiarire è che l'effetto dell'inversione nella sequenza porta a risultati differenti, a seconda della quantità di byte che compongono l'insieme letto o scritto simultaneamente dal microprocessore, come si vede nella figura.

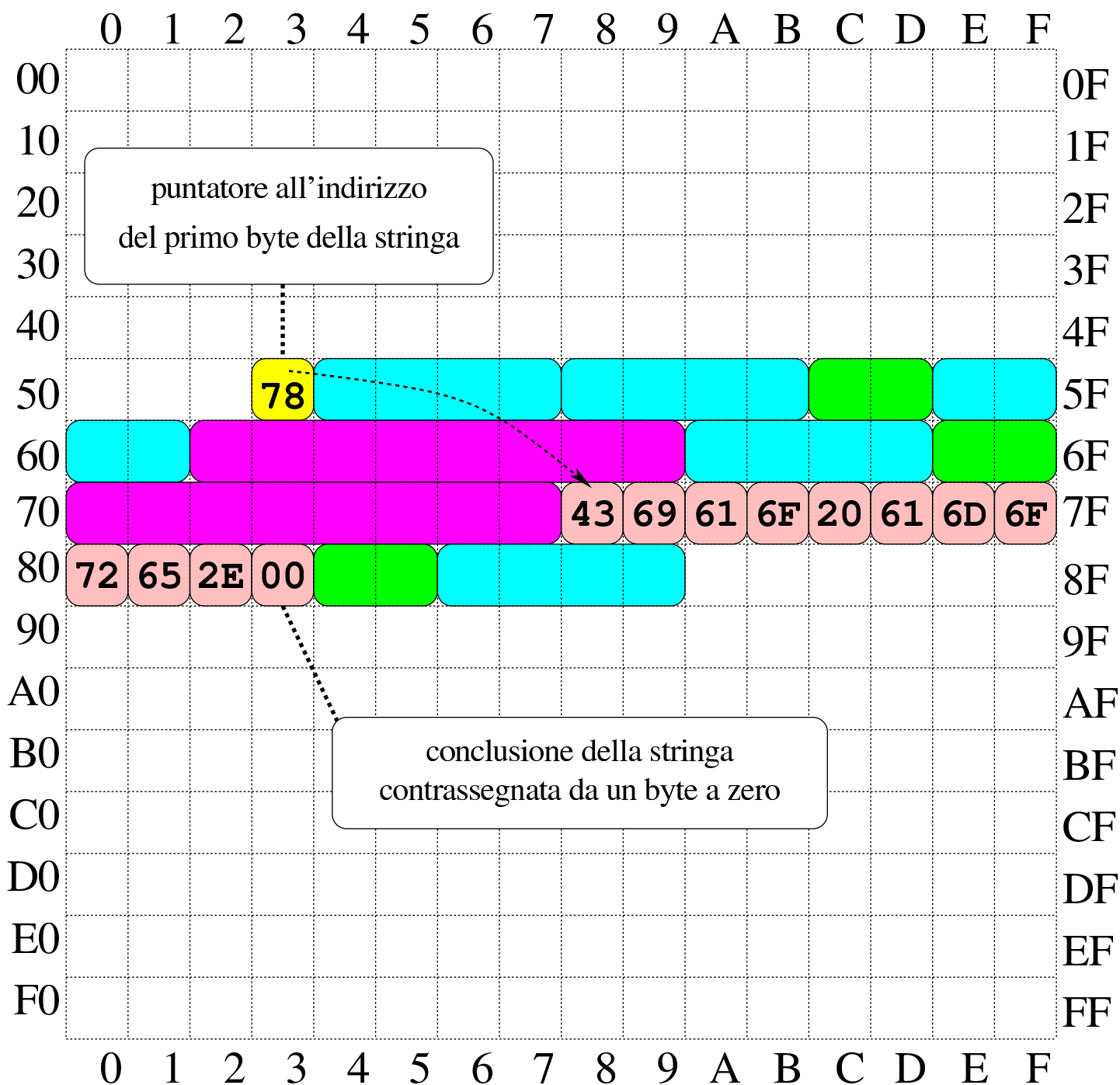
### 63.1.7 Stringhe, array e puntatori



Le stringhe sono rappresentate in memoria come array di caratteri, dove il carattere può impiegare un byte o dimensioni multiple (nel caso di UTF-8, un carattere viene rappresentato utilizzando da uno a quattro byte, a seconda del punto di codifica raggiunto). Il riferimento a una stringa viene fatto come avviene per gli array in generale, attraverso un puntatore all'indirizzo della prima cella di memoria che lo contiene; tuttavia, per non dovere annotare la dimensione di tale array, di norma si conviene che la fine della stringa sia delimitata da un byte a zero, come si vede nell'esempio della figura.



Figura 63.19. Stringa conclusa da un byte a zero (*zero terminated string*), a cui viene fatto riferimento per mezzo di una variabile che contiene il suo indirizzo iniziale. La stringa contiene il testo **‘Ciao amore.’**, secondo la codifica ASCII.



Nella figura si vede che la variabile scalare collocata all'indirizzo  $53_{16}$  contiene un valore da intendere come indirizzo, con il quale si

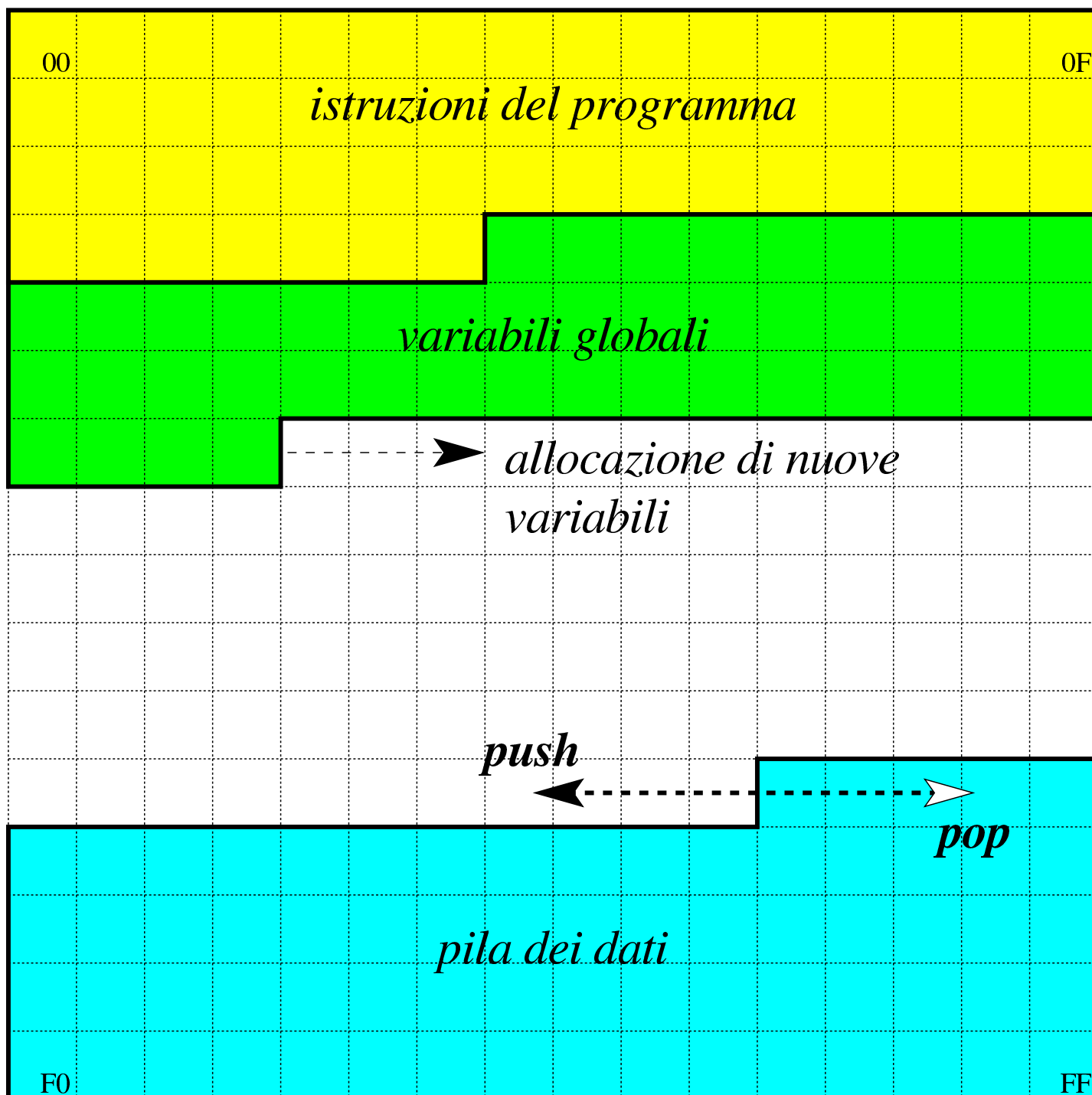
fa riferimento al primo byte dell'array che rappresenta la stringa (in posizione  $78_{16}$ ). La variabile collocata in  $53_{16}$  assume così il ruolo di *variabile puntatore* e, secondo il modello ridotto di memoria della figura, è sufficiente un solo byte per rappresentare un tale puntatore, dal momento che servono soltanto valori da  $00_{16}$  a  $FF_{16}$ .

### 63.1.8 Utilizzo della memoria



La memoria dell'elaboratore viene utilizzata sia per contenere i dati, sia per il codice del programma che li utilizza. Ogni programma ha un proprio spazio in memoria, che può essere reale o virtuale; all'interno di questo spazio, la disposizione delle varie componenti potrebbe essere differente. Nei sistemi che si rifanno al modello di Unix, nella parte più «bassa» della memoria risiede il codice che viene eseguito; subito dopo vengono le variabili globali del programma, mentre dalla parte più «alta» inizia la pila dei dati che cresce verso indirizzi inferiori. Si possono comunque immaginare combinazioni differenti di tale organizzazione, pur rispettando il vincolo di avere tre zone ben distinte per il loro contesto (codice, dati, pila); tuttavia, ci sono situazioni in cui i dati si trovano mescolati al codice, per qualche ragione.

Figura 63.20. Esempio di disposizione delle componenti di un programma in esecuzione in memoria, secondo il modello Unix.



## 63.2 Architettura, linguaggio, contesto virtuale, terminologia

«

Ciò che un microprocessore esegue sono istruzioni in linguaggio macchina, composte secondo la sintassi che il microprocessore stesso è in grado di interpretare. Il linguaggio macchina è fatto esclusivamente di numeri (da gestire in base due) e per questo, di norma, non viene utilizzato direttamente a livello umano.

Quando si deve intervenire al livello più basso possibile della programmazione, ci si avvale generalmente di un linguaggio «assemblatore» (*assembly*), ovvero un linguaggio che, pur rimanendo legato alle caratteristiche del microprocessore e in generale a quelle dell'architettura dell'elaboratore, esprime le istruzioni in una forma simbolica più comprensibile. Naturalmente, un programma scritto secondo un linguaggio assemblatore deve essere elaborato da un compilatore (*assembler*) per generare il linguaggio macchina effettivo.

Non esiste un'architettura standard, né un linguaggio macchina standard e di conseguenza non esiste nemmeno un linguaggio assemblatore standard. Un programma scritto con un linguaggio assemblatore adatto a un certo tipo di architettura non può funzionare in un'architettura differente. Pertanto, se si usa un tale linguaggio, lo si fa soprattutto per quelle cose che altrimenti non potrebbero essere risolte (come il codice necessario all'avvio del sistema operativo).

### 63.2.1 Memoria e registri

«

Le architetture per elaboratore più diffuse prevedono un microprocessore in grado di comunicare con una memoria centrale, organizzata come un vettore di celle, contenenti una quantità uniforme di

bit, accessibili attraverso un indice che ne rappresenta l'indirizzo. Oltre a questo, il microprocessore dispone internamente di alcuni *registri*, ovvero delle celle singole di memoria con compiti più o meno specializzati.

La dimensione dei registri condiziona la capacità del microprocessore di eseguire dei calcoli e la capacità di indirizzamento della memoria. La dimensione dei registri più comuni di un microprocessore corrisponde alla dimensione della *parola* (*word*).

Generalmente, la memoria centrale è organizzata in celle di byte (intesi come gruppi di otto bit), ma possono esistere architetture in cui tali celle corrispondono alla dimensione della parola del microprocessore, o anche altre dimensioni, ma in generale una cella della memoria deve essere contenibile in un registro.

Nella memoria centrale devono risiedere sia i dati da elaborare, sia le istruzioni in linguaggio macchina. Pertanto, un registro molto importante in un microprocessore è quello che tiene traccia, nella memoria centrale, dell'istruzione successiva da eseguire: *instruction pointer* o *program counter*.

Nel caso degli elaboratori x86-32, l'architettura più comune negli anni 1990 prevede parole da 32 bit e una memoria organizzata in byte; pertanto è possibile gestire lo spazio di 4 Gbyte ( $2^{32}$ ). Purtroppo, nella documentazione originale di questo tipo di architettura si usa il termine *word* per identificare una dimensione a 16 bit, come era nel primo microprocessore di quella serie (8088/8086), per motivi di compatibilità.

## 63.2.2 Indicatori o «flag»

«

Un *indicatore*, ovvero un *flag*, è un'informazione costituita da un solo valore binario (*Vero* o *Falso*) che serve a tenere traccia dell'esito delle operazioni svolte all'interno del microprocessore. In generale, gli indicatori sono raccolti assieme in un registro specializzato.

Gli indicatori più importanti in assoluto sono due: «riporto» o *carry* che serve a conoscere l'esito delle somme (e delle sottrazioni) di valori senza segno; «traboccamento» o *overflow* che serve a conoscere l'esito delle somme (e delle sottrazioni) di valori con segno. Bisogna osservare che, tra le varie architetture, non è detto che gli indicatori funzionino sempre nella stessa maniera.

Tabella 63.21. Indicatori comuni tra le varie architetture.

Indicatore ( <i>flag</i> )	Descrizione
<i>carry</i>	È l'indicatore del riporto che diventa utile per le operazioni con valori senza segno.
<i>borrow</i>	È l'indicatore della richiesta del prestito di una cifra nelle sottrazioni che diventa utile per le operazioni con valori senza segno. Di solito questo indicatore è costituito dallo stesso <i>carry</i> , il cui risultato va inteso in questo senso quando si eseguono delle sottrazioni.
<i>overflow</i>	È l'indicatore di traboccamento per le operazioni che riguardano valori con segno.
<i>zero</i>	Viene impostato dopo un'operazione che dà come risultato il valore zero.
<i>sign</i>	In linea di massima, riproduce il bit più significativo di un valore, dopo un'operazione. Se il valore è da intendersi con segno, l'indicatore serve a riprodurre il segno stesso.

Indicatore ( <i>flag</i> )	Descrizione
<i>parity</i>	In linea di massima, si attiva quando l'ultima operazione produce un risultato contenente una quantità pari di bit a uno (ma ciò non significa che il valore corrispondente sia pari).

### 63.2.3 «Opcode»

Nel linguaggio macchina, il codice numerico che descrive le istruzioni è definito *operation code* (codice operazione) e si abbrevia come *opcode* (o solo «op»). La lunghezza complessiva dell'istruzione può cambiare a seconda degli operandi che il codice di operazione prevede di avere.

### 63.2.4 Accesso alla memoria

Le istruzioni fornite al microprocessore (in linguaggio macchina o secondo la simbologia del linguaggio assembler) contengono dati o riferimenti a dei dati. A questo proposito, ogni architettura definisce le proprie tipologie e, di conseguenza, non esiste una denominazione uniforme.

Spesso si distingue tra le modalità di indirizzamento riferite al codice del programma, rispetto a quelle relative ai dati veri e propri. Le forme di indirizzamento più semplici riferite al codice possono essere assolute, quando si specifica un indirizzo preciso, oppure relative, quando si specifica uno spostamento relativo dalla posizione corrente. Si usa l'indirizzamento al codice con i salti e con le chiamate di subroutine. L'indirizzamento ai dati potrebbe comprendere le forme dell'elenco seguente:

- valori numerici costanti, incorporati nell'istruzione, che spesso sono chiamati «immediati»;
- registri da intendere come tali;
- aree di memoria indicate direttamente con un indirizzo;
- aree di memoria indicate da un indirizzo composto da un valore di riferimento e l'aggiunta di un indice o di uno scostamento;
- aree di memoria indicate attraverso un registro che ne contiene l'indirizzo;
- aree di memoria indicate con un indirizzo composto dalla somma tra un valore costante, il contenuto di un registro ed eventualmente uno scostamento.

In generale, il termine «valore immediato» si riferisce a un'informazione numerica costante, incorporata nell'istruzione in linguaggio macchina. Ogni volta che si indica un riferimento fisso alla memoria (di solito lo si fa attraverso un «simbolo», rappresentato da etichetta, che il compilatore traduce in un indirizzo, in uno scostamento o in un dislocamento), sia per ciò che riguarda il codice, sia per i dati veri e propri, si sta utilizzando un valore immediato, anche se è compito del compilatore tradurlo effettivamente in un numero. Tale valore è «immediato» in quanto il microprocessore non deve eseguire alcun calcolo per interpretarlo.

Come si può intuire, le forme più complesse di rappresentazione delle variabili in memoria consentono una scansione utile per rappresentare gli array di dati.



È però importante distinguere i contesti: un conto è l'istruzione macchina, un altro è l'istruzione scritta nel linguaggio assembleatore. Generalmente gli indirizzi della memoria non vengono scritti materialmente in forma numerica, lasciando che sia il compilatore a tradurli nella realtà concreta. Ma questo comporta spesso anche la scelta di un tipo di istruzione macchina rispetto a un altro, in base al contesto, cosa che rimane sempre a carico del compilatore. D'altro canto, certi tipi di indirizzamento complesso, vengono elaborati e semplificati dal compilatore stesso.

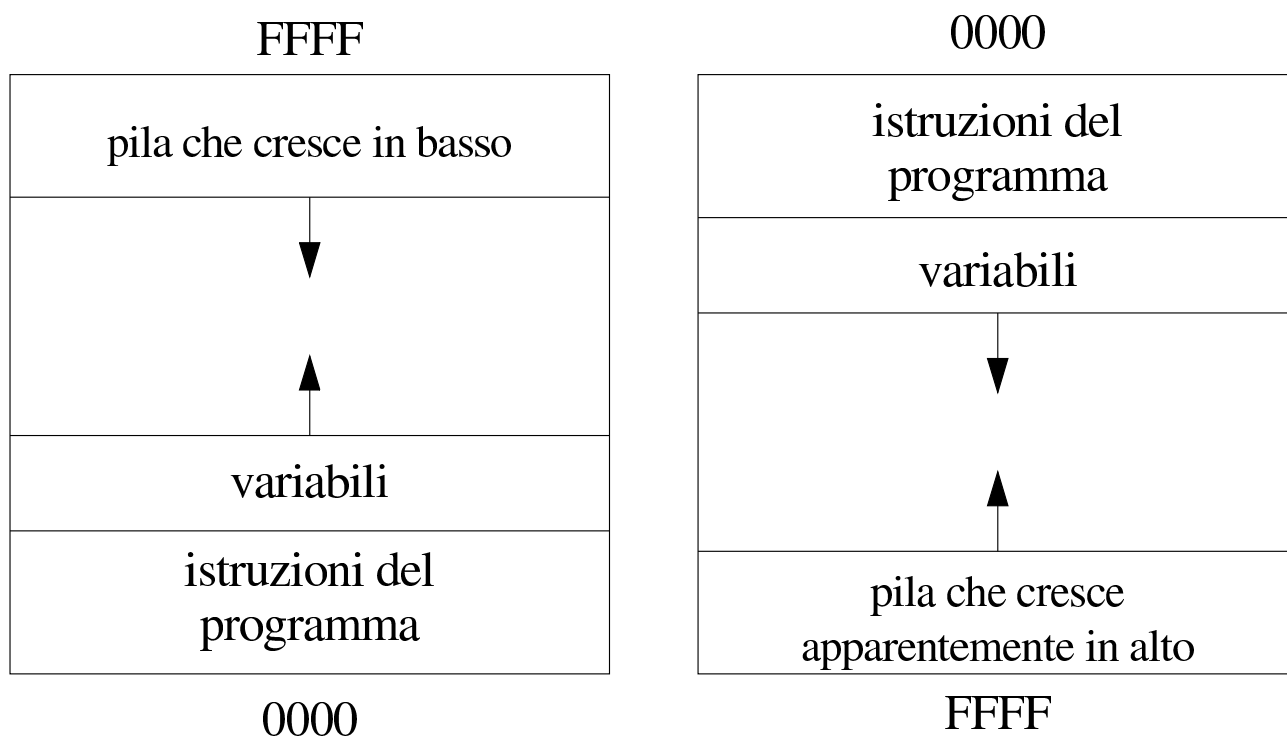
È importante sottolineare che le istruzioni in linguaggio macchina (*opcode*) possono essere diverse, anche se riferite a uno stesso tipo di operazione, quando cambia l'entità di un dislocamento o il tipo di indirizzamento; pertanto, quando si legge il manuale di riferimento per un certo microprocessore, si trova l'elenco delle istruzioni e la descrizione degli operandi previsti, ma non è detto che nel linguaggio assembleatore si debba usare esattamente la stessa modalità.

### 63.2.5 Modello della memoria nei sistemi Unix<sup>1</sup>

Nei sistemi Unix, inclusi i sistemi liberi che si rifanno a quel modello tradizionale, i processi elaborativi vedono la memoria come un solo vettore contenente: le istruzioni da eseguire, lo spazio previsto per i dati e una pila, utilizzata sostanzialmente nel modo descritto nelle sezioni precedenti. La pila inizia però da una posizione elevata di questo vettore e si espande in posizioni inferiori. Pertanto, l'indice della pila viene decrementato quando la si carica di un nuovo elemento (*push*) e viene incrementato quando invece la si scarica (*pop*). Ciò è come dire che la pila è rovesciata e si estende «verso il basso».



Figura 63.22. Semplificazione del modo in cui un processo elaborativo Unix vede la memoria. La dimensione della memoria virtuale a disposizione di un processo elaborativo dipende normalmente dall'architettura dell'elaboratore; il valore indicato nel disegno serve solo come semplificazione. A sinistra si vedono gli indirizzi di memoria partire dal basso ed estendersi in alto; a destra si vede l'opposto. Nella seconda forma visuale la pila cresce «dal basso», ma rimane il fatto che il modo di gestire l'indice sia lo stesso.



La rappresentazione che si vede nella parte sinistra della figura è quella tradizionale, ma se si ragiona «in senso di lettura», potrebbe essere più logico rappresentare gli indirizzi più bassi in alto, progredendo verso il basso. In tal caso, la pila si estende come si è abituati normalmente a pensarla, ma resta il fatto che l'indice di gestione della pila deve essere decrementato per aggiungere degli elementi sulla stessa.

### 63.2.6 Sintassi «AT&T» e «Intel»

Quando si utilizza l'architettura x86 si trovano generalmente compilatori per linguaggi assembler di due tipi: uno conforme allo stile usato nei sistemi Unix del PDP-11; l'altro conforme alla simbologia usata dalla documentazione della casa produttrice dei primi microprocessori di questo tipo. Dal momento che Unix è nato nei laboratori Bell AT&T, la prima notazione è nota come «sintassi AT&T»; per converso, l'altra è la «sintassi Intel».

Generalmente, negli ambienti legati ai sistemi Unix e simili, GNU/Linux incluso, si preferisce usare compilatori con sintassi AT&T.

### 63.2.7 Macchina virtuale

Quando si scrive un programma in linguaggio assembler, occorre tenere in considerazione il contesto di funzionamento. Di norma questo contesto è dato dal sistema operativo, attraverso il quale il programma viene caricato in memoria e poi eseguito.

In effetti, l'uso diretto di un linguaggio assembler è appropriato quando si opera al di fuori del sistema operativo, per esempio, proprio per il codice di avvio di un sistema. Tuttavia, quando si inizia lo studio di un tale linguaggio, i programmi che si realizzano sono fatti per un sistema già funzionante che quindi si sottomettono al controllo di questo.

L'ambiente in cui si trova a funzionare il programma avviato attraverso il sistema operativo è una macchina virtuale che può avere caratteristiche differenti rispetto alla «macchina reale», soprattutto per

ciò che riguarda l'indirizzamento della memoria e per le funzioni a cui è possibile accedere.

### 63.2.8 Compilazione e collegamento

«

Nei sistemi operativi che si rifanno al modello di Unix, la compilazione di un programma scritto secondo un linguaggio assembler segue un procedimento comune. Una prima fase interpreta un file sorgente e produce un file «oggetto», ripetendo eventualmente il procedimento per altri file che servono a produrre lo stesso programma. I file oggetto ottenuti in questa fase sono file binari che non sono ancora pronti per essere eseguiti, in quanto alcune informazioni sono rimaste da definire. Nella seconda fase (nota come *link*) i file oggetto che servono a comporre un certo programma vengono collegati assieme, generando il file eseguibile vero e proprio.

In pratica, un programma eseguibile viene ottenuto da almeno un file oggetto, ma spesso i file oggetto che servono a produrre un programma sono più di uno. Infatti, nei file che costituiscono i sorgenti possono esserci dei riferimenti a zone di memoria e a funzioni descritte in altri file, pertanto è compito della fase di «collegamento» il comporre assieme i file oggetto in modo che questi riferimenti reciproci vengano consolidati.

Secondo la tradizione, in modo predefinito, il compilatore di un linguaggio assembler genera il file oggetto con il nome 'a.out', ma anche il *linker*, ovvero il programma che collega assieme i file oggetto, creerebbe un file eseguibile con lo stesso nome (naturalmente, di solito si dichiara esplicitamente il nome del file da generare). È bene sapere che il nome «a.out» deriva dalle primissime edizioni di Unix e significa *Assembler output*.

Quando si usano linguaggi di programmazione più evoluti rispetto al codice che si rifà direttamente alle caratteristiche del microprocessore, spesso il procedimento di compilazione passa per la produzione di un sorgente in linguaggio assembler, che poi viene compilato secondo la modalità consueta. In ogni caso, se la compilazione prevede la produzione intermedia di file oggetto, teoricamente, questi possono essere collegati assieme ad altri file oggetto prodotti da altri linguaggi. Perché ciò sia possibile effettivamente, occorre comunque che siano compatibili nel modo di condividere la memoria e di eseguire le chiamate delle funzioni, al livello del linguaggio macchina.

Rimane da tenere presente che i file oggetto e i file eseguibili hanno un formato definito da un certo standard. Di questi standard ne esistono molti, anche se nei sistemi Unix e simili si è affermato prevalentemente il formato ELF (*Executable and linkable format*). I primi formati usati nei sistemi Unix sono noti con come «a.out», confondendosi con il nome del file generato in modo predefinito dal compilatore. Si osservi che i compilatori attuali, in mancanza di altre indicazioni, producono file con il nome ‘a.out’, indipendentemente dal formato che questi hanno, formato che può benissimo essere ELF o altro.

### 63.3 Rappresentazione di valori numerici

La memoria di un elaboratore consente di annotare esclusivamente delle cifre binarie e in uno spazio di dimensione prestabilita e fissa. Nelle sezioni successive si descrivono alcune forme di rappresentazione dei valori numerici, nell’ambito di queste limitazioni.



### 63.3.1 Codifica delle singole cifre

«

Un valore numerico potrebbe essere espresso come una stringa di caratteri, corrispondenti alle cifre numeriche che lo rappresentano secondo la notazione in base dieci. Naturalmente, una rappresentazione del genere implica uno spreco di spazio nel sistema di memorizzazione e richiede una trasformazione prima di poter procedere all'esecuzione di calcoli numerici.

Esistono diverse forme di rappresentazioni numeriche, intese come sequenze di cifre in base dieci, che utilizzano quattro bit per ogni cifra. Il sistema più comune è noto con il nome BCD: *Binary coded decimal*.

I sistemi di rappresentazione numerica che utilizzano quattro bit per ogni cifra di un valore in base dieci, si utilizzano per esempio nel linguaggio COBOL, per le variabili scalari di tipo *computational*.

Tabella 63.23. Alcune codifiche per la rappresentazione di cifre numeriche (in base dieci) a gruppi di quattro bit.

Cifra decimale	Codice BCD (8421)	Codice «eccesso 3»	Codice 2421	Codice 5211	Codice 631-1	Codice 732-1
0	0000 <sub>2</sub>	0011 <sub>2</sub>	0000 <sub>2</sub>	0000 <sub>2</sub>	0000 <sub>2</sub> 0011 <sub>2</sub>	0000 <sub>2</sub>
1	0001 <sub>2</sub>	0100 <sub>2</sub>	0001 <sub>2</sub>	0001 <sub>2</sub> 0010 <sub>2</sub>	0010 <sub>2</sub>	0011 <sub>2</sub>
2	0010 <sub>2</sub>	0101 <sub>2</sub>	0010 <sub>2</sub> 1000 <sub>2</sub>	0100 <sub>2</sub> 0011 <sub>2</sub>	0101 <sub>2</sub>	0010 <sub>2</sub>
3	0011 <sub>2</sub>	0110 <sub>2</sub>	0011 <sub>2</sub> 1001 <sub>2</sub>	0101 <sub>2</sub> 0110 <sub>2</sub>	0100 <sub>2</sub>	0100 <sub>2</sub>

Cifra decimale	Codice BCD (8421)	Codice «ec-cesso 3»	Codice 2421	Codice 5211	Codice 631-1	Codice 732-1
4	0100 <sub>2</sub>	0111 <sub>2</sub>	0100 <sub>2</sub> 1010 <sub>2</sub>	0111 <sub>2</sub>	0110 <sub>2</sub>	0111 <sub>2</sub>
5	0101 <sub>2</sub>	1000 <sub>2</sub>	0101 <sub>2</sub> 1011 <sub>2</sub>	1000 <sub>2</sub>	1001 <sub>2</sub>	0110 <sub>2</sub>
6	0110 <sub>2</sub>	1001 <sub>2</sub>	0110 <sub>2</sub> 1100 <sub>2</sub>	1010 <sub>2</sub> 1001 <sub>2</sub>	1000 <sub>2</sub>	1001 <sub>2</sub>
7	0111 <sub>2</sub>	1010 <sub>2</sub>	0111 <sub>2</sub> 1101 <sub>2</sub>	1011 <sub>2</sub> 1100 <sub>2</sub>	1010 <sub>2</sub>	1000 <sub>2</sub>
8	1000 <sub>2</sub>	1011 <sub>2</sub>	1110 <sub>2</sub>	1110 <sub>2</sub> 1101 <sub>2</sub>	1101 <sub>2</sub>	1011 <sub>2</sub>
9	1001 <sub>2</sub>	1100 <sub>2</sub>	1111 <sub>2</sub>	1111 <sub>2</sub>	1100 <sub>2</sub> 1111 <sub>2</sub>	1010 <sub>2</sub>

La codifica BCD e altre sono *codici pesati*, in quanto a ogni bit viene attribuito un peso, da sommare per determinare il valore. I pesi per la codifica BCD sono 8, 4, 2 e 1; pertanto, il codice BCD 1001<sub>2</sub> corrisponde a  $1*8+0*4+0*2+1*1 = 9$ . Nella tabella riepilogativa, i codici pesati sono: BCD, 2421, 5211, 631-1 e 732-1. I nomi usati per questi codici sono costituiti dalla sequenza dei pesi stessi.

Alcuni codici pesati prevedono la rappresentazione di qualche cifra in più in un modo alternativo. Per esempio, nel codice 2421, il numero due si può ottenere sia come 1000<sub>2</sub>, sia come 0010<sub>2</sub>.

I codici pesati come BCD (ovvero 8421), 2421 e 5211, prevedono pesi positivi; i codici come 631-1 e 732-1, prevedono anche pesi negativi. Per esempio, con il codice 732-1, si ottiene il valore uno con il codice 0011<sub>2</sub>, perché il secondo bit (a destra) vale come il numero due, mentre il primo bit (a destra) sottrae una unità.

Dei codici che appaiono nella tabella, il codice a eccesso tre, non è un codice pesato, in quanto corrisponde al codice BCD, a cui si aggiunge il valore tre.

È necessario sottolineare che il codice BCD, a seconda del contesto, può essere riferito anche a un codice a otto bit, dove i primi quattro, più significativi, sono posti a zero.

### 63.3.2 Rappresentazione binaria di numeri interi senza segno

«

Quando si rappresentano dei valori numerici in forma binaria, senza passare per una conversione di ogni singola cifra decimale, si usa tutta la sequenza di bit per il valore. La rappresentazione di un valore intero senza segno coincide normalmente con il valore binario contenuto nella variabile. Pertanto, una variabile della dimensione di 8 bit, può rappresentare valori da zero a  $2^8-1$ :

$00000000_2$  ( $0_{10}$ )

$00000001_2$  ( $1_{10}$ )

$00000010_2$  ( $2_{10}$ )

...

$11111110_2$  ( $254_{10}$ )

$11111111_2$  ( $255_{10}$ )

### 63.3.3 Rappresentazioni binarie superate di numeri interi con segno

«

Un numero binario, inserito nella memoria di un elaboratore, può contenere esclusivamente cifre numeriche; pertanto, la rappresen-



tazione del segno può avvenire solo attraverso cifre numeriche. A partire approssimativamente dal 1965, il segno di un numero intero si rappresenta attraverso il complemento alla base (complemento a due) che ha delle proprietà importanti, ma per comprenderle occorre vedere quali sono state le alternative precedenti.

Il primo modo utilizzato per rappresentare un numero intero con segno è stato quello di attribuire a un bit (probabilmente quello più significativo) il ruolo di indicatore del segno. Per esempio,  $00001010_2$  andrebbe interpretato come  $+10_{10}$ , mentre  $10001010_2$  rappresenterebbe il valore  $-10_{10}$ . In questo modo, disponendo di otto cifre binarie, dovendone riservare una per il segno, si potrebbero rappresentare valori da  $-127$  ( $1111111_2$ ) a  $+127$  ( $0111111_2$ ); inoltre, lo zero potrebbe essere rappresentato indifferentemente come  $00000000_2$  o come  $1000000_2$ .

Il secondo metodo (usato per esempio nel PDP-1) prevede la rappresentazione dei numeri negativi come complemento a uno del valore positivo corrispondente. Il complemento a uno si ottiene invertendo le cifre del numero binario. In questo modo, per esempio,  $00001010_2$  rappresenterebbe sempre il valore  $+10_{10}$ , mentre  $11110101_2$  corrisponderebbe a  $-10_{10}$ . Anche in questo caso la prima cifra rappresenta il segno (dove la cifra uno indica un valore negativo), ma il segno si aggiorna automaticamente con la semplice inversione del valore. Utilizzando il complemento a uno per rappresentare i valori negativi, come nel caso precedente, su otto cifre complessive si possono indicare valori da  $-127$  a  $+127$  e lo zero si può rappresentare ancora in due modi differenti:  $00000000_2$  o  $1111111_2$ .

Disponendo di una variabile per rappresentare valori interi con segno, considerato che il bit più significativo serve a rappresentare il segno stesso, si dispone di un bit in meno per indicare il valore. Pertanto, se si dispone di  $n$  bit, si possono rappresentare valori fino a  $n-1$  bit, ovvero valori fino a  $2^{(n-1)}-1$ . Per i numeri negativi, il calcolo è lo stesso, anche se si considera che si fa riferimento a valori complementati: si può rappresentare fino a  $-(2^{(n-1)}-1)$ .

Il complemento alla base (ovvero il complemento a due) che è invece il metodo attuale per rappresentare i valori interi negativi, ha i vantaggi del metodo del complemento a uno, ma in più ha un solo modo per rappresentare lo zero.

#### 63.3.4 Complemento a due

«

Attualmente, per rappresentare valori interi con segno (positivo o negativo), si utilizza il metodo del complemento alla base, ovvero del complemento a due, dove il primo bit indica sempre il segno. Il complemento a due si ottiene facilmente calcolando prima il complemento a uno e poi aggiungendo una unità al risultato. Per esempio, se si prende un valore positivo rappresentato in otto cifre binarie come  $00010100_2$  (pari a  $+20_{10}$ ), il complemento a uno è  $11101011_2$ ; aggiungendo una unità si ottiene il complemento a due:  $11101100_2$ .

Utilizzando questo metodo, per cambiare di segno a un valore è sufficiente calcolarne il complemento a due (esattamente come si farebbe con il metodo del complemento a uno). Lo si verifica facilmente: riprendendo l'esempio già fatto, partendo da  $-20_{10}$  che si rappresenta come  $11101100_2$ , si calcola prima il complemento a uno, ottenendo

così  $00010011_2$ , quindi si somma una unità e si ottiene  $00010100_2$ , pari a  $+20_{10}$ .

Con il complemento a due, disponendo di  $n$  cifre binarie, si possono rappresentare valori da  $-2^{(n-1)}$  a  $+2^{(n-1)}-1$  ed esiste un solo modo per rappresentare lo zero: quando tutte le cifre binarie sono pari a zero. Infatti, rimanendo nell'ipotesi di otto cifre binarie, il complemento a uno di  $00000000_2$  è  $11111111_2$ , ma aggiungendo una unità per ottenere il complemento a due si ottiene di nuovo  $00000000_2$ , perdendo il riporto.

Si osservi che il valore negativo più grande rappresentabile non può essere trasformato in un valore positivo corrispondente, perché si creerebbe un traboccamento. Per esempio, utilizzando sempre otto bit (segno incluso), il valore minimo che possa essere rappresentato è  $1000000_2$ , pari a  $-128_{10}$ , ma se si calcola il complemento a due, si ottiene di nuovo lo stesso valore binario, che però non è valido. Infatti, il valore positivo massimo che si possa rappresentare in questo caso è solo  $+127_{10}$ .

Figura 63.24. Confronto tra due valori interi con segno.

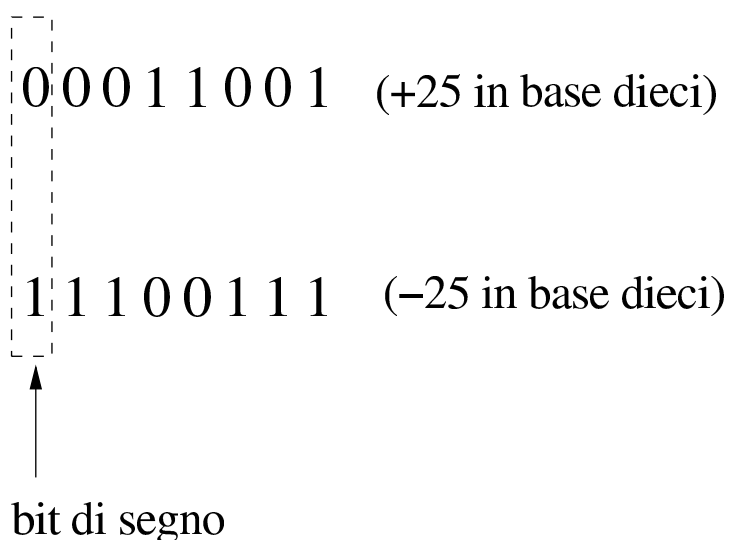
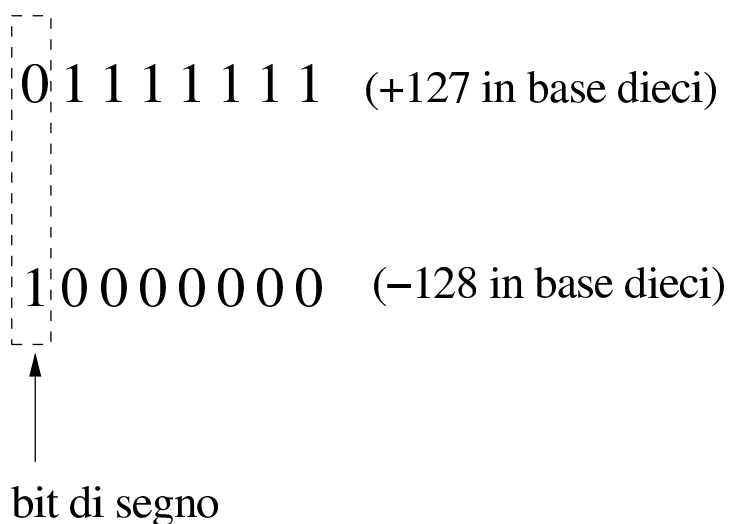


Figura 63.25. Valori massimi rappresentabili con soli otto bit.



Il meccanismo del complemento a due ha il vantaggio di trasformare la sottrazione in una semplice somma algebrica.

### 63.3.5 Rappresentazione binaria di numeri in virgola mobile

«

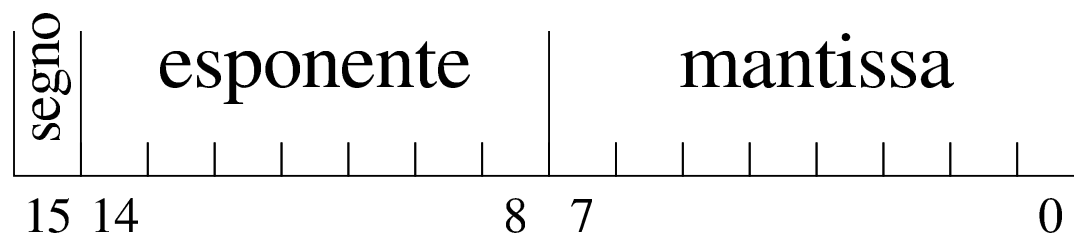
Una forma diffusa per rappresentare dei valori molto grandi, consiste nell'indicare un numero con dei decimali moltiplicato per un valore costante elevato a un esponente intero. Per esempio, per rappresentare il numero 123000000 si potrebbe scrivere  $123 \cdot 10^6$ , oppure anche  $0,123 \cdot 10^9$ . Lo stesso ragionamento vale anche per valori molto piccoli; per esempio 0,000000123 che si potrebbe esprimere come  $0,123 \cdot 10^{-6}$ .

Per usare una notazione uniforme, si può convenire di indicare il numero che appare prima della moltiplicazione per la costante elevata a una certa potenza come un valore che più si avvicina all'unità, essendo minore o al massimo uguale a uno. Pertanto, per gli esempi già mostrati, si tratterebbe sempre di  $0,123 \cdot 10^n$ .

Per rappresentare valori a *virgola mobile* in modo binario, si usa

un sistema simile, dove i bit a disposizione della variabile vengono suddivisi in tre parti: segno, esponente (di una base prestabilita) e mantissa, come nell'esempio che appare nella figura successiva.<sup>2</sup>

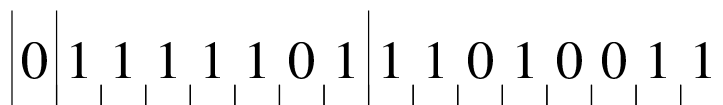
Figura 63.26. Ipotesi di una variabile a 16 bit per rappresentare dei numeri a virgola mobile.



Nella figura si ipotizza la gestione di una variabile a 16 bit per la rappresentazione di valori a virgola mobile. Come si vede dallo schema, il bit più significativo della variabile viene utilizzato per rappresentare il segno del numero; i sette bit successivi si usano per indicare l'esponente (con segno) e gli otto bit finali per la mantissa (senza segno perché indicato nel primo bit), ovvero il valore da moltiplicare per una certa costante elevata all'esponente.

Quello che manca da decidere è come deve essere interpretato il numero della mantissa e qual è il valore della costante da elevare all'esponente indicato. Sempre a titolo di esempio, si conviene che il valore indicato nella mantissa esprima precisamente « $0, \textit{mantissa}$ » e che la costante da elevare all'esponente indicato sia 16 (ovvero  $2^4$ ), che si traduce in pratica nello spostamento della virgola di quattro cifre binarie alla volta.<sup>3</sup>

Figura 63.27. Esempio di rappresentazione del numero  $0,051513671875$  ( $211 \cdot 16^{-3}$ ), secondo le convenzioni stabilite. Si osservi che il valore dell'esponente è negativo ed è così rappresentato come complemento alla base (due) del valore assoluto relativo.



$$+211 \cdot 16^{-3}$$

0,000000000000011010011

Naturalmente, le convenzioni possono essere cambiate: per esempio il segno lo si può incorporare nella mantissa; si può rappresentare l'esponente attraverso un numero al quale deve essere sottratta una costante fissa; si può stabilire un valore diverso della costante da elevare all'esponente; si possono distribuire diversamente gli spazi assegnati all'esponente e alla mantissa.

### 63.3.6 Rappresentazione in virgola mobile IEEE 754

«

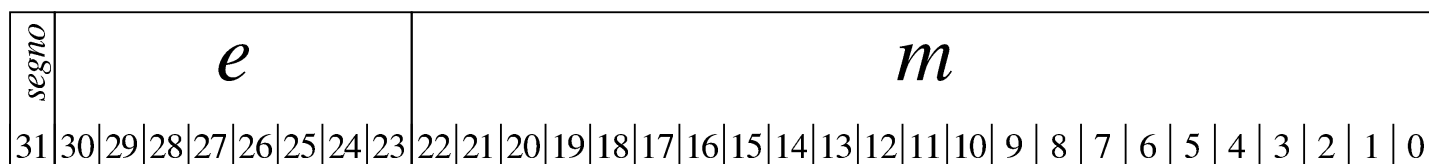
Per la rappresentazione dei valori in virgola mobile esiste uno standard importante, IEEE 754 (ripreso anche da altri enti di standardizzazione), con il quale si definiscono due formati, per la precisione singola e doppia. Secondo questo standard, un valore in virgola mobile a precisione singola richiede 32 bit, mentre per la precisione doppia sono necessari 64 bit. Per prima cosa si definisce un «numero normalizzato», corrispondente a:<sup>4</sup>

$$1, \textit{significante}_2 \times 2^{\textit{esponente}}$$

Di questo si utilizza solo il *significante* (mantissa) e l'*esponente* (caratteristica), omettendo il numero uno iniziale. Nella forma prevista

dallo standard IEEE 754 si annota separatamente il segno del numero, quindi l'esponente, che però è «polarizzato» (nel senso che al valore dell'esponente originario occorre sommare un certo valore fisso), quindi si mettono le cifre del significante (tutte quelle che possono starci). Si osservi che il significante viene indicato sempre come valore assoluto, pertanto non si applica il complemento per i valori negativi; inoltre, l'esponente viene indicato sommando al valore originale un numero fisso che è costituito da tutti i bit a uno, tranne quello più significativo (quando l'esponente è formato da otto bit, il numero da sommare è  $0111111_2$ , pari a  $127_{10}$ ; quando l'esponente è formato da 11 bit, il numero da sommare è  $01111111111_2$ , pari a  $1023_{10}$ ).

Figura 63.28. IEEE 754 a precisione singola.



bit 31            (1)    segno: 0 = positivo; 1 = negativo

bit 23–30        (8)    esponente in eccesso 127 = esponente originale + 127

bit 0–22        (23)    significante (mantissa)

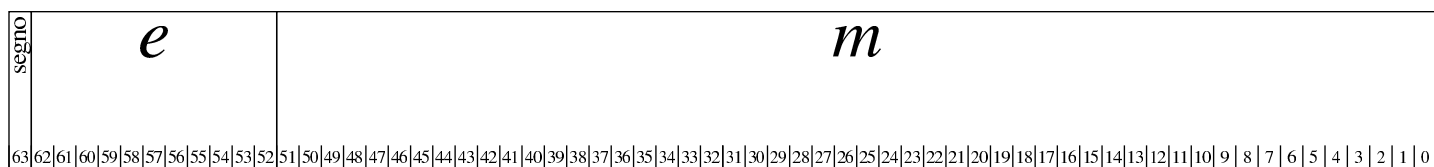
$e = 0$              $m = 0$             zero, che può essere positivo o negativo

$e = 0$              $m \neq 0$             numero «denormalizzato»

$e = 255$            $m = 0$             infinito, che può essere positivo o negativo

$e = 255$            $m \neq 0$             indefinito

Figura 63.29. IEEE 754 a precisione doppia.



bit 63 (1) segno: 0 = positivo; 1 = negativo

bit 52–62 (11) esponente in eccesso 1023 = esponente originale + 1023

bit 0–51 (52) significante (mantissa)

$e = 0$        $m = 0$       zero, che può essere positivo o negativo

$e = 0$        $m \neq 0$       numero «denormalizzato»

$e = 2047$      $m = 0$       infinito, che può essere positivo o negativo

$e = 2047$      $m \neq 0$       indefinito

Conviene fare un esempio con la precisione singola: si vuole rappresentare il valore  $21,11_{10}$ . Si procede convertendo separatamente la parte intera e poi quella decimale. Il numero  $21_{10}$  si converte facilmente in  $10101_2$ , mentre per la parte decimale occorre fare qualche calcolo in più:

$$0,11 \times 2 = 0,22 \rightarrow ,0_2$$

$$0,22 \times 2 = 0,44 \rightarrow ,00_2$$

$$0,44 \times 2 = 0,88 \rightarrow ,000_2$$

$$0,88 \times 2 = 1,76 \rightarrow ,0001_2$$

$$0,76 \times 2 = 1,52 \rightarrow ,00011_2$$

$$0,52 \times 2 = 1,04 \rightarrow ,000111_2$$

$$0,04 \times 2 = 0,08 \rightarrow ,0001110_2$$

$$0,08 \times 2 = 0,16 \rightarrow ,00011100_2$$

$$0,16 \times 2 = 0,32 \rightarrow ,000111000_2$$

$$0,32 \times 2 = 0,64 \rightarrow ,0001110000_2$$

$$0,64 \times 2 = 1,28 \rightarrow ,00011100001_2$$

$$0,28 \times 2 = 0,56 \rightarrow ,000111000010_2$$

$$0,56 \times 2 = 1,12 \rightarrow ,0001110000101_2$$



$$\begin{aligned}
0,12 \times 2 &= 0,24 \rightarrow ,00011100001010_2 \\
0,24 \times 2 &= 0,48 \rightarrow ,000111000010100_2 \\
0,48 \times 2 &= 0,96 \rightarrow ,0001110000101000_2 \\
0,96 \times 2 &= 1,92 \rightarrow ,00011100001010001_2 \\
0,92 \times 2 &= 1,84 \rightarrow ,000111000010100011_2 \\
0,84 \times 2 &= 1,68 \rightarrow ,0001110000101000111_2 \\
0,68 \times 2 &= 1,36 \rightarrow ,00011100001010001111_2 \\
0,36 \times 2 &= 0,72 \rightarrow ,000111000010100011110_2 \\
&\dots
\end{aligned}$$

Pertanto,  $21,11_{10}$  corrisponde approssimativamente a  $10101,000111000010100011110_2$ . Per normalizzare il numero occorre spostare la virgola a sinistra e moltiplicare per una potenza di due:  $1,0101000111000010100011110_2 \times 2^4$ .

A questo punto si prelevano 23 cifre dopo la virgola, ma si richiede un arrotondamento (in questo caso avviene per eccesso), pertanto le cifre che compongono il significante diventano:  $01010001110000101001000_2$ . L'esponente va sommato al valore costante stabilito:  $4+127 = 131$ . Quindi l'esponente si rappresenta così:  $10000011_2$ . Trattandosi di un numero positivo, il bit del segno deve essere zero. Ecco il numero in virgola mobile, a precisione singola, espresso secondo la notazione standard (gli spazi aggiunti servono a facilitarne la lettura):

$$0 \ 10000011 \ 01010001110000101001000_2$$

Con questo metodo, un numero a precisione singola può avere un valore assoluto da  $1_2 \times 2^{-126}$  a  $1,1111\dots_2 \times 2^{+127}$ ; mentre un numero a precisione doppia può avere un valore assoluto da  $1_2 \times 2^{-1022}$  a  $1,1111\dots_2 \times 2^{+1023}$ . Quando si devono rappresentare va-

lori molto bassi, si azzerano i bit dell'esponente e si usa una forma «denormalizzata» di questo tipo per la precisione singola:

$$0, \textit{significante}_2 \times 2^{-127}$$

Per la precisione doppia:

$$0, \textit{significante}_2 \times 2^{-1024}$$

### 63.3.7 Ordine dei byte<sup>5</sup>

«

Generalmente si distinguono i microprocessori in base a una caratteristica legata al modo di ordinare i bit di un numero, presi a gruppi di otto. In pratica, di norma la memoria centrale degli elaboratori è organizzata a celle di otto bit (un byte), mentre il microprocessore è in grado di elaborare dati numerici con una quantità di bit maggiore (ma sempre multipli di otto). Nel momento in cui il microprocessore accede alla memoria centrale per leggere o scrivere un valore, lo fa secondo un ordine che dipende dalla sua progettazione.

Supponendo di avere a che fare con il valore  $13579\text{BDF}_{16}$ , se il microprocessore lo memorizza secondo la stessa sequenza (ovvero memorizza i byte  $13_{16}$ ,  $57_{16}$ ,  $9\text{B}_{16}$  e  $\text{DF}_{16}$ ), allora si dice che la sua architettura è *big endian*; diversamente, se il microprocessore memorizza invertendo la sequenza di byte (quindi  $\text{DF}_{16}$ ,  $9\text{B}_{16}$ ,  $57_{16}$  e  $13_{16}$ ) si dice che questo lavora in modalità *little endian*.

Naturalmente, il microprocessore che scrive in memoria un valore secondo una sequenza di byte invertita, quando va a leggerlo dalla memoria si aspetta di trovarlo invertito nello stesso modo.

Sia chiaro che, all'interno di ogni byte, l'ordine dei bit non viene modificato. Inoltre, nel momento in cui si pensa a un'elaborazione all'interno del microprocessore, con dati contenuti nei suoi registri, non ha importanza conoscere qual è l'ordine dei byte.

## 63.4 Calcoli con i valori binari rappresentati nella forma usata negli elaboratori

Una volta chiarito il modo in cui si rappresentano comunemente i valori numerici elaborati da un microprocessore, in particolare per ciò che riguarda i valori negativi con il complemento a due, occorre conoscere in che modo si trattano o si possono trattare questi dati (indipendentemente dall'ordine dei byte usato).

Questi concetti tornano utili nella programmazione in linguaggio macchina o nei linguaggi assembleri equivalenti, ma servono anche per linguaggi evoluti che conservano una rappresentazione dei valori conforme all'architettura dell'elaboratore.

### 63.4.1 Modifica della quantità di cifre di un numero binario intero

Un numero intero senza segno, espresso con una certa quantità di cifre, può essere trasformato in una quantità di cifre maggiore, aggiungendo degli zeri nella parte più significativa. Per esempio, il numero  $0101_2$  può essere trasformato in  $00000101_2$  senza cambiarne il valore. Nello stesso modo, si può fare una copia di un valore in un contenitore più piccolo, perdendo le cifre più significative, purché queste siano a zero, altrimenti il valore risultante sarebbe alterato.

Quando si ha a che fare con valori interi con segno, nel caso di valori positivi, l'estensione e la riduzione funzionano come per i valori senza segno, con la differenza che nella riduzione di cifre, la prima deve ancora rappresentare un segno positivo. Se invece si ha a che fare con valori negativi, l'aumento di cifre richiede l'aggiunta di cifre a uno nella parte più significativa, mentre la riduzione comporta l'eliminazione di cifre a uno nella parte più significativa, con il vincolo di mantenere inalterato il segno.

Figura 63.30. Aumento e riduzione delle cifre di un numero intero senza segno.

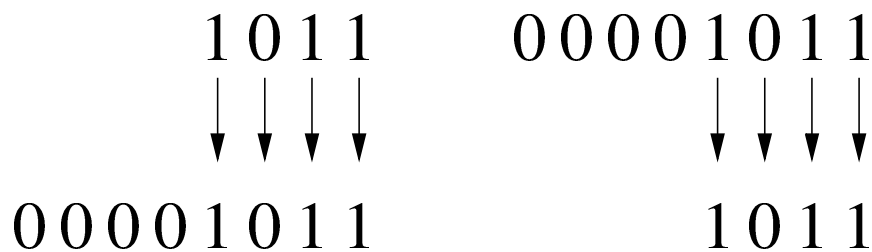


Figura 63.31. Aumento e riduzione delle cifre di un numero intero positivo.

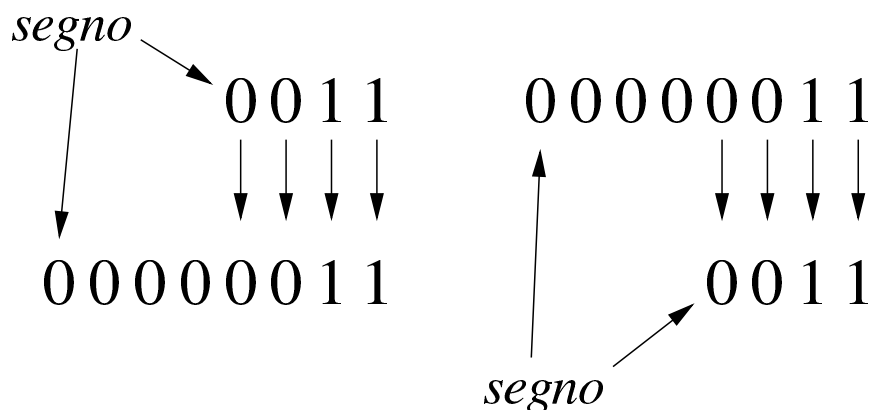
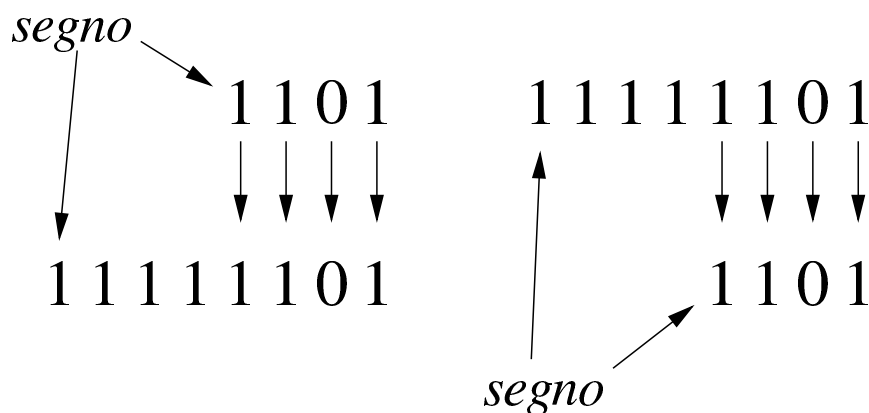


Figura 63.32. Aumento e riduzione delle cifre di un numero intero negativo.



### 63.4.2 Sommatorie con i valori interi con segno

Vengono proposti alcuni esempi che servono a dimostrare le situazioni che si presentano quando si sommano valori con segno, ricordando che i valori negativi sono rappresentati come complemento alla base del valore assoluto corrispondente. «

Figura 63.33. Somma di due valori positivi che genera un risultato valido.

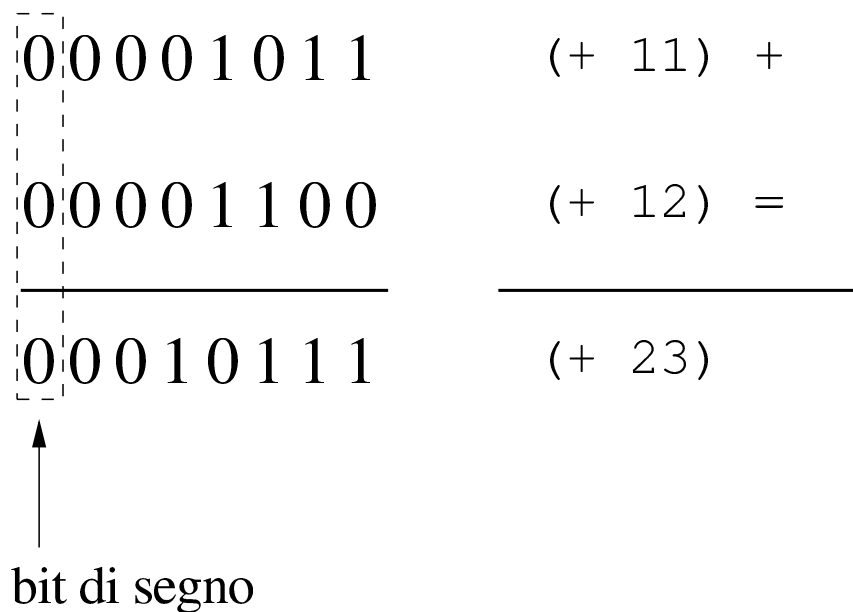


Figura 63.34. Somma di due valori positivi, dove il risultato apparentemente negativo indica la presenza di un traboccamento.

bit di segno	
↓	
0 1 0 0 1 0 1 1	(+ 75) +
0 1 0 0 1 1 0 0	(+ 76) =
-----	-----
1 0 0 1 0 1 1 1	(+ 151)
↓	
traboccamento (overflow)	

Figura 63.35. Somma di un valore positivo e di un valore negativo: il risultato è sempre valido.

0 0 0 0 1 0 1 1	(+ 11) +
1 1 1 1 0 1 0 0	(- 12) =
-----	-----
1 1 1 1 1 1 1 1	(- 1)
↑	
bit di segno	

Figura 63.36. Somma di un valore positivo e di un valore negativo: in tal caso il risultato è sempre valido e se si manifesta un riporto, come in questo caso, va ignorato semplicemente.

0	1	0	0	1	0	1	1	(+ 75) +
1	1	1	1	0	1	0	0	(- 12) =
1	0	0	1	1	1	1	1	(+ 63)

riporto da ignorare ↗

↑ bit di segno

Figura 63.37. Somma di due valori negativi che produce un segno coerente e un riporto da ignorare.

1	1	0	0	1	0	1	1	(- 53) +
1	1	1	1	0	1	0	0	(- 12) =
1	1	0	1	1	1	1	1	(- 65)

riporto da ignorare ↗

↑ bit di segno

Figura 63.38. Somma di due valori negativi che genera un traboccamento, evidenziato da un risultato con un segno incoerente.

bit di segno		
↓	1	0 0 0 1 0 1 1
		(- 117) +
	1	1 1 1 0 1 0 0
		(- 12) =
	1	0 1 1 1 1 1 1
		(- 129)

riporto da  
ignorare

↑  
traboccamento

Dagli esempi mostrati si comprende facilmente che la somma di due valori con segno va fatta ignorando il riporto, perché quello che conta è che il segno risultante sia coerente: se si sommano due valori positivi, perché il risultato sia valido deve essere positivo; se si somma un valore positivo con uno negativo il risultato è sempre valido; se si sommano due valori negativi, perché il risultato sia valido deve rimanere negativo.

### 63.4.3 Somme e sottrazioni con i valori interi senza segno

«

La somma di due numeri interi senza segno avviene normalmente, senza dare un valore particolare al bit più significativo, pertanto, se si genera un riporto, il risultato non è valido (salva la possibilità di considerarlo assieme al riporto). Se invece si vuole eseguire una sottrazione, il valore da sottrarre va «invertito», con il complemento a due, ma sempre evitando di dare un significato particolare al bit più



significativo. Il valore «normale» e quello «invertito» vanno sommati come al solito, ma **se il risultato non genera un riporto**, allora è **sbagliato**, in quanto il sottraendo è più grande del minuendo.

Per comprendere come funziona la sottrazione, si consideri di volere eseguire un'operazione molto semplice:  $1-1$ . Il minuendo (il primo valore) sia espresso come  $00000001_2$ ; il sottraendo (il secondo valore) che sarebbe uguale, va trasformato attraverso il complemento a due, diventando così pari a  $1111111_2$ . A questo punto si sommano algebricamente i due valori e si ottiene  $0000000_2$  con riporto di uno. Il riporto di uno dà la garanzia che il risultato è corretto. Volendo provare a sottrarre un valore più grande, si vede che il riporto non viene ottenuto:  $1-2$ . In questo caso il minuendo si esprime come nell'esempio precedente, mentre il sottraendo è  $00000010_2$  che si trasforma nel complemento a due  $11111110_2$ . Se si sommano i due valori si ottiene semplicemente  $1111111_2$ , senza riporto, ma questo valore che va inteso senza segno è evidentemente errato.

Figura 63.39. Sottrazione tra due numeri interi senza segno, dove il sottraendo ha un valore assoluto minore di quello del minuendo: la presenza del riporto conferma la validità del risultato.

$$\begin{array}{r}
 0011 - \\
 0011 = \\
 \hline
 0000
 \end{array}
 \xrightarrow{\text{complemento}}
 \begin{array}{r}
 0011 + \\
 1101 = \\
 \hline
 10000
 \end{array}$$

1
risultato

*il riporto conferma la validità del risultato  
naturalmente il riporto viene ignorato*

Figura 63.40. Sottrazione tra due numeri interi senza segno, dove il sottraendo ha un valore assoluto maggiore di quello del minuendo: l'assenza di un riporto indica un risultato errato della sottrazione.

$$\begin{array}{r}
 0011 - \\
 0100 = \\
 \hline
 -0001
 \end{array}
 \xrightarrow{\text{complemento}}
 \begin{array}{r}
 0011 + \\
 1100 = \\
 \hline
 01111
 \end{array}$$

*la mancanza del riporto indica un risultato errato*  
*risultato errato*  
*(perché considerato senza segno)*

Sulla base della spiegazione data, c'è però un problema, dovuto al fatto che il complemento a due di un valore a zero dà sempre zero: se si fa la sottrazione con il complemento, il risultato è comunque corretto, ma non si ottiene un riporto.

Figura 63.41. Sottrazione con sottraendo a zero: non si ottiene riporto, ma il risultato è corretto ugualmente.

$$\begin{array}{r}
 0011 - \\
 0000 = \\
 \hline
 -0011
 \end{array}
 \xrightarrow{\text{complemento}}
 \begin{array}{r}
 0011 + \\
 0000 = \\
 \hline
 00011
 \end{array}$$

*in questa situazione particolare, il riporto è zero,*  
*ma il risultato è corretto ugualmente*  
*risultato corretto*

Per correggere questo problema, il complemento a due del numero da sottrarre, va eseguito in due fasi: prima si calcola il complemento a uno, poi si somma il minuendo al sottraendo complementato,

aggiungendo una unità ulteriore. Le figure successive ripetono gli esempi già mostrati, attuando questo procedimento differente.

Figura 63.42. Il complemento a due viene calcolato in due fasi: prima si calcola il complemento a uno, poi si sommano il minuendo e il sottraendo invertito, più una unità.

$$\begin{array}{r}
 0011 - \\
 0011 = \\
 \hline
 0000
 \end{array}
 \xrightarrow{\text{complemento a uno}}
 \begin{array}{r}
 1 + \\
 0011 + \\
 1100 = \\
 \hline
 10000
 \end{array}$$

*risultato*

*il riporto conferma la validità del risultato  
naturalmente il riporto viene ignorato*

$$\begin{array}{r}
 0011 - \\
 0100 = \\
 \hline
 -0001
 \end{array}
 \xrightarrow{\text{complemento a uno}}
 \begin{array}{r}
 1 + \\
 0011 + \\
 1011 = \\
 \hline
 01111
 \end{array}$$

*risultato  
errato*

*la mancanza del riporto indica un risultato errato*

*(perché considerato  
senza segno)*

Figura 63.44. Sottrazione con sottraendo a zero: calcolando il complemento a due attraverso il complemento a uno, si ottiene un riporto coerente.

$$\begin{array}{r}
 0011 - \\
 0000 = \\
 \hline
 -0011
 \end{array}
 \xrightarrow{\text{complemento a uno}}
 \begin{array}{r}
 1 + \\
 0011 + \\
 1111 = \\
 \hline
 10011
 \end{array}$$

*il riporto conferma la validità del risultato naturalmente il riporto viene ignorato*

*risultato corretto*

#### 63.4.4 Somme e sottrazioni in fasi successive

«

Quando si possono eseguire somme e sottrazioni solo con una quantità limitata di cifre, mentre si vuole eseguire un calcolo con numeri più grandi della capacità consentita, si possono suddividere le operazioni in diverse fasi. La somma tra due numeri interi è molto semplice, perché ci si limita a tenere conto del riporto ottenuto nelle fasi precedenti. Per esempio, dovendo sommare  $0101\ 1010\ 1100_2$  a  $1000\ 0101\ 0111_2$  e potendo operare solo a gruppi di quattro bit per volta: si parte dal primo gruppo di bit meno significativo,  $1100_2$  e  $0111_2$ , si sommano i due valori e si ottiene  $0011_2$  con riporto di uno; si prosegue sommando  $1010_2$  con  $0101_2$  aggiungendo il riporto e ottenendo  $0000_2$  con riporto di uno; si conclude sommando  $0101_2$  e  $1000_2$ , aggiungendo il riporto della somma precedente e si ottiene così  $1110_2$ . Quindi, il risultato è  $1110\ 0000\ 0011_2$ .

Figura 63.45. Somma per fasi successive, tenendo conto del riporto.

$$\begin{array}{r}
 010110101100 + \\
 100001010111 = \\
 \hline
 111000000011
 \end{array}
 \quad
 \begin{array}{r}
 \overset{1}{\leftarrow} \\
 0101 + \\
 \hline
 1110
 \end{array}
 \quad
 \begin{array}{r}
 \overset{1}{\leftarrow} \\
 1010 + \\
 \hline
 0101
 \end{array}
 \quad
 \begin{array}{r}
 \overset{1}{\leftarrow} \\
 1100 + \\
 \hline
 0111
 \end{array}
 \quad
 \begin{array}{r}
 \overset{1}{\leftarrow} \\
 10000 \\
 \hline
 10011
 \end{array}
 =$$

*riporto*

Nella sottrazione tra numeri senza segno, il sottraendo va trasformato secondo il complemento a due, quindi si esegue la somma e si considera che ci deve essere un riporto, altrimenti significa che il sottraendo è maggiore del minuendo. Quando si deve eseguire la sottrazione a gruppi di cifre più piccoli di quelli che richiede il valore per essere rappresentato, si può procedere in modo simile a quello che si usa con la somma, con la differenza che «l'assenza del riporto» indica la richiesta di prendere a prestito una cifra.

Per comprendere il procedimento è meglio partire da un esempio. In questo caso si utilizzano i valori già visti, ma invece di sommarli si vuole eseguire la sottrazione. Per la precisione, si intende prendere  $1000\ 0101\ 0111_2$  come minuendo e  $0101\ 1010\ 1100_2$  come sottraendo. Anche in questo caso si suppone di poter eseguire le operazioni solo a gruppi di quattro bit. Si esegue il complemento a due dei tre gruppetti di quattro bit del sottraendo, in modo indipendente, ottenendo:  $1011_2$ ,  $0110_2$ ,  $0100_2$ . A questo punto si eseguono le somme, a partire dal gruppo meno significativo. La prima somma,  $0111_2 + 0100_2$ , dà  $1011_2$ , senza riporto, pertanto occorre prendere a prestito una cifra dal gruppo successivo: ciò significa che va eseguita

la somma del gruppo successivo, sottraendo una unità dal risultato:  $0101_2 + 0110_2 - 0001_2 = 1010_2$ . Anche per il secondo gruppo non si ottiene il riporto della somma, così, anche dal terzo gruppo di bit occorre prendere a prestito una cifra:  $1000_2 + 1011_2 - 0001_2 = 0010_2$ . L'ultima volta la somma genera il riporto (da ignorare) che conferma la correttezza del risultato complessivo, ovvero che la sottrazione è avvenuta con successo.

Va però ricordato il problema legato allo zero, il cui complemento a due dà sempre zero. Se si cambiano i valori dell'esempio, lasciando come minuendo quello precedente,  $1000\ 0101\ 0111_2$ , ma modificando il sottraendo in modo da avere le ultime quattro cifre a zero,  $0101\ 1010\ 0000_2$ , il procedimento descritto non funziona più. Infatti, il complemento a due di  $0000_2$  rimane  $0000_2$  e se si somma questo a  $0111_2$  si ottiene lo stesso valore, ma senza riporti. In questo caso, nonostante l'assenza del riporto, il gruppo dei quattro bit successivi, del sottraendo, va trasformato con il complemento a due, senza togliere l'unità che sarebbe prevista secondo l'esempio precedente. In pratica, per poter eseguire la sottrazione per fasi successive, occorre definire un concetto diverso: il prestito (*borrow*) che non deve scattare quando si sottrae un valore pari a zero.

Se il complemento a due viene ottenuto passando per il complemento a uno, con l'aggiunta di una cifra, si può spiegare in modo più semplice il procedimento della sottrazione per fasi successive: invece di calcolare il complemento a due dei vari tronconi, si calcola semplicemente il complemento a uno e al gruppo meno significativo si aggiunge una unità per ottenere lì l'equivalente di un complemento a due. Successivamente, il riporto delle somme eseguite va aggiunto al gruppo adiacente più significativo, come si farebbe con la somma:

se la sottrazione del gruppo precedente non ha bisogno del prestito di una cifra, si ottiene l'aggiunta di una unità al gruppo successivo.

Figura 63.46. Sottrazione per fasi successive, tenendo conto del prestito delle cifre.

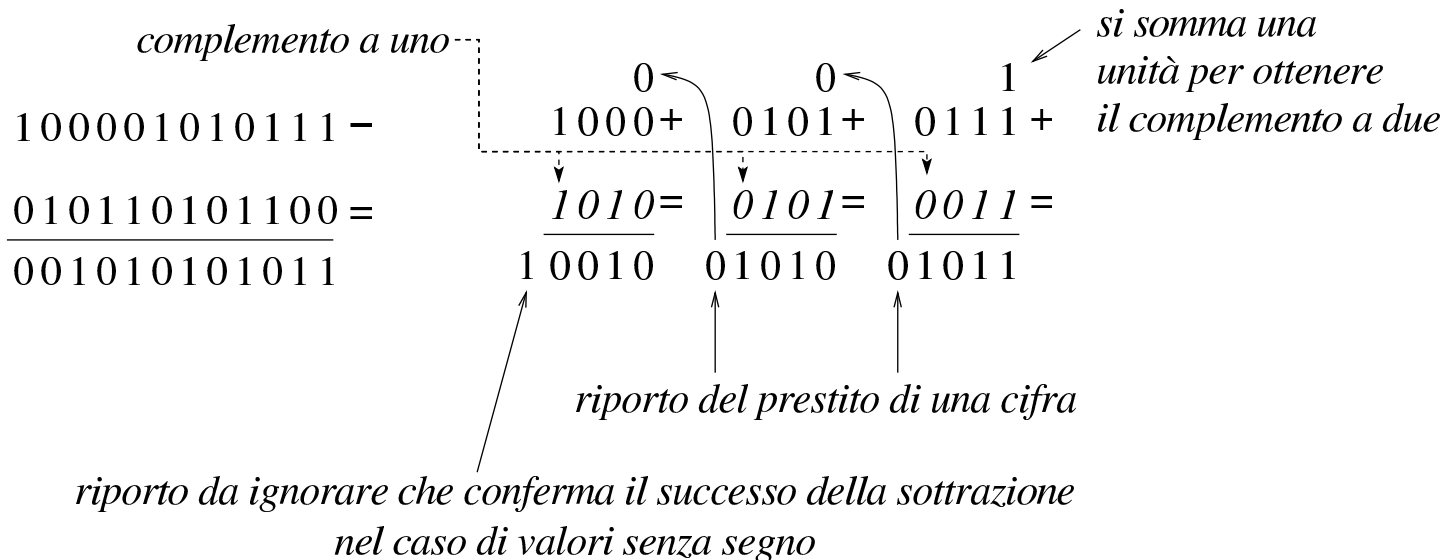
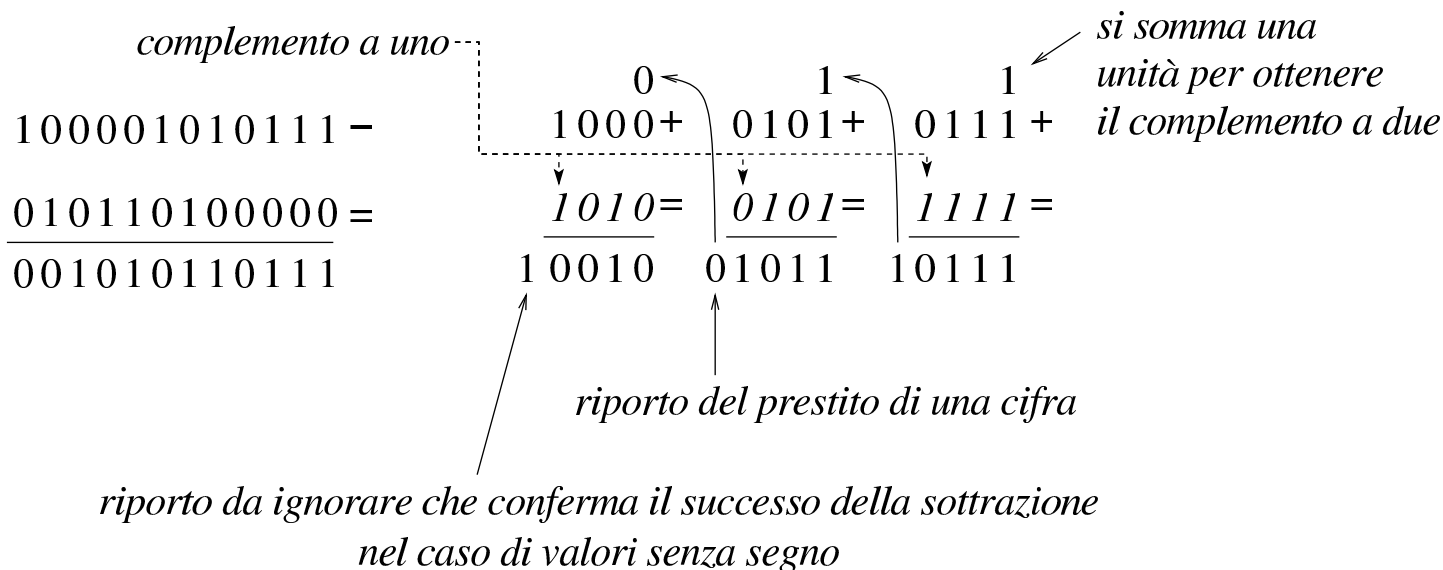


Figura 63.47. Verifica del procedimento anche in presenza di un sottraendo a zero.



La sottrazione per fasi successive funziona anche con valori che, complessivamente, hanno un segno. L'unica differenza sta nel modo di valutare il risultato complessivo: l'ultimo gruppo di cifre a es-

sere considerato (quello più significativo) è quello che contiene il segno ed è il segno del risultato che deve essere coerente, per stabilire se ciò che si è ottenuto è valido. Pertanto, nel caso di valori con segno, il riporto finale si ignora, esattamente come si fa quando la sottrazione avviene in una fase sola, mentre l'esistenza o meno del traboccamento deriva dal confronto della cifra più significativa: se la sottrazione, dopo la trasformazione in somma con il complemento, implica la somma valori con lo stesso segno, il risultato deve ancora avere quel segno, altrimenti c'è il traboccamento.

Se si volessero considerare gli ultimi due esempi come la sottrazione di valori con segno, il minuendo si intenderebbe un valore negativo, mentre il sottraendo sarebbe un valore positivo. Attraverso il complemento si passa alla somma di due valori negativi, ma dal momento che si ottiene un risultato con segno positivo, ciò manifesta un traboccamento, ovvero un risultato errato, perché non contenibile nello spazio disponibile.

### 63.4.5 Indicatori

«

Quando si esegue un calcolo con un microprocessore, oltre al risultato puro e semplice è necessario annotare altre informazioni sull'esito dell'operazione stessa. Come già descritto, una somma può dare luogo a un traboccamento o a un riporto, così come una sottrazione può richiedere un prestito di una cifra. Queste e altre informazioni, che non possono essere incorporate nel risultato di un calcolo, finiscono all'interno di indicatori (*flag*), ovvero di bit singoli, ognuno con un proprio significato preciso. Le figure successive dimostrano il funzionamento degli indicatori più comuni.



Figura 63.48. Somma di interi: se i numeri sono da intendersi senza segno, il risultato non è completo in quanto si genera un riporto; se i numeri sono da intendersi con segno, in tal caso sono negativi, ma la loro somma produce un traboccamento.

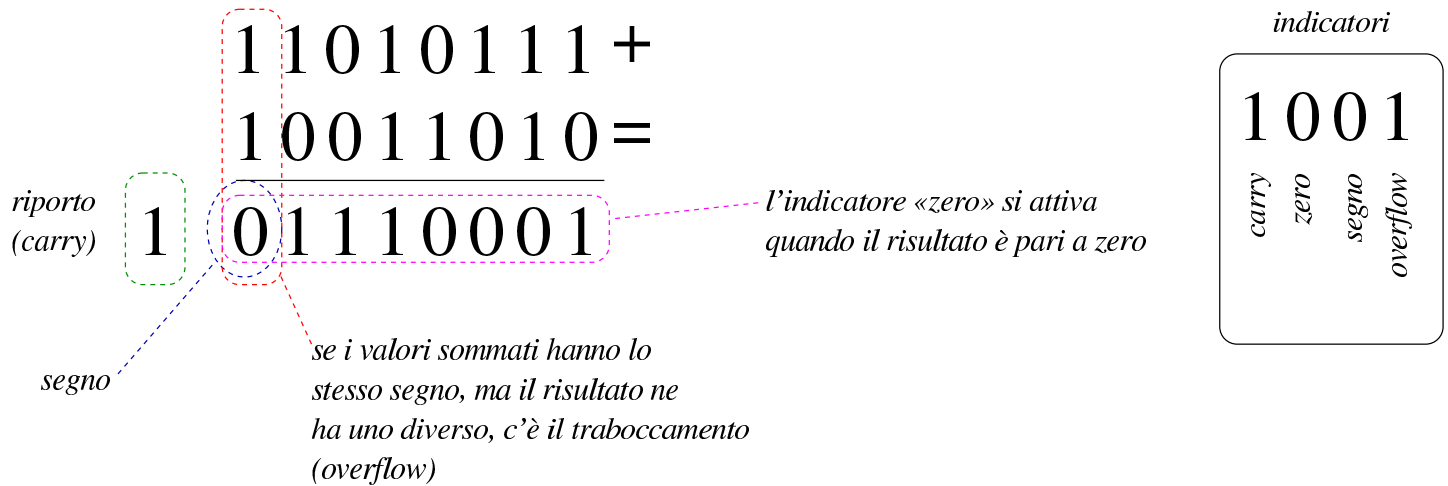


Figura 63.49. Somma di interi: se i numeri sono da intendersi senza segno, il risultato non è completo in quanto si genera un riporto; se i numeri sono da intendersi con segno, in tal caso hanno segni diversi tra di loro e questo impedisce che si crei un traboccamento, inoltre il risultato è zero e si attiva l'indicatore relativo.

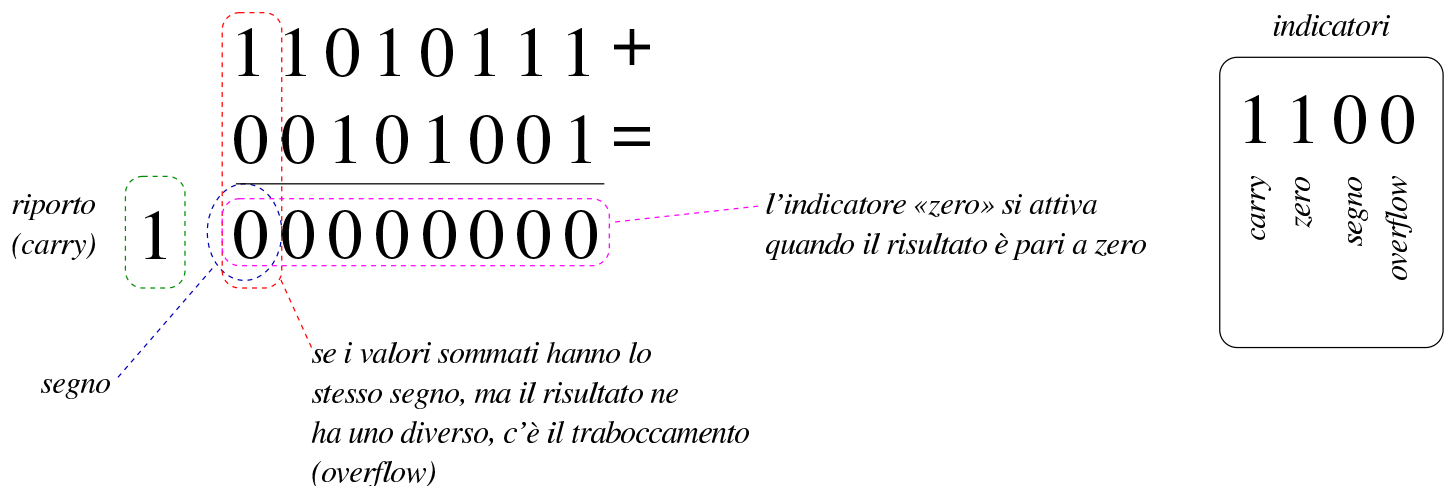
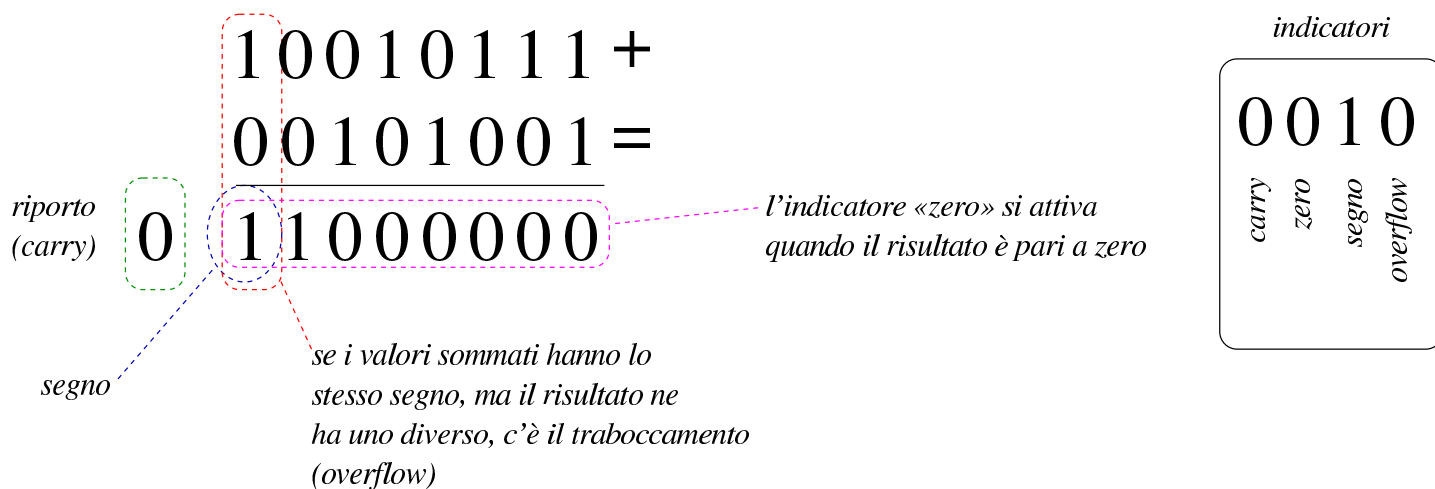


Figura 63.50. Somma di interi: se i numeri sono da intendersi senza segno, il risultato è completo in quanto non si genera un riporto; se i numeri sono da intendersi con segno, in tal caso hanno segni diversi tra di loro e questo impedisce che si crei un traboccamento, inoltre il risultato è negativo e questo attiva l'indicatore di segno.



Come già descritto in altre sezioni, le sottrazioni vanno eseguite calcolando prima il complemento a uno del sottraendo e poi aggiungendo una unità ulteriore. In questo modo, ciò che nella somma rappresenterebbe un riporto, qui va invertito per segnalare la richiesta di un prestito (*borrow*).

Figura 63.51. Sottrazione di interi: se i numeri sono da intendersi senza segno, il risultato è completo in quanto non si genera la richiesta di prestito di una cifra; se i numeri sono da intendersi con segno, si tratta di valori negativi, ma la somma genera un cambiamento di segno, pertanto si attiva l'indicatore di traboccamento.

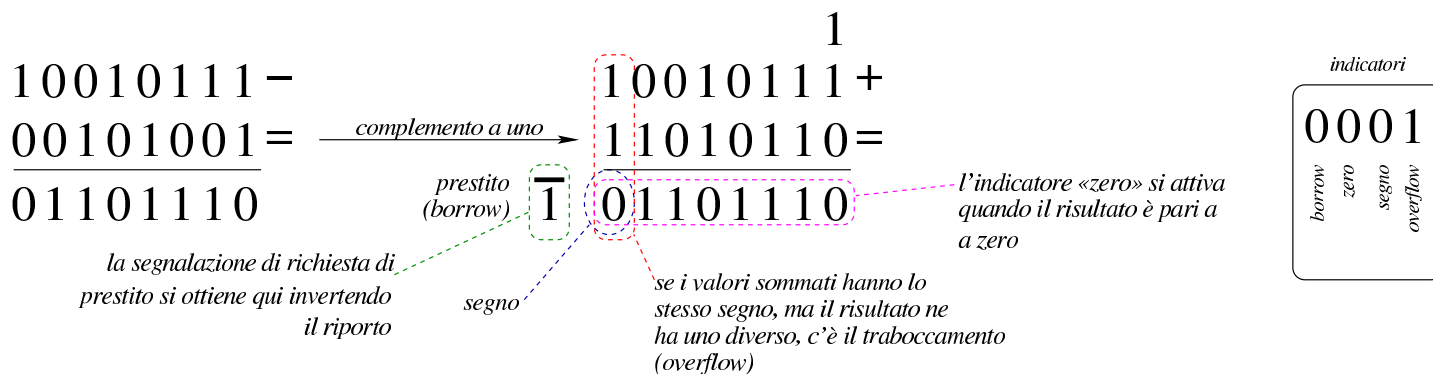
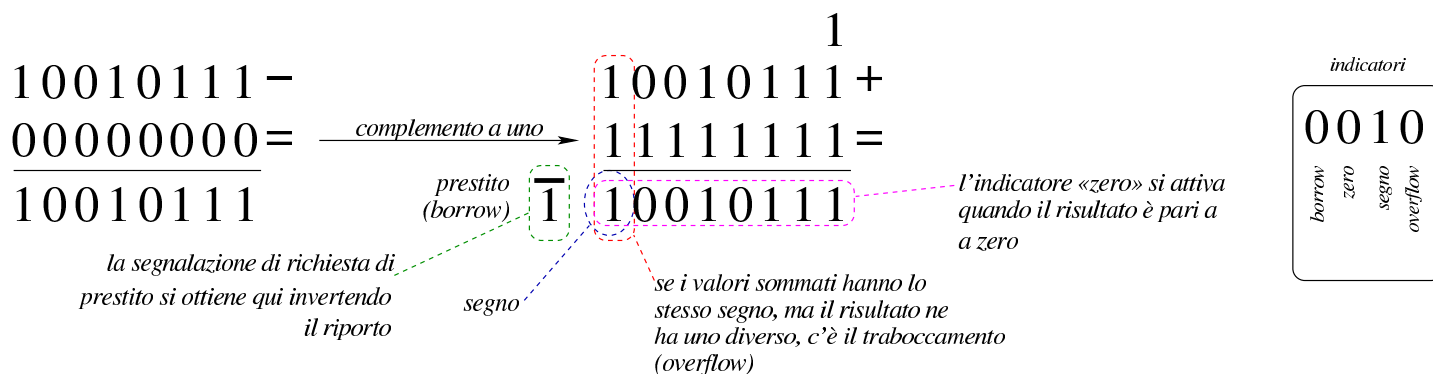


Figura 63.52. Sottrazione di interi: qui viene sottratto zero da un valore. Utilizzando il meccanismo del complemento a uno, aggiungendo una unità alla somma, si evita che scatti la richiesta di prestito, come è logico che sia. In questo caso, dato che il risultato è negativo, si attiva l'indicatore di segno.



## 63.5 Scorrimenti, rotazioni, operazioni logiche

Le operazioni più semplici che si possono compiere con un microprocessore sono quelle che riguardano la logica booleana e lo scorri-

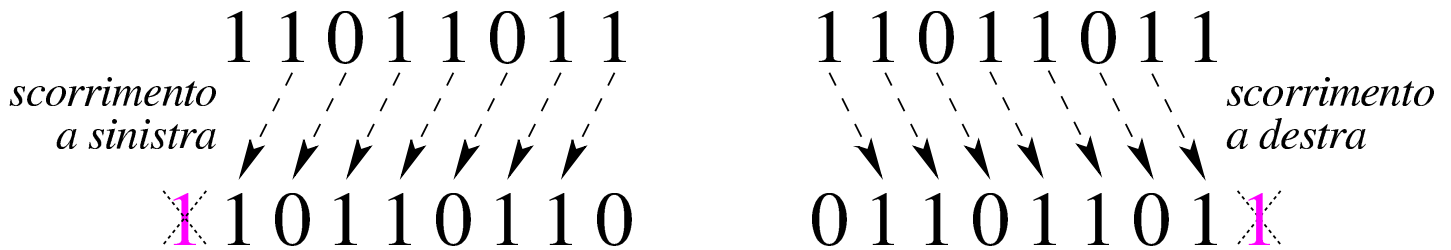
mento dei bit. Proprio per la loro semplicità è importante conoscere alcune applicazioni interessanti di questi procedimenti elaborativi.

### 63.5.1 Scorrimento logico

«

Lo scorrimento «logico» consiste nel fare scalare le cifre di un numero binario, verso sinistra (verso la parte più significativa) o verso destra (verso la parte meno significativa). Nell'eseguire questo scorrimento, da un lato si perde una cifra, mentre dall'altro si acquista uno zero.

Figura 63.53. Scorrimento logico a sinistra, perdendo le cifre più significative e scorrimento logico a destra, perdendo le cifre meno significative.



Lo scorrimento di una posizione verso sinistra corrisponde alla moltiplicazione del valore per due, mentre lo scorrimento a destra corrisponde a una divisione intera per due; scorrimenti di  $n$  posizioni rappresentano moltiplicazioni e divisioni per  $2^n$ . Le cifre che si perdono nello scorrimento a sinistra si possono considerare come il riporto della moltiplicazione, mentre le cifre che si perdono nello scorrimento a destra sono il resto della divisione.

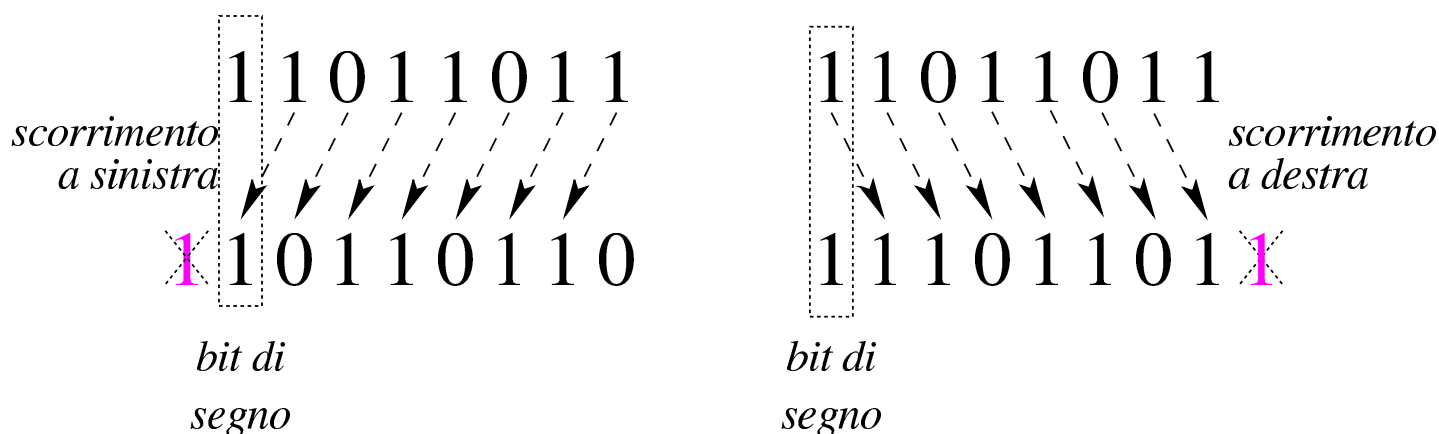
### 63.5.2 Scorrimento aritmetico

«

Il tipo di scorrimento descritto nella sezione precedente, se utilizzato per eseguire moltiplicazioni e divisioni, va bene solo per valori senza segno. Se si intende fare lo scorrimento di un valore con se-

gno, occorre distinguere due casi: lo scorrimento a sinistra è valido se il risultato non cambia di segno; lo scorrimento a destra implica il mantenimento del bit che rappresenta il segno e l'aggiunta di cifre uguali a quella che rappresenta il segno stesso.

Figura 63.54. Scorrimento aritmetico a sinistra e a destra, di un valore negativo.

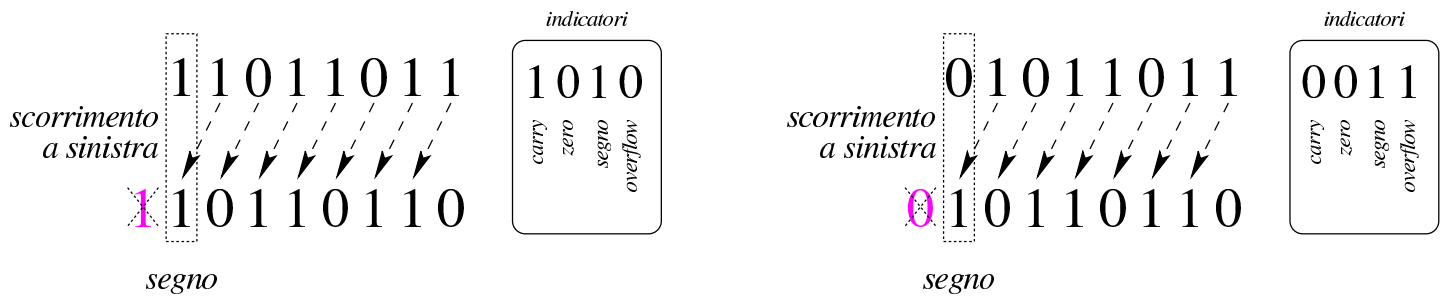


### 63.5.3 Scorrimento e indicatori

Tenendo conto che gli scorrimenti si eseguono sempre per una sola posizione alla volta e che rappresentano una moltiplicazione o una divisione per due, tornano utili gli stessi indicatori descritti a proposito di somme e sottrazioni. Come già accennato, il riporto viene usato per segnalare la cifra che viene perduta (sia per lo scorrimento verso sinistra, sia per quello verso destra), mentre l'indicatore di traboccamento (*overflow*) serve a segnalare che il risultato cambia di segno.

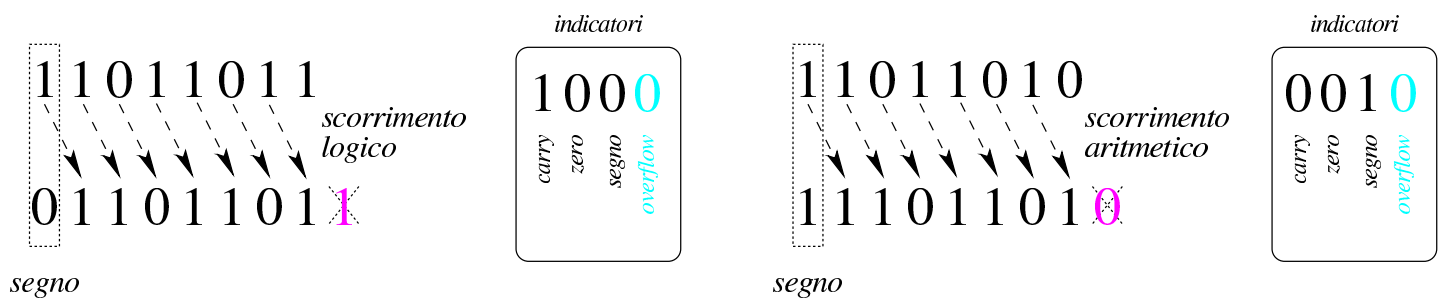
Lo scorrimento aritmetico verso sinistra avviene nello stesso modo di quello «logico». Nel caso il valore che viene fatto scorrere sia considerato privo di segno, il risultato della moltiplicazione per due è valido se non si presenta un riporto; se invece il valore ha un segno, il risultato è «corretto» se il segno non è cambiato.

Figura 63.55. Scorrimento a sinistra. Nel lato sinistro si vede che il risultato non è valido se si tratta di un valore senza segno, in quanto si presenta un riporto, mentre sarebbe valido se fosse un valore con segno, perché questo non cambia. A destra, invece, si vede un valore che se è da intendere senza segno, dà un risultato corretto, mentre se ha il segno, il risultato non è più valido perché il segno si inverte (*overflow*).



Lo scorrimento verso destra avviene in modo diverso se il valore va inteso con segno o senza segno, perché se si presta attenzione al segno si usa lo scorrimento aritmetico che inserisce a sinistra cifre uguali al segno precedente. Pertanto, nello scorrimento a destra si considera solo il resto, che finisce in pratica nello stesso indicatore del riporto.

Figura 63.56. Nel lato sinistro si vede uno scorrimento «logico» che produce un resto, mentre in quello destro si vede uno scorrimento aritmetico che, in questo caso, non produce alcun resto.



## 63.5.4 Moltiplicazione

La moltiplicazione si ottiene attraverso diverse fasi di scorrimento e somma di un valore, dove però il risultato richiede un numero doppio di cifre rispetto a quelle usate per il moltiplicando e il moltiplicatore. Il procedimento di moltiplicazione deve avvenire sempre con valori senza segno. Se i valori si intendono con segno, quando sono negativi occorre farne prima il complemento a due, in modo da portarli a valori positivi, quindi occorre decidere se il risultato va preso così come viene o se va invertito a sua volta con il complemento a due: se i valori moltiplicati hanno segno diverso tra loro, il risultato deve essere trasformato con il complemento a due per renderlo negativo, altrimenti il risultato è sempre positivo.

Figura 63.57. Moltiplicazione.

*moltiplicazione di valori senza segno*

$$\begin{array}{r}
 1011 \times \\
 1101 = \\
 \hline
 00001011 + \\
 00000000 + \\
 00101100 + \\
 01011000 = \\
 \hline
 10001111
 \end{array}$$

*moltiplicazione di valori con segno diverso*

$$\begin{array}{r}
 1011 \times \xrightarrow{\text{complemento a due}} 0101 \times \\
 0111 = \qquad \qquad \qquad 0111 = \\
 \hline
 11011101 \xleftarrow{\text{complemento a due}} \begin{array}{r}
 00000101 + \\
 00001010 + \\
 00010100 + \\
 00000000 = \\
 \hline
 00100011
 \end{array}
 \end{array}$$

## 63.5.5 Divisione

La divisione si ottiene attraverso diverse fasi di scorrimento di un valore, che di volta in volta viene sottratto al dividendo, ma solo se la sottrazione è possibile effettivamente. Il procedimento di divisione deve avvenire sempre con valori senza segno. Se i valori si intendono

con segno, quando sono negativi occorre farne prima il complemento a due, in modo da portarli a valori positivi, quindi occorre decidere se il risultato va preso così come viene o se va invertito a sua volta con il complemento a due: se dividendo e divisore hanno segni diversi tra loro, il risultato deve essere trasformato con il complemento a due per renderlo negativo, altrimenti il risultato è sempre positivo.

Figura 63.58. Divisione: i valori sono intesi senza segno.

$$11011101 \div 0110 = 00100100$$

$11000000$   
 $00011101$   
 $00000000$   
 $00011101$   
 $00000000$   
 $00011101$   
 $00011000$   
 $00000101$   
 $00000000$   
 $00000101$   
 $00000000$   
 $00000101$  *resto della divisione intera*

221 : 6 = 36  
con il resto di 5

### 63.5.6 Rotazione

«

La rotazione è uno scorrimento dove le cifre che si perdono da una parte rientrano dall'altra. Esistono due tipi di rotazione; uno «normale» e l'altro che include nella rotazione il bit del riporto. Dal momento che la rotazione non si presta per i calcoli matematici, di solito non viene considerato il segno.



Figura 63.59. Rotazione normale.

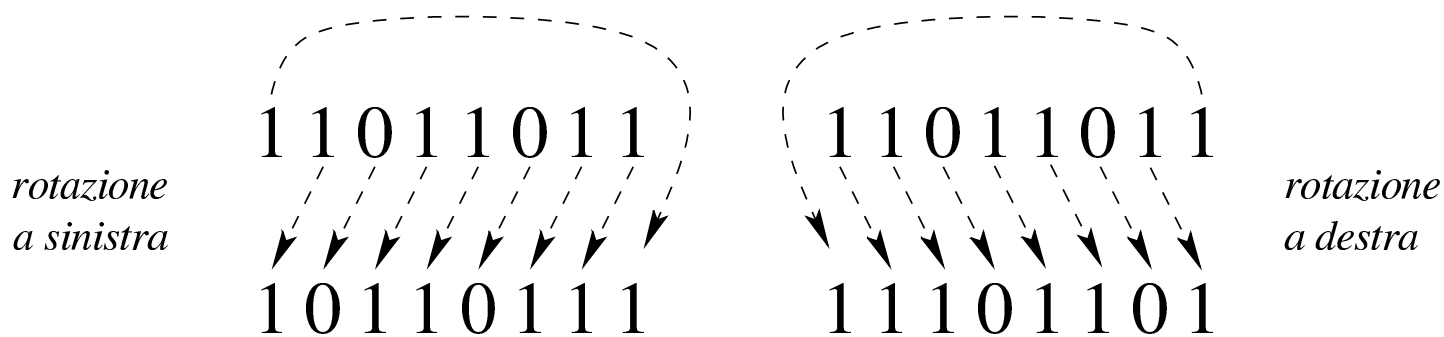
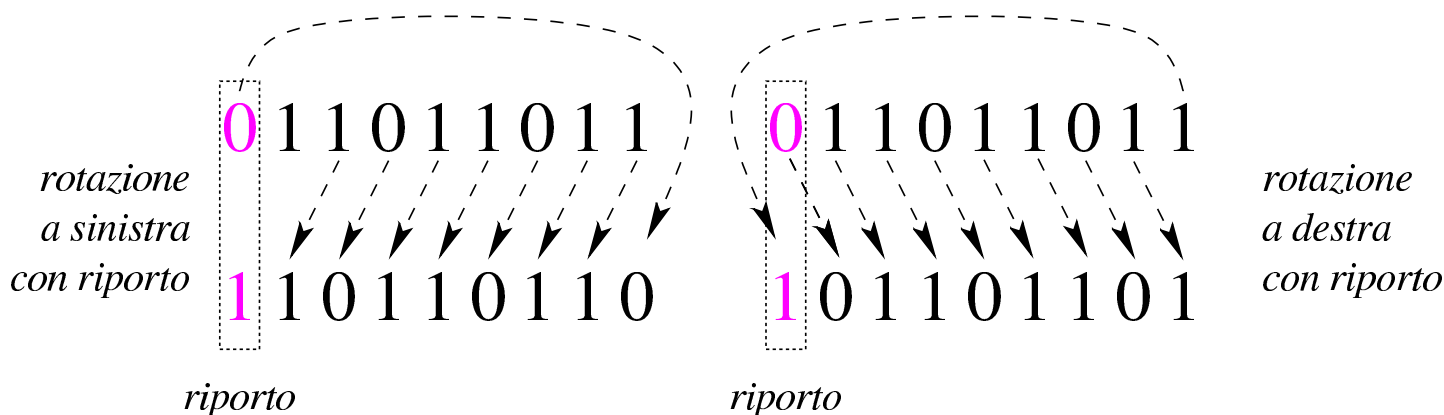


Figura 63.60. Rotazione con riporto.



### 63.5.7 Rotazione e indicatori

Le rotazioni non sono riconducibili a operazioni matematiche, ma si usa ugualmente l'indicatore del riporto per conservare la cifra persa; inoltre, l'indicatore di traboccamento può servire per annotare un'ipotesi di cambiamento di segno.



Figura 63.61. Rotazione normale. La cifra che fuoriesce da un lato e rientra dall'altro, rimane annotata nell'indicatore di riporto; nel caso dell'esempio di rotazione a destra, l'indicatore di traboccamento segnala che la cifra più significativa è diversa rispetto alla fase precedente.

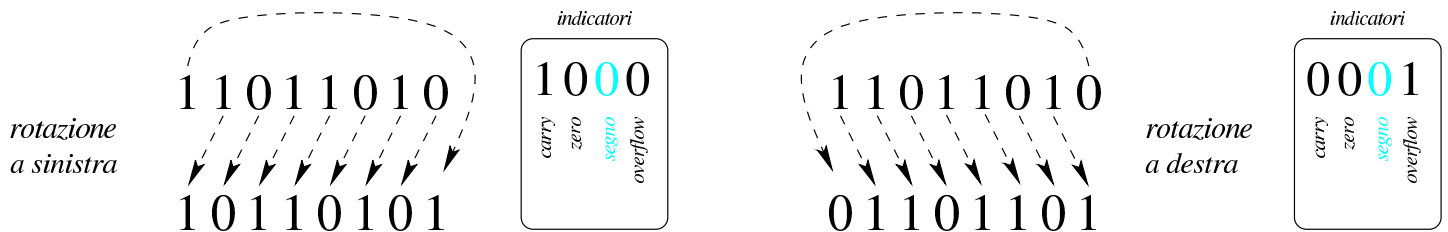
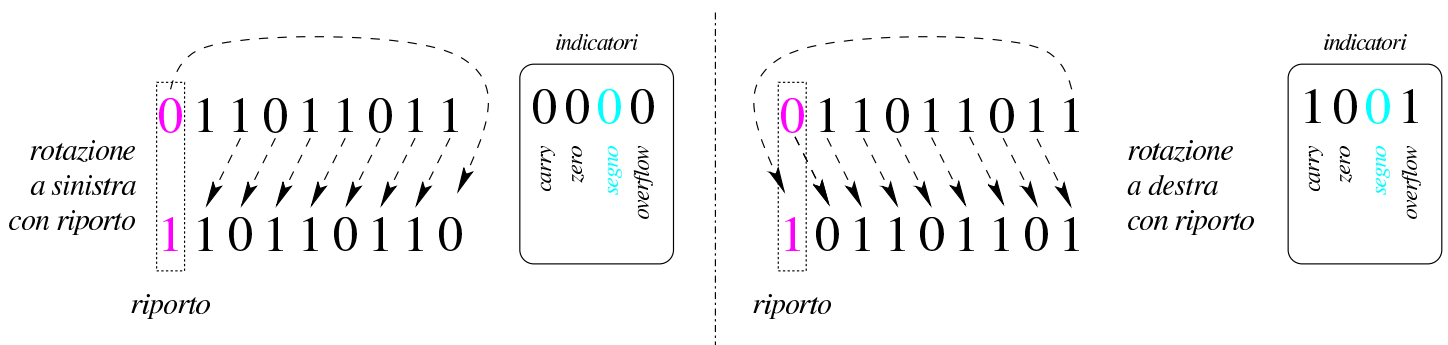


Figura 63.62. Rotazione con riporto. La cifra che fuoriesce da un lato entra nel riporto, mentre dall'altro lato entra la cifra conservata nel riporto precedente; nel caso dell'esempio di rotazione a destra, l'indicatore di traboccamento segnala che la cifra più significativa è diversa rispetto alla fase precedente.



### 63.5.8 Operatori logici



Gli operatori logici si possono applicare anche a valori composti da più cifre binarie.

Figura 63.63. AND e OR.

$$\begin{array}{r}
 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ a \\
 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ b \\
 \hline
 0\ 1\ 0\ 0\ 1\ 0\ 0\ 1\ a\ AND\ b
 \end{array}
 \qquad
 \begin{array}{r}
 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ a \\
 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ b \\
 \hline
 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ a\ OR\ b
 \end{array}$$

Figura 63.64. XOR e NOT.

$$\begin{array}{r}
 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ a \\
 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ b \\
 \hline
 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ a\ XOR\ b
 \end{array}
 \qquad
 \begin{array}{r}
 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ a \\
 \hline
 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ NOT\ a
 \end{array}$$

È importante osservare che l'operatore NOT esegue in pratica il complemento a uno di un valore.

Capita spesso di trovare in un sorgente scritto in un linguaggio assembler un'istruzione che assegna a un registro il risultato dell'operatore XOR su se stesso. Ciò si fa, evidentemente, per azzerarne il contenuto, quando, probabilmente, l'assegnamento esplicito di un valore a un registro richiede una frazione di tempo maggiore per la sua esecuzione.

Figura 63.65. XOR per azzerare i valori.

$$\begin{array}{r}
 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ a \\
 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ a \\
 \hline
 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ a\ XOR\ a
 \end{array}$$

## 63.5.9 Intervenire su bit singoli

«

Quando si lavora con valori binari composti da una quantità prestabilita di cifre, per intervenire singolarmente o comunque solo parzialmente sulle stesse occorre predisporre delle maschere da abbinare poi con un operatore logico appropriato. Segue la descrizione di alcuni esempi.

- Si vuole attivare il quarto bit (contando a partire dalla cifra meno significativa) nella variabile  $x$ .

$$x := x \text{ OR } 8_{10}$$

$$1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ x$$

$$1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ x$$

$$\underline{0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 8_{10} \text{ (maschera)}}$$

$$\underline{0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 8_{10} \text{ (maschera)}}$$

$$1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ x \text{ OR } 8_{10}$$

$$1\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ x \text{ OR } 8_{10}$$

- Si vuole disattivare il quarto bit (contando a partire dalla cifra meno significativa) nella variabile  $x$ .

$$x := x \text{ AND (NOT } 8_{10})$$

$$\underline{0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 8_{10}}$$

$$1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ x$$

$$1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ \text{NOT } 8_{10}$$

$$1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ \text{NOT } 8_{10} \text{ (maschera)}$$

$$\underline{1\ 1\ 0\ 1\ 0\ 0\ 1\ 1\ x \text{ AND (NOT } 8_{10})}$$

- Si vuole invertire il quarto bit (contando a partire dalla cifra meno significativa) nella variabile  $x$ .

$$x := x \text{ XOR } 8_{10}$$

$$1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ x$$

$$1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ x$$

$$\underline{0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 8_{10} \text{ (maschera)}}$$

$$\underline{0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 8_{10} \text{ (maschera)}}$$

$$1\ 1\ 0\ 1\ 0\ 0\ 1\ 1\ x \text{ XOR } 8_{10}$$

$$1\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ x \text{ XOR } 8_{10}$$

- Si vuole dividere un valore per otto (che è una potenza di due, ovvero  $2^3$ ), calcolando il quoziente intero e il resto. Per farlo occorre far scorrere il valore verso destra, di tre posizioni. Tenendo conto che le cifre che vengono espulse sono quelle che rappresentano il resto, questo lo si può ottenere con una maschera pari a sette ( $2^3 - 1$ ), abbinata con l'operatore AND.

*divisione per otto, ottenuta con lo scorrimento  
a destra di tre cifre*

$$\begin{array}{r}
 11011011 \div \\
 00001000 = \\
 \hline
 00011011 \quad 011 \\
 \text{divisione intera} \quad \text{resto}
 \end{array}$$

*calcolo del resto di una divisione per otto*

$$\begin{array}{r}
 11011011 \quad x \\
 00000111 \quad 8_{10} - 1 \\
 \hline
 00000011 \quad x \text{ AND } (8-1)
 \end{array}$$

### 63.5.10 Somme e sottrazioni abbinata agli operatori logici

Esiste una proprietà interessante della sottrazione, quando viene abbinata all'operatore logico AND. Come si vede nella figura successiva, quando si riduce un valore di una unità, quella che prima era la cifra a uno meno significativa passa a zero, mentre le cifre precedenti passano a uno. Così facendo, se si abbinano i due valori (quello originale e quello ridotto di una unità) con l'operatore AND, si ottiene un nuovo valore in cui, semplicemente, la cifra meno significativa a uno passa a zero:

$$\begin{array}{r}
 x = 01101100 \\
 x-1 = 01101011 \\
 x \text{ AND } (x-1) = 01101000
 \end{array}$$

Per converso, se si incrementa di una unità un valore e poi si abbinata l'operatore logico OR (tra il valore originale e quello incrementato

di una unità), si ottiene di portare a uno la cifra meno significativa che prima era a zero:

$$\begin{array}{r}
 x = 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1 \\
 x+1 = 0\ 1\ 1\ 0\ 1\ 1\ 0\ 0 \\
 x\ OR\ (x+1) = 0\ 1\ 1\ 0\ 1\ 1\ 1\ 1
 \end{array}$$

## 63.6 Confronti attraverso la sottrazione

«

Il confronto tra due valori avviene provando a sottrarne uno dall'altro. In un microprocessore, l'esito di una sottrazione, come mostrato dagli indicatori comuni, consente di confrontare i valori originali.

### 63.6.1 Confronto di valori senza segno

«

Se si esegue una sottrazione e si attiva l'indicatore del risultato zero, senza la presenza di una richiesta del prestito di una cifra, i valori sono uguali; se il valore ottenuto è diverso da zero e non c'è alcuna richiesta di prestito, vuol dire che il sottraendo ha un valore inferiore al minuendo; negli altri casi, il sottraendo ha un valore maggiore del minuendo.

Figura 63.72. Confronto di valori senza segno.

$  \begin{array}{r}  a\ 1\ 0\ 1\ 1 - \\  b\ 1\ 0\ 1\ 0 = \\  \hline  0\ 0\ 0\ 1 \\  a > b  \end{array}  $	<p style="text-align: center;"><i>indicatori</i></p> <div style="border: 1px solid black; border-radius: 10px; padding: 5px; display: inline-block;"> <span style="font-size: 1.2em; color: green;">0000</span>  <small>barrow</small>  <small>zero</small>  <small style="color: green;">segno</small>  <small style="color: green;">overflow</small> </div>	$  \begin{array}{r}  a\ 1\ 0\ 1\ 1 - \\  b\ 1\ 0\ 1\ 1 = \\  \hline  0\ 0\ 0\ 0 \\  a = b  \end{array}  $	<p style="text-align: center;"><i>indicatori</i></p> <div style="border: 1px solid black; border-radius: 10px; padding: 5px; display: inline-block;"> <span style="font-size: 1.2em; color: green;">0100</span>  <small>barrow</small>  <small>zero</small>  <small style="color: green;">segno</small>  <small style="color: green;">overflow</small> </div>	$  \begin{array}{r}  a\ 1\ 0\ 1\ 1 - \\  b\ 1\ 1\ 0\ 0 = \\  \hline  1\ 1\ 1\ 1 \\  a < b  \end{array}  $	<p style="text-align: center;"><i>indicatori</i></p> <div style="border: 1px solid black; border-radius: 10px; padding: 5px; display: inline-block;"> <span style="font-size: 1.2em; color: green;">1010</span>  <small>barrow</small>  <small>zero</small>  <small style="color: green;">segno</small>  <small style="color: green;">overflow</small> </div>
-----------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## 63.6.2 Confronto di valori con segno

Il confronto tra valori con segno avviene in modo meno intuitivo di quello che invece lo ignora. Qui non si considera l'indicatore del prestito di una cifra, mentre vanno considerati al suo posto gli indicatori di segno e di traboccamento, che possono essere uguali o meno tra di loro. Pertanto: se il risultato della sottrazione dà zero, i valori confrontati sono uguali; se il risultato della sottrazione è diverso da zero, se gli indicatori di segno e di traboccamento sono uguali, vuol dire che il sottraendo è inferiore del minuendo; diversamente il sottraendo è superiore al minuendo.

Figura 63.73. Confronto di valori con segno.

$\begin{array}{r} a \ 0111 - \\ b \ 1000 = \\ \hline 1111 \\ a > b \end{array}$	<div style="text-align: center; margin-bottom: 5px;"><i>indicatori</i></div> <div style="border: 1px solid black; padding: 5px; display: inline-block;"> <math>0011</math>  <i>borrow</i>    <i>zero</i>    <i>segno</i>    <i>overflow</i> </div>	$\begin{array}{r} a \ 1011 - \\ b \ 1011 = \\ \hline 0000 \\ a = b \end{array}$	<div style="text-align: center; margin-bottom: 5px;"><i>indicatori</i></div> <div style="border: 1px solid black; padding: 5px; display: inline-block;"> <math>0100</math>  <i>borrow</i>    <i>zero</i>    <i>segno</i>    <i>overflow</i> </div>	$\begin{array}{r} a \ 1011 - \\ b \ 0100 = \\ \hline 0111 \\ a < b \end{array}$	<div style="text-align: center; margin-bottom: 5px;"><i>indicatori</i></div> <div style="border: 1px solid black; padding: 5px; display: inline-block;"> <math>0001</math>  <i>borrow</i>    <i>zero</i>    <i>segno</i>    <i>overflow</i> </div>
---------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Dal momento che il meccanismo del confronto di valori con segno può essere difficile da comprendere con pochi esempi, si aggiunge un prospetto con i confronti fra tutti i valori che si possono rappresentare con due soli bit, sia senza segno, sia con segno. Nel prospetto viene mostrata la sottrazione e l'addizione dopo l'inversione del sottraendo, inoltre sono annotati tutti i riporti e i prestiti parziali.

Figura 63.74. Verifica di tutti i casi di confronto per valori a due bit.

	minuendo	sottraendo	risultato	confronto	sottrazione con prestito di cifre	somma del sottraendo dopo la trasformazione con il complemento a uno e l'aggiunta di una unità	borrow	zero	segno	overflow
con segno	1	-	-2 = 3	1 > -2	$\begin{array}{r} 1\ 0 \\ 0\ 1\ - \\ \underline{1\ 1} \\ 1\ 1 \end{array}$	$\begin{array}{r} 0\ 1\ 1 \\ 0\ 1\ + \\ \underline{0\ 1} \\ 1\ 1 \end{array}$	1	0	1	1
senza segno	1	-	2 = -1	1 < 2	$\begin{array}{r} 1\ 0 \\ 0\ 1\ - \\ \underline{1\ 1} \\ 1\ 1 \end{array}$	$\begin{array}{r} 0\ 1\ 1 \\ 0\ 1\ + \\ \underline{0\ 1} \\ 1\ 1 \end{array}$	1	0	1	1
con segno	1	-	-1 = 2	1 > -1	$\begin{array}{r} 1\ 0 \\ 0\ 1\ - \\ \underline{1\ 1} \\ 1\ 0 \end{array}$	$\begin{array}{r} 0\ 1\ 1 \\ 0\ 1\ + \\ \underline{0\ 0} \\ 1\ 0 \end{array}$	1	0	1	1
senza segno	1	-	3 = -2	1 < 3	$\begin{array}{r} 1\ 0 \\ 0\ 1\ - \\ \underline{1\ 1} \\ 1\ 0 \end{array}$	$\begin{array}{r} 0\ 1\ 1 \\ 0\ 1\ + \\ \underline{0\ 0} \\ 1\ 0 \end{array}$	1	0	1	1
con segno	1	-	0 = 1	1 > 0	$\begin{array}{r} 0\ 0 \\ 0\ 1\ - \\ \underline{0\ 0} \\ 0\ 1 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 0\ 1\ + \\ \underline{1\ 1} \\ 0\ 1 \end{array}$	0	0	0	0
senza segno	1	-	0 = 1	1 > 0	$\begin{array}{r} 0\ 0 \\ 0\ 1\ - \\ \underline{0\ 0} \\ 0\ 1 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 0\ 1\ + \\ \underline{1\ 1} \\ 0\ 1 \end{array}$	0	0	0	0
con segno	1	-	1 = 0	1 = 1	$\begin{array}{r} 0\ 0 \\ 0\ 1\ - \\ \underline{0\ 1} \\ 0\ 0 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 0\ 1\ + \\ \underline{1\ 0} \\ 0\ 0 \end{array}$	0	1	0	0
senza segno	1	-	1 = 0	1 = 1	$\begin{array}{r} 0\ 0 \\ 0\ 1\ - \\ \underline{0\ 1} \\ 0\ 0 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 0\ 1\ + \\ \underline{1\ 0} \\ 0\ 0 \end{array}$	0	1	0	0
con segno	0	-	-2 = 2	0 > -2	$\begin{array}{r} 1\ 0 \\ 0\ 0\ - \\ \underline{1\ 0} \\ 1\ 0 \end{array}$	$\begin{array}{r} 0\ 1\ 1 \\ 0\ 0\ + \\ \underline{0\ 1} \\ 1\ 0 \end{array}$	1	0	1	1
senza segno	0	-	2 = -2	0 < 2	$\begin{array}{r} 1\ 0 \\ 0\ 0\ - \\ \underline{1\ 0} \\ 1\ 0 \end{array}$	$\begin{array}{r} 0\ 1\ 1 \\ 0\ 0\ + \\ \underline{0\ 1} \\ 1\ 0 \end{array}$	1	0	1	1
con segno	0	-	-1 = 1	0 > -1	$\begin{array}{r} 1\ 1 \\ 0\ 0\ - \\ \underline{1\ 1} \\ 0\ 1 \end{array}$	$\begin{array}{r} 0\ 0\ 1 \\ 0\ 0\ + \\ \underline{0\ 0} \\ 0\ 1 \end{array}$	1	0	0	0
senza segno	0	-	3 = -3	0 < 3	$\begin{array}{r} 1\ 1 \\ 0\ 0\ - \\ \underline{1\ 1} \\ 0\ 1 \end{array}$	$\begin{array}{r} 0\ 0\ 1 \\ 0\ 0\ + \\ \underline{0\ 0} \\ 0\ 1 \end{array}$	1	0	0	0
con segno	0	-	0 = 0	0 = 0	$\begin{array}{r} 0\ 0 \\ 0\ 0\ - \\ \underline{0\ 0} \\ 0\ 0 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 0\ 0\ + \\ \underline{1\ 1} \\ 0\ 0 \end{array}$	0	1	0	0
senza segno	0	-	0 = 0	0 = 0	$\begin{array}{r} 0\ 0 \\ 0\ 0\ - \\ \underline{0\ 0} \\ 0\ 0 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 0\ 0\ + \\ \underline{1\ 1} \\ 0\ 0 \end{array}$	0	1	0	0
con segno	0	-	1 = -1	0 < 1	$\begin{array}{r} 1\ 1 \\ 0\ 0\ - \\ \underline{0\ 1} \\ 1\ 1 \end{array}$	$\begin{array}{r} 0\ 0\ 1 \\ 0\ 0\ + \\ \underline{1\ 0} \\ 1\ 1 \end{array}$	1	0	1	0
senza segno	0	-	1 = -1	0 < 1	$\begin{array}{r} 1\ 1 \\ 0\ 0\ - \\ \underline{0\ 1} \\ 1\ 1 \end{array}$	$\begin{array}{r} 0\ 0\ 1 \\ 0\ 0\ + \\ \underline{1\ 0} \\ 1\ 1 \end{array}$	1	0	1	0
con segno	-1	-	-2 = 1	-1 > -2	$\begin{array}{r} 0\ 0 \\ 1\ 1\ - \\ \underline{1\ 0} \\ 0\ 1 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 1\ 1\ + \\ \underline{0\ 1} \\ 0\ 1 \end{array}$	0	0	0	0
senza segno	3	-	2 = 1	3 > 2	$\begin{array}{r} 0\ 0 \\ 1\ 1\ - \\ \underline{1\ 0} \\ 0\ 1 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 1\ 1\ + \\ \underline{0\ 1} \\ 0\ 1 \end{array}$	0	0	0	0
con segno	-1	-	-1 = 0	-1 = -1	$\begin{array}{r} 0\ 0 \\ 1\ 1\ - \\ \underline{1\ 1} \\ 0\ 0 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 1\ 1\ + \\ \underline{0\ 0} \\ 0\ 0 \end{array}$	0	1	0	0
senza segno	3	-	3 = 0	3 = 3	$\begin{array}{r} 0\ 0 \\ 1\ 1\ - \\ \underline{1\ 1} \\ 0\ 0 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 1\ 1\ + \\ \underline{0\ 0} \\ 0\ 0 \end{array}$	0	1	0	0
con segno	-1	-	0 = -1	-1 < 0	$\begin{array}{r} 0\ 0 \\ 1\ 1\ - \\ \underline{0\ 0} \\ 1\ 1 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 1\ 1\ + \\ \underline{1\ 1} \\ 1\ 1 \end{array}$	0	0	1	0
senza segno	3	-	0 = 3	3 > 0	$\begin{array}{r} 0\ 0 \\ 1\ 1\ - \\ \underline{0\ 0} \\ 1\ 1 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 1\ 1\ + \\ \underline{1\ 1} \\ 1\ 1 \end{array}$	0	0	1	0
con segno	-1	-	1 = -2	-1 < 1	$\begin{array}{r} 0\ 0 \\ 1\ 1\ - \\ \underline{0\ 1} \\ 1\ 0 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 1\ 1\ + \\ \underline{1\ 0} \\ 1\ 0 \end{array}$	0	0	1	0
senza segno	3	-	1 = 2	3 > 1	$\begin{array}{r} 0\ 0 \\ 1\ 1\ - \\ \underline{0\ 1} \\ 1\ 0 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 1\ 1\ + \\ \underline{1\ 0} \\ 1\ 0 \end{array}$	0	0	1	0
con segno	-2	-	-2 = 0	-2 = -2	$\begin{array}{r} 0\ 0 \\ 1\ 0\ - \\ \underline{1\ 0} \\ 0\ 0 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 1\ 0\ + \\ \underline{0\ 1} \\ 0\ 0 \end{array}$	0	1	0	0
senza segno	2	-	2 = 0	2 = 2	$\begin{array}{r} 0\ 0 \\ 1\ 0\ - \\ \underline{1\ 0} \\ 0\ 0 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 1\ 0\ + \\ \underline{0\ 1} \\ 0\ 0 \end{array}$	0	1	0	0
con segno	-2	-	-1 = -1	-2 < -1	$\begin{array}{r} 1\ 1 \\ 1\ 0\ - \\ \underline{1\ 1} \\ 1\ 1 \end{array}$	$\begin{array}{r} 0\ 0\ 1 \\ 1\ 0\ + \\ \underline{0\ 0} \\ 1\ 1 \end{array}$	1	0	1	0
senza segno	2	-	3 = -1	2 < 3	$\begin{array}{r} 1\ 1 \\ 1\ 0\ - \\ \underline{1\ 1} \\ 1\ 1 \end{array}$	$\begin{array}{r} 0\ 0\ 1 \\ 1\ 0\ + \\ \underline{0\ 0} \\ 1\ 1 \end{array}$	1	0	1	0
con segno	-2	-	0 = -2	-2 < 0	$\begin{array}{r} 0\ 0 \\ 1\ 0\ - \\ \underline{0\ 0} \\ 1\ 0 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 1\ 0\ + \\ \underline{1\ 1} \\ 1\ 0 \end{array}$	0	0	1	0
senza segno	2	-	0 = 2	2 > 0	$\begin{array}{r} 0\ 0 \\ 1\ 0\ - \\ \underline{0\ 0} \\ 1\ 0 \end{array}$	$\begin{array}{r} 1\ 1\ 1 \\ 1\ 0\ + \\ \underline{1\ 1} \\ 1\ 0 \end{array}$	0	0	1	0
con segno	-2	-	1 = -3	-2 < 1	$\begin{array}{r} 0\ 1 \\ 1\ 0\ - \\ \underline{0\ 1} \\ 0\ 1 \end{array}$	$\begin{array}{r} 1\ 0\ 1 \\ 1\ 0\ + \\ \underline{1\ 0} \\ 0\ 1 \end{array}$	0	0	0	1
senza segno	2	-	1 = 1	2 > 1	$\begin{array}{r} 0\ 1 \\ 1\ 0\ - \\ \underline{0\ 1} \\ 0\ 1 \end{array}$	$\begin{array}{r} 1\ 0\ 1 \\ 1\ 0\ + \\ \underline{1\ 0} \\ 0\ 1 \end{array}$	0	0	0	1



## 63.7 Riferimenti

- Jonathan Bartlett, *Programming from the ground up*, 2003, <http://savannah.nongnu.org/projects/pgubook/>
- Paul A. Carter, *PC Assembly Language*, 2006, <http://www.drpaulcarter.com/pcasm/>
- Wikipedia, *Addressing mode*, [http://en.wikipedia.org/wiki/Addressing\\_mode](http://en.wikipedia.org/wiki/Addressing_mode)
- Mario Italiani, Giuseppe Serazzi, *Elementi di informatica*, ETAS libri, 1973, ISBN 8845303632
- Sandro Petrizzelli, *Appunti di elettronica digitale*, [http://users.libero.it/sandry/Digitale\\_01.pdf](http://users.libero.it/sandry/Digitale_01.pdf)
- Tony R. Kuphaldt, *Lessons In Electric Circuits*, <http://www.faqs.org/docs/electric/>
- Wikipedia, *Sistema numerico binario* [http://it.wikipedia.org/wiki/Sistema\\_numerico\\_binario](http://it.wikipedia.org/wiki/Sistema_numerico_binario)
- Wikipedia, *IEEE 754*, [http://it.wikipedia.org/wiki/IEEE\\_754](http://it.wikipedia.org/wiki/IEEE_754)

<sup>1</sup> Questa sezione riprende e in parte ripete, per maggiore chiarezza, un concetto già presentato nella sezione [63.1.8](#).

<sup>2</sup> Nel contesto riferito alla definizione di un numero in virgola mobile, si possono usare indifferentemente i termini *mantissa* o *significante*, così come sono indifferenti i termini *caratteristica* o *esponente*.

<sup>3</sup> Si osservi che lo standard IEEE 754 utilizza una «mantissa normalizzata» che indica la frazione di valore tra uno e due: «1,*mantissa*».

<sup>4</sup> Considerato che si tratta di un numero da esprimere in base due, il valore che viene moltiplicato per la potenza di due è un numero che va da uno a due.

<sup>5</sup> Per completezza, questa sezione riprende un concetto già descritto nella sezione [63.1.6](#).

# Microprocessori x86-32



64.1	Terminologia impropria .....	183
64.2	Registri principali fino ai 32 bit 183	
64.3	Sintesi delle istruzioni principali .....	186
64.4	Primo approccio al linguaggio assembler per x86 .	213
64.4.1	Il primo programma .....	213
64.4.2	Utilizzo di GDB .....	217
64.4.3	Modalità «TUI» .....	226
64.4.4	Utilizzo di DDD .....	231
64.4.5	Alcune istruzioni comuni .....	233
64.4.6	Dimensione dei dati nelle istruzioni .....	236
64.4.7	Direttive per il compilatore .....	238
64.4.8	Sezioni del sorgente .....	241
64.4.9	Usare GNU AS con la notazione Intel .....	242
64.5	Esempi con le «quattro operazioni» .....	244
64.5.1	Somma .....	248
64.5.2	Sottrazione .....	259
64.5.3	Moltiplicazione senza segno .....	266
64.5.4	Moltiplicazione con segno .....	269
64.5.5	Divisione .....	272
64.6	Esempi con gli «spostamenti» .....	275
64.6.1	Scorrimento logico .....	277

64.6.2	Scorrimento aritmetico .....	281
64.6.3	Rotazione .....	284
64.6.4	Rotazione con riporto .....	287
64.7	Esempi con i confronti .....	290
64.8	Le istruzioni di salto .....	298
64.8.1	Portata del salto .....	298
64.8.2	Salto incondizionato .....	299
64.8.3	Salto condizionato dallo stato di un indicatore 300	
64.8.4	Salto condizionato da un confronto 301	
64.8.5	Cicli .....	304
64.9	Esempi di programmi con strutture di controllo .....	305
64.9.1	Somma attraverso l'incremento unitario ...	306
64.9.2	Moltiplicazione attraverso la somma .....	309
64.9.3	Divisione attraverso la sottrazione .....	312
64.9.4	Elevamento a potenza .....	314
64.9.5	Moltiplicazione attraverso lo scorrimento e la somma 316	
64.9.6	Conteggio dei bit a uno .....	319
64.10	Funzioni .....	322
64.10.1	Esempio banale di chiamata .....	322
64.10.2	Salvataggio dei registri prima della chiamata ..	326
64.10.3	Passaggio di parametri attraverso la pila .....	327

64.10.4	Utilizzo del registro «EBP» .....	334
64.10.5	Allocazione dello spazio per le variabili locali e preservazione dei registri .....	341
64.10.6	Convenzioni di chiamata .....	348
64.10.7	Nota sugli array «locali» .....	349
64.11	Esempi di funzioni ricorsive .....	350
64.11.1	Elevamento a potenza .....	350
64.11.2	Fattoriale .....	354
64.12	Indirizzamento dei dati .....	358
64.12.1	Gestione di array .....	360
64.12.2	Istruzione «LEA» .....	366
64.13	Rappresentazione dei dati in memoria attraverso un esempio .....	368
64.14	Esempi con gli array .....	375
64.14.1	Ricerca sequenziale .....	375
64.14.2	Ricerca binaria .....	380
64.14.3	Bubblesort .....	386
64.15	Calcoli con gli indirizzi in fase di compilazione .....	391
64.15.1	Distanza tra due indirizzi .....	391
64.15.2	Riempimento di spazio inutilizzato .....	393
64.16	Interazione con il sistema operativo .....	395
64.16.1	Parametri di chiamata del programma .....	395
64.16.2	Funzioni del sistema operativo .....	398

64.16.3	Esempi di lettura e scrittura con i flussi standard	400									
64.17	Riferimenti	405									
.ascii	239	.bss	241	.byte	239	.data	241	.equ	238	.int	239
.lcomm	239	.text	241	ADC	189 255	ADD	189 235 248	AH	183	AL	183
AND	193	AX	183	BH	183	BL	183	BP	183	BSWAP	187
BX	183	CALL	196 322 350	CBW	187	CDQ	187	CH	183	CL	183
CLC	200	CMC	200	CMP	200 290 301 306	CWDE	187	CX	183	db	239
dd	239	DEC	189 236 306	DH	183	DI	183	DIV	189 272	DL	183
DX	183	EAX	183	EBP	183 334	EBX	183	ECX	183	EDI	183
EDX	183	EFLAGS	183	EIP	183	ENTER	341 350	equ	238	ESI	183
ESP	183	FLAGS	183	IDIV	189 272	IMUL	189 269	INC	189 236 306	INT	196 213
IP	183	JA	201 301	JAE	201 301	JB	201 301 312	JBE	201 301	JC	201 300
JCXZ	201	JE	201 301 306	JG	201 301	JGE	201 301	JL	201 301	JLE	201 301
JMP	201 299 309	JNA	201 301	JNAE	201 301	JNB	201 301	JNBE	201 301	JNC	201 300 322
JNE	201 301	JNG	201 301	JNGE	201 301	JNL	201 301	JNLE	301	JNO	201 300
JNP	201 300	JNS	201 300	JNZ	201 300 306	JO	201 300	JP	201 300	JS	201 300
JZ	201 300 314	LEA	187 366	LEAVE	341 350	LOOP	211 304 306	LOOPE	211 304	LOOPNE	211 304
LOOPNZ	211 304	LOOPZ	211 304	MOV	187 213 234	MOVSX	187	MOVZX	187 322	MUL	189 266
NEG	189 261	NOP	187	NOT	193	OR	193	POP	196 326	POPA	196 341 350
POPAD	196	POPF	196	PUSH	196 326	PUSHA	196 341 350	PUSHF	196	RCL	193 287
RCR	193 287	resb	239	resd	239	resw	239	RET	196 322 350	ROL	193 284
ROR	193 284	SAL	193 281	SAR	193 281	SBB	189 263	SETA	205	SETAE	205
SETB	205	SETBE	205	SETC	205	SETE	205	SETG	205	SETGE	205
SETL	205	SETLE	205	SETNA	205	SETNAE	205	SETNB	205	SETNBE	205
SETNC	205	SETNE	205								

SETNG 205 SETNGE 205 SETNL 205 SETNLE 205 SETNO 205  
 SETNS 205 SETNZ 205 SETO 205 SETS 205 SETZ 205 SHL 193  
 277 316 SHR 193 277 316 SI 183 SP 183 SUB 189 235 259 TEST  
 200 XCHG 187 XOR 193

## 64.1 Terminologia impropria

È bene ricordare che nella documentazione standard sui microprocessori x86 si usa una terminologia coerente, ma impropria, riferita alla dimensione delle unità di dati. In particolare, il problema nasce dal fatto che originariamente questi microprocessori avevano parole da 16 bit, così è stato associato il termine *word* a 16 bit ed è rimasta tale l'associazione anche con la trasformazione successiva a 32 bit. Pertanto, generalmente valgono le convenzioni riportate nella tabella successiva.

Tabella 64.1. Terminologia associata alla dimensione dei dati nei microprocessori x86.

Definizione o abbreviazione	Dimensione dei dati	Definizione o abbreviazione	Dimensione dei dati
«b», byte	8 bit	«w», <i>word</i>	16 bit
«d», «dw», <i>dword</i> , <i>double word</i>	32 bit	«q», «qw», <i>qword</i> , <i>quad word</i>	64 bit

## 64.2 Registri principali fino ai 32 bit

I registri dei microprocessori x86 sono stati inizialmente da 16 bit; successivamente, quelli principali sono stati estesi a 32 bit. Alcuni

registri hanno una funzione ben precisa; gli altri sono utilizzabili per scopi generali, ma in pratica ognuno ha un compito preferenziale.

Tutti i registri che sono stati estesi da 16 bit; a 32 bit; hanno due nomi: uno riferito alla porzione dei 16 bit meno significativi, l'altro che riguarda il registro nel suo complesso. Inoltre, per quattro registri in particolare, è possibile individuare anche i due byte che compongono la parte meno significativa. Per esempio, il registro *EAX* ha una dimensione di 32 bit, di cui è possibile individuare i 16 bit meno significativi con il nome *AX*, ma in più, il nome *AL* individua il byte meno significativo di *AX* mentre *AH* ne individua quello più significativo.

Il registro *EIP* (o *IP* nei microprocessori a 16 bit) viene gestito automaticamente e serve a contenere l'indirizzo dell'istruzione successiva da eseguire. In effetti, la gestione manuale di tale registro non sarebbe conveniente, dal momento che la dimensione delle istruzioni in linguaggio macchina varia in base al tipo e agli operandi.

Il registro *ESP* (o *SP* nei microprocessori a 16 bit) individua l'ultimo elemento della pila dei dati e può essere gestito manualmente, sapendo che questo indice si deve spostare a gruppi di quattro byte (o due byte nella versione a 16 bit) e che la pila cresce diminuendo l'indice.

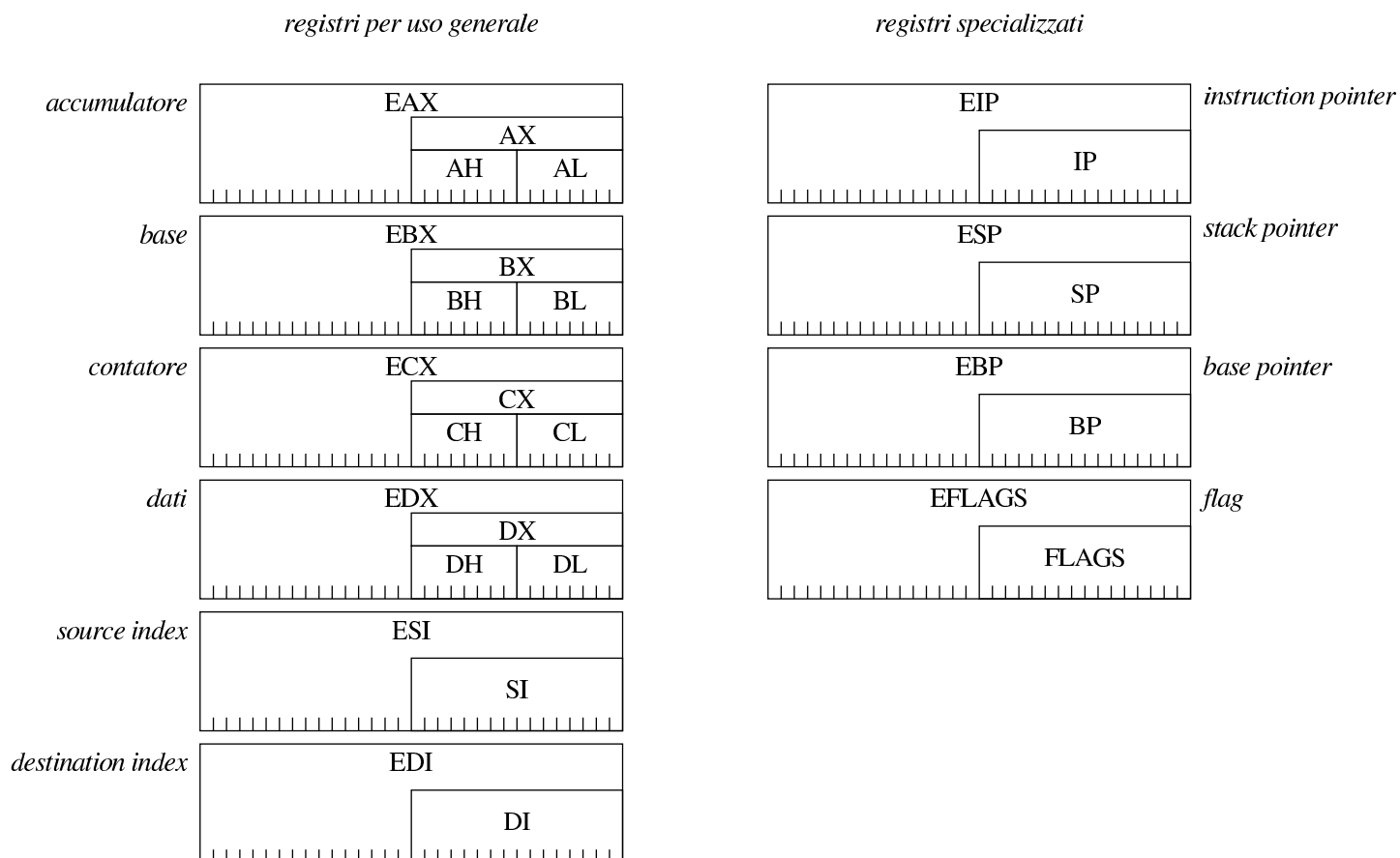
Il registro *EBP* (o *BP* nei microprocessori a 16 bit) si affianca all'indice della pila, per tenere conto della posizione raggiunta all'inizio di una funzione.

Il registro *EFLAGS* (o *FLAGS* nei microprocessori a 16 bit) raccoglie i vari indicatori che descrivono l'esito delle operazioni svolte. In particolare sono importanti gli indicatori che appaiono nella tabella



## 64.3.

Figura 64.2. Registri principali.

Tabella 64.3. Gli indicatori principali, contenuti nel registro **FLAGS**.

Indicatore ( <i>flag</i> )	Descrizione
<i>C carry</i>	È l'indicatore del riporto per le operazioni con valori senza segno. In particolare si attiva dopo una somma che genera un riporto e dopo una sottrazione che richiede il prestito di una cifra (in tal caso si chiama anche <i>borrow</i> ).
<i>O overflow</i>	È l'indicatore di traboccamento per le operazioni che riguardano valori con segno.
<i>Z zero</i>	Viene impostato dopo un'operazione che dà come risultato il valore zero.

Indicatore ( <i>flag</i> )	Descrizione
<i>S sign</i>	Riproduce il bit più significativo di un valore, dopo un'operazione. Se il valore è da intendersi con segno, l'indicatore serve a riprodurre il segno stesso.
<i>P parity</i>	Si attiva quando l'ultima operazione produce un risultato i cui otto bit meno significativi contengono una quantità pari di cifre a uno.

### 64.3 Sintesi delle istruzioni principali

«

Nelle tabelle successive vengono annotate le istruzioni più semplici che possono essere utilizzate con i microprocessori x86, raggruppate secondo il contesto a cui appartengono. In modo particolare sono assenti le istruzioni per i calcoli in virgola mobile e quelle per la gestione delle stringhe.

L'ordine in cui sono specificati gli operandi è quello «Intel», ovvero appare prima la destinazione e poi l'origine. Le sigle usate per definire i tipi di operandi sono: *reg* per «registro»; *mem* per «memoria»; *imm* per «immediato» (costante numerica).

Nella colonna degli indicatori appare: il simbolo «#» per annotare che l'indicatore relativo può essere modificato dall'istruzione; il simbolo «t» per annotare che lo stato precedente dell'indicatore viene considerato dall'istruzione; zero o uno se l'indicatore viene impostato in un certo modo; il simbolo «?» se l'effetto dell'istruzione sull'indicatore è indefinito.

Tabella 64.4. Assegnamenti, scambi, conversioni e istruzione nulla.

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
NOP		Istruzione nulla.	c p z s o . . . . .
MOV	<i>reg, reg</i> <i>reg, mem</i> <i>reg, imm</i> <i>mem, reg</i> <i>mem, imm</i>	Copia il valore dell'origine nella destinazione. Origine e destinazione devono avere la stessa quantità di bit. <i>dst := org</i>	c p z s o . . . . .
LEA	<i>reg32, mem</i>	<i>load effective address</i> Mette nel registro l'indirizzo della memoria. <i>dst := indirizzo(org)</i>	c p z s o . . . . .
MOVSX	<i>reg16, reg8</i> <i>reg16, mem8</i> <i>reg32, reg8</i> <i>reg32, mem8</i> <i>reg32, reg16</i> <i>reg32, mem16</i>	Tratta il valore nell'origine come un numero con segno e lo estende in modo da occupare tutto lo spazio della destinazione.	c p z s o . . . . .
MOVZX	<i>reg16, reg8</i> <i>reg16, mem8</i> <i>reg32, reg8</i> <i>reg32, mem8</i> <i>reg32, reg16</i> <i>reg32, mem16</i>	Tratta il valore nell'origine come un numero senza segno e lo estende in modo da occupare tutto lo spazio della destinazione.	c p z s o . . . . .

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
XCHG	<i>reg, reg</i> <i>reg, mem</i> <i>mem, reg</i>	Scambia i valori. <i>dst ::=: org</i>	c p z s o . . . . .
CBW		Converte un intero con segno, della dimensione di 8 bit, contenuto in <i>AL</i> , in modo da occupare tutto <i>AX</i> (da 8 bit a 16 bit). L'espansione tiene conto del segno. <i>AX := AL</i>	c p z s o . . . . .
CWDE		Converte un intero con segno, della dimensione di 16 bit, contenuto in <i>AX</i> , in modo da occupare <i>EAX</i> (da 16 bit a 32 bit). L'espansione tiene conto del segno. <i>EAX := AX</i>	c p z s o . . . . .
CDQ		Converte un intero con segno, della dimensione di 32 bit, contenuto in <i>EAX</i> , in modo da occupare la somma di <i>EDX:EAX</i> (da 32 bit a 64 bit). L'espansione tiene conto del segno. <i>EDX:EAX := EAX</i>	c p z s o . . . . .

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
BSWAP	<i>reg32</i>	Inverte l'ordine dei byte contenuti nel registro: quello meno significativo diventa il più significativo; la coppia interna si scambia.	c p z s o · · · · ·

Tabella 64.5. Operazioni aritmetiche.

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
NEG	<i>reg mem</i>	Inverte il segno di un numero, attraverso il complemento a due. <i>dst := -dst</i>	c p z s o # # # # #

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
ADD	<i>reg, reg</i> <i>reg, mem</i> <i>reg, imm</i> <i>mem, reg</i> <i>mem, imm</i>	Somma di interi, con o senza segno, ignorando il riporto precedente. Se i valori si intendono con segno, è importante l'esito dell'indicatore di traboccamento ( <i>overflow</i> ), se invece i valori sono da intendersi senza segno, è importante l'esito dell'indicatore di riporto ( <i>carry</i> ). $dst := org + dst$	c p z s o # # # # #
SUB	<i>reg, reg</i> <i>reg, mem</i> <i>reg, imm</i> <i>mem, reg</i> <i>mem, imm</i>	Sottrazione di interi con o senza segno, ignorando il riporto precedente. $dst := org - dst$	c p z s o # # # # #
ADC	<i>reg, reg</i> <i>reg, mem</i> <i>reg, imm</i> <i>mem, reg</i> <i>mem, imm</i>	Somma di interi, con o senza segno, aggiungendo anche il riporto precedente (l'indicatore <i>carry</i> ). $dst := org + dst + c$	c p z s o t . . . . # # # # #
SBB	<i>reg, reg</i> <i>reg, mem</i> <i>reg, imm</i> <i>mem, reg</i> <i>mem, imm</i>	Sottrazione di interi, con o senza segno, tenendo conto del «prestito» precedente (l'indicatore <i>carry</i> ). $dst := org + dst - c$	c p z s o t . . . . # # # # #

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
INC	<i>reg mem</i>	Incrementa di una unità un intero. <i>dst++</i>	c p z s o · # # # #
DEC	<i>reg mem</i>	Decrementa di una unità un valore intero. <i>dst--</i>	c p z s o · # # # #
MUL	<i>reg mem</i>	Moltiplicazione intera senza segno. L'operando è il moltiplicatore, mentre il moltiplicando è costituito da registri prestabiliti. <i>AX := AL*src</i> <i>DX:AX := AX*src</i> <i>EDX:EAX := EAX*src</i>	c p z s o # ? ? ? #
DIV	<i>reg mem</i>	Divisione intera senza segno. L'operando è il divisore, mentre il dividendo è costituito da registri prestabiliti. <i>AL := AX/src AH := resto</i> <i>AX := DX:AX/src</i> <i>DX := resto</i> <i>EAX := EDX:EAX/src</i> <i>EDX := resto</i>	c p z s o ? ? ? ? ?

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
IMUL	<i>reg mem</i>	<p>Moltiplicazione intera con segno. In questo caso l'operando è il moltiplicatore, mentre il moltiplicando è costituito da registri prestabiliti.</p> <p><b><i>AX := AL*src</i></b>  <b><i>DX:AX := AX*src</i></b>  <b><i>EDX:EAX := EAX*src</i></b></p>	<p>c p z s o # ? ? ? #</p>
IDIV	<i>reg mem</i>	<p>Divisione intera con segno. L'operando è il divisore, mentre il dividendo è costituito da registri prestabiliti.</p> <p><b><i>AL := AX/src AH := resto</i></b>  <b><i>AX := DX:AX/src</i></b>  <b><i>DX := resto</i></b>  <b><i>EAX := EDX:EAX/src</i></b>  <b><i>EDX := resto</i></b></p>	<p>c p z s o ? ? ? ? ?</p>



Tabella 64.6. Operazioni logiche.

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
NOT	<i>reg</i> <i>mem</i>	NOT di tutti i bit dell'operando. <i>dst := NOT dst</i>	c p z s o · · · · ·
AND	<i>reg, reg</i> <i>reg, mem</i>	AND, OR, o XOR, tra tutti i bit dei due operandi. <i>dst := org AND dst</i>	c p z s o
OR	<i>reg, imm</i> <i>mem, reg</i>	<i>dst := org OR dst</i>	0 # # # 0
XOR	<i>mem, imm</i>	<i>dst := org XOR dst</i>	

Tabella 64.7. Scorrimenti e rotazioni.

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
SHL SHR	<i>reg, 1</i> <i>mem, 1</i> <i>reg</i> <i>mem</i>	Fa scorrere i bit, rispettivamente verso sinistra o verso destra (l'ultima cifra perduta finisce nell'indicatore del riporto). Se appare un solo operando, la rotazione viene eseguita <b>CL</b> volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o # . . . #
SAL	<i>reg, 1</i> <i>mem, 1</i> <i>reg</i> <i>mem</i>	Funziona esattamente come 'SHL', ma esiste in quanto è la controparte di 'SAR', riferendosi a uno scorrimento aritmetico.	c p z s o # . . . #

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
SAR	<i>reg, 1</i> <i>mem, 1</i> <i>reg</i> <i>mem</i>	Fa scorrere i bit verso destra (l'ultima cifra perduta finisce nell'indicatore del riporto), mantenendo il segno originale. Se appare un solo operando, la rotazione viene eseguita <b>CL</b> volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o # . . . .
RCL RCR	<i>reg, 1</i> <i>mem, 1</i> <i>reg</i> <i>mem</i>	Ruota i bit, rispettivamente verso sinistra o verso destra, utilizzando anche l'indicatore di riporto ( <i>carry</i> ). Se appare un solo operando, la rotazione viene eseguita <b>CL</b> volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o t . . . . # . . . #

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
ROL ROR	<i>reg, 1</i> <i>mem, 1</i> <i>reg</i> <i>mem</i>	Ruota i bit, rispettivamente verso sinistra o verso destra. Se appare un solo operando, la rotazione viene eseguita <i>CL</i> volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o # . . . #

Tabella 64.8. Chiamate e gestione della pila.

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
INT	<i>imm8</i>	Esegue una chiamata attraverso un'interruzione.	c p z s o . . . . .
CALL	<i>reg</i> <i>mem</i> <i>imm</i>	Inserisce nella pila l'indirizzo dell'istruzione successiva e salta all'indirizzo indicato.	c p z s o . . . . .

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
RET		Estrae dalla pila l'indirizzo dell'istruzione da raggiungere e salta a quella (serve a concludere una chiamata eseguita con ' <b>CALL</b> ').	c p z s o · · · · ·
PUSH	<i>reg mem</i>	Inserisce nella pila il valore (della dimensione di un registro comune).	c p z s o · · · · ·
POP	<i>reg mem</i>	Estrae dalla pila l'ultimo valore inserito (della dimensione di un registro comune).	c p z s o · · · · ·
PUSHF		Inserisce nella pila l'insieme del registro degli indicatori ( <b>FLAGS</b> o <b>EFLAGS</b> ).	c p z s o · · · · ·
POPF		Estrae dalla pila l'insieme del registro degli indicatori ( <b>FLAGS</b> o <b>EFLAGS</b> ), aggiornando di conseguenza il registro stesso.	c p z s o · · · · ·

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
PUSHA  PUSHAD		<p>Inserisce nella pila i registri principali: <b>PUSHA</b> inserisce nella pila i registri da 16 bit, mentre <b>PUSHAD</b> li inserisce a 32 bit. In sequenza vengono inseriti: <b>AX, CX, DX, BX, SP, BP, SI, DI</b>; ovvero: <b>EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI</b>.</p> <p>Si osservi che di solito si usa solo <b>PUSHA</b>, lasciando al compilatore la responsabilità di scegliere quale istruzione è appropriata per il contesto.</p>	<p>c p z s o  . . . . .</p>

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
POPA  POPAD		<p>Ripristina i registri principali, estraendo i contenuti dalla pila: <b>'POPA'</b> ripristina i registri da 16 bit, mentre <b>'POPAD'</b> riguarda quelli da 32 bit. In sequenza vengono estratti: <b>DI, SI, BP, SP</b> viene eliminato senza aggiornare il registro, <b>BX, DX, CX, AX</b>; ovvero: <b>EDI, ESI, EBP, ESP</b> viene eliminato senza aggiornare il registro, <b>EBX, EDX, ECX, EAX</b>. Come si vede, anche se <b>'PUSHA'</b> e <b>'PUSHAD'</b> salvano l'indice della pila, in pratica questo indice non viene ripristinato.</p> <p>Si osservi che di solito si usa solo <b>'POPA'</b>, lasciando al compilatore la responsabilità di scegliere quale istruzione è appropriata per il contesto.</p>	c p z s o . . . . .

Tabella 64.9. Indicatori e confronti.

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
CLC		Azzerare l'indicatore del riporto ( <i>carry</i> ), senza intervenire negli altri indicatori.	c p z s o 0 . . . .
CMC		Inverte il valore dell'indicatore del riporto ( <i>carry</i> ).	c p z s o # . . . .
CMP	<i>reg, reg</i> <i>reg, mem</i> <i>reg, imm</i> <i>mem, reg</i> <i>mem, imm</i>	Confronta due valori interi. La comparazione avviene simulando la sottrazione dell'origine dalla destinazione, senza però modificare gli operandi, ma aggiornando gli indicatori, come se fosse avvenuta una sottrazione vera e propria. <i>dst - org</i>	c p z s o # # # # #
TEST	<i>reg, reg</i> <i>reg, imm</i> <i>mem, reg</i> <i>mem, imm</i>	AND dei due valori senza conservare il risultato. Serve solo a ottenere l'aggiornamento degli indicatori. <i>dst AND org</i>	c p z s o 0 # # # 0



Tabella 64.10. Salti.

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
JMP	<i>reg</i> <i>mem</i> <i>imm</i>	Salto incondizionato all'indirizzo indicato.	c p z s o . . . . .
JA JNBE	<i>imm</i>	Dopo un confronto di valori senza segno, salta se la destinazione era maggiore dell'origine. CMP <i>dst, org</i> IF <i>dst &gt; org</i> THEN go to <i>imm</i>	c p z s o t . t . .
JAE JNB	<i>imm</i>	Dopo un confronto di valori senza segno, salta se la destinazione era maggiore o uguale all'origine. CMP <i>dst, org</i> IF <i>dst &gt;= org</i> THEN go to <i>imm</i>	c p z s o t . t . .
JB JNAE	<i>imm</i>	Dopo un confronto di valori senza segno, salta se la destinazione era minore dell'origine. CMP <i>dst, org</i> IF <i>dst &lt; org</i> THEN go to <i>imm</i>	c p z s o t . t . .

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
JBE JNA	<i>imm</i>	Dopo un confronto di valori senza segno, salta se la destinazione era minore o uguale all'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst</i> &lt;= <i>org</i></b> <b>THEN go to <i>imm</i></b>	c p z s o t . t . .
JE	<i>imm</i>	Dopo un confronto, indipendentemente dal segno, salta se la destinazione era uguale all'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst</i> == <i>org</i></b> <b>THEN go to <i>imm</i></b>	c p z s o . . t . .
JNE	<i>imm</i>	Dopo un confronto, indipendentemente dal segno, salta se la destinazione era diversa dall'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst</i> != <i>org</i></b> <b>THEN go to <i>imm</i></b>	c p z s o . . t . .

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
JG JNLE	<i>imm</i>	Dopo un confronto con segno, salta se la destinazione era maggiore dell'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst &gt; org</i></b> <b>THEN go to <i>imm</i></b>	c p z s o · · t t t
JGE JNL	<i>imm</i>	Dopo un confronto con segno, salta se la destinazione era maggiore o uguale all'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst &gt;= org</i></b> <b>THEN go to <i>imm</i></b>	c p z s o · · t t t
JL JNGE	<i>imm</i>	Dopo un confronto con segno, salta se la destinazione era minore dell'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst &lt; org</i></b> <b>THEN go to <i>imm</i></b>	c p z s o · · t t t

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
JLE JNG	<i>imm</i>	Dopo un confronto con segno, salta se la destinazione era minore o uguale all'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst</i> &lt;= <i>org</i></b> <b>THEN go to <i>imm</i></b>	c p z s o · · t t t
JC JNC	<i>imm</i>	Salta se l'indicatore del riporto ( <i>carry</i> ), rispettivamente, è attivo, oppure non è attivo.	c p z s o t · · · ·
JO JNO	<i>imm</i>	Salta se l'indicatore di traboccamento ( <i>overflow</i> ), rispettivamente, è attivo, oppure non è attivo.	c p z s o · · · · t
JS JNS	<i>imm</i>	Salta se l'indicatore di segno ( <i>sign</i> ), rispettivamente, è attivo, oppure non è attivo.	c p z s o · · · t ·
JZ JNZ	<i>imm</i>	Salta se l'indicatore di zero, rispettivamente, è attivo, oppure non è attivo.	c p z s o · · t · ·
JP JNP	<i>imm</i>	Salta se l'indicatore di parità, rispettivamente, è attivo, oppure non è attivo.	c p z s o · t · · ·

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry,</i> <i>parity,</i> <i>zero, sign,</i> <i>overflow</i>
JCXZ	<i>imm</i>	Salta se il valore contenuto nel registro <i>CX</i> è pari a zero.	c p z s o . . . . .

Tabella 64.11. Impostazione del valore in base all'esito di un confronto.

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry,</i> <i>parity,</i> <i>zero, sign,</i> <i>overflow</i>
SETA  SETNBE	<i>reg8</i> <i>mem8</i>	Dopo un confronto di valori senza segno, imposta a uno il registro o la memoria indicati, se la destinazione era maggiore dell'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst &gt; org</i></b> <b>THEN <i>x:=1</i></b> <b>ELSE <i>x:=0</i></b>	c p z s o . . . . .

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
SETAE  SETNB	<i>reg8</i> <i>mem8</i>	Dopo un confronto di valori senza segno, imposta a uno il registro o la memoria indicati, se la destinazione era maggiore o uguale all'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst</i> &gt;= <i>org</i></b> <b>THEN <i>x</i>:=1</b> <b>ELSE <i>x</i>:=0</b>	c p z s o · · · · ·
SETB  SETNAE	<i>reg8</i> <i>mem8</i>	Dopo un confronto di valori senza segno, imposta a uno il registro o la memoria indicati, se la destinazione era minore dell'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst</i> &lt; <i>org</i></b> <b>THEN <i>x</i>:=1</b> <b>ELSE <i>x</i>:=0</b>	c p z s o · · · · ·

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
SETBE  SETNA	<i>reg8</i> <i>mem8</i>	Dopo un confronto di valori senza segno, imposta a uno il registro o la memoria indicati, se la destinazione era minore o uguale all'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst</i> &lt;= <i>org</i></b> <b>THEN <i>x</i>:=1</b> <b>ELSE <i>x</i>:=0</b>	c p z s o · · · · ·
SETE	<i>reg8</i> <i>mem8</i>	Dopo un confronto, indipendentemente dal segno, imposta a uno il registro o la memoria indicati, se la destinazione era uguale all'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst</i> == <i>org</i></b> <b>THEN <i>x</i>:=1</b> <b>ELSE <i>x</i>:=0</b>	c p z s o · · · · ·

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
SETNE	<i>reg8</i> <i>mem8</i>	Dopo un confronto, indipendentemente dal segno, imposta a uno il registro o la memoria indicati, se la destinazione era diversa dall'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst</i> <math>\neq</math> <i>org</i></b> <b>THEN <i>x</i>:=1</b> <b>ELSE <i>x</i>:=0</b>	c p z s o · · · · ·
SETG  SETNLE	<i>reg8</i> <i>mem8</i>	Dopo un confronto con segno, imposta a uno il registro o la memoria indicati, se la destinazione era maggiore dell'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst</i> &gt; <i>org</i></b> <b>THEN <i>x</i>:=1</b> <b>ELSE <i>x</i>:=0</b>	c p z s o · · · · ·



Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry,</i> <i>parity,</i> <i>zero, sign,</i> <i>overflow</i>
SETGE  SETNL	<i>reg8</i> <i>mem8</i>	Dopo un confronto con segno, imposta a uno il registro o la memoria indicati, se la destinazione era maggiore o uguale all'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst</i> &gt;= <i>org</i></b> <b>THEN <i>x</i>:=1</b> <b>ELSE <i>x</i>:=0</b>	c p z s o · · · · ·
SETL  SETNGE	<i>reg8</i> <i>mem8</i>	Dopo un confronto con segno, imposta a uno il registro o la memoria indicati, se la destinazione era minore dell'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst</i> &lt; <i>org</i></b> <b>THEN <i>x</i>:=1</b> <b>ELSE <i>x</i>:=0</b>	c p z s o · · · · ·

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
SETLE SETNG	<i>reg8</i> <i>mem8</i>	Dopo un confronto con segno, imposta a uno il registro o la memoria indicati, se la destinazione era minore o uguale all'origine. <b>CMP <i>dst, org</i></b> <b>IF <i>dst</i> &lt;= <i>org</i></b> <b>THEN <i>x</i>:=1</b> <b>ELSE <i>x</i>:=0</b>	c p z s o · · · · ·
SETC SETNC	<i>reg8</i> <i>mem8</i>	Imposta a uno il registro o la memoria indicati, se l'indicatore del riporto ( <i>carry</i> ), rispettivamente, è attivo, oppure non è attivo.	c p z s o t · · · ·
SETO SETNO	<i>reg8</i> <i>mem8</i>	Imposta a uno il registro o la memoria indicati, se l'indicatore di traboccamento ( <i>overflow</i> ), rispettivamente, è attivo, oppure non è attivo.	c p z s o · · · · t
SETS SETNS	<i>reg8</i> <i>mem8</i>	Imposta a uno il registro o la memoria indicati, se l'indicatore di segno ( <i>sign</i> ), rispettivamente, è attivo, oppure non è attivo.	c p z s o · · · t ·

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
SETZ SETNZ	<i>reg8</i> <i>mem8</i>	Imposta a uno il registro o la memoria indicati, se l'indicatore di zero, rispettivamente, è attivo, oppure non è attivo.	c p z s o · · t · ·

Tabella 64.12. Iterazioni

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
LOOP	<i>imm8</i>	Senza alterare gli indicatori, decrementa di una unità il registro ' <b>ECX</b> ' (o solo ' <b>CX</b> ', se viene richiesta una dimensione più piccola), quindi, se il registro è ancora diverso da zero, salta all'indirizzo cui fa riferimento l'operando.	c p z s o · · · · ·

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
LOOPE  LOOPZ	<i>imm8</i>	Senza alterare gli indicatori, decrementa di una unità il registro ' <b>ECX</b> ' (o solo ' <b>CX</b> ', se viene richiesta una dimensione più piccola), quindi, se il registro è ancora diverso da zero e l'indicatore «zero» è attivo, salta all'indirizzo cui fa riferimento l'operando.	c p z s o . . t . .
LOOPNE  LOOPNZ	<i>imm8</i>	Senza alterare gli indicatori, decrementa di una unità il registro ' <b>ECX</b> ' (o solo ' <b>CX</b> ', se viene richiesta una dimensione più piccola), quindi, se il registro è ancora diverso da zero e l'indicatore «zero» non è attivo, salta all'indirizzo cui fa riferimento l'operando.	c p z s o . . t . .

## 64.4 Primo approccio al linguaggio assembleatore per x86

Per scrivere programmi con il linguaggio assembleatore in un sistema GNU/Linux, è necessario disporre del compilatore: serve GNU AS<sup>1</sup> (GAS) per la sintassi AT&T oppure NASM<sup>2</sup> per quella Intel. <<

### 64.4.1 Il primo programma

Viene mostrato il listato di un programma molto semplice, il cui scopo è unicamente quello di concludere il proprio funzionamento restituendo un valore, attraverso una funzione del sistema operativo. Ne vengono preparate due versioni: una adatta a GNU AS e l'altra per NASM. <<

Listato 64.13. File 'fine-gas.s', adatto per GNU AS.

```
1      #
2      # fine-gas.s
3      #
4      .section .data
5      #
6      .section .text
7      .globl _start
8      #
9      _start:
10     mov    $1, %eax
11     mov    $7, %ebx
12     int   $0x80
```

## Listato 64.14. File ‘fine-nasm.s’, adatto per NASM.

```
1      ;
2      ; fine-nasm.s
3      ;
4      section .data
5      ;
6      section .text
7      global _start
8      ;
9      _start:
10     mov eax, 1
11     mov ebx, 7
12     int 0x80
```

Per compilare il file si genera prima un file oggetto, quindi si passa per il *linker* (il «collegatore»), ovvero il programma che collega i file oggetto che devono comporre il file eseguibile finale. Si comincia con il sorgente per GNU AS:

```
$ as -o fine-gas.o fine-gas.s [Invio]
```

```
$ ld -o fine-gas fine-gas.o [Invio]
```

Per quanto riguarda il sorgente per NASM:

```
$ nasm -f elf -o fine-nasm.o fine-nasm.s [Invio]
```

```
$ ld -o fine-nasm fine-nasm.o [Invio]
```

In entrambi gli esempi, viene generato un file oggetto in formato ELF (*Executable and linkable format*), cosa che deve essere richiesta esplicitamente nel secondo caso, mentre nel primo è implicita. Pertanto, come si vede, il programma ‘**ld**’ viene usato sempre nello stesso modo.

Il programma generato (**'fine-gas'** o **'fine-nasm'**) si limita a chiamare una funzione del sistema operativo, con la quale conclude il suo lavoro restituendo il valore numerico sette. Lo si può verificare ispezionando il parametro **'\$?'** della shell:

```
$ ./fine-gas [Invio]
```

```
$ echo $? [Invio]
```

7

I due programmi sono perfettamente uguali nel modo di disporsi nelle righe del file sorgente, pertanto vengono descritti senza fare distinzioni.

Le prime tre righe sono commenti, ignorati dal compilatore; la quarta riga contiene una direttiva per il compilatore che lo avverte dell'inizio della zona usata per descrivere l'uso della memoria (che in questo caso non viene usata affatto); la quinta riga è un altro commento; la sesta riga avverte il compilatore dell'inizio del codice del programma.

La settima riga serve ad avvisare il compilatore che il simbolo rappresentato dall'etichetta denominata **'\_start'** individua l'indirizzo dell'istruzione iniziale da rendere pubblica, pertanto deve rimanere rintracciabile nel file oggetto generato dalla compilazione. L'ottava riga è un altro commento e la nona riga dichiara il simbolo **'\_start'** già nominato.

Per il compilatore, l'etichetta '**\_start**' indica convenzionalmente il punto di inizio del programma e il simbolo corrispondente deve essere reso pubblico, perché '**ld**' deve sapere da che parte si comincia (soprattutto quando più file oggetto si collegano assieme in un solo eseguibile).

Nella decima riga si assegna il valore uno al registro *EAX* e nell'undicesima si assegna il valore sette al registro *EBX*; infine, nell'ultima riga, si esegue un'interruzione all'indirizzo  $80_{16}$ . In pratica, le ultime tre righe servono a eseguire la chiamata di una funzione del sistema operativo. La funzione è individuata dal numero uno che deve essere collocato nel registro *EAX* e il parametro, rappresentato dal valore di uscita, da collocare nel registro *EBX*. La chiamata effettiva avviene con l'interruzione all'indirizzo  $80_{16}$ .

Dopo aver compilato il programma e ottenuto il file eseguibile, si può dare un'occhiata al suo contenuto con l'aiuto di `Objdump`:<sup>3</sup>

```
$ objdump --disassemble fine-gas [Invio]
```

```
fine-gas:      file format elf32-i386
```

```
Disassembly of section .text:
```

```
08048074 <_start>:
```

```
8048074:      b8 01 00 00 00      mov     $0x1,%eax
8048079:      bb 07 00 00 00      mov     $0x7,%ebx
804807e:      cd 80               int     $0x80
```

Usato in questo modo, `Objdump` disassembla il programma e mostra anche le istruzioni nel linguaggio macchina vero e proprio, con gli indirizzi di memoria virtuale che verrebbero utilizzati durante il



funzionamento. Eventualmente si può richiedere espressamente di disassemblare utilizzando una notazione Intel:

```
$ objdump --disassemble -M intel fine-gas [Invio]
```

```
fine-gas:      file format elf32-i386
```

```
Disassembly of section .text:
```

```
08048074 <_start>:
```

```
8048074:      b8 01 00 00 00      mov     eax,0x1
8048079:      bb 07 00 00 00      mov     ebx,0x7
804807e:      cd 80              int     0x80
```

## 64.4.2 Utilizzo di GDB

Per poter iniziare lo studio di un linguaggio assembler è praticamente indispensabile saper utilizzare un *debugger*, ovvero uno strumento che permetta di eseguire passo per passo il proprio programma, consentendo di verificare lo stato dei registri ed eventualmente della memoria. Infatti, con un linguaggio assembler, operazioni «semplici» come l'emissione di informazioni attraverso lo schermo diventano invece molto complicate.

Nei sistemi GNU è disponibile GDB (GNU debugger)<sup>4</sup>. Per capire come utilizzarlo, si prenda nuovamente l'esempio di programma introduttivo, aggiungendo qualche piccola modifica:

```
1  #
2  # fine-gas.s
3  #
4  .section .data
5  #
6  .section .text
7  .globl _start
8  #
9  _start:
10     mov    $1, %eax
11     bp1:
12     mov    $7, %ebx
13     bp2:
14     int   $0x80
```

```
1  ;
2  ; fine-nasm.s
3  ;
4  segment .data
5  ;
6  segment .text
7  global _start
8  ;
9  _start:
10     mov  eax, 1
11     bp1:
12     mov  ebx, 7
13     bp2:
14     int  0x80
```

Rispetto al file originale sono state aggiunte due etichette, ‘**bp1**’ e ‘**bp2**’ (il cui nome è stato scelto arbitrariamente, ma in questo caso ricorda il termine *breakpoint*), collocate tra le istruzioni che si traducono in codici del linguaggio macchina. Naturalmente, il file

va ricompilato nel modo già descritto; poi, una volta ottenuto il file eseguibile, lo si avvia all'interno di GDB:

```
$ gdb fine-gas [Invio]
```

```
GNU gdb 6.5-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or distribute
copies of it under certain conditions. Type "show copying"
to see the conditions.
There is absolutely no warranty for GDB. Type "show
warranty" for details. This GDB was configured as
"i486-linux-gnu"...(no debugging symbols found)
Using host libthread_db library
"/lib/tls/libthread_db.so.1".
```

A questo punto GDB presenta un invito, dal quale è possibile inserire dei comandi in modo interattivo.

```
(gdb)
```

Di solito, la prima cosa da fare consiste nel definire degli «stop» (*breakpoint*), dove il programma deve essere fermato automaticamente. È per questa ragione che sono state aggiunte delle etichette nel sorgente: per poter associare a quei simboli dei punti di sospensione dell'esecuzione.

```
(gdb) break bp1 [Invio]
```

```
Breakpoint 1 at 0x8048079
```

```
(gdb) break bp2 [Invio]
```

```
Breakpoint 2 at 0x804807e
```

Una volta fissati gli stop, si può «avviare» il programma, che viene così sospeso in corrispondenza del primo di questi punti:

```
(gdb) run [Invio]
```

```
Starting program: /home/tizio/fine-gas  
(no debugging symbols found)
```

```
Breakpoint 1, 0x08048079 in bp1 ()
```

Si ispezionano i registri:

```
(gdb) info registers [Invio]
```

```
eax          0x1          1  
ecx          0x0          0  
edx          0x0          0  
ebx          0x0          0  
esp          0xbff50080    0xbff50080  
ebp          0x0          0x0  
esi          0x0          0  
edi          0x0          0  
eip          0x8048079    0x8048079 <bp1>  
eflags      0x292        [ AF SF IF ]  
cs          0x73         115  
ss          0x7b         123  
ds          0x7b         123  
es          0x7b         123  
fs          0x0          0  
gs          0x0          0
```

Si può verificare che il registro *EAX* contiene il valore uno, come dovrebbe effettivamente in questa posizione. Per far proseguire il programma fino al prossimo stop si usa il comando ‘**continue**’:

```
(gdb) continue [Invio]
```

Breakpoint 2, 0x0804807e in bp2 ()

Si ispezionano nuovamente i registri:

(gdb) **info registers** [Invio]

```

eax          0x1          1
ecx          0x0          0
edx          0x0          0
ebx          0x7          7
esp          0xbfcefe20      0xbfcefe20
ebp          0x0          0x0
esi          0x0          0
edi          0x0          0
eip          0x804807e      0x804807e <bp2>
eflags      0x292          [ AF SF IF ]
cs          0x73          115
ss          0x7b          123
ds          0x7b          123
es          0x7b          123
fs          0x0          0
gs          0x0          0

```

Si può vedere che a questo punto il registro **EBX** risulta impostato con il valore previsto. Si lascia concludere il programma e si termina l'attività con GDB:

(gdb) **continue** [Invio]

Continuing.

Program exited with code 07.

(gdb) **quit** [Invio]

Il procedimento descritto vale per il programma compilato nel modo

«normale», sia attraverso GNU AS, sia attraverso NASM. Ma per avere una visione più chiara di ciò che si fa, occorre abbinare il sorgente originale. Questo può essere fatto con GNU AS, utilizzando l'opzione '**--gstabs**', oppure con NASM mettendo l'opzione '**-g**'. Qui si mostra solo il caso di GNU AS:

```
$ as --gstabs -o fine-gas.o fine-gas.s [Invio]
```

```
$ ld -o fine-gas fine-gas.o [Invio]
```

```
$ gdb fine-gas [Invio]
```

```
GNU gdb 6.5-debian
```

```
Copyright (C) 2006 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. Type "show copying" to see the conditions.
```

```
There is absolutely no warranty for GDB. Type "show warranty" for details. This GDB was configured as
```

```
"i486-linux-gnu"...(no debugging symbols found)
```

```
Using host libthread_db library
```

```
"/lib/tls/libthread_db.so.1".
```

```
(gdb) break bp1 [Invio]
```

```
Breakpoint 1 at 0x8048079: file fine-gas.s, line 12.
```

```
(gdb) break bp2 [Invio]
```

```
Breakpoint 2 at 0x804807e: file fine-gas.s, line 14.
```

```
(gdb) run [Invio]
```

```
Starting program: /home/tizio/fine-gas
```

```
Breakpoint 1, bp1 () at fine-gas.s:12
```

```
12          mov $7, %ebx
```

```
Current language: auto; currently asm
```

Come si vede dall'esempio, si ottengono più informazioni collegate al sorgente originale; in particolare si può sapere qual è la prossima istruzione che verrebbe eseguita. Questa volta si decide di procedere eseguendo un'istruzione alla volta, con il comando '**stepi**':

```
(gdb) stepi [Invio]
```

```
bp2 () at fine-gas.s:14
```

```
14          int $0x80
```

Dato che il programma è molto breve, con la prossima istruzione si va a concluderne il funzionamento:

```
(gdb) stepi [Invio]
```

```
Program exited with code 07.
```

```
(gdb) quit [Invio]
```

Tabella 64.34. Alcuni comandi interattivi per GDB.

Comando	Descrizione
<code>set args <i>argomento</i>...</code>	Definisce gli argomenti della riga di comando del programma.
<code>r [<i>argomento</i>] ...</code> <code>run [<i>argomento</i>] ...</code>	Avvia l'esecuzione del programma, eventualmente con gli argomenti indicati.

Comando	Descrizione
b <i>funzione</i>  break <i>funzione</i>	Definisce uno stop all'esecuzione del programma in corrispondenza del simbolo indicata. Questo simbolo viene definito precisamente «funzione» in quanto riguarda la zona che descrive le istruzioni del programma e non quelle dei dati.
b <i>riga</i>  break <i>riga</i>	Definisce uno stop all'esecuzione del programma in corrispondenza della riga indicata, riferita al file sorgente. Per poter usare questo comando occorre compilare il programma con i riferimenti al file sorgente.
cl [ <i>funzione</i> ]  clear [ <i>funzione</i> ]	Elimina uno stop associato a un certo simbolo, oppure, se c'è, elimina quello della posizione corrente.
cl <i>riga</i>  clear <i>riga</i>	Elimina uno stop associato alla riga indicata.
d [ <i>n</i> ] ...  disable [ <i>n</i> ] ...	Disabilita gli stop indicati per numero, o tutti se non ne viene indicato alcuno.
enable [ <i>n</i> ] ...	Riabilita gli stop indicati per numero, o tutti se non ne viene indicato alcuno.



Comando	Descrizione
d [n] ... delete [n] ...	Elimina gli stop indicati per numero, o tutti se non ne viene indicato alcuno.
si stepi	Esegue la prossima istruzione-macchina e poi si ferma nuovamente. Se l'istruzione è una chiamata di funzione, passa all'inizio della stessa.
ni nexti	Esegue la prossima istruzione-macchina e poi si ferma nuovamente. <b>Se l'istruzione è una chiamata di funzione, questa viene saltata.</b>
c continue	Riprende l'esecuzione di un programma dopo uno stop (che può poi fermarsi nuovamente allo stop successivo, se c'è).
kill	Interrompe definitivamente l'esecuzione del programma.
i r info registers	Mostra lo stato dei registri.
bt backtrace	Mostra lo stato della pila, precisamente mostra gli elementi che compongono lo <i>stack frame</i> .

Comando	Descrizione
<pre>p [/f] [(tipo)] <i>variabile</i> print [/f] [(tipo)] <i>variabile</i></pre>	<p>Mostra il valore contenuto in memoria, in corrispondenza del simbolo specificato (<i>variabile</i>). Se non si tratta di una variabile scalare di dimensione «standard», occorre specificarne il formato, tra parentesi tonde, esattamente prima del nome; se si vuole visualizzare il valore contenuto nella variabile in modo diverso da quello predefinito, occorre aggiungere l'opzione <i>/f</i>, dove la lettera <i>f</i> specifica il tipo di rappresentazione.</p>
<pre>p /f (tipo [n]) <i>variabile</i> print /f (tipo [n]) <i>variabile</i></pre>	<p>Se nella definizione del formato si mette un valore numerico tra parentesi quadre, si intende visualizzare <i>n</i> valori di quel tipo a partire dalla posizione indicata dal nome della variabile. Ciò consente di visualizzare il contenuto di un array.</p>

### 64.4.3 Modalità «TUI»

«

GDB, se è stato compilato per includere tale funzionalità, può essere usato con l'opzione `-tui`, con la quale si ottiene una suddivisione dello schermo in finestre:

```
$ gdb -tui fine-gas [Invio]
```

```
+--fine-gas.s-----+
|10          mov  $1, %eax      |
|11      bp1:                    |
|12          mov  $7, %ebx      |
|13      bp2:                    |
|14          int  $0x80         |
|15                                     |
|16                                     |
|17                                     |
|18                                     |
|19                                     |
|20                                     |
|21                                     |
|22                                     |
|23                                     |
+-----+
```

exec No process In: Line: ?? PC: 0x0

GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details. This GDB was configured as "i486-linux-gnu"...Using host libthread\_db library "/lib/tls/libthread\_db.so.1".

(gdb) **break bp1** [*Invio*]

(gdb) **break bp2** [*Invio*]

```
+--fine-gas.s-----+
|10          mov  $1, %eax      |
|11      bp1:                    |
b+ |12          mov  $7, %ebx    |
|13      bp2:                    |
b+ |14          int  $0x80      |
|15                                     |
|16                                     |
|17                                     |
|18                                     |
|19                                     |
|20                                     |
|21                                     |
|22                                     |
|23                                     |
+-----+
```

exec No process In: Line: ?? PC: 0x0

This GDB was configured as "i486-linux-gnu"...Using host  
libthread\_db library "/lib/tls/libthread\_db.so.1".

(gdb) break bp1

Breakpoint 1 at 0x8048079: file fine-gas.s, line 12.

(gdb) break bp2

Breakpoint 2 at 0x804807e: file fine-gas.s, line 14.

(gdb) **run** [*Invio*]

```

+--fine-gas.s-----+
|10          mov  $1, %eax      |
|11      bp1:                  |
B+>|12          mov  $7, %ebx   |
|13      bp2:                  |
b+ |14          int  $0x80      |
|15                                     |
|16                                     |
|17                                     |
|18                                     |
|19                                     |
|20                                     |
|21                                     |
|22                                     |
|23                                     |
+-----+

```

```
child process 9699 In: bp1          Line: 12   PC: 0x8048079
```

```
(gdb) break bp2
```

```
Breakpoint 2 at 0x804807e: file fine-gas.s, line 14.
```

```
(gdb) run
```

```
Starting program: /home/tizio/fine-gas
```

```
Breakpoint 1, bp1 () at fine-gas.s:12
```

```
Current language:  auto; currently asm
```

Durante il funzionamento si può anche attivare una finestra con lo stato dei registri, attraverso il comando **'layout reg'**:

```
(gdb) layout reg [Invio]
```

```

+--Register group: general-----+
|eax          0x1      1          |
|ecx          0x0      0          |
|edx          0x0      0          |
|ebx          0x0      0          |
|esp          0xbfa77b90      0xbfa77b90 |
|ebp          0x0      0x0        |
+-----+
      |10          mov   $1, %eax      |
      |11          bp1:                |
B+> |12          mov   $7, %ebx      |
      |13          bp2:                |
b+  |14          int   $0x80         |
      |15          |                  |
      |16          |                  |
      +-----+
child process 9699 In: bp1          Line: 12   PC: 0x8048079
Breakpoint 2 at 0x804807e: file fine-gas.s, line 14.
(gdb) run
Starting program: /home/tizio/fine-gas

Breakpoint 1, bp1 () at fine-gas.s:12
Current language:  auto; currently asm
(gdb) layout reg

```

(gdb) **quit** [Invio]

Durante il funzionamento normale di GDB, se è prevista tale modalità di funzionamento, questa può essere attivata anche con la combinazione di tasti [Ctrl x][a].

#### 64.4.4 Utilizzo di DDD

GDB è lo strumento fondamentale per il controllo del funzionamento di un programma, ma al suo fianco se ne possono aggiungere altri che consentono di avere una visione più «semplice». Per esempio, DDD,<sup>5</sup> ovvero *Data display debugger* è un programma frontale che si avvale di GDB, ma lo fa attraverso un'interfaccia grafica che consente di tenere sotto controllo più cose, simultaneamente.

```
$ ddd fine-gas [Invio]
```

Prendendo lo stesso esempio di programma già usato nella sezione precedente, compilato in modo da avere i riferimenti al sorgente, ecco come si può presentare DDD dopo che sono stati fissati gli stop e dopo che il programma è stato avviato, in modo che si arresti al primo di questi:

The screenshot shows a debugger window with the following components:

- Assembly Code:**

```

1 #
2 # fine-gas.s
3 #
4 .section .data
5 #
6 .section .text
7 .globl _start
8 #
9 _start:
10 mov $1, %eax
11 bp1:
12 mov $7, %ebx
13 bp2:
14 int $0x80

```
- Control Menu:**
  - Run
  - Interrupt
  - Step | Stepi
  - Next | Nexti
  - Until | Finish
  - Cont | Kill
  - Up | Down
  - Undo | Redo
  - Edit | Make
- Registers Window:**

Register	Value	Comment
eax	0x1	1
ecx	0x0	0
edx	0x0	0
ebx	0x0	0
esp	0xbfa3ca00	0xbfa3ca00
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0
eip	0x8048079	0x8048079
eflags	0x200292	[ AF ]
cs	0x73	115
ss	0x7b	123
- Assembler Dump:**

```

Dump of assembler code for function bp1:
0x08048079 <bp1+0>: mov $0x7,%ebx
End of assembler dump.

```
- Command Line:**

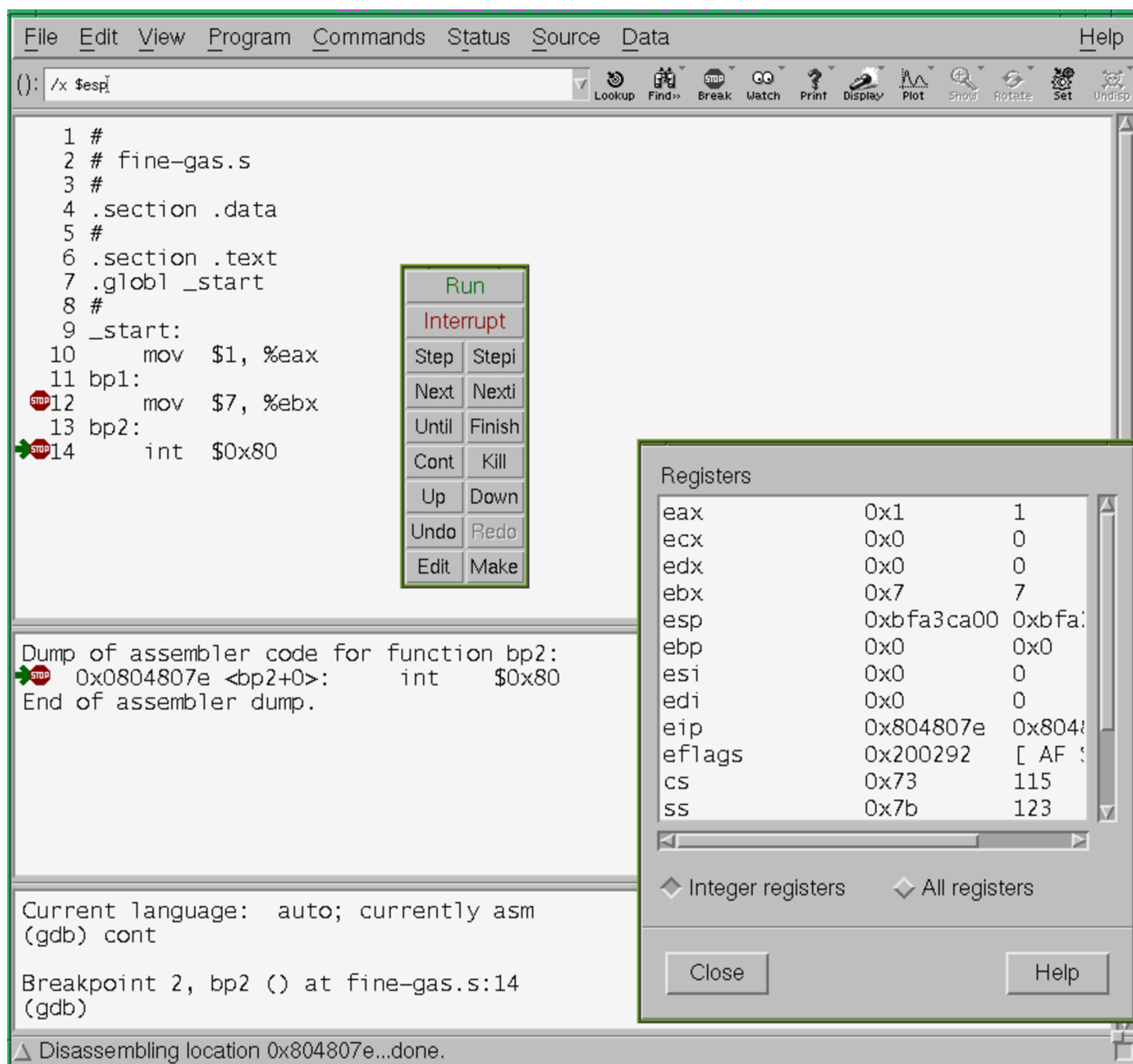
```

(gdb) run
Breakpoint 1, bp1 () at fine-gas.s:12
Current language: auto; currently asm
(gdb)

```
- Status Bar:** Showing integer registers only.

Ecco la situazione al livello del secondo stop:





### 64.4.5 Alcune istruzioni comuni

Le istruzioni del linguaggio assembleatore che si traducono direttamente in istruzioni del linguaggio macchina hanno una forma uniforme: un nome mnemonico seguito dagli operandi.

*mnemonico* *operando* [, *operando*] ...

Per esempio, l'istruzione seguente sottrae il valore contenuto nel registro *EAX* da quello di *EBX*, mettendo il risultato nel registro *EBX* ( $EBX=EBX-EAX$ ):

```
GNUAS sub %eax, %ebx
```

```
NASM sub ebx, eax
```

In questo caso il nome mnemonico è 'SUB', mentre i nomi dei registri sono gli operandi (eventualmente, il codice corrispondente in linguaggio macchina sarebbe  $29C3_{16}$ ). Nelle sezioni successive si descrivono alcune istruzioni comuni.

### 64.4.5.1 MOV

«

L'istruzione 'MOV' serve a copiare il contenuto dell'origine nella destinazione. Gli operandi possono essere registri, aree di memoria e costanti numeriche, tenendo conto che le costanti numeriche possono figurare solo nell'origine e che si può fare riferimento ad aree di memoria in una sola posizione (nell'origine o nella destinazione).

```
GNUAS mov origine, destinazione
```

```
NASM mov destinazione, origine
```

L'esempio seguente copia il contenuto del registro *EAX* all'interno di *EBX*:

```
GNUAS mov %eax, %ebx
```

```
NASM mov ebx, eax
```

## 64.4.5.2 ADD

L'istruzione '**ADD**' serve a eseguire la somma di valori interi, con o senza segno. Gli operandi possono essere registri, aree di memoria e costanti numeriche, tenendo conto che le costanti numeriche possono figurare solo nell'origine e che si può fare riferimento ad aree di memoria in una sola posizione: nell'origine o nella destinazione.

`GNU AS` **add** *origine*, *destinazione*

`NASM` **add** *destinazione*, *origine*

L'esempio seguente aggiunge al registro *EAX* una unità:

`GNU AS` **add** \$1, %eax

`NASM` **add** eax, 1

## 64.4.5.3 SUB

L'istruzione '**SUB**' serve a eseguire la sottrazione di valori interi, con o senza segno. Gli operandi possono essere registri, aree di memoria e costanti numeriche, tenendo conto che le costanti numeriche possono figurare solo nell'origine e che si può fare riferimento ad aree di memoria in una sola posizione: nell'origine o nella destinazione.

`GNU AS` **sub** *origine*, *destinazione*

`NASM` **sub** *destinazione*, *origine*

L'esempio seguente sottrae il valore del registro *EAX* da quello di *EBX*, mettendo il risultato in *EBX*:

`GNU AS` **sub** %eax, %ebx

`NASM` **sub** ebx, eax

#### 64.4.5.4 INC e DEC

«

Le istruzioni ‘**INC**’ e ‘**DEC**’ servono, rispettivamente, a incrementare e a decrementare di una unità il valore dell’operando, che può essere un registro o un’area di memoria. Dal momento che c’è un solo operando, non c’è differenza tra la sintassi di GNU AS e di NASM per quanto riguarda l’ordine degli stessi:

**inc** *destinazione*

**dec** *destinazione*

I due esempi seguenti, rispettivamente, incrementano e decrementano di una unità il valore del registro **EAX**:

- `GNUAS inc %eax`

`NASM inc eax`

- `GNUAS dec %eax`

`NASM dec eax`

#### 64.4.6 Dimensione dei dati nelle istruzioni

«

Di norma, nelle istruzioni che elaborano dati, come quelle descritte nelle sezioni precedenti, se un registro si trova a essere origine o destinazione di qualcosa, implicitamente si intende che i dati debbano essere della sua dimensione. Per esempio, scrivendo ‘**mov \$1, %eax**’ si intende che la costante numerica sia precisamente  $0000\ 0001_{16}$  (ovvero  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2$ ), perché la destinazione è da 32 bit.

Quando il contesto non è sufficiente a stabilire di quanti bit deve essere fatto il valore che si elabora, l'istruzione va precisata. Nel caso di GNU AS si aggiunge un suffisso al nome mnemonico che distingue l'operazione, suffisso che è composto da una sola lettera. Per esempio, **'mov'** diventa **'movl'** per chiarire che si tratta di dati da 32 bit. Nel caso di NASM si aggiunge una parola chiave dopo il nome mnemonico.

Tabella 64.41. Suffissi per precisare la quantità di bit coinvolti nelle istruzioni, secondo la sintassi di GNU AS. Negli esempi, **'mem'** è il nome di un simbolo (un'etichetta) che rappresenta un indirizzo di memoria.

Suffisso	Significato	Dimensione in bit	Esempi
b	byte	8 bit	<code>mov \$1 %al</code> <code>movb \$1 mem</code> <code>pushb mem</code>
w	<i>word</i>	16 bit	<code>mov \$1 %ax</code> <code>movw \$1 mem</code> <code>pushw mem</code>
l	<i>long</i>	32 bit	<code>mov \$1 %eax</code> <code>movl \$1 mem</code> <code>pushl mem</code>

Tabella 64.42. Parole chiave per precisare la quantità di bit coinvolti nelle istruzioni, secondo la sintassi di NASM. Negli esempi, **'mem'** è un simbolo che rappresenta un indirizzo di memoria.

Parola chiave	Dimensione in bit	Esempi
byte	8 bit	<pre>mov al, 1 mov byte [mem], 1 push byte [mem]</pre>
word	16 bit	<pre>mov ax, 1 mov word ax, 1 mov word [mem], 1 push word [mem]</pre>
long	32 bit	<pre>mov eax, 1 mov long [mem], 1 push long [mem]</pre>

## 64.4.7 Direttive per il compilatore

«

Nel sorgente in linguaggio assembler è possibile inserire delle direttive (o pseudoistruzioni) che il compilatore può interpretare per la costruzione corretta del file oggetto. Si usano queste direttive in particolare per definire delle costanti e delle aree di memoria.

### 64.4.7.1 Commenti

«

Le righe vuote e quelle bianche vengono ignorate dal compilatore; inoltre, è possibile inserire dei commenti preceduti da un carattere che viene riconosciuto come prefisso, con il quale si annulla il significato di ciò che appare da quel punto fino alla fine della riga:

`GNUAS #commento`

`NASM ; commento`

### 64.4.7.2 Direttiva «**equ**»

La direttiva '**equ**' viene usata per definire delle costanti simboliche (attraverso un nome, ovvero un'etichetta) da utilizzare poi nelle istruzioni, al posto del dato corrispondente. In linea di principio non dovrebbe essere possibile ridefinire le costanti.

`GNUAS .equ nome , valore`

`NASM nome equ valore`

Nell'esempio seguente si associa al simbolo '**ADDRESS**' il numero otto:

`GNUAS .equ ADDRESS , 8`

`NASM ADDRESS equ 8`

### 64.4.7.3 Direttive di dichiarazione dei dati

Attraverso delle direttive del compilatore si definiscono delle aree di memoria, a cui si fa riferimento nelle istruzioni attraverso dei nomi simbolici (etichette). In questo modo il compilatore attribuisce il loro indirizzo e lo sostituisce nella fase di assemblaggio. Si distingue tra dati che devono essere inizializzati preventivamente con un certo valore e dati il cui valore iniziale è indifferente. Quando si tratta di dati privi di valore iniziale, le informazioni necessarie consistono solo nel nome e nella dimensione dell'area di memoria:

`GNUAS .lcomm nome , dimensione_in_byte`

`NASM nome resb dimensione_in_byte`

`NASM nome resw dimensione_in_multipli_di_16_bit`

`NASM nome resd dimensione_in_multipli_di_32_bit`

Segue la descrizione di alcuni esempi.

- `GNUAS .lcomm BUFFER, 500`

`NASM BUFFER resb 500`

Dichiara localmente un'area di memoria da 500 byte, nominata provvisoriamente **'BUFFER'**.

- `GNUAS .lcomm NUMERO, 4`

`NASM NUMERO resd 1`

Dichiara localmente un'area di memoria da 4 byte (32 bit), nominata provvisoriamente **'NUMERO'**.

Quando si tratta di dati da inizializzare, le informazioni necessarie alla dichiarazione consistono nel nome e nel contenuto, mentre la dimensione deriva dalla pseudoistruzione scelta:

`GNUAS nome: .ascii stringa [, stringa] ...`

`GNUAS nome: .byte byte [, byte] ...`

`GNUAS nome: .int intero [, intero] ...`

`NASM nome db byte [, byte]`

`NASM nome dd intero [, intero]`

Segue la descrizione di alcuni esempi.

- `GNUAS X1: .byte 65`

`NASM X1 db 65`



Inizializza l'area di memoria identificata temporaneamente con il nome **'X1'**, con il valore  $65_{10}$ .

- `GNUAS X2: .byte 'A'`

`NASM X2 db 'A'`

Inizializza l'area di memoria identificata temporaneamente con il nome **'X2'**, inserendo il codice corrispondente alla lettera «A».

- `GNUAS X3: .int 12345`

`NASM X3 dd 12345`

Inizializza l'area di memoria identificata temporaneamente con il nome **'X3'**, inserendo il valore 12345.

- `GNUAS X4: .ascii 'Ciao!', 0`

`NASM X4 db 'Ciao', 0`

Inizializza l'area di memoria a partire dall'indirizzo identificato temporaneamente con il nome **'X3'**, inserendo una stringa, terminata con il valore zero.

## 64.4.8 Sezioni del sorgente

Il sorgente si suddivide in sezioni, le quali descrivono la struttura del file-oggetto che si deve andare a produrre. Per quanto riguarda il formato ELF di un sistema GNU/Linux, si distingue l'area del codice, da quella dei dati; inoltre, nell'ambito dei dati si distingue la parte di quelli preinizializzati e di quelli che non lo sono.<sup>6</sup>



```
.section .data
    dichiarazione_dati_inizializzati
.section .bss
    dichiarazione_dati_non_definiti
.section .text
    istruzioni_del_programma
```

```
section .data
    dichiarazione_dati_inizializzati
section .bss
    dichiarazione_dati_non_definiti
section .text
    istruzioni_del_programma
```

I due modelli sintattici si riferiscono rispettivamente a GNU AS e a NASM.

#### 64.4.9 Usare GNU AS con la notazione Intel

«

È possibile chiedere al compilatore GNU AS di interpretare il sorgente secondo la sintassi «Intel». Per questo si usa la direttiva `‘.intel_syntax noprefix’` (l’opzione `‘noprefix’` serve a consentire di annotare i nomi dei registri senza il prefisso `‘%’`). L’esempio visto nella sezione [64.4.1](#) potrebbe essere modificato nel modo seguente:

```
#  
.section .data  
#  
.section .text  
.globl _start  
#  
.intel_syntax noprefix  
_start:  
    mov    eax, 1  
    mov    ebx, 7  
    int   0x80
```

La direttiva fa sì che il sorgente venga interpretato secondo la sintassi Intel a partire dal punto in cui si trova. Dal momento che l'effetto riguarda solo l'interpretazione delle istruzioni di codice che si traduce in linguaggio macchina (pertanto il modo di dare le altre direttive continua a essere quello normale di GNU AS), è conveniente piazzare la direttiva '**.intel\_syntax noprefix**' all'inizio della sezione '**.text**', come si vede nell'esempio.

Naturalmente è possibile tornare alla sintassi AT&T con una direttiva analoga: '**.att\_syntax**'. Lo si può vedere nell'esempio successivo:

```
#
.section .data
#
.section .text
.globl _start
#
.intel_syntax noprefix
_start:
    mov    eax, 1
.att_syntax
    mov    $7, %ebx
    int   $0x80
```

## 64.5 Esempi con le «quattro operazioni»

«

Vengono mostrati esempi di programmi estremamente banali, per dimostrare il funzionamento delle istruzioni con cui si eseguono le «quattro operazioni» su valori interi, attraverso l'aiuto di GDB. Per la compilazione, se si utilizza GNU AS è bene ricordare di inserire l'opzione '**--gstabs**', mentre con NASM è bene aggiungere l'opzione '**-g**', in modo da poter gestire più facilmente GDB, disponendo dei riferimenti al sorgente:

```
$ as --gstabs -o nome.o nome.s [Invio]
```

```
$ nasm -g -f elf -o nome.o nome.s [Invio]
```

Tabella 64.5. Operazioni aritmetiche.

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
NEG	<i>reg</i> <i>mem</i>	Inverte il segno di un numero, attraverso il complemento a due. <i>dst := -dst</i>	c p z s o # # # # #
ADD	<i>reg, reg</i> <i>reg, mem</i> <i>reg, imm</i> <i>mem, reg</i> <i>mem, imm</i>	Somma di interi, con o senza segno, ignorando il riporto precedente. Se i valori si intendono con segno, è importante l'esito dell'indicatore di traboccamento ( <i>overflow</i> ), se invece i valori sono da intendersi senza segno, è importante l'esito dell'indicatore di riporto ( <i>carry</i> ). <i>dst := org + dst</i>	c p z s o # # # # #
SUB	<i>reg, reg</i> <i>reg, mem</i> <i>reg, imm</i> <i>mem, reg</i> <i>mem, imm</i>	Sottrazione di interi con o senza segno, ignorando il riporto precedente. <i>dst := org - dst</i>	c p z s o # # # # #
ADC	<i>reg, reg</i> <i>reg, mem</i> <i>reg, imm</i> <i>mem, reg</i> <i>mem, imm</i>	Somma di interi, con o senza segno, aggiungendo anche il riporto precedente (l'indicatore <i>carry</i> ). <i>dst := org + dst + c</i>	c p z s o t . . . . # # # # #

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
SBB	<i>reg, reg</i> <i>reg, mem</i> <i>reg, imm</i> <i>mem, reg</i> <i>mem, imm</i>	Sottrazione di interi, con o senza segno, tenendo conto del «prestito» precedente (l'indicatore <i>carry</i> ). <i>dst := org + dst - c</i>	c p z s o t . . . . # # # # #
INC	<i>reg</i> <i>mem</i>	Incrementa di una unità un intero. <i>dst++</i>	c p z s o . # # # #
DEC	<i>reg</i> <i>mem</i>	Decrementa di una unità un valore intero. <i>dst--</i>	c p z s o . # # # #
MUL	<i>reg</i> <i>mem</i>	Moltiplicazione intera senza segno. L'operando è il moltiplicatore, mentre il moltiplicando è costituito da registri prestabiliti. <i>AX := AL * src</i> <i>DX:AX := AX * src</i> <i>EDX:EAX := EAX * src</i>	c p z s o # ? ? ? #

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
DIV	<i>reg mem</i>	Divisione intera senza segno. L'operando è il divisore, mentre il dividendo è costituito da registri prestabiliti. $AL := AX/src$ $AH := resto$ $AX := DX:AX/src$ $DX := resto$ $EAX := EDX:EAX/src$ $EDX := resto$	c p z s o ? ? ? ? ?
IMUL	<i>reg mem</i>	Moltiplicazione intera con segno. In questo caso l'operando è il moltiplicatore, mentre il moltiplicando è costituito da registri prestabiliti. $AX := AL*src$ $DX:AX := AX*src$ $EDX:EAX := EAX*src$	c p z s o # ? ? ? #

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
IDIV	<i>reg mem</i>	Divisione intera con segno. L'operando è il divisore, mentre il dividendo è costituito da registri prestabiliti. <i>AL := AX/src AH := resto</i> <i>AX := DX:AX/src</i> <i>DX := resto</i> <i>EAX := EDX:EAX/src</i> <i>EDX := resto</i>	c p z s o ? ? ? ? ?

### 64.5.1 Somma

«

Viene proposto un programma che si limita a sommare due numeri (interi positivi) definiti in memoria e a restituire il risultato (ammesso che non sia troppo grande) attraverso il valore di uscita. Il programma viene mostrato sia nella forma adatta a GNU AS, sia in quella conforme a NASM. Le righe dei due listati coincidono.

```

1  # op1 + op2
2  #
3  .section .data
4  op1:    .int    15
5  op2:    .int     5
6  #
7  .section .text
8  .globl _start
9  #

```



```

10  _start:
11      mov op1, %edx    # Accumula il primo addendo in EDX.
12      add op2, %edx    # Somma il secondo addendo in EDX.
13  bp1:
14      mov $1,    %eax  # Restituisce il valore contenuto
15      mov %edx, %ebx  # in EDX come valore di uscita,
16      int $0x80      # attraverso la chiamata di sistema
17                          # 1 (exit).

```

```

1  ; op1 + op2
2  ;
3  section .data
4  op1:    dd 15
5  op2:    dd 5
6  ;
7  section .text
8  global _start
9  ;
10 _start:
11     mov edx, [op1]    ; Accumula il primo addendo in EDX.
12     add edx, [op2]    ; Somma il secondo addendo in EDX.
13  bp1:
14     mov eax, 1        ; Restituisce il valore contenuto
15     mov ebx, edx      ; in EDX come valore di uscita,
16     int 80h          ; attraverso la chiamata di sistema
17                          ; 1 (exit).

```

Nelle righe 4 e 5 vengono dichiarate due aree di memoria della dimensione di un registro (32 bit), associando rispettivamente i nomi ‘**op1**’ e ‘**op2**’, a indicare il primo e il secondo operando della somma. Nella riga 11 il contenuto della memoria che rappresenta il primo operando della somma, viene inserito nel registro **EDX**, mentre nella riga 12 si somma quanto è rappresentato dal secondo operando,

nello stesso registro.

Si osservi che per indicare l'indirizzo di memoria è stata usata la modalità di indirizzamento diretta. In pratica, il compilatore sostituisce i nomi 'op1' e 'op2' con l'indirizzo di memoria a cui fanno riferimento.

Al termine, nelle righe da 14 a 16, si prepara la chiamata di sistema 'exit', passando il risultato in modo che venga usato come valore di uscita. Se il risultato della somma è inferiore o uguale a 255, può essere letto.

I programmi sono uguali, a parte qualche piccola differenza nell'allocazione della memoria. Si può controllare con Objdump. Si suppone che il programma sia stato compilato con il nome 'add':

```
$ objdump --disassemble add[Invio]
```

```
add:      file format elf32-i386
```

```
Disassembly of section .text:
```

```
08048074 <_start>:
```

```
8048074:      8b 15 8c 90 04 08      mov     0x804908c,%edx
804807a:      03 15 90 90 04 08      add     0x8049090,%edx
```

```
08048080 <bp1>:
```

```
8048080:      b8 01 00 00 00      mov     $0x1,%eax
8048085:      89 d3              mov     %edx,%ebx
8048087:      cd 80              int     $0x80
```

Si può controllare il funzionamento del programma, avviandolo e

verificando poi il valore di uscita:

```
$ ./add ; echo $? [Invio]
```

20

Si analizza il funzionamento del programma con GDB:

```
$ gdb add [Invio]
```

```
(gdb) break bp1 [Invio]
```

```
Breakpoint 1 at 0x8048080: file add.s, line 14.
```

```
(gdb) run [Invio]
```

```
Starting program: /home/tizio/add
```

```
Breakpoint 1, bp1 () at add.s:14
```

```
14      mov $1, %eax # Restituisce il valore contenuto in EDX
```

```
Current language: auto; currently asm
```

```
(gdb) info registers [Invio]
```

eax	0x0	0
ecx	0x0	0
<b>edx</b>	<b>0x14</b>	<b>20</b>
ebx	0x0	0
esp	0xbf9dcb10	0xbf9dcb10
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0
eip	0x8048080	0x8048080 <bp1>
eflags	0x216	[ PF AF IF ]
cs	0x73	115
ss	0x7b	123
ds	0x7b	123
es	0x7b	123

```
fs          0x0      0
gs          0x0      0
```

(gdb) **stepi** [*Invio*]

```
bp1 () at add.s:15
15          mov %edx, %ebx # come valore di uscita, attraverso
```

(gdb) **stepi** [*Invio*]

```
bp1 () at add.s:16
16          int $0x80 # chiamata di sistema 1 (exit).
```

(gdb) **info registers** [*Invio*]

```
eax          0x1      1
ecx          0x0      0
edx          0x14     20
ebx          0x14     20
esp          0xbfba64d0 0xbfba64d0
ebp          0x0      0x0
esi          0x0      0
edi          0x0      0
eip          0x8048087 0x8048087 <bp1+7>
eflags      0x216     [ PF AF IF ]
cs          0x73     115
ss          0x7b     123
ds          0x7b     123
es          0x7b     123
fs          0x0      0
gs          0x0      0
```

(gdb) **stepi** [*Invio*]<sup>7</sup>

```
Program exited with code 024.
```

(gdb) **quit** [Invio]

### 64.5.1.1 Traboccamento



Si può modificare leggermente il programma proposto, allo scopo di causare un traboccamento, in modo da vedere cosa accade nei registri:

3	.section .data
4	op1: .int 0x7FFFFFFF # 0b01111111111111111111111111111111
5	op2: .int 0x00000001 # 0b00000000000000000000000000000001

3	section .data
4	op1: dd 0x7FFFFFFF ; 01111111111111111111111111111111b
5	op2: dd 0x00000001 ; 00000000000000000000000000000001b

Compilando il programma ed eseguendolo con l'ausilio di GDB si può verificare che la somma di quei due valori trasforma il risultato in un valore apparentemente negativo, cosa che è indice di un traboccamento:

```

eax          0x0          0
ecx          0x0          0
edx        0x80000000      -2147483648
ebx          0x0          0
esp          0xbfa6f390    0xbfa6f390
ebp          0x0          0x0
esi          0x0          0
edi          0x0          0
eip          0x8048080    0x8048080 <bp1>
eflags      0xa96        [ PF AF SF IF OF ]
cs          0x73          115
ss          0x7b          123
ds          0x7b          123
es          0x7b          123

```

```
fs          0x0      0
gs          0x0      0
```

A ogni modo il fatto viene sottolineato dall'indicatore di traboccamento (*overflow*). In questo caso non interviene l'indicatore di riporto (*carry*), perché se il risultato fosse da intendersi senza segno, sarebbe ancora corretto.

### 64.5.1.2 Riporto

«

Si può modificare leggermente il programma proposto, allo scopo di causare un riporto, in modo da vedere cosa accade nei registri:

3	.section .data
4	op1: .int 0xFFFFFFFF # 0b11111111111111111111111111111110
5	op2: .int 0x00000002 # 0b00000000000000000000000000000010

3	section .data
4	op1: dd 0xFFFFFFFF ; 11111111111111111111111111111110b
5	op2: dd 0x00000002 ; 00000000000000000000000000000001b

In questo caso, la somma dei due valori supera proprio di una unità la capienza del registro, che alla fine risulta a zero, ma l'indicatore di riporto (*carry*) segnala l'accaduto:

```
eax          0x0      0
ecx          0x0      0
edx        0x0      0
ebx          0x0      0
esp          0xbfbb44e0    0xbfbb44e0
ebp          0x0      0x0
esi          0x0      0
edi          0x0      0
eip          0x8048080    0x8048080 <bp1>
```

eflags	0x257	[ <b>CF</b> PF AF ZF IF ]
cs	0x73	115
ss	0x7b	123
ds	0x7b	123
es	0x7b	123
fs	0x0	0
gs	0x0	0

Si può osservare che è assente l'indicatore di traboccamento (*overflow*), perché se la somma fosse avvenuta tra due numeri con segno, il risultato sarebbe corretto.

### 64.5.1.3 Somma di valori molti grandi

Viene mostrato un altro esempio, dove la somma riguarda valori molto grandi, divisi tra due registri. I due listati sono equivalenti e compatibili, riga per riga: «

```

1  # op1 + op2
2  #
3  .section .data
4  op1:      .quad    0x00FFFFFFFFFFFFFFFF
5  op2:      .quad    0x0000000000000001
6  #
7  .section .text
8  .globl _start
9  #
10 _start:
11     mov op1 , %eax # Accumula metà del primo addendo
12                        # in EAX.
13     mov op1+4, %edx # Accumula il resto del primo
14                        # addendo in EDX.
15     add op2, %eax # Somma metà del secondo addendo
16                        # in EAX.
17  bp1:

```

```

18     adc op2+4, %edx # Somma il resto del secondo addendo
19                                     # in EDX. Il risultato atteso in
20                                     # EDX:EAX è: 0x01000000:0x00000000
21 bp2:
22     mov $1,      %eax # Conclude il funzionamento del
23     mov $0,      %ebx # programma restituendo zero in
24     int $0x80      # ogni caso.

```

```

1     ; op1 + op2
2     ;
3     section .data
4     op1:      dd  0xFFFFFFFF, 0x00FFFFFF ; 0x00FFFFFFFFFFFFFFFF
5     op2:      dd  0x00000001, 0x00000000 ; 0x0000000000000001
6     ;
7     section .text
8     global _start
9     ;
10    _start:
11    mov eax, [op1]      ; Accumula metà del primo addendo
12                                ; in EAX.
13    mov edx, [op1+4]   ; Accumula il resto del primo
14                                ; addendo in EDX.
15    add eax, [op2]     ; Somma metà del secondo addendo
16                                ; in EAX.
17 bp1:
18    adc edx, [op2+4]   ; Somma il resto del secondo
19                                ; addendo in EDX. Il risultato
20                                ; atteso in EDX:EAX è:
21                                ; 0x01000000:0x00000000
22 bp2:
23    mov eax, 1         ; Conclude il funzionamento
24    mov edx, 0         ; del programma restituendo
25    int 80h           ; zero in ogni caso.

```



In questo programma ci sono delle complicazioni che vanno descritte, cominciando preferibilmente dalla versione per GNU AS (il primo listato). Nelle righe 4 e 5 vengono dichiarati due numeri molto grandi, da 64 bit. Successivamente, nelle righe da 11 a 15, si fa riferimento a questi due numeri, prendendoli a pezzi. Per la precisione, nella riga 11 si copiano i primi 32 bit a partire dall'indirizzo a cui fa riferimento l'etichetta '**op1**' (sono solo 32 bit perché l'istruzione copia il valore in un registro di tale dimensione); nella riga 12 si copiano gli altri 32 bit, indicando che dall'indirizzo dell'etichetta '**op1**' occorre spostarsi in avanti di quattro byte. Lo stesso ragionamento si fa per il secondo operando.

L'ordine in cui sono prelevati i dati è importante e occorre riflettere su questo fatto. Il registro *EAX* viene caricato con la porzione meno significativa del numero, che in memoria si trova nei «primi» 32 bit. Ciò avviene perché si intende che il microprocessore operi ordinando i byte in memoria secondo la modalità *little endian*.

Nel secondo listato, quello per NASM, non essendo possibile indicare un numero completo da 64 bit, si è reso necessario spezzarlo in due. In questo caso, per mantenere la stessa struttura dell'altro listato, i due tronconi sono stati messi in fila, apparentemente in ordine inverso, per riprodurre la stessa sequenza *little endian* complessiva.

Una volta compreso il modo in cui i dati sono prelevati dalla memoria, ci si può soffermare sulle istruzioni di somma: il troncone meno significativo viene sommato con l'istruzione '**ADD**', mentre per quello più significativo si usa '**ADC**' che aggiunge anche il riporto, se c'è.

Alla fine, una volta compilato il programma, con GDB è possibile

eseguirlo fino al simbolo evidenziato dall'etichetta **'bp1'** per verificare l'effetto della prima addizione, quindi lo si può fare proseguire fino al simbolo **'bp2'**, per verificare che la coppia di registri ***EDX:EAX*** contenga il risultato corretto:

```
(gdb) break bp1 [Invio]
```

```
(gdb) break bp2 [Invio]
```

```
(gdb) run [Invio]
```

```
(gdb) info registers [Invio]
```

<b>eax</b>	<b>0x0</b>	<b>0</b>
ecx	0x0	0
edx	0xffffffff	16777215
ebx	0x0	0
esp	0xbfa89bb0	0xbfa89bb0
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0
eip	0x8048085	0x8048085 <bp1>
eflags	0x257	[ <b>CF</b> PF AF ZF IF ]
cs	0x73	115
ss	0x7b	123
ds	0x7b	123
es	0x7b	123
fs	0x0	0
gs	0x0	0

La prima somma ha prodotto uno zero nel registro ***EAX***, con riporto.

```
(gdb) continue [Invio]
```

```
(gdb) info registers [Invio]
```



```

8      .globl _start
9      #
10     _start:
11         mov op1, %edx # Accumula il minuendo in EDX.
12         sub op2, %edx # Riduce EDX del sottraendo.
13     bp:
14         mov $1,    %eax # Restituisce il valore contenuto in
15         mov %edx, %ebx # EDX come valore di uscita,
16         int $0x80      # attraverso la chiamata di sistema
17                             # 1 (exit).

```

```

1      ; op1 - op2
2      ;
3      section .data
4      op1: dd 0x00000001 ; 00000000000000000000000000000001b
5      op2: dd 0x00000002 ; 00000000000000000000000000000010b
6      ;
7      section .text
8      global _start
9      ;
10     _start:
11         mov edx, [op1] ; Accumula il minuendo in EDX.
12         sub edx, [op2] ; Riduce EDX del sottraendo.
13     bp1:
14         mov eax, 1      ; Restituisce il valore contenuto in EDX
15         mov ebx, edx    ; come valore di uscita, attraverso la
16         int 80h        ; chiamata di sistema 1 (exit).

```

Rispetto agli esempi che riguardano la somma, qui, nella riga 12 si utilizza l'istruzione '**SUB**'. Come si può comprendere, dal momento che si sottrae una grandezza maggiore di quella contenuta nel minuendo, si ottiene un valore negativo, oppure, se i valori sono da intendersi senza segno, si ottiene un «riporto», come richiesta del prestito di una cifra oltre quella più significativa. Con GDB, in cor-

rispondenza del simbolo **'bp1'** si possono vedere i registri e si può verificare che è scattato il riporto (*carry*) e il segno:

```
...
edx                0xffffffff      -1
...
eflags            0x297      [ CF PF AF SF IF ]
...
```

Se si lascia concludere il programma, il valore di uscita che si ottiene è  $255_{10}$ , pari a  $FF_{16}$ , ovvero  $377_8$ , ovvero  $1111111_2$ , dal momento che si possono ottenere solo otto bit.

### 64.5.2.1 Inversione del segno

Si può vedere cosa accade se, invece di usare l'istruzione **'SUB'**, si cambia il segno al sottraendo e lo si somma semplicemente all'altro operando: «

```
1  # op1 - op2
2  #
3  .section .data
4  op1:    .int    0x00000001    # 0b00000000000000000000000000000001
5  op2:    .int    0x00000002    # 0b00000000000000000000000000000010
6  #
7  .section .text
8  .globl _start
9  #
10 _start:
11     mov op2, %edx # Accumula il sottraendo in EDX.
12     neg %edx     # Ne inverte il segno con il complemento a due.
13 bp1:
14     add op1, %edx # Somma il minuendo a EDX.
15 bp2:
16     mov $1,    %eax # Restituisce il valore contenuto in EDX
17     mov %edx, %ebx # come valore di uscita, attraverso la
18     int $0x80     # chiamata di sistema 1 (exit).
```

```

1      ; op1 - op2
2      ;
3      section .data
4      op1:    dd      0x00000001    ; 00000000000000000000000000000001b
5      op2:    dd      0x00000002    ; 00000000000000000000000000000010b
6      ;
7      section .text
8      global _start
9      ;
10     _start:
11         mov edx, [op2] ; Accumula il sottraendo in EDX.
12         neg edx      ; Ne inverte il segno con il complemento a due.
13     bp1:
14         add edx, [op1] ; Somma il minuendo a EDX.
15     bp2:
16         mov eax, 1     ; Restituisce il valore contenuto in EDX
17         mov ebx, edx   ; come valore di uscita, attraverso la
18         int 80h       ; chiamata di sistema 1 (exit).

```

Rispetto a quanto fatto nel caso precedente, qui, nella riga 11 viene sommato il valore del sottraendo nel registro **EDX**, di cui viene invertito il segno nella riga 12. Successivamente, nella riga 14 viene sommato il valore del minuendo. Il risultato è lo stesso, ma gli indicatori si comportano in modo differente durante il procedimento. In corrispondenza del simbolo '**bp1**' si può vedere quanto segue:

```

...
edx          0xffffffffe      -2
...
eflags      0x293      [ CF AF SF IF ]
...

```

Si può osservare che l'inversione del segno ha prodotto un riporto, oltre alla segnalazione del segno, ammesso che questo vada considerato.

In corrispondenza del simbolo **'bp2'** è appena stata eseguita la somma del minuendo. Si può osservare che questa volta non si ottiene alcun riporto:

```
...
edx                0xffffffff    -1
...
eflags            0x293    [ PF SF IF ]
...
```

### 64.5.2.2 Sottrazione per fasi successive

Viene proposto un esempio di sottrazione da svolgere in due fasi, perché il valore non è contenibile in un solo registro. Si vuole eseguire:  $10000000000000000_{16} - 0FFFFFFFFFFFFFFFF_{16}$ .

```
1  # op1 + op2
2  #
3  .section .data
4  op1:    .quad    0x10000000000000000
5  op2:    .quad    0x0FFFFFFFFFFFFFFFF
6  #
7  .section .text
8  .globl _start
9  #
10 _start:
11     mov op1    , %eax # Accumula metà del primo valore
12                          # in EAX.
13     mov op1+4, %edx # Accumula il resto del primo
14                          # valore in EDX.
15 bp1:
16     sub op2,    %eax # Sottrae metà del secondo valore
17                          # in EAX.
18 bp2:
19     sbb op2+4, %edx # Sottrae il resto del secondo
```

```

20                                     # valore in EDX.
21 bp3:
22     mov $1,      %eax # Conclude il funzionamento del
23     mov $0,      %ebx # programma restituendo zero
24     int $0x80      # in ogni caso.

```

```

1     ; op1 + op2
2     ;
3     section .data
4     op1:      dd  0x00000000, 0x10000000 ; 0x1000000000000000
5     op2:      dd  0xFFFFFFFF, 0x0FFFFFFF ; 0xFFFFFFFFFFFFFFF
6     ;
7     section .text
8     global _start
9     ;
10    _start:
11        mov eax, [op1]      ; Accumula metà del primo valore
12                                ; in EAX.
13        mov edx, [op1+4]   ; Accumula il resto del primo
14                                ; valore in EDX.
15    bp1:
16        sub eax, [op2]      ; Sottrae metà del secondo valore
17                                ; in EAX.
18    bp2:
19        sbb edx, [op2+4]   ; Sottrae il resto del secondo
20                                ; valore in EDX.
21    bp3:
22        mov eax, 1          ; Conclude il funzionamento del
23        mov edx, 0          ; programma restituendo zero
24        int 0x80           ; in ogni caso.

```

Il valore del minuendo viene copiato, in due pezzi, nei registri **EDX:EAX**; quindi si sottraggono i 32 bit inferiori del sottraendo al registro **EAX** e infine si sottraggono i 32 bit più significativi del



sottraendo dal registro *EDX*, tenendo conto del riporto precedente.  
In corrispondenza del simbolo '**bp1**', il minuendo è stato copiato nei registri *EDX:EAX*:

```
...
eax          0x0          0
...
edx          0x10000000    268435456
...
eflags      0x292      [ AF SF IF ]
...
```

Al punto di '**bp2**' è stata eseguita la sottrazione dei 32 bit inferiori, causando un riporto, da intendersi come richiesta di un prestito:

```
...
eax       0x1       1
...
edx          0x10000000    268435456
...
eflags      0x213      [ CF AF IF ]
...
```

Al punto di '**bp3**' è stata completata la sottrazione:

```
...
eax          0x1          1
...
edx       0x0       0
...
eflags      0x256      [ PF AF ZF IF ]
...
```

### 64.5.3 Moltiplicazione senza segno

&lt;&lt;

Nella moltiplicazione si distingue il fatto che si consideri il segno o meno. Quando si esegue una moltiplicazione senza segno si usa l'istruzione '**MUL**' con l'indicazione di un solo operando, perché gli altri sono impliciti. Nella moltiplicazione il contenitore del risultato deve essere più capiente di ciò che è stato usato per produrlo. Si distinguono questi casi:

```
AX := AL*src
```

```
DX:AX := AX*src
```

```
EDX:EAX := EAX*src
```

In pratica, l'origine deve essere di pari dimensioni del moltiplicando, costituito, rispettivamente da: *AL*, *AX* o *EAX*.

```
1  # op1 * op2
2  #
3  .section .data
4  op1:      .short  0x8008
5  op2:      .short  0x2002
6  #
7  .section .bss
8  .lcomm prodotto, 4
9  #
10 .section .text
11 .globl _start
12 #
13 _start:
```



```
18  bp1:
19      mov  [prodotto], ax    ; Copia in memoria la prima
20                          ; parte del risultato.
21      mov  [prodotto+2], dx ; Copia in memoria la
22                          ; seconda parte del
23                          ; risultato.
24      mov  eax, [prodotto]  ; Copia il risultato
25                          ; dalla memoria a EAX.
26  bp2:
27      mov  eax, 1           ; Restituisce il valore
28                          ; contenuto in EAX
29      mov  ebx, eax        ; come valore di uscita,
30                          ; attraverso la
31      int  0x80           ; chiamata di sistema
32                          ; 1 (exit).
```

In questo esempio i valori da moltiplicare sono della dimensione di 16 bit e sono, rispettivamente,  $8008_{16}$  e  $2002_{16}$ . Moltiplicando questi due valori si deve ottenere  $10020010_{16}$ . In pratica si deve eseguire una moltiplicazione del tipo  $DX:AX := AX * src$ .

Rispetto a esempi già visti nelle sezioni precedenti, in questo si dichiara un'area di memoria non inizializzata, nella riga numero 8, per contenere almeno quattro byte (32 bit), con il nome simbolico 'prodotto'. All'interno di questa area di memoria si vuole ricostruire il risultato della moltiplicazione in modo che occupi un gruppo continuo di 32 bit.

Nella riga 15 si esegue la moltiplicazione, utilizzando come operando direttamente la memoria. Tuttavia, per farlo, occorre specificare la dimensione di questo operando, altrimenti verrebbe presa in considerazione un'area più grande del voluto.

In corrispondenza del punto **'bp1'** si può vedere il risultato della moltiplicazione diviso tra ***DX*** e ***AX***:

```
...
eax          0x10          16
...
edx          0x1002       4098
...
eflags      0xa93        [ CF AF SF IF OF ]
...
```

Nelle righe 17 e 18 viene copiato il risultato in memoria, ricomponendolo nell'ordine corretto, osservando che si usa una rappresentazione dei valori numerici in modalità *little endian*, quindi la parte meno significativa viene copiata prima. In corrispondenza del punto **'bp2'** il risultato della moltiplicazione è tutto contenuto nel registro ***EAX***:

```
...
eax          0x10020010      268566544
...
edx          0x1002       4098
...
eflags      0xa93        [ CF AF SF IF OF ]
...
```

#### 64.5.4 Moltiplicazione con segno

La moltiplicazione con segno si ottiene con un'istruzione differente, **'IMUL'**, che però va usata con la stessa modalità di quella senza segno. Sarebbe possibile usare più di un operando con questa istruzione, senza bisogno di operandi impliciti, ma in generale non è conveniente perché c'è il rischio di creare confusione sulla dimensione di questi operandi.



```

1      # op1 * op2
2      #
3      .section .data
4      op1:      .short    0x0007
5      op2:      .short   -0x0001
6      #
7      .section .bss
8      .lcomm prodotto, 4
9      #
10     .section .text
11     .globl _start
12     #
13     _start:
14         mov    op1, %ax      # Accumula il moltiplicando
15                                 # in AX.
16         imulw op2          # Moltiplica il secondo valore
17                                 # per AX (implicito).
18     bp1:
19         mov %ax, prodotto  # Copia in memoria la prima
20                                 # parte del risultato.
21         mov %dx, prodotto+2 # Copia in memoria la seconda
22                                 # parte del risultato.
23         mov prodotto, %eax  # Copia il risultato dalla
24                                 # memoria a EAX.
25     bp2:
26         mov $1,    %eax      # Restituisce il valore
27                                 # contenuto in EAX
28         mov %eax, %ebx      # come valore di uscita,
29                                 # attraverso la
30         int $0x80          # chiamata di sistema 1 (exit).

```

```

1      ; op1 * op2
2      ;
3      section .data
4      op1:      dw          0x0007

```

```
5   op2:    dw    -0x0001
6   ;
7   section .bss
8   prodotto resb 4
9   ;
10  section .text
11  global _start
12  ;
13  _start:
14      mov     ax, [op1]           ; Accumula il
15                                     ; moltiplicando in AX.
16      imul  word [op2]           ; Moltiplica il secondo
17                                     ; valore per AX
18                                     ; (implicito).
19  bp1:
20      mov     [prodotto], ax      ; Copia in memoria la
21                                     ; prima parte del
22                                     ; risultato.
23      mov     [prodotto+2], dx    ; Copia in memoria la
24                                     ; seconda parte del
25                                     ; risultato.
26      mov     eax, [prodotto]     ; Copia il risultato
27                                     ; dalla memoria a EAX.
28  bp2:
29      mov     eax, 1              ; Restituisce il valore
30                                     ; contenuto in EAX
31      mov     ebx, eax            ; come valore di uscita,
32                                     ; attraverso la
33      int     0x80               ; chiamata di sistema
34                                     ; 1 (exit).
```

Rispetto a esempi già visti, questo utilizza una costante numerica negativa (riga 5); in pratica è il compilatore che la trasforma nel complemento a due, in modo automatico. Per il resto, tutto pro-

cede come nell'esempio della moltiplicazione intera, a parte l'uso dell'istruzione '**IMUL**'.

In corrispondenza del punto '**bp1**' si può vedere il risultato della moltiplicazione, distribuito tra **DX:AX**:

```
...
eax          0xffff9    65529
...
edx          0xffff    65535
...
eflags      0x292    [ AF SF IF ]
...
```

In '**bp2**' il risultato è completo nel registro **EAX**:

```
...
eax          0xffffffff9    -7
...
eflags      0x292    [ AF SF IF ]
...
```

### 64.5.5 Divisione

«

Per la divisione si usa un meccanismo simile a quello della moltiplicazione, ma opposto:

$$AL := AX/src \quad AH := resto$$

$$AX := DX:AX/src \quad DX := resto$$

$$EAX := EDX:EAX/src \quad EDX := resto$$



In questo esempio si parte da valori che occupano 32 bit, azzerando inizialmente **EDX** perché il dividendo non è così grande da richiederne l'utilizzo.

```

1  # op1 / op2
2  #
3  .section .data
4  op1:    .int    0x00010001
5  op2:    .int    0x00000002
6  #
7  .section .text
8  .globl _start
9  #
10 _start:
11     mov  op1, %eax # Accumula il dividendo in EAX.
12     mov  $0,  %edx # Azzera EDX.
13     divl op2      # Divide EDX:EAX per il divisore.
14 bp1:
15     mov  $1,  %eax # Restituisce il valore contenuto
16     mov  %edx, %ebx # in EDX come valore di uscita,
17     int  $0x80    # attraverso la chiamata di sistema
18                               # 1 (exit).

```

```

1  ; op1 * op2
2  ;
3  section .data
4  op1:    dd      0x00010001
5  op2:    dd      0x00000002
6  ;
7  section .text
8  global _start
9  ;
10 _start:
11     mov  eax, [op1] ; Accumula il dividendo in EAX.
12     mov  edx, 0     ; Azzera EDX.

```

13	div long [op2]	; Divide EDX:EAX per il divisore.
14	bp1:	
15	mov eax, 1	; Restituisce il valore contenuto
16	mov edx, eax	; in EDX come valore di uscita,
17	int 0x80	; attraverso la chiamata di sistema
18		; 1 (exit).

In corrispondenza del punto ‘**bp1**’ si può leggere il risultato della divisione in *EAX* e il resto in *EDX*:

```

...
eax          0x8000    32768
...
edx          0x1      1
...
eflags      0x212    [ AF IF ]
...

```

Per quanto riguarda la divisione con segno, tutto procede nello stesso modo, a parte il fatto che si utilizza l’istruzione ‘**IDIV**’:

10	_start:	
11	mov op1, %eax	# Accumula il dividendo in EAX.
12	mov \$0, %edx	# Azzera EDX.
13	idivl op2	# Divide EDX:EAX per il divisore.

10	_start:	
11	mov eax, [op1]	; Accumula il dividendo in EAX.
12	mov edx, 0	; Azzera EDX.
13	idiv long [op2]	; Divide EDX:EAX per il divisore.

## 64.6 Esempi con gli «spostamenti»

Vengono mostrati esempi di programmi estremamente banali, per dimostrare il funzionamento delle istruzioni con cui si eseguono gli spostamenti di bit (scorrimenti e rotazioni), attraverso l'aiuto di GDB. Per la compilazione, se si utilizza GNU AS è bene ricordare di inserire l'opzione '**--gstabs**', mentre con NASM è bene aggiungere l'opzione '**-g**', in modo da poter gestire più facilmente GDB, disponendo dei riferimenti al sorgente:

```
$ as --gstabs -o nome.o nome.s [Invio]
```

```
$ nasm -g -f elf -o nome.o nome.s [Invio]
```

Tabella 64.7. Scorrimenti e rotazioni.

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
SHL SHR	<i>reg, 1</i> <i>mem, 1</i> <i>reg</i> <i>mem</i>	Fa scorrere i bit, rispettivamente verso sinistra o verso destra (l'ultima cifra perduta finisce nell'indicatore del riporto). Se appare un solo operando, la rotazione viene eseguita <b>CL</b> volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o # . . . #

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
SAL	<i>reg, 1 mem, 1 reg mem</i>	Funziona esattamente come 'SHL', ma esiste in quanto è la controparte di 'SAR', riferendosi a uno scorrimento aritmetico.	c p z s o # . . . #
SAR	<i>reg, 1 mem, 1 reg mem</i>	Fa scorrere i bit verso destra (l'ultima cifra perduta finisce nell'indicatore del riporto), mantenendo il segno originale. Se appare un solo operando, la rotazione viene eseguita <i>CL</i> volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o # . . . .

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione	Indicatori principali: <i>carry, parity, zero, sign, overflow</i>
RCL RCR	<i>reg, 1</i> <i>mem, 1</i> <i>reg</i> <i>mem</i>	Ruota i bit, rispettivamente verso sinistra o verso destra, utilizzando anche l'indicatore di riporto ( <i>carry</i> ). Se appare un solo operando, la rotazione viene eseguita <b>CL</b> volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o t . . . . # . . . #
ROL ROR	<i>reg, 1</i> <i>mem, 1</i> <i>reg</i> <i>mem</i>	Ruota i bit, rispettivamente verso sinistra o verso destra. Se appare un solo operando, la rotazione viene eseguita <b>CL</b> volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o # . . . #

### 64.6.1 Scorrimento logico

Viene proposto un esempio in cui si vede l'uso delle istruzioni '**SHL**' e '**SHR**'. Si parte da un valore che viene fatto scorrere a sinistra, poi viene ripristinato e quindi viene fatto scorrere a destra.



```

# Scorrimento logico (logic shift)
#
.section .data
op1:    .byte    0b01001100        # 0x4C
#
.section .text
.globl _start
#
_start:
    mov  op1, %al    # Inizializza AL con il valore di
                    # partenza.
    shl  $1, %al    # Sposta a sinistra le cifre di una
                    # posizione.
bp1:    # AL = 0b10011000 0x98 overflow
    shl  $1, %al    # Sposta a sinistra le cifre di una
                    # posizione.
bp2:    # AL = 0b00110000 0x30 carry, overflow
    mov  op1, %al    # Inizializza AL con il valore di
                    # partenza.
    shr  $1, %al    # Sposta a destra le cifre di una
                    # posizione.
bp3:    # AL = 0b00100110 0x26
    shr  $2, %al    # Sposta a destra le cifre di due
                    # posizioni.
bp4:    # AL = 0b00001001 0x09 carry
    mov  $0, %ebx    # Restituisce il valore contenuto in AL
    mov  %al, %bl    # copiandolo nella parte inferiore del
    mov  $1, %eax    # registro EBX ed eseguendo la chiamata
    int  $0x80      # di sistema 1 (exit).

```

```

; Scorrimento logico (logic shift)
;
section .data
op1:    dd        01001100b        ; 0x4C
;

```

```

section .text
global _start
;
_start:
    mov  al, [op1] ; Inizializza AL con il valore di
                    ; partenza.
    shl  al, 1     ; Sposta a sinistra le cifre di una
                    ; posizione.
bp1:                ; AL = 10011000b 0x98 overflow
    shl  al, 1     ; Sposta a sinistra le cifre di una
                    ; posizione.
bp2:                ; AL = 00110000b 0x30 carry, overflow
    mov  al, [op1] ; Inizializza AL con il valore di
                    ; partenza.
    shr  al, 1     ; Sposta a destra le cifre di una
                    ; posizione.
bp3:                ; AL = 00100110b 0x26
    shr  al, 2     ; Sposta a destra le cifre di due
                    ; posizioni.
bp4:                ; AL = 00001001b 0x09 carry
    mov  ebx, 0    ; Restituisce il valore contenuto in AL
    mov  bl, al    ; copiandolo nella parte inferiore del
    mov  eax, 1    ; registro EBX ed eseguendo la chiamata
    int  0x80     ; di sistema 1 (exit).

```

Viene mostrato lo stato del registro *EAX* e degli indicatori, nei vari punti di sospensione previsti:

## bp1

```

eax          0x98          152
eflags      0xa92        [ AF SF IF OF ]

```

Bisogna tenere presente che l'ultima istruzione eseguita riguarda solo la porzione *AL* del registro *EAX*; pertanto, dal momento

che il bit più significativo è a uno, si attiva l'indicatore del segno e anche quello del traboccamento, dato che, se lo scorrimento riguardasse un valore con segno, il valore ottenuto non sarebbe più valido (perché a questo punto si sarebbe trasformato in un numero negativo).

## bp2

```
eax          0x30      48
eflags      0xa17     [ CF PF AF IF OF ]
```

Sempre tenendo conto che le istruzioni eseguite riguardano solo *AL*, si può osservare che la perdita della cifra più significativa a uno si traduce nell'attivazione del riporto; inoltre, se si trattasse di un valore con segno, anche questa volta si sarebbe verificata un'inversione (da negativo a positivo) e per questo si attiva ancora l'indicatore di traboccamento.

## bp3

```
eax          0x26      38
eflags      0x212     [ AF IF ]
```

Questo scorrimento a destra fa perdere solo delle cifre a zero, pertanto il segno non cambia e non c'è alcun riporto (resto).

## bp4

```
eax          0x9       9
eflags      0x217     [ CF PF AF IF ]
```

Questo ulteriore scorrimento, di due posizioni, comporta la fuoriuscita di una cifra a uno che implica l'attivazione dell'indicatore del riporto (resto).



## 64.6.2 Scorrimento aritmetico



Viene proposto un esempio analogo a quello della sezione precedente, in cui si vede l'uso delle istruzioni '**SAL**' e '**SAR**'. Si parte da un valore negativo che viene fatto scorrere a sinistra, poi viene ripristinato e quindi viene fatto scorrere a destra.

```
# Scorrimento aritmetico
#
.section .data
op1:    .byte    0b11001100    # 0xCC
#
.section .text
.globl _start
#
_start:
    mov  op1, %al    # Inizializza AL con il valore di
                    # partenza.
    sal  $1,  %al    # Sposta a sinistra le cifre di una
                    # posizione.
bp1:    # AL = 0b10011000 0x98 carry
    sal  $1,  %al    # Sposta a sinistra le cifre di una
                    # posizione.
bp2:    # AL = 0b00110000 0x30 carry, overflow
    mov  op1, %al    # Inizializza AL con il valore di
                    # partenza.
    sar  $1,  %al    # Sposta a destra le cifre di una
                    # posizione.
bp3:    # AL = 0b11100110 0xE6
    sar  $2,  %al    # Sposta a destra le cifre di due
                    # posizioni.
bp4:    # AL = 0b11111001 0xF9 carry
    mov  $0,   %ebx  # Restituisce il valore contenuto in AL
    mov  %al,  %bl   # copiandolo nella parte inferiore del
    mov  $1,   %eax  # registro EBX ed eseguendo la chiamata
```

```
int $0x80      # di sistema 1 (exit).
```

```
; Scorrimento aritmetico
;
section .data
op1:      dd      11001100b      ; 0xCC
;
section .text
global _start
;
_start:
    mov  al, [op1] ; Inizializza AL con il valore di
                    ; partenza.
    sal  al, 1     ; Sposta a sinistra le cifre di una
                    ; posizione.
bp1:      ; AL = 10011000b 0x98 carry
    sal  al, 1     ; Sposta a sinistra le cifre di una
                    ; posizione.
bp2:      ; AL = 00110000b 0x30 carry, overflow
    mov  al, [op1] ; Inizializza AL con il valore di
                    ; partenza.
    sar  al, 1     ; Sposta a destra le cifre di una
                    ; posizione.
bp3:      ; AL = 11100110b 0xE6
    sar  al, 2     ; Sposta a destra le cifre di due
                    ; posizioni.
bp4:      ; AL = 11111001b 0xF9 carry
    mov  ebx, 0    ; Restituisce il valore contenuto in AL
    mov  bl, al    ; copiandolo nella parte inferiore del
    mov  eax, 1    ; registro EBX ed eseguendo la chiamata
    int  0x80      ; di sistema 1 (exit).
```

Viene mostrato lo stato del registro *EAX* e degli indicatori, nei vari punti di sospensione previsti:

**bp1**

```

eax          0x98      152
eflags      0x293     [ CF AF SF IF ]

```

Bisogna tenere presente che l'ultima istruzione eseguita riguarda solo la porzione *AL* del registro *EAX*; pertanto, dal momento che il bit più significativo è a uno, è attivo l'indicatore del segno e anche quello del riporto, avendo perduto una cifra a uno. Il numero non ha cambiato di segno e quindi l'indicatore di traboccamento non è attivo.

**bp2**

```

eax          0x30      48
eflags      0xa17     [ CF PF AF IF OF ]

```

Sempre tenendo conto che le istruzioni eseguite riguardano solo *AL*, si può osservare che anche questa volta è stata persa una cifra più significativa a uno, essendo attivo l'indicatore del riporto, ma la cosa più importante è che adesso il valore ha cambiato di segno e così si vede attivo anche l'indicatore di traboccamento.

**bp3**

```

eax          0xe6      230
eflags      0x292     [ AF SF IF ]

```

Questo scorrimento a destra fa perdere solo delle cifre a zero, pertanto non c'è alcun riporto (resto).

**bp4**

```

eax          0xf9      249
eflags      0x297     [ CF PF AF SF IF ]

```

Questo ulteriore scorrimento, di due posizioni, comporta la fuoriuscita di una cifra a uno che implica l'attivazione dell'indicatore del riporto (resto).

### 64.6.3 Rotazione

«

Viene proposto un esempio analogo a quello delle sezioni precedenti, in cui si vede l'uso delle istruzioni 'ROL' e 'ROR'. Si parte da un valore che viene fatto ruotare a sinistra, poi viene ripristinato e quindi viene fatto ruotare a destra.

```
# Rotazione
#
.section .data
op1:    .byte    0b11001100    # 0xCC
#
.section .text
.globl _start
#
_start:
    mov  op1, %al    # Inizializza AL con il valore di
                    # partenza.
    rol  $1, %al    # Ruota a sinistra le cifre di una
                    # posizione.
bp1:    # AL = 0b10011001 0x99 carry
    rol  $1, %al    # Ruota a sinistra le cifre di una
                    # posizione.
bp2:    # AL = 0b00110011 0x33 carry, overflow
    mov  op1, %al    # Inizializza AL con il valore di
                    # partenza.
    ror  $1, %al    # Ruota a destra le cifre di una
                    # posizione.
bp3:    # AL = 0b01100110 0x66 overflow
    ror  $2, %al    # Ruota a destra le cifre di due
```

```

                                # posizioni.
bp4:                                # AL = 0b10011001 0x99 carry, overflow
    mov $0,    %ebx # Restituisce il valore contenuto in AL
    mov %al,   %bl  # copiandolo nella parte inferiore del
    mov $1,    %eax # registro EBX ed eseguendo la chiamata
    int $0x80      # di sistema 1 (exit).

```

```

; Rotazione
;
section .data
op1:    dd    11001100b    ; 0xCC
;
section .text
global _start
;
_start:
    mov  al, [op1] ; Inizializza AL con il valore di
                    ; partenza.
    rol  al, 1     ; Ruota a sinistra le cifre di una
                    ; posizione.
bp1:                                ; AL = 10011001b 0x99 carry
    rol  al, 1     ; Ruota a sinistra le cifre di una
                    ; posizione.
bp2:                                ; AL = 00110011b 0x33 carry, overflow
    mov  al, [op1] ; Inizializza AL con il valore di
                    ; partenza.
    ror  al, 1     ; Ruota a destra le cifre di una
                    ; posizione.
bp3:                                ; AL = 01100110b 0x66 overflow
    ror  al, 2     ; Ruota a destra le cifre di due
                    ; posizioni.
bp4:                                ; AL = 10011001b 0x99 carry, overflow
    mov  ebx, 0    ; Restituisce il valore contenuto in AL
    mov  bl, al    ; copiandolo nella parte inferiore del
    mov  eax, 1    ; registro EBX ed eseguendo la chiamata

```

```
int 0x80 ; di sistema 1 (exit).
```

Viene mostrato lo stato del registro *EAX* e degli indicatori, nei vari punti di sospensione previsti:

### bp1

```
eax          0x99      153
eflags      0x293     [ CF AF SF IF ]
```

Si deve tenere conto che l'ultima istruzione eseguita riguarda solo la porzione *AL* del registro *EAX*. La rotazione ha fatto uscire a sinistra una cifra a uno che rientra nella parte destra. La cifra spostata la si ritrova nell'indicatore di riporto.

### bp2

```
eax          0x33      51
eflags      0xa93     [ CF AF SF IF OF ]
```

Questa ulteriore rotazione a sinistra fa uscire un'altra cifra a uno, che viene segnalata nel riporto, ma il bit più significativo passa a zero e quindi l'indicatore di traboccamento viene attivato.

### bp3

```
eax          0x66      102
eflags      0xa92     [ AF SF IF OF ]
```

Questa rotazione a destra fa perdere solo una cifra a zero, pertanto non c'è alcun riporto (resto), ma dal momento che rientra a sinistra, l'indicatore di traboccamento manifesta l'ipotesi di cambiamento del segno.

## bp4

```

eax          0x99      153
eflags      0xa93     [ CF AF SF IF OF ]

```

Questa ulteriore rotazione, di due posizioni, comporta la fuoriuscita di una cifra a uno che implica l'attivazione dell'indicatore del riporto (resto). Dal momento che la cifra rientra a sinistra, l'indicatore di traboccamento segnala l'ipotesi di cambiamento del segno.

### 64.6.4 Rotazione con riporto

Viene proposto un esempio analogo a quello della sezione precedente, in cui si vede l'uso delle istruzioni **RCL** e **RCR**. Si parte da un valore che viene fatto ruotare a sinistra, poi viene ripristinato e quindi viene fatto ruotare a destra. Qui viene usata anche l'istruzione **CLC** per azzerare il riporto.

```

# Rotazione con riporto
#
.section .data
op1:    .byte    0b11001101    # 0xCD
#
.section .text
.globl _start
#
_start:
    clc          # Azzerare il riporto.
    mov  op1, %al # Inizializza AL con il valore di
                # partenza.
    rcl  $1, %al  # Ruota a sinistra le cifre di una
                # posizione.
bp1:
    # AL = 0b10011010 0x9A carry
    rcl  $1, %al  # Ruota a sinistra le cifre di una

```

```

                                # posizione.
bp2:                            # AL = 0b00110101 0x35 carry, overflow
    clc                          # Azzera il riporto.
    mov  op1, %al                # Inizializza AL con il valore di
                                # partenza.
    rcr  $1, %al                # Ruota a destra le cifre di una
                                # posizione.
bp3:                            # AL = 0b01100110 0x66 carry, overflow
    rcr  $2, %al                # Ruota a destra le cifre di due
                                # posizioni.
bp4:                            # AL = 0b01011001 0x59 carry, overflow
    mov  $0, %ebx               # Restituisce il valore contenuto in AL
    mov  %al, %bl               # copiandolo nella parte inferiore del
    mov  $1, %eax               # registro EBX ed eseguendo la chiamata
    int  $0x80                  # di sistema 1 (exit).

```

```

; Rotazione con riporto
;
section .data
op1:    dd      11001101b      ; 0xCD
;
section .text
global _start
;
_start:
    clc                      ; Azzera il riporto.
    mov  al, [op1]           ; Inizializza AL con il valore di
                                ; partenza.
    rcl  al, 1                ; Ruota a sinistra le cifre di una
                                ; posizione.
bp1:                            ; AL = 10011010b 0x9A carry
    rcl  al, 1                ; Ruota a sinistra le cifre di una
                                ; posizione.
bp2:                            ; AL = 00110101b 0x35 carry, overflow
    clc                      ; Azzera il riporto.

```



```

    mov  al, [op1] ; Inizializza AL con il valore di
                    ; partenza.
    rcr  al, 1     ; Ruota a destra le cifre di una
                    ; posizione.
bp3:
    rcr  al, 2     ; Ruota a destra le cifre di due
                    ; posizioni.
bp4:
    mov  ebx, 0    ; Restituisce il valore contenuto in AL
    mov  bl, al    ; copiandolo nella parte inferiore del
    mov  eax, 1    ; registro EBX ed eseguendo la chiamata
    int  0x80     ; di sistema 1 (exit).

```

Viene mostrato lo stato del registro *EAX* e degli indicatori, nei vari punti di sospensione previsti:

### bp1

```

eax          0x9a      154
eflags      0x293     [ CF AF SF IF ]

```

Si deve tenere conto che l'ultima istruzione eseguita riguarda solo la porzione *AL* del registro *EAX*. La rotazione ha fatto uscire a sinistra una cifra a uno che passa nel riporto, mentre a destra entra il valore precedente del riporto (zero).

### bp2

```

eax          0x35      53
eflags      0xa93     [ CF AF SF IF OF ]

```

Questa ulteriore rotazione a sinistra fa uscire un'altra cifra a uno che passa nel, mentre entra a destra la cifra a uno del riporto precedente. Dopo la rotazione il bit più significativo passa a zero e quindi l'indicatore di traboccamento viene attivato.

**bp3**

```

eax          0x66      102
eflags      0xa93     [ CF AF SF IF OF ]

```

Questa rotazione a destra fa perdere una cifra a uno che si trasferisce del riporto; inoltre, dato che il riporto precedente era zero, a sinistra si inserisce una cifra a zero, la quale fa scattare l'indicatore di traboccamento (nell'ipotesi di un valore con segno).

**bp4**

```

eax          0x59      89
eflags      0xa93     [ CF AF SF IF OF ]

```

Questa ulteriore rotazione, di due posizioni, comporta la fuoriuscita di una cifra a zero e poi a uno che implica l'attivazione dell'indicatore del riporto (resto). In precedenza c'era stato un riporto che attualmente risulta inserito subito dopo la cifra più significativa. Dal momento l'ultimo scorrimento (dei due eseguiti qui) fa cambiare la cifra più significativa, si attiva l'indicatore di traboccamento.

## 64.7 Esempi con i confronti

«

Viene mostrato un esempio di programma, con l'unico scopo di dimostrare il funzionamento dell'istruzione di confronto, attraverso l'aiuto di GDB. Per la compilazione, se si utilizza GNU AS è bene ricordare di inserire l'opzione '**--gstabs**', mentre con NASM è bene aggiungere l'opzione '**-g**', in modo da poter gestire più facilmente GDB, disponendo dei riferimenti al sorgente:

```
$ as --gstabs -o nome.o nome.s [Invio]
```

```
$ nasm -g -f elf -o nome.o nome.s [Invio]
```

```
# Confronto
#
.section .text
.globl _start
#
_start:
    mov $0b01000000, %al      # A = 64      A = 64
    mov $0b10000000, %bl      # B = 128     B = -128
    cmp %bl, %al              # carry, segno, overflow
bp1:
    mov $0b01000000, %al      # A = 64      A = 64
    mov $0b11000000, %bl      # B = 192     B = -64
    cmp %bl, %al              # carry, segno, overflow
bp2:
    mov $0b01000000, %al      # A = 64      A = 64
    mov $0b00000000, %bl      # B = 0       B = 0
    cmp %bl, %al              #
bp3:
    mov $0b01000000, %al      # A = 64      A = 64
    mov $0b01000000, %bl      # B = 64     B = 64
    cmp %bl, %al              # zero
bp4:
    mov $0b00000000, %al      # A = 0       A = 0
    mov $0b10000000, %bl      # B = 128     B = -128
    cmp %bl, %al              # carry, segno, overflow
bp5:
    mov $0b00000000, %al      # A = 0       A = 0
    mov $0b11000000, %bl      # B = 192     B = -64
    cmp %bl, %al              # carry
bp6:
    mov $0b00000000, %al      # A = 0       A = 0
    mov $0b00000000, %bl      # B = 0       B = 0
    cmp %bl, %al              # zero
bp7:
```

```

    mov $0b00000000, %al      # A = 0      A = 0
    mov $0b01000000, %bl      # B = 64     B = 64
    cmp %bl, %al             # carry, segno
bp8:
    mov $0b11000000, %al      # A = 192    A = -64
    mov $0b10000000, %bl      # B = 128    B = -128
    cmp %bl, %al             #
bp9:
    mov $0b11000000, %al      # A = 192    A = -64
    mov $0b11000000, %bl      # B = 192    B = -64
    cmp %bl, %al             # zero
bp10:
    mov $0b11000000, %al      # A = 192    A = -64
    mov $0b00000000, %bl      # B = 0      B = 0
    cmp %bl, %al             # segno
bp11:
    mov $0b11000000, %al      # A = 192    A = -64
    mov $0b01000000, %bl      # B = 64     B = 64
    cmp %bl, %al             # segno
bp12:
    mov $0b10000000, %al      # A = 128    A = -128
    mov $0b10000000, %bl      # B = 128    B = -128
    cmp %bl, %al             # zero
bp13:
    mov $0b10000000, %al      # A = 128    A = -128
    mov $0b11000000, %bl      # B = 192    B = -64
    cmp %bl, %al             # carry, segno
bp14:
    mov $0b10000000, %al      # A = 128    A = -128
    mov $0b00000000, %bl      # B = 0      B = 0
    cmp %bl, %al             # segno
bp15:
    mov $0b10000000, %al      # A = 128    A = -128
    mov $0b01000000, %bl      # B = 64     B = 64

```

```

    cmp %bl, %al                # overflow
bp16:
    mov $0, %ebx                # Conclude il funzionamento con la
    mov $1, %eax                # chiamata di sistema 1 (exit).
    int $0x80                   #

```

```

; Confronto
;
section .text
global _start
;
_start:
    mov al, 01000000b           ; A = 64      A = 64
    mov bl, 10000000b           ; B = 128     B = -128
    cmp al, bl                  ; carry, segno, overflow
bp1:
    mov al, 01000000b           ; A = 64      A = 64
    mov bl, 11000000b           ; B = 192     B = -64
    cmp al, bl                  ; carry, segno, overflow
bp2:
    mov al, 01000000b           ; A = 64      A = 64
    mov bl, 00000000b           ; B = 0       B = 0
    cmp al, bl                  ;
bp3:
    mov al, 01000000b           ; A = 64      A = 64
    mov bl, 01000000b           ; B = 64      B = 64
    cmp al, bl                  ; zero
bp4:
    mov al, 00000000b           ; A = 0       A = 0
    mov bl, 10000000b           ; B = 128     B = -128
    cmp al, bl                  ; carry, segno, overflow
bp5:
    mov al, 00000000b           ; A = 0       A = 0
    mov bl, 11000000b           ; B = 192     B = -64
    cmp al, bl                  ; carry

```

```

bp6:
    mov al, 00000000b    ; A = 0      A = 0
    mov bl, 00000000b    ; B = 0      B = 0
    cmp al, bl          ; zero
bp7:
    mov al, 00000000b    ; A = 0      A = 0
    mov bl, 01000000b    ; B = 64     B = 64
    cmp al, bl          ; carry, segno
bp8:
    mov al, 11000000b    ; A = 192    A = -64
    mov bl, 10000000b    ; B = 128    B = -128
    cmp al, bl          ;
bp9:
    mov al, 11000000b    ; A = 192    A = -64
    mov bl, 11000000b    ; B = 192    B = -64
    cmp al, bl          ; zero
bp10:
    mov al, 11000000b    ; A = 192    A = -64
    mov bl, 00000000b    ; B = 0      B = 0
    cmp al, bl          ; segno
bp11:
    mov al, 11000000b    ; A = 192    A = -64
    mov bl, 01000000b    ; B = 64     B = 64
    cmp al, bl          ; segno
bp12:
    mov al, 10000000b    ; A = 128    A = -128
    mov bl, 10000000b    ; B = 128    B = -128
    cmp al, bl          ; zero
bp13:
    mov al, 10000000b    ; A = 128    A = -128
    mov bl, 11000000b    ; B = 192    B = -64
    cmp al, bl          ; carry, segno
bp14:
    mov al, 10000000b    ; A = 128    A = -128

```

```

    mov bl, 00000000b           ; B = 0           B = 0
    cmp al, bl                 ; segno
bp15:
    mov al, 10000000b         ; A = 128          A = -128
    mov bl, 01000000b         ; B = 64           B = 64
    cmp al, bl                 ; overflow
bp16:
    mov ebx, 0                ; Conclude il funzionamento con la
    mov eax, 1                 ; chiamata di sistema 1 (exit).
    int 0x80                   ;

```

Tra i commenti si possono osservare i valori confrontati, interpretandoli sia con segno, sia senza segno, assieme all'effetto atteso sugli indicatori. Viene mostrato lo stato degli indicatori nei vari punti di sospensione previsti, con l'ausilio di GDB:

## bp1

```

eax          0x40          64
ebx          0x80          128
eflags      0xa87         [ CF PF SF IF OF ]

```

## bp2

```

eax          0x40          64
ebx          0xc0          192
eflags      0xa83         [ CF SF IF OF ]

```

## bp3

```

eax          0x40          64
ebx          0x0           0
eflags      0x202         [ IF ]

```

**bp4**

eax	0x40	64
ebx	0x40	64
eflags	0x246	[ PF ZF IF ]

**bp5**

eax	0x0	0
ebx	0x80	128
eflags	0xa83	[ CF SF IF OF ]

**bp6**

eax	0x0	0
ebx	0xc0	192
eflags	0x203	[ CF IF ]

**bp7**

eax	0x0	0
ebx	0x0	0
eflags	0x246	[ PF ZF IF ]

**bp8**

eax	0x0	0
ebx	0x40	64
eflags	0x287	[ CF PF SF IF ]

**bp9**

eax	0xc0	192
ebx	0x80	128
eflags	0x202	[ IF ]

**bp10**

eax	0xc0	192
ebx	0xc0	192
eflags	0x246	[ PF ZF IF ]



**bp11**

eax	0xc0	192
ebx	0x0	0
eflags	0x286	[ PF SF IF ]

**bp12**

eax	0xc0	192
ebx	0x40	64
eflags	0x282	[ SF IF ]

**bp13**

eax	0x80	128
ebx	0x80	128
eflags	0x246	[ PF ZF IF ]

**bp14**

eax	0x80	128
ebx	0xc0	192
eflags	0x287	[ CF PF SF IF ]

**bp15**

eax	0x80	128
ebx	0x0	0
eflags	0x282	[ SF IF ]

**bp16**

eax	0x80	128
ebx	0x40	64
eflags	0xa02	[ IF OF ]

## 64.8 Le istruzioni di salto

«

Con il linguaggio macchina, le strutture di controllo si realizzano solo attraverso le istruzioni di salto. Una di queste istruzioni è incondizionata, mentre le altre sono sottoposte al verificarsi di una condizione. A loro volta, le istruzioni di salto condizionato si dividono in due gruppi: uno riferito al controllo dello stato di un certo indicatore, l'altro riferito virtualmente a un confronto di valori.

### 64.8.1 Portata del salto

«

In generale, le istruzioni di salto hanno un solo operando, costituito dal riferimento all'indirizzo di memoria da raggiungere. Il compilatore traduce il riferimento all'indirizzo di memoria in un «dislocamento», ovvero nella quantità di byte da percorrere, in avanti o indietro. A questo proposito, ciò che rappresenta il riferimento alla memoria può avere dimensioni diverse e questo significa che l'istruzione può richiedere di precisare la dimensione del numero che rappresenta tale dislocamento. In modo predefinito, la dimensione del numero usato per indicare il dislocamento è quella di un registro comune.

Si osservi che la possibilità di dichiarare esplicitamente l'entità del dislocamento dipende dal compilatore; d'altro canto, sarebbe compito del compilatore determinarlo automaticamente.

## 64.8.2 Salto incondizionato



Il salto incondizionato si ottiene con l'istruzione '**JMP**':

```
JMP imm
```

Con un linguaggio assembler, il valore immediato richiesto come operando si ottiene con un simbolo, ovvero indicando il nome di un'etichetta già dichiarata altrove nel sorgente:

```
# Salto incondizionato
#
.section .text
.globl _start
#
_start:
    mov $1, %ebx
bp1:
    jmp bp3
bp2:
    mov $2, %ebx      # Questa istruzione non viene eseguita.
bp3:
    mov $1, %eax      # Conclude il funzionamento con la
    int $0x80         # chiamata di sistema 1 (exit).
```

```
; Salto incondizionato
;
section .text
global _start
;
_start:
    mov ebx, 1
bp1:
    jmp bp3
bp2:
```

```

    mov ebx, 2          ; Questa istruzione non viene eseguita.
bp3:
    mov eax, 1          ; Conclude il funzionamento con la
    int 0x80           ; chiamata di sistema 1 (exit).

```

In questo esempio si vede che l'istruzione contenuta tra i punti 'bp2' e 'bp3' non viene mai eseguita.

### 64.8.3 Salto condizionato dallo stato di un indicatore

«

Un gruppo di istruzioni di salto condizionato dipende dallo stato di un certo indicatore. Queste istruzioni hanno la forma seguente, dove la lettera *x* identifica l'indicatore da controllare:

*Jx imm*

*JNx imm*

Per esempio, l'istruzione '**JZ**' salta se l'indicatore zero è attivo, mentre '**JNZ**' salta se l'indicatore zero non è attivo.

Tabella 64.135. Salti condizionati dallo stato di un indicatore particolare.

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione
JC JNC	<i>imm</i>	Salta se l'indicatore del riporto ( <i>carry</i> ), rispettivamente, è attivo, oppure non è attivo.

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione
JO JNO	<i>imm</i>	Salta se l'indicatore di traboccamento ( <i>overflow</i> ), rispettivamente, è attivo, oppure non è attivo.
JS JNS	<i>imm</i>	Salta se l'indicatore di segno ( <i>sign</i> ), rispettivamente, è attivo, oppure non è attivo.
JZ JNZ	<i>imm</i>	Salta se l'indicatore di zero, rispettivamente, è attivo, oppure non è attivo.
JP JNP	<i>imm</i>	Salta se l'indicatore di parità, rispettivamente, è attivo, oppure non è attivo.

#### 64.8.4 Salto condizionato da un confronto

Il gruppo più importante di istruzioni di salto condizionato dipende da un confronto, che di solito si realizza con l'istruzione '**CMP**'. In pratica, l'istruzione '**CMP**' simula una sottrazione, aggiornando gli indicatori come se si trattasse di una sottrazione vera e propria; successivamente, l'istruzione di salto condizionato verifica gli indicatori e si comporta di conseguenza.

Da quanto descritto si deve comprendere che, anche se le istruzioni di questo tipo richiamano l'idea del confronto tra due valori, in pratica dipendono da indicatori che possono essere stati modificati da istruzioni di tipo differente.

Dal momento che, ai fini del confronto tra due valori, la valutazione degli indicatori va fatta diversamente a seconda che si tratti di interi senza segno o con segno, queste istruzioni sono suddivise in sottogruppi.

Tabella 64.136. Salti condizionati dall'esito di un confronto con valori interi senza segno.

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione
JA JNBE	<i>imm</i>	Dopo un confronto di valori senza segno, salta se la destinazione era maggiore dell'origine. CMP <i>dst, org</i> IF <i>dst &gt; org</i> THEN go to <i>imm</i>
JAE JNB	<i>imm</i>	Dopo un confronto di valori senza segno, salta se la destinazione era maggiore o uguale all'origine. CMP <i>dst, org</i> IF <i>dst &gt;= org</i> THEN go to <i>imm</i>
JB JNAE	<i>imm</i>	Dopo un confronto di valori senza segno, salta se la destinazione era minore dell'origine. CMP <i>dst, org</i> IF <i>dst &lt; org</i> THEN go to <i>imm</i>

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione
JBE  JNA	<i>imm</i>	Dopo un confronto di valori senza segno, salta se la destinazione era minore o uguale all'origine. CMP <i>dst, org</i> IF <i>dst &lt;= org</i> THEN go to <i>imm</i>

Tabella 64.137. Salti condizionati dall'esito di un confronto con valori interi, indipendentemente dal segno.

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione
JE	<i>imm</i>	Dopo un confronto, indipendentemente dal segno, salta se la destinazione era uguale all'origine. CMP <i>dst, org</i> IF <i>dst == org</i> THEN go to <i>imm</i>
JNE	<i>imm</i>	Dopo un confronto, indipendentemente dal segno, salta se la destinazione era diversa dall'origine. CMP <i>dst, org</i> IF <i>dst != org</i> THEN go to <i>imm</i>

Tabella 64.138. Salti condizionati dall'esito di un confronto con valori interi con segno.

Nome	Operandi: <i>dst, org1, org2</i>	Descrizione
JG JNLE	<i>imm</i>	Dopo un confronto con segno, salta se la destinazione era maggiore dell'origine. CMP <i>dst, org</i> IF <i>dst &gt; org</i> THEN go to <i>imm</i>
JGE JNL	<i>imm</i>	Dopo un confronto con segno, salta se la destinazione era maggiore o uguale all'origine. CMP <i>dst, org</i> IF <i>dst &gt;= org</i> THEN go to <i>imm</i>
JL JNGE	<i>imm</i>	Dopo un confronto con segno, salta se la destinazione era minore dell'origine. CMP <i>dst, org</i> IF <i>dst &lt; org</i> THEN go to <i>imm</i>
JLE JNG	<i>imm</i>	Dopo un confronto con segno, salta se la destinazione era minore o uguale all'origine. CMP <i>dst, org</i> IF <i>dst &lt;= org</i> THEN go to <i>imm</i>

### 64.8.5 Cicli



È possibile realizzare dei cicli enumerativi attraverso delle istruzioni simili a quelle di salto condizionato, dove in pratica, dopo il blocco di istruzioni da reiterare, viene verificata la condizione e quindi, eventualmente, si ripete il ciclo. Si distinguono tre casi:



```
LOOP imm
```

```
LOOPE imm | LOOPZ imm
```

```
LOOPNE imm | LOOPNZ imm
```

In tutte le situazioni, il valore immediato che viene fornito come operando si riferisce al dislocamento dell'indirizzo di memoria da raggiungere (cosa che viene tradotta dal compilatore, sostituendo il simbolo con il numero appropriato). Il dislocamento consentito è breve ( $\pm 128$  byte), quindi non si possono realizzare cicli contenenti tante istruzioni.

In tutte le situazioni, l'istruzione decrementa prima il registro **ECX** (oppure solo **CX**, se viene dichiarato l'uso di una dimensione «breve», ovvero a soli 16 bit<sup>8</sup>) senza alterare gli indicatori, quindi verifica se tale registro è diverso da zero. Nel caso dell'istruzione '**LOOP**', il fatto che il registro, dopo il decremento di una unità, contenga ancora un valore diverso da zero, è sufficiente per far scattare la ripetizione del ciclo; nel caso di '**LOOPE**' o di '**LOOPZ**', è necessario anche che l'indicatore zero sia attivo; per converso, con '**LOOPNE**' o '**LOOPNZ**', è necessario anche che l'indicatore zero non sia attivo.

## 64.9 Esempi di programmi con strutture di controllo

Vengono mostrati esempi di programmi estremamente banali, per dimostrare il funzionamento delle strutture di controllo, basate sostanzialmente su istruzioni di salto condizionato, attraverso l'aiuto



di GDB. Per la compilazione, se si utilizza GNU AS è bene ricordare di inserire l'opzione '**--gstabs**', mentre con NASM è bene aggiungere l'opzione '**-g**', in modo da poter gestire più facilmente GDB, disponendo dei riferimenti al sorgente:

```
$ as --gstabs -o nome.o nome.s [Invio]
```

```
$ nasm -g -f elf -o nome.o nome.s [Invio]
```

### 64.9.1 Somma attraverso l'incremento unitario

«

Viene mostrato un programma che esegue la somma di due valori interi senza segno, incrementando progressivamente il primo addendo, di una unità, corrispondentemente alla riduzione di una unità del secondo. Quando il secondo addendo è stato ridotto a zero, il primo contiene il risultato della somma. Il ciclo con cui si incrementa il primo addendo è controllato dall'istruzione '**LOOP**':

```
# op1 + op2
#
.section .data
op1:    .int    0x00000007    # Intero senza segno.
op2:    .int    0x00000002    # Intero senza segno.
#
.section .text
.globl _start
#
_start:
    mov op1, %eax    # Primo addendo.
    mov op2, %ecx    # Secondo addendo.
bp1:
    cmp $0, %ecx    # Se il secondo addendo è zero, non
    je end_do_somma # esegue il ciclo di incrementi.
do_somma:
```

```

    inc %eax          # Aggiunge una unità a EAX.
    loop do_somma    # Ripete il ciclo se nel frattempo
                    # ECX non si è azzerato.
end_do_somma:
    mov %eax, %ebx   # Restituisce il valore
    mov $1, %eax    # ottenuto dalla somma.
    int $0x80       #

```

```

; op1 + op2
;
section .data
op1:    dd        0x00000007    ; Intero senza segno.
op2:    dd        0x00000002    ; Intero senza segno.
;
section .text
global _start
;
_start:
    mov eax, [op1]   ; Primo addendo.
    mov ecx, [op2]   ; Secondo addendo.
bp1:
    cmp ecx, 0       ; Se il secondo addendo è zero, non
    je end_do_somma ; esegue il ciclo di incrementi.
do_somma:
    inc eax          ; Aggiunge una unità a EAX.
    loop do_somma   ; Ripete il ciclo se nel frattempo
                    ; ECX non si è azzerato.
end_do_somma:
    mov ebx, eax     ; Restituisce il valore ottenuto
    mov eax, 1       ; dalla somma.
    int 0x80        ;

```

Come si può vedere, il ciclo di incremento è racchiuso tra i simboli **‘do\_somma’** e **‘end\_do\_somma’**; inoltre, prima di entrare nel ciclo

di somma, viene verificato che il secondo addendo sia diverso da zero, perché se è pari a zero, viene saltato il ciclo di somma, dal momento che *EAX* contiene già il valore corretto.

Viene mostrata una seconda versione del ciclo, dove si sostituisce l'istruzione '**LOOP**' con altre istruzioni di salto condizionato:

```

    cmp $0, %ecx      # Se il secondo addendo è zero, non
    je end_do_somma # esegue il ciclo di incrementi.
do_somma:
    inc %eax         # Aggiunge una unità a EAX.
    dec %ecx         # Riduce ECX di una unità.
    cmp $0, %ecx     # Se il secondo addendo è ancora diverso
    jnz do_somma     # da zero, allora ripete il ciclo.
end_do_somma:

```

```

    cmp ecx, 0       ; Se il secondo addendo è zero, non
    je end_do_somma ; esegue il ciclo di incrementi.
do_somma:
    inc eax         ; Aggiunge una unità a EAX.
    dec ecx         ; Riduce ECX di una unità.
    cmp ecx, 0     ; Se il secondo addendo è ancora diverso
    jnz do_somma   ; da zero, allora ripete il ciclo.
end_do_somma:

```

Viene mostrata una terza versione del ciclo, dove il controllo di uscita avviene solo all'inizio:

```

do_somma:
    cmp $0, %ecx     # Se il secondo addendo è zero, esce dal
    je end_do_somma # ciclo di incrementi.
    dec %ecx         # Riduce di una unità ECX.
    inc %eax         # Aggiunge una unità a EAX.
    jmp do_somma     # Ritorna all'inizio del ciclo.
end_do_somma:

```

```

do_somma:
    cmp ecx, 0      ; Se il secondo addendo è zero, esce dal
    je end_do_somma ; ciclo di incrementi.
    dec ecx        ; Riduce di una unità ECX.
    inc eax        ; Aggiunge una unità a EAX.
    jmp do_somma   ; Ritorna all'inizio del ciclo.
end_do_somma:

```

## 64.9.2 Moltiplicazione attraverso la somma

Viene mostrato un programma che esegue la moltiplicazione di due valori interi senza segno, sommando progressivamente il moltiplicando a un registro che inizialmente è pari a zero, corrispondentemente alla riduzione di una unità del moltiplicatore. Quando il moltiplicatore è stato ridotto a zero, il registro che viene incrementato contiene il risultato della moltiplicazione. Il ciclo è controllato dall'istruzione **'LOOP'**:

```

# op1 * op2
#
.section .data
op1:    .int    0x00000007    # Intero senza segno.
op2:    .int    0x00000003    # Intero senza segno.
#
.section .text
.globl _start
#
_start:
    mov op1, %eax    # Moltiplicando.
    mov op2, %ecx    # Moltiplicatore.
    mov $0, %ebx     # Risultato.
bp1:
    cmp $0, %ecx     # Se il moltiplicatore è zero,
    je end_do_moltiplica # non esegue il ciclo di somme.

```

```
do_moltiplica:
    add %eax, %ebx        # Aggiunge il moltiplicando al
                          # risultato.
    loop do_moltiplica   # Ripete il ciclo se nel frattempo
                          # ECX non si è azzerato.
end_do_moltiplica:
    mov $1, %eax         # Restituisce il valore ottenuto
    int $0x80           # dalla moltiplicazione.
```

```
; op1 * op2
;
section .data
op1:    dd    0x00000007    ; Intero senza segno.
op2:    dd    0x00000003    ; Intero senza segno.
;
section .text
global _start
;
_start:
    mov eax, [op1]        ; Moltiplicando.
    mov ecx, [op2]        ; Moltiplicatore.
    mov ebx, 0           ; Risultato.
bp1:
    cmp ecx, 0           ; Se il moltiplicatore è zero,
    je end_do_moltiplica ; non esegue il ciclo di somme.
do_moltiplica:
    add ebx, eax         ; Aggiunge il moltiplicando al
                          ; risultato.
    loop do_moltiplica   ; Ripete il ciclo se nel frattempo
                          ; ECX non si è azzerato.
end_do_moltiplica:
    mov eax, 1           ; Restituisce il valore ottenuto
    int 0x80            ; dalla moltiplicazione.
```

Viene mostrata una seconda versione del ciclo, dove si sostituisce l'istruzione '**LOOP**' con altre istruzioni di salto condizionato:

```

    cmp $0, %ecx          # Se il moltiplicatore è zero,
    je end_do_moltiplica # non esegue il ciclo di somme.
do_moltiplica:
    add %eax, %ebx        # Aggiunge il moltiplicando al
                          # risultato.
    cmp $0, %ecx          # Se il moltiplicatore è ancora
    jnz do_moltiplica     # diverso da zero, allora ripete
                          # il ciclo.
end_do_moltiplica:

```

```

    cmp ecx, 0           ; Se il moltiplicatore è zero,
    je end_do_moltiplica ; non esegue il ciclo di somme.
do_moltiplica:
    add ebx, eax         ; Aggiunge il moltiplicando al
                          ; risultato.
    cmp ecx, 0           ; Se il moltiplicatore è ancora
    jnz do_moltiplica    ; diverso da zero, allora ripete
                          ; il ciclo.
end_do_moltiplica:

```

Viene mostrata una terza versione del ciclo, dove il controllo di uscita avviene solo all'inizio:

```

do_moltiplica:
    cmp $0, %ecx          # Se il moltiplicatore è zero,
    je end_do_moltiplica # esce dal ciclo di somme.
    dec %ecx              # Riduce di una unità il
                          # moltiplicatore.
    add %eax, %ebx        # Aggiunge il moltiplicando al
                          # risultato.
    jmp do_moltiplica     # Ritorna all'inizio del ciclo.
end_do_moltiplica:

```

```

do_moltiplica:
    cmp ecx, 0          ; Se il moltiplicatore è zero,
    je end_do_moltiplica ; esce dal ciclo di somme.
    dec ecx            ; Riduce di una unità il
                        ; moltiplicatore.
    add ebx, eax       ; Aggiunge il moltiplicando al
                        ; risultato.
    jmp do_moltiplica  ; Ritorna all'inizio del ciclo.
end_do_moltiplica:

```

### 64.9.3 Divisione attraverso la sottrazione



Viene mostrato un programma che esegue la divisione di due valori interi senza segno, sottraendo progressivamente il divisore a un registro che inizialmente è pari al valore del dividendo, corrispondentemente all'incremento di una unità del risultato (a partire da zero). Quando il registro che viene ridotto, di volta in volta, del valore del divisore, diventa minore del divisore, la divisione termina. Viene proposta una sola versione, con un ciclo controllato da una condizione iniziale:

```

# op1 / op2
#
.section .data
op1:    .int    33      # Intero senza segno.
op2:    .int    12      # Intero senza segno.
#
.section .text
.globl _start
#
_start:
    mov op1,  %eax     # Dividendo.
    mov op2,  %ecx     # Divisore.

```



```

    mov $0,    %ebx    # Risultato.
    mov %eax, %edx    # Resto.
do_dividi:
    cmp %ecx, %edx    # Se il resto è minore del
    jb end_do_dividi # divisore, conclude il ciclo di
                    # sottrazioni.

    sub %ecx, %edx    # Sottrae al resto il divisore.
    inc %ebx          # Incrementa il risultato di una
                    # unità.

    jmp do_dividi     # Torna all'inizio del ciclo.
end_do_dividi:
    mov $1,    %eax    # Restituisce il valore ottenuto
    int $0x80         # dalla moltiplicazione.

```

```

; op1 / op2
;
section .data
op1:    dd    33    ; Intero senza segno.
op2:    dd    12    ; Intero senza segno.
;
section .text
global _start
;
_start:
    mov eax, [op1]    ; Dividendo.
    mov ecx, [op2]    ; Divisore.
    mov ebx, 0        ; Risultato.
    mov edx, eax      ; Resto.
do_dividi:
    cmp edx, ecx      ; Se il resto è minore del
    jb end_do_dividi ; divisore, conclude il ciclo di
                    ; sottrazioni.

    sub edx, ecx      ; Sottrae al resto il divisore.
    inc ebx           ; Incrementa il risultato di una
                    ; unità.

```

```

    jmp do_dividi          ; Torna all'inizio del ciclo.
end_do_dividi:
    mov eax, 1            ; Restituisce il valore ottenuto
    int 0x80             ; dalla moltiplicazione.

```

## 64.9.4 Elevamento a potenza



Viene mostrato un programma che calcola la potenza di due numeri interi senza segno, moltiplicando progressivamente la base, corrispondentemente al decremento di un contatore che parte dal valore dell'esponente. Quando il contatore raggiunge lo zero, il ciclo di moltiplicazioni termina e il risultato della potenza viene emesso come valore di uscita, ammesso che sia abbastanza piccolo da poter essere rappresentato:

```

# op1 / op2
#
.section .data
op1:    .int    5      # Intero senza segno.
op2:    .int    3      # Intero senza segno.
#
.section .text
.globl _start
#
_start:
    mov op1, %esi     # Base.
    mov op2, %edi     # Esponente.
    mov $0, %edx      # Risultato: EDX:EAX
    mov $1, %eax      #
    mov %edi, %ecx    # Contatore.
do_potenza:
    cmp $0, %ecx      # Se il contatore ha raggiunto lo
    jz end_do_potenza # zero, conclude il ciclo di

```

```

                                # moltiplicazioni.
    mul %esi                    # EDX:EAX := EAX*ESI.
    dec %ecx                    # Riduce di una unità il contatore.
    jmp do_potenza             # Torna all'inizio del ciclo.
end_do_potenza:
    mov %eax, %ebx             # Restituisce il valore della potenza,
    mov $1, %eax              # ammesso che sia abbastanza piccolo
    int $0x80                 # da poter essere rappresentato come
                                # valore di uscita.

```

```

; op1 / op2
;
section .data
op1:    dd    5        ; Intero senza segno.
op2:    dd    3        ; Intero senza segno.
;
section .text
global _start
;
_start:
    mov esi, [op1]        ; Base.
    mov edi, [op2]        ; Esponente.
    mov edx, 0            ; Risultato: EDX:EAX
    mov eax, 1            ;
    mov ecx, edi          ; Contatore.
do_potenza:
    cmp ecx, 0            ; Se il contatore ha raggiunto lo
    jz end_do_potenza    ; zero, conclude il ciclo di
                                ; moltiplicazioni.
    mul esi                ; EDX:EAX := EAX*ESI.
    dec ecx                ; Riduce di una unità il contatore.
    jmp do_potenza        ; Torna all'inizio del ciclo.
end_do_potenza:
    mov ebx, eax          ; Restituisce il valore della potenza,
    mov eax, 1            ; ammesso che sia abbastanza piccolo

```

```
int 0x80          ; da poter essere rappresentato come
                  ; valore di uscita.
```

## 64.9.5 Moltiplicazione attraverso lo scorrimento e la somma

«

Viene mostrato un programma che calcola il prodotto di due numeri interi senza segno, sommando progressivamente il moltiplicando che viene fatto scorrere verso sinistra. Per comprendere il procedimento occorre fare mente locale al modo in cui la moltiplicazione verrebbe eseguita a mano: il moltiplicando viene sommato al risultato (che inizialmente è pari a zero) se la cifra meno significativa del moltiplicatore è pari a uno; successivamente il moltiplicando viene fatto scorrere verso sinistra di una posizione e lo si somma nuovamente al risultato se la cifra successiva del moltiplicatore è pari a uno; quindi si continua fino a che si esauriscono le cifre del moltiplicatore.

Nel programma mostrato, durante il ciclo di somme, il moltiplicatore viene fatto scorrere verso destra, in modo da poter verificare il valore della cifra espulsa attraverso l'indicatore di riporto (*carry*). Così facendo, quando il moltiplicatore è pari a zero, il ciclo di somme può terminare.

```
# op1 * op2
#
.section .data
op1:    .byte    0x07, 0x00 # little endian = 0x0007
                                # intero senza segno.
op2:    .byte    0x03, 0x00 # little endian = 0x0003
                                # intero senza segno.
#
.section .text
```

```

.globl _start
#
_start:
    movzx op1, %edx    # Moltiplicando.
    movzx op2, %ecx    # Moltiplicatore.
    mov    $0, %eax    # Risultato.
do_mol:
    cmp    $0, %ecx    # Se il moltiplicatore è pari a
    jz     end_do_mol  # zero, il ciclo deve terminare.
    shr   $1, %ecx    # Fa scorrere a destra il
    jnc   end_do_mol_somma # moltiplicatore e se
                                # l'indicatore di riporto non è
                                # attivo, salta la somma.
do_mol_somma:
    add   %edx, %eax   # Aggiunge il moltiplicando al
                                # risultato.
end_do_mol_somma:
    shl   $1, %edx    # Fa scorrere il moltiplicando
                                # a sinistra.
    jmp   do_mol      # Torna all'inizio del ciclo.
end_do_mol:
    mov   %eax, %ebx   # Restituisce il valore del
    mov   $1, %eax    # prodotto, ammesso che sia
    int   $0x80       # abbastanza piccolo da poter
                                # essere rappresentato come
                                # valore di uscita.

```

```

; op1 * op2
;
section .data
op1:    dw      0x0007    ; Intero senza segno.
op2:    dw      0x0003    ; Intero senza segno.
;
section .text
global _start

```

```
;
_start:
    movzx edx, word [op1] ; Moltiplicando.
    movzx ecx, word [op2] ; Moltiplicatore.
    mov  eax,      0      ; Risultato.
do_mol:
    cmp  ecx,      0      ; Se il moltiplicatore è pari a
    jz   end_do_mol     ; zero, il ciclo deve terminare.
    shr  ecx,      1      ; Fa scorrere a destra il
    jnc  end_do_mol_somma ; moltiplicatore e se
                                ; l'indicatore di riporto non è
                                ; attivo, salta la somma.
do_mol_somma:
    add  eax,      edx    ; Aggiunge il moltiplicando al
                                ; risultato.
end_do_mol_somma:
    shl  edx,      1      ; Fa scorrere il moltiplicando a
                                ; sinistra.
    jmp  do_mol         ; Torna all'inizio del ciclo.
end_do_mol:
    mov  ebx,      eax    ; Restituisce il valore del
    mov  eax,      1      ; prodotto, ammesso che sia
    int  0x80           ; abbastanza piccolo da poter
                                ; essere rappresentato come
                                ; valore di uscita.
```

Come si può vedere, moltiplicando e moltiplicatore sono variabili da 16 bit, in modo da avere la certezza che il risultato del prodotto sia contenibile in un registro. Nel caso del primo listato, fatto per GNU AS, lo spazio in memoria viene dichiarato come sequenza di due byte, perché manca la possibilità di definire esplicitamente un intero «corto».

## 64.9.6 Conteggio dei bit a uno

Viene proposto un esempio di programma che conta quanti bit a uno compongono un certo valore numerico. Per farlo, si usa lo scorrimento (in questo caso è a destra, ma non farebbe differenza) e quindi viene contato il riporto (l'indicatore di riporto è attivo se fuoriesce una cifra a uno).

```
#
.section .data
op1:      .int 44
#
.section .text
.globl _start
#
_start:
    mov    op1, %eax      # EAX contiene il numero di cui
                        # contare i bit a 1.
    mov    $0, %ecx      # ECX è il contatore dei bit a uno.
do_conta:
    cmp    $0, %eax      # Se EAX è a zero, il conteggio dei
    jz     end_do_conta  # bit si conclude.
    shr   $1, %eax       # Fa scorrere a destra EAX.
    adc   $0, %ecx       # ECX = ECX + 0 + carry.
    jmp   do_conta      # Riprende il ciclo.
end_do_conta:
    mov   %ecx, %ebx     # Restituisce la quantità di bit.
    mov   $1, %eax      #
    int   $0x80         #
```

```
;
section .data
op1:      dd 44
;
section .text
```

```

global _start
;
_start:
    mov    eax, [op1]    ; EAX contiene il numero di cui
                        ; contare i bit a 1.
    mov    ecx, 0        ; ECX è il contatore dei bit a uno.
do_conta:
    cmp    eax, 0        ; Se EAX è a zero, il conteggio dei
    jz     end_do_conta ; bit si conclude.
    shr   eax, 1        ; Fa scorrere a destra EAX.
    adc   ecx, 0        ; ECX = ECX + 0 + carry.
    jmp   do_conta      ; Riprende il ciclo.
end_do_conta:
    mov    ebx, ecx      ; Restituisce la quantità di bit.
    mov    eax, 1        ;
    int   0x80          ;

```

Viene proposto un metodo alternativo che utilizza la sottrazione e l'abbinamento con l'operatore logico AND (è descritto nella sezione [63.5.10](#)):

```

#
.section .data
op1:      .int 44
#
.section .text
.globl _start
#
_start:
    mov    op1, %eax     # EAX contiene il numero di cui
                        # contare i bit a 1.
    mov    $0, %ecx     # ECX è il contatore dei bit a uno.
do_conta:
    cmp    $0, %eax     # Se EAX è a zero, il conteggio
    jz     end_do_conta # dei bit si conclude.

```



```

    mov    %eax, %edx    # Fa una copia in EDX.
    dec    %eax         # Decrementa EAX di una unità.
    and    %edx, %eax   # EAX := EAX AND EDX.
    inc    %ecx         # ECX++
    jmp    do_conta     # Riprende il ciclo.
end_do_conta:
    mov    %ecx, %ebx   # Restituisce la quantità di bit.
    mov    $1, %eax    #
    int    $0x80       #

```

```

;
section .data
opl:      dd 44
;
section .text
global _start
;
_start:
    mov    eax, [opl]   ; EAX contiene il numero di cui
                       ; contare i bit a 1.
    mov    ecx, 0      ; ECX è il contatore dei bit a uno.
do_conta:
    cmp    eax, 0      ; Se EAX è a zero, il conteggio dei
    jz    end_do_conta ; bit si conclude.
    mov    edx, eax    ; Fa una copia in EDX.
    dec    eax         ; Decrementa EAX di una unità.
    and    eax, edx    ; EAX := EAX AND EDX.
    inc    ecx         ; ECX++
    jmp    do_conta    ; Riprende il ciclo.
end_do_conta:
    mov    ebx, ecx    ; Restituisce la quantità di bit.
    mov    eax, 1      ;
    int    0x80        ;

```



```
8      #
9      .section .text
10     .globl _start
11     #-----
12     _start:
13         movzx op1, %edx      # Moltiplicando.
14         movzx op2, %ecx      # Moltiplicatore.
15         mov $0, %eax         # Risultato.
16     bp1:
17         call f_mol          # Esegue la moltiplicazione:
18                                     # EAX = EDX * ECX
19     bp2:
20         mov %eax, %ebx       # Restituisce il valore del
21         mov $1, %eax         # prodotto, ammesso che sia
22         int $0x80            # abbastanza piccolo da poter
23                                     # essere rappresentato come
24                                     # valore di uscita.
25     #-----
26     # Moltiplicazione di due numeri interi.
27     # EAX = EDX * ECX
28     # I registri EDX e ECX vengono alterati durante il
29     # procedimento.
30     #
31     f_mol:
32         cmp $0, %ecx         # Se il moltiplicatore è
33         jz f_end_mol         # pari a zero, il ciclo
34                                     # deve terminare.
35         shr $1, %ecx         # Fa scorrere a destra il
36         jnc end_do_somma     # moltiplicatore e se
37                                     # l'indicatore di riporto
38                                     # non è attivo, salta
39                                     # la somma.
40     do_somma:
41         add %edx, %eax       # Aggiunge il moltiplicando
```

```

42                                     # al risultato.
43  end_do_somma:
44      shl    $1,    %edx                # Fa scorrere il
45                                     # moltiplicando a sinistra.
46      jmp    f_mol                    # Torna all'inizio della
47                                     # funzione.
48  f_end_mol:
49      ret                            # Torna all'istruzione
50                                     # successiva alla chiamata.

```

```

1   ; op1 * op2
2   ;
3   section .data
4   op1:    dw    0x0007    ; Intero senza segno.
5   op2:    dw    0x0003    ; Intero senza segno.
6   ;
7   section .text
8   global _start
9   ;-----
10  _start:
11      movzx edx, word [op1]    ; Moltiplicando.
12      movzx ecx, word [op2]    ; Moltiplicatore.
13      mov    eax,    0        ; Risultato.
14  bp1:
15      call  f_mol        ; Esegue la moltiplicazione:
16                                     ; EAX = EDX * ECX
17  bp2:
18      mov    ebx,    eax      ; Restituisce il valore
19      mov    eax,    1        ; del prodotto, ammesso che
20      int   0x80             ; sia abbastanza piccolo da
21                                     ; poter essere rappresentato
22                                     ; come valore di uscita.
23  ;-----
24  ; Moltiplicazione di due numeri interi.

```

```
25 ; EAX = EDX * ECX
26 ; I registri EDX e ECX vengono alterati durante il
27 ; procedimento.
28 ;
29 f_mol:
30     cmp     ecx,      0      ; Se il moltiplicatore è
31     jz     f_end_mol      ; pari a zero, il ciclo deve
32                                     ; terminare.
33     shr     ecx,      1      ; Fa scorrere a destra il
34     jnc     end_do_somma   ; moltiplicatore e se
35                                     ; l'indicatore di riporto
36                                     ; non è attivo, salta
37                                     ; la somma.
38 do_somma:
39     add     eax,      edx    ; Aggiunge il moltiplicando
40                                     ; al risultato.
41 end_do_somma:
42     shl     edx,      1      ; Fa scorrere il
43                                     ; moltiplicando a sinistra.
44     jmp     f_mol         ; Torna all'inizio della
45                                     ; funzione.
46 f_end_mol:
47     ret                ; Torna all'istruzione
48                                     ; successiva alla chiamata.
```

Si osservi che il programma viene eseguito a partire dalla riga 11 e che si conclude alla riga 19. La subroutine, ovvero la funzione che esegue la moltiplicazione, si trova in un gruppo di istruzioni successive, il cui inizio è segnalato dal simbolo '**f\_mol**'.

La funzione riceve il moltiplicando e il moltiplicatore attraverso due registri, utilizzando a sua volta un altro registro per restituire il risultato.

## 64.10.2 Salvataggio dei registri prima della chiamata



Ogni funzione ha la necessità di elaborare dati senza interferire con il resto del programma; in pratica, ogni funzione deve poter utilizzare i registri con una certa libertà. Nell'esempio precedente vengono utilizzati dei registri per passare alla funzione i valori da moltiplicare, ma all'interno della funzione il contenuto dei registri viene modificato. Per lo scopo dell'esempio, il fatto che i registri *ECX* e *EDX* vengano modificati, non produce effetti collaterali, ma in un programma più complesso potrebbe essere il caso di salvaguardare il contenuto originale dei registri prima della chiamata di una funzione. L'esempio successivo mostra una variante del codice contenuto tra i simboli 'bp1' e 'bp2', allo scopo di conservare una copia dei registri che contengono il moltiplicando e il moltiplicatore, per ripristinarla dopo la chiamata:

```
bp1:
    push    %ecx    # Salva il moltiplicatore nella pila.
    push    %edx    # Salva il moltiplicando nella pila.
    call   f_mol   # Esegue la moltiplicazione: EAX = EDX * ECX
    pop     %edx    # Recupera il moltiplicando dalla pila.
    pop     %ecx    # Recupera il moltiplicatore dalla pila.
bp2:
```

```
bp1:
    push    ecx     ; Salva il moltiplicatore nella pila.
    push    edx     ; Salva il moltiplicando nella pila.
    call   f_mol   ; Esegue la moltiplicazione: EAX = EDX * ECX
    pop     edx     ; Recupera il moltiplicando dalla pila.
    pop     ecx     ; Recupera il moltiplicatore dalla pila.
bp2:
```

Come si può intendere, il recupero dei valori dalla pila deve avvenire in senso inverso.

### 64.10.3 Passaggio di parametri attraverso la pila

Per rendere più libera la funzione dal programma chiamante, conviene utilizzare la stessa pila per il passaggio dei parametri. In pratica, dopo avere salvato i registri che contengono dati importanti (ammesso che ciò vada fatto), occorre accumulare nella pila gli argomenti della chiamata della funzione, secondo un ordine convenuto per la funzione stessa. All'interno della funzione, poi, si vanno a pescare questi valori per usarli nell'elaborazione.

```
# op1 * op2
#
.section .data
op1:      .byte    0x07, 0x00 # little endian = 0x0005
                                # intero senza segno.
op2:      .byte    0x03, 0x00 # little endian = 0x0003
                                # intero senza segno.

#
.section .text
.globl _start
#-----
_start:
    movzx op1,    %edx # Moltiplicando.
    movzx op2,    %ecx # Moltiplicatore.
bp1:
    push  %ecx      # Salva il moltiplicatore nella pila.
    push  %edx      # Salva il moltiplicando nella pila.
    #
    push  %ecx      # Inserisce il secondo parametro nella
                    # pila.
    push  %edx      # Inserisce il primo parametro nella
```

```
                                # pila.
call  f_mol                    # Esegue la moltiplicazione:
                                # EAX = EDX * ECX
add   $4,    %esp              # Espelle il primo parametro della
                                # chiamata.
add   $4,    %esp              # Espelle il secondo parametro della
                                # chiamata.
#
pop   %edx                    # Recupera il moltiplicando dalla pila.
pop   %ecx                    # Recupera il moltiplicatore dalla
                                # pila.
bp2:
mov   %eax, %ebx              # Restituisce il valore del prodotto,
mov   $1,    %eax              # ammesso che sia abbastanza piccolo
int   $0x80                    # da poter essere rappresentato come
                                # valore di uscita.

#-----
# Moltiplicazione di due numeri interi.
# f_mol (a, b) => EAX
# EAX = a * b
#
f_mol:
mov   4(%esp), %edx           # Copia il primo parametro in EDX.
mov   8(%esp), %ecx           # Copia il secondo parametro in ECX.
mov   $0,     %eax            # Azzera EAX per sicurezza.
do_mol:
cmp   $0,     %ecx            # Se il moltiplicatore è pari a
jz   end_do_mol              # zero, il ciclo deve terminare.
shr  $1,     %ecx            # Fa scorrere a destra il
jnc  end_do_somma            # moltiplicatore e se l'indicatore
                                # di riporto non è attivo, salta
                                # la somma.
do_somma:
add  %edx,    %eax            # Aggiunge il moltiplicando al
```



```

                                # risultato.
end_do_somma:
    shl    $1,    %edx # Fa scorrere il moltiplicando a
                                # sinistra.
    jmp    do_mol    # Torna all'inizio del ciclo di
                                # moltiplicazione.
end_do_mol:
    ret        # Torna all'istruzione successiva
                                # alla chiamata.

```

```

; op1 * op2
;
section .data
op1:    dw      0x0007    ; Intero senza segno.
op2:    dw      0x0003    ; Intero senza segno.
;
section .text
global _start
;-----
_start:
    movzx  edx, word [op1]    ; Moltiplicando.
    movzx  ecx, word [op2]    ; Moltiplicatore.
bp1:
    push  ecx    ; Salva il moltiplicatore nella pila.
    push  edx    ; Salva il moltiplicando nella pila.
    ;
    push  ecx    ; Inserisce il secondo parametro nella
                    ; pila.
    push  edx    ; Inserisce il primo parametro nella
                    ; pila.
    call  f_mol    ; Esegue la moltiplicazione:
                    ; EAX = EDX * ECX
    add   esp, 4    ; Espelle il primo parametro della
                    ; chiamata.

```

```

    add    esp, 4    ; Espelle il secondo parametro della
                    ; chiamata.

    ;
    pop    edx      ; Recupera il moltiplicando dalla pila.
    pop    ecx      ; Recupera il moltiplicatore dalla pila.
bp2:
    mov    ebx, eax ; Restituisce il valore del prodotto,
    mov    eax, 1   ; ammesso che sia abbastanza piccolo
    int   0x80     ; da poter essere rappresentato come
                    ; valore di uscita.

;-----
; Moltiplicazione di due numeri interi.
; f_mol (a, b) => EAX
; EAX = a * b
;
f_mol:
    mov    edx, [esp+4] ; Copia il primo parametro in EDX.
    mov    ecx, [esp+8] ; Copia il secondo parametro in ECX.
    mov    eax, 0      ; Azzera EAX per sicurezza.
    ;
do_mol:
    cmp    ecx, 0      ; Se il moltiplicatore è pari a zero,
    jz     end_do_mol  ; il ciclo deve terminare.
    shr   ecx, 1      ; Fa scorrere a destra il
    jnc   end_do_somma ; moltiplicatore e se l'indicatore di
                    ; riporto non è attivo, salta
                    ; la somma.
do_somma:
    add    eax, edx    ; Aggiunge il moltiplicando al
                    ; risultato.
end_do_somma:
    shl   edx, 1      ; Fa scorrere il moltiplicando a
                    ; sinistra.
    jmp   do_mol      ; Torna all'inizio del ciclo di

```

```

                                ; moltiplicazione.
end_do_mol:
    ret                            ; Torna all'istruzione successiva
                                ; alla chiamata.

```

Come si vede, rispetto alla versione precedente dello stesso programma, il risultato del prodotto, calcolato all'interno della funzione, continua a essere restituito attraverso il registro *EAX*, ma sarebbe stato possibile accumulare nella pila, prima della chiamata, un valore in più, da considerare poi come il risultato generato dalla funzione.

All'inizio della funzione vengono recuperati i valori che costituiscono gli argomenti della chiamata. Tale operazione viene eseguita attraverso queste istruzioni:

```

mov 4(%esp), %edx # Copia il primo parametro in EDX.
mov 8(%esp), %ecx # Copia il secondo parametro in ECX.

```

```

mov edx, [esp+4] ; Copia il primo parametro in EDX.
mov ecx, [esp+8] ; Copia il secondo parametro in ECX.

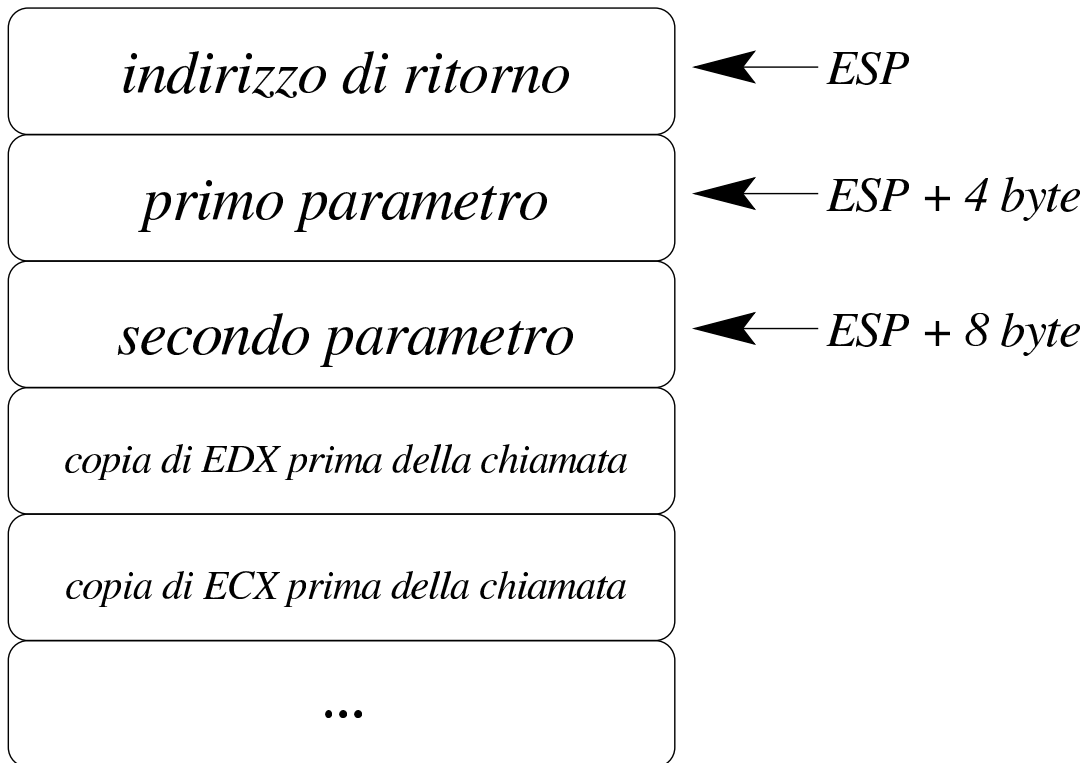
```

L'operando '*4(%esp)*', ovvero '*[esp+4]*', individua l'indirizzo di memoria corrispondente al valore del registro *ESP*, più quattro byte.

Il registro *ESP* è l'indice della pila (*stack pointer*) che punta all'ultimo elemento presente (quello in cima alla pila). Nei sistemi Unix (compresi i sistemi GNU) la pila parte da una posizione elevata della memoria e «cresce» utilizzando indirizzi che invece decrescono. Pertanto, considerato che l'ultimo elemento della pila è l'indirizzo di ritorno, l'elemento immediatamente precedente lo si raggiunge quattro byte dopo (32 bit) e quello ancora precedente si trova otto

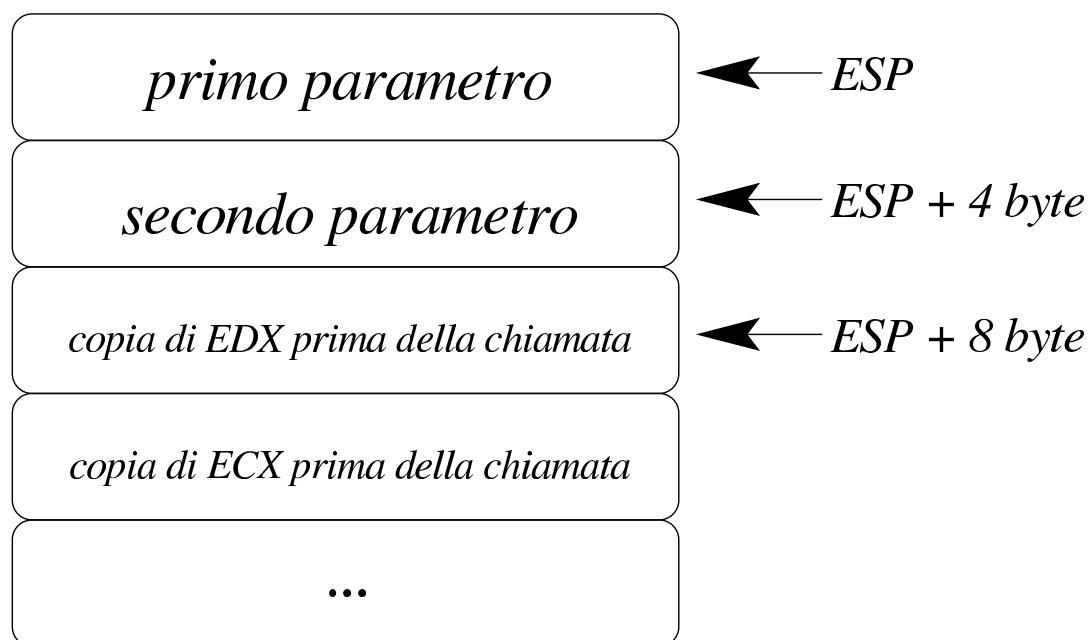
byte dopo la posizione finale.

Figura 64.169. Situazione della pila in corrispondenza del simbolo '`f_mol`'.



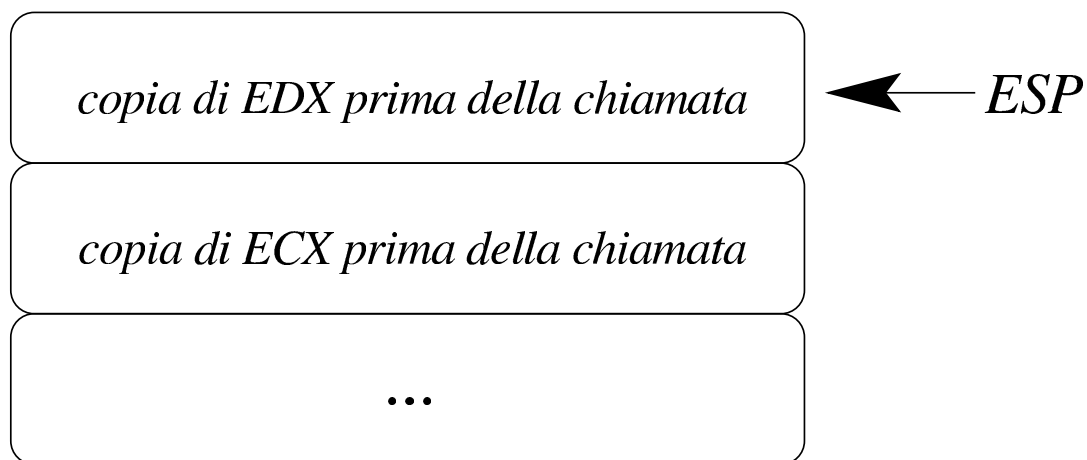
La funzione elabora il prodotto dei valori forniti come argomento e ne lascia il risultato nel registro *EAX*. Al ritorno, la pila si presenta come si vede nella figura successiva:

Figura 64.170. Situazione della pila immediatamente dopo la chiamata della funzione.



Come si vede, occorre espellere dalla pila i parametri usati per la chiamata. Dal momento che non c'è bisogno di rileggere il loro valore, ci si limita a decrementare l'indice della pila, ovvero si incrementa il valore del registro *ESP* a gruppi di quattro byte alla volta.

Figura 64.171. Situazione della pila dopo l'espulsione dei parametri della chiamata.



Naturalmente, considerato che la funzione non altera i valori ac-

cumulati nella pila, la chiamata potrebbe essere semplificata un po':

```

bp1:
    push    %ecx    # Salva ECX, inserendolo come secondo
                  # parametro nella pila.
    push    %edx    # Salva EDX, inserendolo come primo
                  # parametro nella pila.
    call   f_mol   # Esegue la moltiplicazione: EAX = EDX * ECX
    pop     %edx    # Recupera EDX dalla pila.
    pop     %ecx    # Recupera ECX dalla pila.
bp2:

```

```

bp1:
    push    ecx     ; Salva ECX, inserendolo come secondo
                  ; parametro nella pila.
    push    edx     ; Salva EDX, inserendolo come primo
                  ; parametro nella pila.
    call   f_mol   ; Esegue la moltiplicazione: EAX = EDX * ECX
    pop     edx     ; Recupera EDX dalla pila.
    pop     ecx     ; Recupera ECX dalla pila.
bp2:

```

#### 64.10.4 Utilizzo del registro «EBP»

«

Perché una funzione possa gestire delle variabili «locali», ovvero tali da avere un campo di azione limitato alla funzione stessa, senza lasciare tracce al ritorno dalla chiamata, si deve usare la pila aggiungendovi altri elementi. Questo fatto complica l'accesso ai parametri della chiamata, perché durante l'esecuzione delle istruzioni della funzione, l'indice della pila può spostarsi. A questo proposito, all'inizio di una funzione, conviene conservare una copia del registro *ESP* in un altro registro apposito: *EBP* (*base pointer*). In pra-

tica, l'indice contenuto in **EBP** dovrebbe essere sempre usato per rappresentare la posizione in cui si trova l'indirizzo di ritorno della funzione in cui ci si trova.

Viene riproposto il programma già presentato nella sezione precedente, con le semplificazioni già descritte a proposito della chiamata e con le modifiche relative all'uso del registro **EBP**.

```
# op1 * op2
#
.section .data
op1:    .byte    0x07, 0x00 # little endian = 0x0005
                                # intero senza segno.
op2:    .byte    0x03, 0x00 # little endian = 0x0003
                                # intero senza segno.

#
.section .text
.globl _start
#-----
_start:
    movzx op1,    %edx # Moltiplicando.
    movzx op2,    %ecx # Moltiplicatore.
    mov    $0,    %eax # Risultato.
bp1:
    push  %ebp      # Salva il registro EBP
                                # prima della chiamata.
    push  %ecx      # Inserisce il moltiplicando nella
                                # pila.
    push  %edx      # Inserisce il moltiplicatore nella
                                # pila.
    call  f_mol     # Esegue la moltiplicazione:
                                # EAX = EDX * ECX
    pop   %edx      # Recupera il moltiplicando dalla pila.
    pop   %ecx      # Recupera il moltiplicatore dalla
                                # pila.
```





```

mov %ebp, %esp # Ripristina ESP, espellendo
                # le variabili locali.
ret            # Torna all'istruzione successiva
                # alla chiamata.

```

```

; op1 * op2
;
section .data
op1:  dw      0x0007      ; Intero senza segno.
op2:  dw      0x0003      ; Intero senza segno.
;
section .text
global _start
;-----
_start:
    movzx edx, word [op1]  ; Moltiplicando.
    movzx ecx, word [op2]  ; Moltiplicatore.
bp1:
    push ebp           ; Salva il registro EBP prima
                        ; della chiamata.
    push ecx             ; Inserisce il moltiplicando nella pila.
    push edx             ; Inserisce il moltiplicatore nella pila.
    call f_mol           ; Esegue la moltiplicazione: EAX = EDX * ECX
    pop  edx             ; Recupera il moltiplicando dalla pila.
    pop  ecx             ; Recupera il moltiplicatore dalla pila.
    pop  ebp           ; Recupera il registro EBP dopo
                        ; la chiamata.
bp2:
    mov  ebx,  eax       ; Restituisce il valore del prodotto,
    mov  eax,  1         ; ammesso che sia abbastanza piccolo
    int  0x80           ; da poter essere rappresentato come
                        ; valore di uscita.
;-----
; Moltiplicazione di due numeri interi.

```

```

; f_mol (a, b) => EAX
; EAX = a * b
;
f_mol:
    mov    ebp,    esp    ; Copia ESP in EBP.
    mov    edx,    [ebp+4] ; Copia il primo parametro in EDX.
    mov    ecx,    [ebp+8] ; Copia il secondo parametro
                        ; in ECX.
    mov    eax,    0      ; Azzera EAX per sicurezza.
    ;
do_mol:
    cmp    ecx,    0      ; Se il moltiplicatore è pari a
    jz     end_do_mol    ; zero, il ciclo deve terminare.
    shr   ecx,    1      ; Fa scorrere a destra il
                        ; moltiplicatore e se l'indicatore
    jnc   end_do_somma  ; di riporto non è attivo, salta
                        ; la somma.
do_somma:
    add   eax,    edx    ; Aggiunge il moltiplicando al
                        ; risultato.
end_do_somma:
    shl   edx,    1      ; Fa scorrere il moltiplicando a
                        ; sinistra.
    jmp   do_mol        ; Torna all'inizio del ciclo di
                        ; moltiplicazione.
end_do_mol:
    mov   esp,    ebp    ; Ripristina ESP, espellendo
                        ; le variabili locali.
    ret                                ; Torna all'istruzione successiva
                        ; alla chiamata.

```

Nei listati appena mostrati si vede che **EBP** viene salvato prima della chiamata e ripristinato successivamente. Esiste però un altro modo, più diffuso, per cui il registro **EBP** va salvato nella pila all'inizio

della funzione, con le conseguenze che ciò comporta:

```

...
bp1:
    push    %ecx    # Inserisce il moltiplicando nella pila.
    push    %edx    # Inserisce il moltiplicatore nella pila.
    call   f_mol   # Esegue la moltiplicazione: EAX = EDX * ECX
    pop     %edx    # Recupera il moltiplicando dalla pila.
    pop     %ecx    # Recupera il moltiplicatore dalla pila.
bp2:
...
f_mol:
    push    %ebp      # Salva il registro EBP.
bp3:
    mov     %esp,    %ebp    # Copia ESP in EBP.
    mov     8(%ebp), %edx    # Copia il primo parametro in EDX.
    mov     12(%ebp), %ecx   # Copia il secondo parametro
                             # in ECX.
    mov     $0,     %eax    # Azzera EAX per sicurezza.
...
end_do_mol:
    mov     %ebp,    %esp    # Ripristina ESP, espellendo le
                             # variabili locali.
    pop     %ebp      # Riporta il registro EBP allo
                             # stato precedente.
    ret                               # Torna all'istruzione successiva
                             # alla chiamata.

```

```

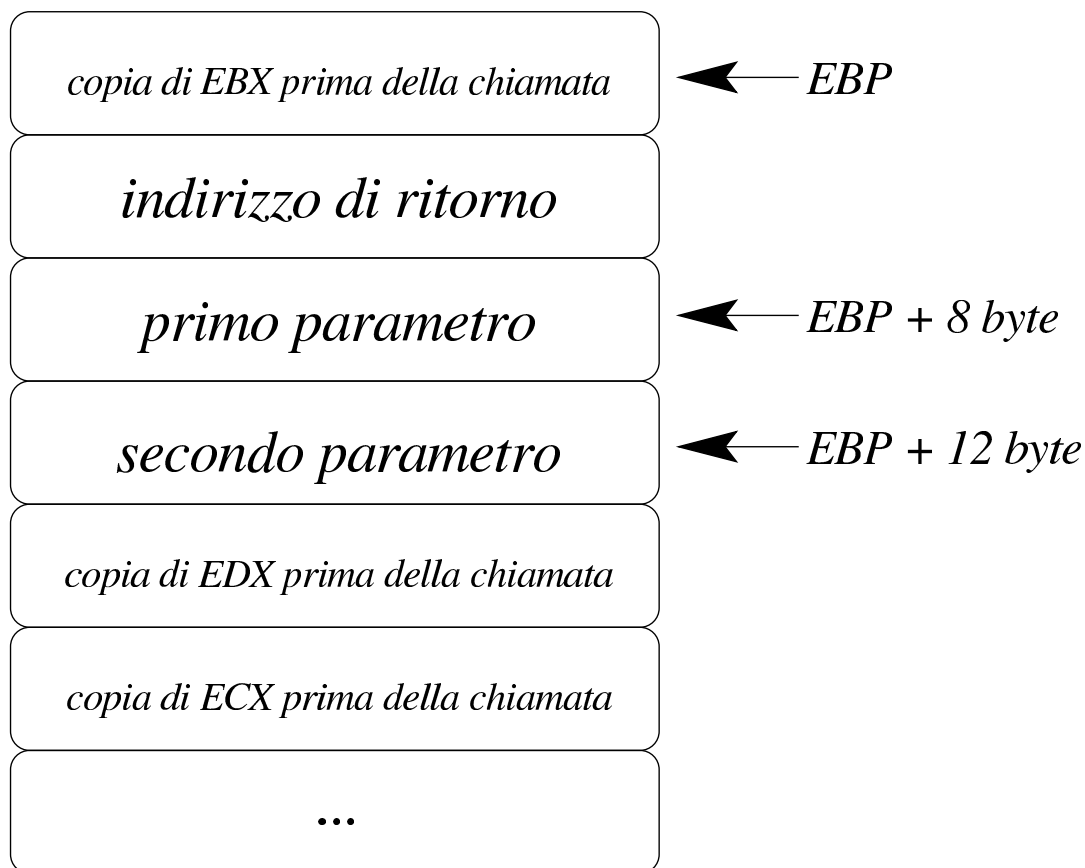
...
bp1:
    push    ecx     ; Inserisce il moltiplicando nella pila.
    push    edx     ; Inserisce il moltiplicatore nella pila.
    call   f_mol   ; Esegue la moltiplicazione: EAX = EDX * ECX
    pop     edx     ; Recupera il moltiplicando dalla pila.
    pop     ecx     ; Recupera il moltiplicatore dalla pila.

```

```
bp2:
...
f_mol:
    push  ebp                ; Salva il registro EBP.
bp3:
    mov    ebp,    esp        ; Copia ESP in EBP.
    mov    edx,    [ebp+8]    ; Copia il primo parametro
                                ; in EDX.
    mov    ecx,    [ebp+12]   ; Copia il secondo parametro
                                ; in ECX.
    mov    eax,    0          ; Azzera EAX per sicurezza.
...
end_do_mol:
    mov    esp,    ebp        ; Ripristina ESP, espellendo le
                                ; variabili locali.
    pop   ebp                ; Ripristina il registro EBP.
    ret                                ; Torna all'istruzione successiva
                                ; alla chiamata.
```

In tal caso, in corrispondenza del simbolo '**bp3**', la pila ha i contenuti che sono schematizzati nella figura successiva.

Figura 64.178. Situazione della pila in corrispondenza del simbolo 'bp3'.



#### 64.10.5 Allocazione dello spazio per le variabili locali e preservazione dei registri

Come accennato nella sezione precedente, una volta salvato il valore di **EBP** nella pila e assegnatovi il valore di **ESP**, le variabili locali possono essere accumulate nella pila quando servono. Tuttavia, di solito si preferisce definire subito lo spazio utilizzato dalle variabili locali. Per esempio, supponendo di averne due, la pila potrebbe mostrarsi come nella figura successiva.

«



```

op2:    .byte    0x03, 0x00    # little endian = 0x0003
                                     # intero senza segno.

#
.section .text
.globl _start
#-----
_start:
    movzx op1,    %edx # Moltiplicando.
    movzx op2,    %ecx # Moltiplicatore.
    mov    $0,    %eax # Risultato.
bp1:
    push    %ecx    # Inserisce il moltiplicando nella
                    # pila.
    push    %edx    # Inserisce il moltiplicatore nella
                    # pila.
    call   f_mol    # Esegue la moltiplicazione:
                    # EAX = EDX * ECX
    add    $8, %esp # Espelle i parametri
                    # di chiamata.
bp2:
    mov    %eax, %ebx # Restituisce il valore del prodotto,
    mov    $1,  %eax # ammesso che sia abbastanza piccolo
    int   $0x80     # da poter essere rappresentato come
                    # valore di uscita.
#-----
# Moltiplicazione di due numeri interi.
# f_mol (a, b) => EAX
# EAX = a * b
#
f_mol:
    push %ebp          # Salva il registro EBP.
    mov  %esp,        %ebp # Copia ESP in EBP.
    sub  $4,        %esp # Crea lo spazio per
                    # una variabile locale.

```

```
pusha                # Salva i registri
                        # principali.

#
mov  8(%ebp), %edx     # Copia il primo parametro in EDX.
mov  12(%ebp), %ecx    # Copia il secondo parametro in ECX.
mov  $0,      %eax     # Azzera EAX per sicurezza.
#
do_mol:
  cmp  $0,      %ecx   # Se il moltiplicatore è pari a
  jz   end_do_mol     # zero, il ciclo deve terminare.
  shr  $1,      %ecx   # Fa scorrere a destra il
  jnc  end_do_somma   # moltiplicatore e se l'indicatore
                        # di riporto non è attivo, salta
                        # la somma.

do_somma:
  add  %edx,    %eax   # Aggiunge il moltiplicando al
                        # risultato.

end_do_somma:
  shl  $1,      %edx   # Fa scorrere il moltiplicando a
                        # sinistra.
  jmp  do_mol        # Torna all'inizio del ciclo di
                        # moltiplicazione.

end_do_mol:
  mov  %eax, -4(%ebp) # Copia EAX nella variabile
                        # locale prevista.

  popa                # Ripristina i registri principali.
  mov  -4(%ebp), %eax # Rimette a posto il valore di EAX
                        # che deve essere restituito.
  mov  %ebp,    %esp   # Ripristina ESP, espellendo le
                        # variabili locali.
  pop  %ebp          # Riporta il registro EBP allo
                        # stato precedente.
  ret                # Torna all'istruzione successiva
                        # alla chiamata.
```



```

; op1 * op2
;
section .data
op1:    dw      0x0007      ; Intero senza segno.
op2:    dw      0x0003      ; Intero senza segno.
;
section .text
global _start
;-----
_start:
    movzx edx, word [op1]    ; Moltiplicando.
    movzx ecx, word [op2]    ; Moltiplicatore.
bp1:
    push  ecx                ; Inserisce il moltiplicando
                            ; nella pila.
    push  edx                ; Inserisce il moltiplicatore
                            ; nella pila.
    call  f_mol              ; Esegue la moltiplicazione:
                            ; EAX = EDX * ECX
    add esp,      8        ; Espelle i parametri
                            ; della chiamata.
bp2:
    mov   ebx,      eax      ; Restituisce il valore del
    mov   eax,      1        ; prodotto, ammesso che sia
    int   0x80              ; abbastanza piccolo da poter
                            ; essere rappresentato come
                            ; valore di uscita.
;-----
; Moltiplicazione di due numeri interi.
; f_mol (a, b) => EAX
; EAX = a * b
;
f_mol:
    push  ebp                ; Salva il registro EBP.

```

```
mov    ebp,      esp      ; Copia ESP in EBP.
sub   esp,      4        ; Crea lo spazio per una
                           ; variabile locale.
pusha                                ; Salva i registri principali.
;
mov    edx,      [ebp+8]   ; Copia il primo parametro
                           ; in EDX.
mov    ecx,      [ebp+12] ; Copia il secondo parametro in
                           ; ECX.
mov    eax,      0        ; Azzera EAX per sicurezza.
;
do_mol:
  cmp    ecx,      0        ; Se il moltiplicatore è pari
  jz     end_do_mol        ; a zero, il ciclo deve
                           ; terminare.
  shr   ecx,      1        ; Fa scorrere a destra il
  jnc   end_do_somma      ; moltiplicatore e se
                           ; l'indicatore di riporto non
                           ; è attivo, salta la somma.
do_somma:
  add   eax,      edx      ; Aggiunge il moltiplicando al
                           ; risultato.
end_do_somma:
  shl   edx,      1        ; Fa scorrere il moltiplicando
                           ; a sinistra.
  jmp   do_mol           ; Torna all'inizio del ciclo
                           ; di moltiplicazione.
end_do_mol:
  mov   [ebp-4],  eax      ; Copia EAX nella variabile
                           ; locale prevista.
  popa                                ; Ripristina i registri
                           ; principali.
  mov   eax,      [ebp-4] ; Rimette a posto il valore
                           ; di EAX che deve essere
```

```

; restituito.
mov   esp,    ebp    ; Ripristina ESP, espellendo
                        ; le variabili locali.
pop   ebp          ; Ripristina il registro EBP.
ret                                ; Torna all'istruzione
                        ; successiva alla chiamata.

```

Come si vede, avendo usato la coppia di istruzioni ‘**PUSHA**’, ‘**POPA**’, alla fine occorre prendersi cura del risultato che è già disponibile nel registro *EAX*: infatti viene salvato prima nello spazio riservato per la variabile locale, quindi vengono ripristinati tutti i registri (tutti eccetto ‘**ESP**’) e ancora viene ripristinato *EAX*, che deve trasmettere il valore alla chiamata.

A questo punto occorre sapere che le istruzioni seguenti possono essere sostituite dall’istruzione ‘**ENTER**’, dove *n* rappresenta una quantità di byte:

```

push  %ebp
mov   %esp, %ebp
sub   $n, %esp

```

```

push  ebp
mov   ebp, esp
sub   esp, n

```

Per la precisione, il rimpiazzo avviene come nei due brani seguenti: si osservi che in questo caso, gli attributi di ‘**ENTER**’ non vengono invertiti nelle due sintassi.

```

enter $n, $0

```

```
enter n, 0
```

Le istruzioni che invece va a rimpiazzare ‘**LEAVE**’ sono quelle seguenti:

```
mov    %ebp, %esp
pop    %ebp
```

```
mov    esp, ebp
pop    ebp
```

Logicamente, ‘**LEAVE**’ non richiede operandi, quindi si usa nello stesso modo nelle due sintassi:

```
leave
```

### 64.10.6 Convenzioni di chiamata

«

Da quanto descritto si comprende che si possono usare diversi modi per chiamare una funzione, ma anche se esistono delle modalità equivalenti, occorre definire una convenzione. In generale, per chi scrive programmi in un sistema compatibile con la tradizione Unix, la cosa migliore è uniformarsi alle convenzioni di chiamata del linguaggio C (precisamente servono quelle usate dal proprio compilatore), in modo da poter mettere assieme programmi scritti in parte in linguaggio assembleatore e in parte anche in C. Di solito, le regole per chi scrive funzioni in linguaggio assembleatore sono sostanzialmente quelle dell’ultimo esempio mostrato nella sezione precedente:

- i parametri vanno messi sulla pila in ordine inverso, in modo tale che prima della chiamata appaia in cima il primo parametro;

- all'inizio della funzione, occorre accumulare nella pila il contenuto di ***EBP***, che deve essere ripristinato immediatamente prima del ritorno;
- all'interno della funzione si accede ai parametri contenuti nella pila, senza estrarli dalla stessa e senza modificarli, perché le modifiche non verrebbero considerate;
- i registri principali devono essere preservati (ogni compilatore ha la sua politica e non si può dire, in generale, quali siano);
- la funzione deve restituire il risultato della sua elaborazione attraverso ***EAX***, oppure, se si richiede una dimensione più grande, deve essere usata la coppia ***EDX:EAX***.<sup>10</sup>

Nel caso si vogliano utilizzare funzioni scritte in linguaggio C, all'interno di un programma scritto in linguaggio assembler, occorre verificare quali registri le funzioni scritte in C non preservano (oltre alla coppia ***EDX:EAX***, usata per restituire il risultato della chiamata). In generale, si può considerare che le funzioni scritte in linguaggio C potrebbero alterare i registri ***EAX***, ***ECX*** e ***EDX***. Se però si vuole avere la certezza assoluta sul contenuto dei registri dopo la chiamata di una funzione realizzata con un altro linguaggio, conviene organizzarsi salvando tutti quelli che si stanno utilizzando prima della chiamata e ripristinandoli subito dopo, come in parte è stato mostrato.

#### 64.10.7 Nota sugli array «locali»

Generalmente, un array viene gestito attraverso uno spazio di memoria condiviso da tutto il programma, dove le funzioni che devono manipolarlo ricevono l'indirizzo di questo, tra i parametri della

chiamata. Tuttavia, nel caso si volesse gestire, all'interno di una funzione, un array locale, il cui contenuto viene abbandonato alla conclusione della stessa, l'unico modo per ottenere ciò è attraverso la pila dei dati. In pratica, come per le variabili locali scalari, andrebbe riservato un certo spazio aumentando la dimensione della pila in modo adeguato, per poi scandire tale spazio con indici appropriati.

## 64.11 Esempi di funzioni ricorsive

&lt;&lt;

Vengono mostrati alcuni esempi molto semplici in cui si applica la ricorsione, adatti alla compilazione con GNU AS e NASM.

### 64.11.1 Elevamento a potenza

&lt;&lt;

Viene proposta una soluzione ricorsiva del problema dell'elevamento a potenza. In pratica,  $x^y$  è pari a  $x \cdot x^{(y-1)}$ , ma in particolare: se  $x$  è pari a zero, il risultato è zero; altrimenti, se  $y$  è pari a zero, il risultato è uno.

All'interno della funzione '**f\_pwr**' viene riservato lo spazio per una sola variabile locale, che serve a conservare il valore di *EAX*, per poi recuperarlo quando si ripristinano tutti i registri, prima della conclusione della funzione stessa.

```
# op1 ^ op2
#
.section .data
op1:    .int    3
op2:    .int    4
#
.section .text
.globl _start
#
# Main.
```

```
#
_start:
    mov    op1, %esi    # Base.
    mov    op2, %edi    # Esponente.
bp1:
    push  %edi          # f_pwr (ESI, EDI) ==> EAX
    push  %esi          #
    call  f_pwr         #
    add   $8, %esp      #
bp2:
    mov   %eax, %ebx    # Restituisce il valore della potenza,
    mov   $1, %eax      # ammesso che sia abbastanza piccolo
    int  $0x80          # da poter essere rappresentato come
                        # valore di uscita.

#
# Potenza di due numeri interi senza segno.
# f_pwr (a, b) ==> EAX
# EAX = a ^ b
#
f_pwr:
    enter $4, $0
    pusha
    #
    mov   8(%ebp), %esi # Base.
    mov   12(%ebp), %edi # Esponente.
    #
    cmp   $0, %esi     # Se la base è pari a 0,
    jz   f_pwr_end_0   # restituisce 0.
    #
    cmp   $0, %edi     # Se l'esponente è pari a 0,
    jz   f_pwr_end_1   # restituisce 1.
    #
    dec  %edi          # Riduce l'esponente di una unità.
    push %edi          # f_pwr (ESI, EDI) ==> EAX
```

```
    push  %esi          #
    call  f_pwr         #
    add   $8, %esp      #
    mul   %esi          # EDX:EAX = EAX*ESI
    mov   %eax, -4(%ebp) # Salva il risultato.
    jmp   f_pwr_end_X   # Conclude la funzione.
    #
f_pwr_end_0:
    popa                # Conclude la funzione con EAX = 0.
    mov   $0, %eax      #
    leave                #
    ret                 #
f_pwr_end_1:
    popa                # Conclude la funzione con EAX = 1.
    mov   $1, %eax      #
    leave                #
    ret                 #
f_pwr_end_X:
    popa                # Conclude la funzione con EAX pari
    mov   -4(%ebp), %eax # al valore salvato nella variabile
    leave                # locale.
    ret                 #
```

```
; op1 ^ op2
;
section .data
op1:    dd     3
op2:    dd     4
;
section .text
global _start
;
; Main.
;
_start:
```



```
    mov     esi, [op1] ; Base.
    mov     edi, [op2] ; Esponente.
bp1:
    push   edi          ; f_pwr (ESI, EDI) ==> EAX
    push   esi          ;
    call   f_pwr        ;
    add    esp, 8       ;
bp2:
    mov     ebx, eax    ; Restituisce il valore della potenza,
    mov     eax, 1     ; ammesso che sia abbastanza piccolo
    int    0x80        ; da poter essere rappresentato come
                        ; valore di uscita.

;
; Potenza di due numeri interi senza segno.
; f_pwr (a, b) ==> EAX
; EAX = a ^ b
;
f_pwr:
    enter  4,0
    pusha
    ;
    mov     esi, [ebp+8] ; Base.
    mov     edi, [ebp+12] ; Esponente.
    ;
    cmp     esi, 0      ; Se la base è pari a 0,
    jz     f_pwr_end_0 ; restituisce 0.
    ;
    cmp     edi, 0      ; Se l'esponente è pari a 0,
    jz     f_pwr_end_1 ; restituisce 1.
    ;
    dec    edi          ; Riduce l'esponente di una unità.
    push   edi          ; f_pwr (ESI, EDI) ==> EAX
    push   esi          ;
    call   f_pwr        ;
```

```

    add    esp, 8          ;
    mul    esi             ; EDX:EAX = EAX*ESI
    mov    [ebp-4], eax    ; Salva il risultato.
    jmp    f_pwr_end_X    ; Conclude la funzione.
    ;
f_pwr_end_0:
    popa                  ; Conclude la funzione con EAX = 0.
    mov    eax, 0         ;
    leave                   ;
    ret                    ;
f_pwr_end_1:
    popa                  ; Conclude la funzione con EAX = 1.
    mov    eax, 1         ;
    leave                   ;
    ret                    ;
f_pwr_end_X:
    popa                  ; Conclude la funzione con EAX pari
    mov    eax, [ebp-4]   ; al valore salvato nella variabile
    leave                   ; locale.
    ret                    ;

```

## 64.11.2 Fattoriale



Viene proposta una soluzione ricorsiva del problema del fattoriale. In pratica,  $x!$  è pari a  $x \cdot (x-1)!$ , ma in particolare: se  $x$  è pari a 1, il risultato è uno.

All'interno della funzione '**f\_fact**' viene riservato lo spazio per una sola variabile locale, che serve a conservare il valore di **EAX**, per poi recuperarlo quando si ripristinano tutti i registri, prima della conclusione della funzione stessa.

```

# op1!
#

```

```
.section .data
op1:    .int    5
#
.section .text
.globl _start
#
# Main.
#
_start:
    mov     op1, %esi    # ESI contiene il valore di cui si
                        # vuole calcolare il fattoriale.
bp1:
    push   %esi         # f_fact (ESI) ==> EAX
    call  f_fact        #
    add    $4, %esp     #
bp2:
    mov    %eax, %ebx   # Restituisce il valore del fattoriale,
    mov    $1, %eax     # ammesso che sia abbastanza piccolo
    int    $0x80        # da poter essere rappresentato come
                        # valore di uscita.

#
# Fattoriale di un numero senza segno.
# f_fatt (a) ==> EAX
# EAX = a!
#
f_fact:
    enter $4, $0
    pusha
    #
    mov    8(%ebp), %edi # Valore di cui calcolare il
                        # fattoriale.
    cmp    $1, %edi     # Il fattoriale di 1 è 1.
    jz    f_fact_end_1  #
    #
```

```

    mov    %edi, %esi    # ESI contiene il valore di cui si
    dec    %esi         # vuole il fattoriale, ridotto di
                        # una unità.
    push  %esi         # f_fact (ESI) ==> EAX
    call  f_fact       #
    add   $4, %esp     #
    mul   %edi         # EDX:EAX = EAX*EDI
    mov   %eax, -4(%ebp) # Salva il risultato.
    jmp   f_fact_end_X # Conclude la funzione.
    #
f_fact_end_1:
    popa                # Conclude la funzione con EAX = 1.
    mov   $1, %eax     #
    leave                #
    ret                 #
f_fact_end_X:
    popa                # Conclude la funzione con EAX pari
    mov  -4(%ebp), %eax # al valore salvato nella variabile
    leave                # locale.
    ret                 #

```

```

; op1!
;
section .data
op1:    dd        5
;
section .text
global _start
;
; Main.
;
_start:
    mov   esi, [op1] ; ESI contiene il valore di cui si
                    ; vuole calcolare il fattoriale.
bp1:

```

```
    push esi          ; f_fact (ESI) ==> EAX
    call f_fact      ;
    add esp, 4       ;
bp2:
    mov ebx, eax     ; Restituisce il valore del fattoriale,
    mov eax, 1       ; ammesso che sia abbastanza piccolo
    int 0x80         ; da poter essere rappresentato come
                    ; valore di uscita.
;
; Fattoriale di un numero senza segno.
; f_fatt (a) ==> EAX
; EAX = a!
;
f_fact:
    enter 4,0
    pusha
;
    mov edi, [ebp+8] ; Valore di cui calcolare il
                    ; fattoriale.
    cmp edi, 1       ; Il fattoriale di 1 è 1.
    jz f_fact_end_1 ;
;
    mov esi, edi     ; ESI contiene il valore di cui si
    dec esi          ; vuole il fattoriale, ridotto di
                    ; una unità.
    push esi         ; f_fact (ESI) ==> EAX
    call f_fact      ;
    add esp, 4       ;
    mul edi          ; EDX:EAX = EAX*EDI
    mov [ebp-4], eax ; Salva il risultato.
    jmp f_fact_end_X ; Conclude la funzione.
;
f_fact_end_1:
    popa             ; Conclude la funzione con EAX = 1.
```

```

    mov eax, 1          ;
    leave              ;
    ret                ;
f_fact_end_X:
    popa               ; Conclude la funzione con EAX pari
    mov eax, [ebp-4]   ; al valore salvato nella variabile
    leave              ; locale.
    ret                ;

```

## 64.12 Indirizzamento dei dati

«

In generale, con l'architettura x86 ci si preoccupa di definire il modo in cui fare riferimento ai dati nel sorgente in linguaggio assembleatore (incluso ciò che riguarda la pila), mentre per il riferimento alle istruzioni si usano simboli che il compilatore traduce normalmente in indirizzi relativi (il «dislocamento»). Viene mostrata una tabella che riepiloga i vari modi con cui è possibile fare riferimento ai dati, secondo le due sintassi più comuni.

Tabella 64.193. Indirizzamento dei dati.

Esempio AT&T	Esempio Intel	Descrizione
\$21	21	Si sta facendo riferimento al numero $21_{10}$ in modo letterale.
\$0x15	0x15	
\$025	025	
\$0b10101	10101b	

Esempio AT&T	Esempio Intel	Descrizione
$\$nome\_simbolo$	$nome\_simbolo$	Si fa riferimento all'indirizzo corrispondente al simbolo, ovvero al numero che costituisce tale indirizzo.
$nome\_simbolo$	$[nome\_simbolo]$	Si fa riferimento all'area di memoria che inizia in corrispondenza del simbolo.
$nome\_simbolo \pm n$	$[nome\_simbolo \pm n]$	Si fa riferimento all'area di memoria che inizia in corrispondenza del simbolo, $\pm n$ byte.
$\%eax$	$eax$	Si sta facendo riferimento al registro <b>EAX</b> , in qualità di variabile.
$(\%eax)$	$[eax]$	Si sta facendo riferimento all'area di memoria che inizia a partire dall'indirizzo contenuto nel registro <b>EAX</b>
$\pm n (\%eax)$	$[eax \pm n]$	Si sta facendo riferimento all'area di memoria che inizia a partire dall'indirizzo contenuto nel registro <b>EAX</b> , $\pm n$ byte.

Esempio AT&T	Esempio Intel	Descrizione
<i>nome_simbolo</i> (%eax)	[eax+ <i>nome_simbolo</i> ]	Si sta facendo riferimento all'area di memoria che inizia a partire dall'indirizzo a cui fa riferimento il simbolo, più il contenuto del registro <i>EAX</i> .
( <i>nome_simbolo</i> ± <i>n</i> ) (%eax)	[eax+ <i>nome_simbolo</i> ± <i>n</i> ]	Si sta facendo riferimento all'area di memoria che inizia a partire dall'indirizzo a cui fa riferimento il simbolo, più il contenuto del registro <i>EAX</i> , ± <i>n</i> byte.
(%eax, %ebx, <i>n</i> )	[eax+%ebx* <i>n</i> ]	Si sta facendo riferimento all'area di memoria che inizia a partire da <i>EAX</i> +( <i>EBX</i> · <i>n</i> ).
± <i>m</i> (%eax, %ebx, <i>n</i> )	[eax+%ebx* <i>n</i> ± <i>m</i> ]	Si sta facendo riferimento all'area di memoria che inizia a partire da ( <i>EAX</i> +( <i>EBX</i> · <i>n</i> ))± <i>m</i> .

### 64.12.1 Gestione di array



Per comprendere l'uso della forma più complessa di indirizzamento, si prenda l'esempio seguente, in cui appare un simbolo, denominato '**record**', che descrive una sequenza di valori, contenenti anche ciò che si vuole considerare una matrice di numeri.

```
#
.section .data
```



```

record:      .ascii "matrice"
              .int    0,  1,  2,  3,  4,  5,  6,  7,  8,  9
              .int    10, 11, 12, 13, 14, 15, 16, 17, 18, 19
              .int    20, 21, 22, 23, 24, 25, 26, 27, 28, 29
              .int    30, 31, 32, 33, 34, 35, 36, 37, 38, 39
              .int    40, 41, 42, 43, 44, 45, 46, 47, 48, 49
              .int    50, 51, 52, 53, 54, 55, 56, 57, 58, 59
              .int    60, 61, 62, 63, 64, 65, 66, 67, 68, 69
              .int    70, 71, 72, 73, 74, 75, 76, 77, 78, 79
              .int    80, 81, 82, 83, 84, 85, 86, 87, 88, 89
              .int    90, 91, 92, 93, 94, 95, 96, 97, 98, 99
dimensione_int: .int    4      # 32 bit
dimensione_riga: .int    40     # 4 * 10 byte
riga:          .int    3      # da 0 a 9
colonna:       .int    5      # da 0 a 9
#
.section .text
.globl _start
#
_start:
    mov    riga, %eax      # Legge le coordinate di riga e
    mov    colonna, %ecx   # colonna, copiandole in EAX e ECX.
    mull   dimensione_riga # EDX:EAX := EAX * 40.
    mov    (record+7) (%eax,%ecx,4), %ebx
                                # EBX := array[riga,colonna].
bp1:
    mov    $1, %eax       # Restituisce il valore trovato
    int    $0x80          # nella matrice.

```

```

;
section .data
record:      db      "matrice"
              dd      0,  1,  2,  3,  4,  5,  6,  7,  8,  9
              dd      10, 11, 12, 13, 14, 15, 16, 17, 18, 19
              dd      20, 21, 22, 23, 24, 25, 26, 27, 28, 29

```

```

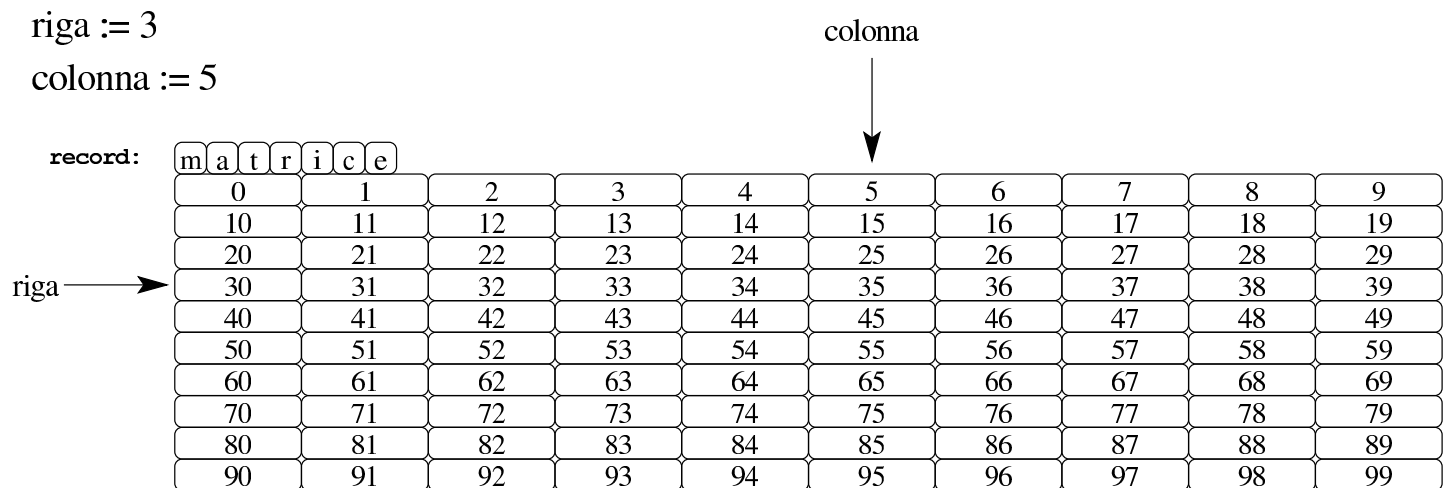
        dd    30, 31, 32, 33, 34, 35, 36, 37, 38, 39
        dd    40, 41, 42, 43, 44, 45, 46, 47, 48, 49
        dd    50, 51, 52, 53, 54, 55, 56, 57, 58, 59
        dd    60, 61, 62, 63, 64, 65, 66, 67, 68, 69
        dd    70, 71, 72, 73, 74, 75, 76, 77, 78, 79
        dd    80, 81, 82, 83, 84, 85, 86, 87, 88, 89
        dd    90, 91, 92, 93, 94, 95, 96, 97, 98, 99
dimensione_int: dd    4      ; 32 bit
dimensione_riga: dd    40    ; 4 * 10 byte
riga:          dd    3      ; da 0 a 9
colonna:       dd    5      ; da 0 a 9
;
section .text
global _start
;
_start:
    mov     eax, [riga]      ; Legge le coordinate di riga
    mov     ecx, [colonna]  ; e colonna, copiandole in
                           ; EAX e ECX.
    mul     long [dimensione_riga] ; EDX:EAX := EAX * 40.
    mov     ebx, [record+7+eax+ecx*4]
                           ; EBX := array[riga,colonna].
bp1:
    mov     eax, 1          ; Restituisce il valore
    int     0x80           ; trovato nella matrice.

```

In pratica, **'record'** individua l'inizio di una struttura composta da una stringa di sette byte, seguita da un centinaio di interi da 32 bit, suddivisi idealmente in righe da 10 unità. In pratica, una «riga» di questi numeri occupa 40 byte.

Per accedere a un certo elemento della matrice contenuta nel «record», occorre considerare uno scostamento iniziale di 7 byte, quindi occorre calcolare la posizione dell'elemento, secondo lo schema

che si vede nella figura successiva.



$$7 + (\text{riga} \times 40) + (\text{colonna} \times 4)$$

Viene proposto un altro esempio equivalente, ma realizzato gestendo in modo diverso l'indirizzamento. La dichiarazione dei dati è la stessa:

```

...
_start:
    mov     riga, %eax           # Legge le coordinate di riga.
    mull   dimensione_riga     # EDX:EAX := EAX * 40.
    mov     %eax, %ecx         # ECX := riga * 40.
    #
    mov     colonna, %eax       # Legge le coordinate di colonna.
    mull   dimensione_int      # EDX:EAX := EAX * 4.
    add    %eax, %ecx          # ECX := (riga*40)+(colonna*4)
    #
    add    $7, %ecx           # ECX := 7+(riga*40)+(colonna*4)
    mov    record(%ecx), %ebx # EBX := array[riga,colonna].
bp1:
    mov    $1, %eax           # Restituisce il valore trovato
    int   $0x80              # nella matrice.

```

```

...
_start:
    mov eax, [riga]                ; Legge le coordinate di riga.
    mul long [dimensione_riga]    ; EDX:EAX := EAX * 40.
    mov ecx, eax                  ; ECX := riga * 40.
    ;
    mov eax, [colonna]           ; Legge le coord. di colonna.
    mul long [dimensione_int]    ; EDX:EAX := EAX * 4.
    add ecx, eax                 ; ECX := (riga*40)+(colonna*4)
    ;
    add ecx, 7                  ; ECX := 7+(riga*40)+(colonna*4)
    mov ebx, [record+ecx]      ; EBX := array[riga,colonna].
bp1:
    mov eax, 1                   ; Restituisce il valore trovato
    int 0x80                     ; nella matrice.

```

Come si vede, si arriva a mettere nel registro *ECX* il risultato di  $7+(riga \times 40)+(colonna \times 4)$ ; pertanto, alla fine si aggiunge solo l'indirizzo iniziale della struttura e si ottiene, complessivamente, l'indirizzo del valore cercato nella matrice.

Un altro esempio, dove si vede un altro modo di usare l'indirizzamento indiretto fornito dal linguaggio macchina:

```

...
_start:
    mov    riga, %eax             # Legge le coordinate di riga.
    mull  dimensione_riga       # EDX:EAX := EAX * 40.
    mov    %ecx, %ecx            # ECX := riga * 40.
    #
    mov    colonna, %eax         # Legge le coordinate di colonna.
    mull  dimensione_int       # EDX:EAX := EAX * 4.
    add    %ecx, %ecx           # ECX := (riga*40)+(colonna*4)
    #
    add    $7, %ecx             # ECX := 7+(riga*40)+(colonna*4)

```

```

mov    $record, %edx    # EDX := indirizzo iniziale di
                        # "record".
mov    (%edx,%ecx), %ebx # EBX := array[riga,colonna].
bp1:
mov    $1, %eax        # Restituisce il valore trovato
int    $0x80          # nella matrice.

```

```

...
_start:
mov    eax, [riga]      ; Legge le coordinate di riga.
mul    long [dimensione_riga] ; EDX:EAX := EAX * 40.
mov    ecx, eax        ; ECX := riga * 40.
;
mov    eax, [colonna]   ; Legge le coordinate di colonna.
mul    long [dimensione_int] ; EDX:EAX := EAX * 4.
add    ecx, eax        ; ECX := (riga*40)+(colonna*4)
;
add    ecx, 7          ; ECX := 7+(riga*40)+(colonna*4)
mov    edx, record     ; EDX := indirizzo iniziale di "record".
mov    ebx, [edx+ecx]  ; EBX := array[riga,colonna].
bp1:
mov    eax, 1          ; Restituisce il valore trovato
int    0x80          ; nella matrice.

```

Infine, un esempio dove si fa in modo che *ECX* contenga l'indirizzo completo dell'elemento cercato:

```

...
_start:
mov    riga, %eax      # Legge le coordinate di riga.
mull   dimensione_riga # EDX:EAX := EAX * 40.
mov    %eax, %ecx     # ECX := riga * 40.
#
mov    colonna, %eax   # Legge le coordinate di colonna.
mull   dimensione_int # EDX:EAX := EAX * 4.

```

```

add    %eax, %ecx      # ECX := (riga*40)+(colonna*4)
#
add    $7, %ecx       # ECX := 7+(riga*40)+(colonna*4)
add    $record, %ecx  # ECX := indirizzo completo.
mov    (%ecx), %ebx   # EBX := array[riga,colonna].
bp1:
mov    $1, %eax       # Restituisce il valore trovato
int    $0x80          # nella matrice.

```

```

...
_start:
mov    eax, [riga]    ; Legge le coordinate di riga.
mul    long [dimensione_riga] ; EDX:EAX := EAX * 40.
mov    ecx, eax      ; ECX := riga * 40.
;
mov    eax, [colonna] ; Legge le coord. di colonna.
mul    long [dimensione_int] ; EDX:EAX := EAX * 4.
add    ecx, eax      ; ECX := (riga*40)+(colonna*4)
;
add    ecx, 7        ; ECX := 7+(riga*40)+(colonna*4)
add    ecx, record   ; ECX := indirizzo completo.
mov    ebx, [ecx]    ; EBX := array[riga,colonna].
bp1:
mov    eax, 1        ; Restituisce il valore trovato
int    0x80          ; nella matrice.

```

## 64.12.2 Istruzione «LEA»



L'istruzione '**LEA**' consente di calcolare un indirizzo di memoria e di salvarlo in un registro. Per esempio, le due istruzioni successive, una con '**MOV**' e l'altra con '**LEA**', fanno la stessa cosa:

```

mov    (%ecx), %ebx
lea    %ecx, %ebx

```

Ovvero:

```
mov    ebx, [ecx]
lea    ebx, ecx
```

Pertanto, con l'istruzione '**LEA**', ciò che appare nell'operando che svolge il ruolo di «origine» viene considerato solo per il suo indirizzo e copiato nella destinazione. La differenza sostanziale, rispetto a '**MOV**', sta nel poter usare le espressioni di indirizzamento indiretto. Per riprendere l'esempio usato precedentemente per mostrare come accedere a una matrice di numeri, si potrebbe fare una cosa come si vede nel listato successivo, anche se sarebbe poco utile in tale circostanza particolare:

```
...
_start:
    mov    riga, %eax        # Legge le coordinate di riga e
    mov    colonna, %ecx    # colonna, copiandole in EAX e ECX.
    #
    mull  dimensione_riga # EDX:EAX := EAX * 40.
    #
    lea    (record+7) (%eax,%ecx,4), %edx # EDX contiene
                                           # l'indirizzo
                                           # dell'elemento.
    mov    (%edx), %ebx      # EBX := array[riga,colonna].
bp1:
    mov    $1, %eax        # Restituisce il valore trovato
    int   $0x80           # nella matrice.
```

```
...
_start:
    mov    eax, [riga]     ; Legge le coordinate di riga e
    mov    ecx, [colonna] ; colonna, copiandole in EAX e ECX.
    ;
```

```

mul    long [dimensione_riga] ; EDX:EAX := EAX * 40.
;
lea    ebx, [record+7+eax+ecx*4] ; EDX contiene
;                                ; l'indirizzo
;                                ; dell'elemento.
mov    ebx, [edx]                ; EBX := array[riga,colonna].
bp1:
mov    eax, 1                    ; Restituisce il valore trovato
int    0x80                      ; nella matrice.

```

## 64.13 Rappresentazione dei dati in memoria attraverso un esempio

«

Viene proposto un esempio in cui si utilizzano i tipi principali di variabili inizializzate in memoria, con due listati equivalenti, il primo adatto a GNU AS mentre il secondo è per NASM. In particolare, nel secondo listato, non potendo dichiarare variabili da 64 bit, sono state usate coppie di variabili a 32 bit, inizializzate tenendo conto dell'inversione *little endian*.

```

#
.section .data
numero8:    .byte    0b00010010          # 0x12          # 18
numero16:   .short   0b0001001000110100 # 0x1234       # 4660
numero32:   .int     0b00010010001101000101011001111000
;                                # 0x12345678
;                                # 305419896
numero64:   .quad    0x123456789ABCDEF0
carattere:  .byte    'A'
stringa:    .ascii   "testo"
caratteri:  .byte    'T', 'E', 'S', 'T', 'O'
;                                .skip 3, 0xFF
;                                .rept 4

```



```

        .byte   'Z'
        .endr

numer18: .byte   0x12, 0x34, 0x56, 0x78, 0x9A
numer16: .short  0xBCDE, 0xF012, 0x3456, 0x789A
numer32: .int    0xBCDEF012, 0x3456789A, 0xBCDEF012
numer64: .quad   0x3456789ABCDEF012, 0x3456789ABCDEF012
fine:    .byte   'F', 'I', 'N', 'E'
#
.section .text
.globl _start
#
_start:
    mov     $0, %ebx
    mov     $1, %eax
    int     $0x80

```

```

;
section .data
numero8:  db      00010010b           ; 0x12           = 18
numero16: dw      0001001000110100b   ; 0x1234        = 4660
numero32: dd      00010010001101000101011001111000b
                                           ; 0x12345678
                                           ; = 305419896
numero64: dd      0x9ABCDEF0, 0x12345678 ; 0x123456789ABCDEF0
carattere: db     'A'
stringa:  db     "testo"
caratteri: db     'T', 'E', 'S', 'T', 'O'
times 3   db     0xFF
times 4   db     'Z'
;
;
numer18:  db     0x12, 0x34, 0x56, 0x78, 0x9A
numer16:  dw     0xBCDE, 0xF012, 0x3456, 0x789A
numer32:  dd     0xBCDEF012, 0x3456789A, 0xBCDEF012

```

```
numeri64:    dd    0xBCDEF012, 0x3456789A, 0xBCDEF012,  
             0x3456789A  
fine:       db    'F', 'I', 'N', 'E'  
;  
section .text  
global _start  
;  
_start:  
    mov     ebx, 0  
    mov     eax, 1  
    int    0x80
```

Il programma in questione non fa alcunché e serve solo per verificare con GDB come sono rappresentati i dati in memoria e come questi si possono leggere. Complessivamente, le variabili occupano 78 byte, secondo lo schema che si vede nella figura successiva, dove vengono riprodotte anche le inversioni dovute alla rappresentazione *little endian*:

numero8 12	numero16 34 12	numero32 78 56 34 12	numero64 0F
DE BC 9A 78 56 34 12			carattere 41 <sup>A</sup>
stringa <i>t e s t o</i> 74 65 73 74 6F	caratteri <i>T E S</i> 54 45 53		
<i>T O</i> 54 4F	FF FF FF	<i>Z Z Z</i> 5A 5A 5A	
<i>Z</i> 5A	numeri8 12 34 56 78 9A	numeri16 DE BC	
12 F0 56 34 9A 78			numeri32 12 F0
DE BC 9A 78 56 34			12 F0
DE BC		numeri64 12 F0 DE BC 9A 78	
56 34		12 F0 DE BC 9A 78	
56 34		fine <i>F I N E</i> 46 49 4E 45	

Per leggere la memoria già inizializzata con GDB non è necessario avviare il programma, in quanto, appena aperto, questa è subito accessibile:

```
$ gdb nome_programma [Invio]
```

```
(gdb) print /x (char[78])numero8 [Invio]
```

```
$1 = {0x12, 0x34, 0x12, 0x78, 0x56, 0x34, 0x12, 0xf0, 0xde,  
0xbc, 0x9a, 0x78, 0x56, 0x34, 0x12, 0x41, 0x74, 0x65,  
0x73, 0x74, 0x6f, 0x54, 0x45, 0x53, 0x54, 0x4f, 0xff,  
0xff, 0xff, 0x5a, 0x5a, 0x5a, 0x5a, 0x12, 0x34, 0x56,  
0x78, 0x9a, 0xde, 0xbc, 0x12, 0xf0, 0x56, 0x34, 0x9a,  
0x78, 0x12, 0xf0, 0xde, 0xbc, 0x9a, 0x78, 0x56, 0x34,  
0x12, 0xf0, 0xde, 0xbc, 0x12, 0xf0, 0xde, 0xbc, 0x9a,  
0x78, 0x56, 0x34, 0x12, 0xf0, 0xde, 0xbc, 0x9a, 0x78,  
0x56, 0x34, 0x46, 0x49, 0x4e, 0x45}
```

Quello che viene richiesto con il comando appena mostrato è di mostrare la memoria a partire dall'indirizzo corrispondente al simbolo **'numero8'** (pertanto dall'inizio), in forma di array di byte, composto da 78 elementi. Si può verificare la corrispondenza tra quanto ottenuto e la figura mostrata in precedenza.

Con i comandi successivi si ispezionano le variabili, singolarmente. È sempre necessario dichiarare la dimensione a cui si è interessati, quando questa è diversa da quella predefinita (32 bit). Non è possibile ispezionare le variabili da 64 bit in modo complessivo, pertanto occorre arrangiarsi, come se fossero array di due numeri a 32 bit. Nei comandi mostrati appare spesso l'opzione **'/f'**, con la quale si dichiara il tipo di rappresentazione che si vuole ottenere.

```
(gdb) print /x (char)numero8 [Invio]
```

```
$2 = 0x12
```

```
(gdb) print (char)numero8 [Invio]
```

```
$3 = 18 '\022'
```

```
(gdb) print /x (short)numero16 [Invio]
```

```
$4 = 0x1234
```

```
(gdb) print (short)numero16 [Invio]
```

```
$5 = 4660
```

```
(gdb) print /x (int)numero32 [Invio]
```

```
$6 = 0x12345678
```

```
(gdb) print (int)numero32 [Invio]
```

```
$7 = 305419896
```

```
(gdb) print /x (int[2])numero64 [Invio]
```

```
$8 = {0x9abcdef0, 0x12345678}
```

```
(gdb) print (char)carattere [Invio]
```

```
$9 = 65 'A'
```

```
(gdb) print (char[11])carattere [Invio]
```

```
$10 = "AtestoTESTO"
```

```
(gdb) print (char[5])stringa [Invio]
```

```
$11 = "testo"
```

```
(gdb) print (char[5])caratteri [Invio]
```

```
$12 = "TESTO"
```

```
(gdb) print /x (char[12])caratteri [Invio]
```

```
$13 = {0x54, 0x45, 0x53, 0x54, 0x4f, 0xff, 0xff, 0xff, ↵  
↵0x5a, 0x5a, 0x5a, 0x5a}
```

```
(gdb) print /x (char[5])numer18 [Invio]
```

```
$14 = {0x12, 0x34, 0x56, 0x78, 0x9a}
```

```
(gdb) print /d (char[5])numer18 [Invio]
```

```
$15 = {18, 52, 86, 120, -102}
```

```
(gdb) print /x (short[4])numer16 [Invio]
```

```
$16 = {0xbcde, 0xf012, 0x3456, 0x789a}
```

```
(gdb) print /d (short[4])numer16 [Invio]
```

```
$17 = {-17186, -4078, 13398, 30874}
```

```
(gdb) print /d (unsigned short[4])numer16 [Invio]
```

```
$18 = {48350, 61458, 13398, 30874}
```

```
(gdb) print /x (int[3])numer32 [Invio]
```

```
$19 = {0xbcdef012, 0x3456789a, 0xbcdef012}
```

```
(gdb) print /d (int[3])numer32 [Invio]
```

```
$20 = {-1126240238, 878082202, -1126240238}
```

```
(gdb) print /d (unsigned int[3])numer32 [Invio]
```

```
$21 = {3168727058, 878082202, 3168727058}
```

```
(gdb) print /x (int[4])numer64 [Invio]
```

```
$22 = {0xbcdef012, 0x3456789a, 0xbcdef012, 0x3456789a}
```

```
(gdb) print (char[4])fine [Invio]
```

```
$23 = "FINE"
```

## 64.14 Esempi con gli array

Vengono mostrati alcuni esempi elementari di scansione di array, adatti alla compilazione con GNU AS e NASM. Si dà per scontato che si sappia utilizzare GDB per ispezionare la memoria e leggere, in particolare, il contenuto degli array stessi.

Negli esempi viene usata la direttiva ‘**.equ**’, o ‘**equ**’, per associare una sigla al livello in cui si trovano i dati nella pila (più precisamente nello *stack frame*).

Tutti gli esempi sono mostrati con listati a coppie: uno valido per GNU AS e l’altro per NASM.

### 64.14.1 Ricerca sequenziale

Viene mostrato un esempio di programma contenente una funzione che esegue una ricerca sequenziale all’interno di un array di interi, senza segno. Il metodo utilizzato si rifà a quanto descritto in pseudocodifica nella sezione [62.4.1](#). Il risultato della scansione viene emesso attraverso il valore restituito dal programma; ciò che si ottiene è precisamente l’indice dell’elemento trovato, oppure  $-1$  se nessun elemento corrisponde.

```
# Ricerca sequenziale.
#
.section .data
lista: .int 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 # Interi senza
                                           # segno.

a:      .int 0          # Indice minimo.
z:      .int 9          # Indice massimo.
#
.section .text
.globl _start
```

```
#
# Main.
#
_start:
    push    z           # f_rs ($lista, $7, a, z) ==> EAX
    push    a           # Si cerca il valore 7 nell'array
    push    $7          # «lista», tra gli indici «a» e «z».
    push    $lista     # Viene restituito l'indice
    call    f_rs       # dell'elemento trovato, oppure
    add     $16, %esp   # -1 se non è presente.
bp1:
    mov     %eax, %ebx  # Restituisce l'indice trovato,
    mov     $1, %eax   # ammesso che sia abbastanza piccolo
    int    $0x80       # da poter essere rappresentato come
                       # valore di uscita.

#
# Ricerca sequenziale all'interno di una lista di valori.
# f_rs (lista, x, a, z) ==> EAX
# Al termine EAX contiene l'indice del valore trovato,
# oppure -1 se questo non c'è.
#
f_rs:
    enter  $4, $0
    pusha
    .equ   rs_i,      -4           # Gli si associa EAX.
    .equ   rs_lista,  8           # Gli si associa ESI.
    .equ   rs_x,      12          # Gli si associa EDX.
    .equ   rs_a,      16
    .equ   rs_z,      20
    #
    mov    rs_lista(%ebp), %esi   # ESI contiene l'indirizzo
                                   # dell'array.
    mov    rs_x(%ebp), %edx      # EDX contiene il valore
                                   # cercato.
```



```
#
mov    rs_a(%ebp),    %eax    # EAX viene usato come
                                # indice di scansione.
f_rs_loop:
    cmp    rs_z(%ebp),    %eax    # Se EAX è maggiore
                                # dell'indice massimo,
    ja    f_rs_non_trovato    # l'elemento cercato non
                                # c'è.

#
    cmp    (%esi,%eax,4),    %edx    # Se il valore cercato
    je    f_rs_trovato    # corrisponde a quello
                                # dell'indice corrente,
                                # termina la scansione.

#
    inc    %eax    # Incrementa l'indice
    jmp    f_rs_loop    # di scansione e salta
                                # all'inizio del ciclo.

#
f_rs_non_trovato:
    popa    # Conclude la funzione con
    mov    $-1, %eax    # EAX = -1.
    leave    #
    ret     #

f_rs_trovato:
    mov    %eax, rs_i(%ebp)    # Salva EAX nella variabile
                                # locale prevista.
    popa    # Conclude la funzione con EAX
    mov    rs_i(%ebp), %eax    # pari al valore salvato nella
    leave    # variabile locale.
    ret     #
```

```
; Ricerca sequenziale.
;
section .data
```

```
lista: dd 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 ; Interi senza
                                           ; segno.

a:      dd 0          ; Indice minimo.
z:      dd 9          ; Indice massimo.
;
section .text
global _start
;
; Main.
;
_start:
    push long [z] ; f_rs ($lista, $7, a, z) ==> EAX
    push long [a] ; Si cerca il valore 7 nell'array
    push long 7   ; «lista», tra gli indici «a» e «z».
    push lista   ; Viene restituito l'indice dell'elemento
    call f_rs    ; trovato, oppure -1 se non è presente.
    add esp, 16  ;

bp1:
    mov ebx, eax ; Restituisce l'indice trovato,
    mov eax, 1   ; ammesso che sia abbastanza piccolo
    int 0x80    ; da poter essere rappresentato come
                ; valore di uscita.

;
; Ricerca sequenziale all'interno di una lista di valori.
; f_rs (lista, x, a, z) ==> EAX
; Al termine EAX contiene l'indice del valore trovato,
; oppure -1 se questo non c'è.
;
f_rs:
    enter 4, 0
    pusha
    rs_i      equ -4          ; Gli si associa EAX.
    rs_lista  equ 8          ; Gli si associa ESI.
    rs_x      equ 12         ; Gli si associa EDX.
```

```

rs_a      equ 16
rs_z      equ 20
;
mov  esi, [rs_lista+ebp] ; ESI contiene l'indirizzo
                        ; dell'array.
mov  edx, [rs_x+ebp]    ; EDX contiene il valore
                        ; cercato.
;
mov  eax, [rs_a+ebp]    ; EAX viene usato come indice
                        ; di scansione.
f_rs_loop:
cmp  eax, [rs_z+ebp]    ; Se EAX è maggiore
ja  f_rs_non_trovato   ; dell'indice massimo,
                        ; l'elemento cercato non c'è.
;
cmp  edx, [esi+eax*4]   ; Se il valore cercato
je  f_rs_trovato       ; corrisponde a quello
                        ; dell'indice corrente,
                        ; termina la scansione.
;
inc  eax                ; Incrementa l'indice di
jmp  f_rs_loop          ; scansione e salta
                        ; all'inizio del ciclo.
;
f_rs_non_trovato:
popa                    ; Conclude la funzione con EAX = -1.
mov  eax, -1            ;
leave                    ;
ret                      ;
f_rs_trovato:
mov  [rs_i+ebp], eax    ; Salva EAX nella variabile locale
                        ; prevista.
popa                    ; Conclude la funzione con EAX pari
mov  eax, [rs_i+ebp]    ; al valore salvato nella variabile

```

```

leave          ; locale.
ret           ;

```

## 64.14.2 Ricerca binaria

«

Viene mostrato un esempio di programma contenente una funzione che esegue una ricerca binaria all'interno di un array ordinato, di interi con segno. Il metodo utilizzato si rifà a quanto descritto in pseudocodifica nella sezione [62.4.2](#). Il risultato della scansione viene emesso attraverso il valore restituito dal programma; ciò che si ottiene è precisamente l'indice dell'elemento trovato, oppure  $-1$  se nessun elemento corrisponde.

```

# Ricerca binaria.
#
.section .data
lista: .int 1, 3, 4, 7, 9, 10, 11, 22, 23, 44 # Interi con
                                           # segno.

a:      .int 0          # Indice minimo.
z:      .int 9          # Indice massimo.
#
.section .text
.globl _start
#
# Main.
#
_start:
    push z              # f_rb ($lista, $7, a, z) ==> EAX
    push a              # Si cerca il valore 7 nell'array
    push $7            # ordinato «lista», tra gli indici
    push $lista        # «a» e «z». Viene restituito l'indice
    call f_rb          # dell'elemento trovato, oppure -1 se
    add $16, %esp      # non è presente.

```

```
bp1:
    mov  %eax, %ebx # Restituisce l'indice trovato,
    mov  $1, %eax  # ammesso che sia abbastanza piccolo
    int  $0x80     # da poter essere rappresentato come
                  # valore di uscita.

#
# Ricerca binaria all'interno di una lista ordinata di
# valori.
# f_rb (lista, x, a, z) ==> EAX
# Al termine EAX contiene l'indice del valore trovato,
# oppure -1 se questo non c'è.
#
f_rb:
    enter $4, $0
    pusha
    .equ rb_m,      -4          # Gli si associa EAX.
    .equ rb_lista,  8          # Gli si associa ESI.
    .equ rb_x,     12          # Gli si associa EDX.
    .equ rb_a,     16
    .equ rb_z,     20
    #
    mov  rb_lista(%ebp), %esi # ESI contiene l'indirizzo
                              # dell'array.
    mov  rb_x(%ebp),     %edx # EDX contiene il valore
                              # cercato.
                              # EAX viene usato come
                              # «elemento centrale».

    #
    mov  rb_a(%ebp), %eax  # Calcola l'indice
    add  rb_z(%ebp), %eax  # dell'elemento centrale e lo
    sar  $1, %eax         # mette in EAX. Lo scorrimento
                              # a destra serve a dividere
                              # per due EAX.

    #
```

```

bp2:
    cmp    rb_a(%ebp), %eax    # Se EAX, ovvero l'indice,
    jnb   f_rb_non_trovato    # è minore dell'indice minimo,
                                # l'elemento non c'è.

    #
    cmp    (%esi,%eax,4), %edx # Se il valore cercato è
    jnl   f_rb_minore        # minore di quello trovato,
    jg    f_rb_maggiore     # cerca nella parte inferiore;
    je    f_rb_fine         # se è maggiore cerca in
                                # quella superiore; se è
                                # uguale, l'elemento è stato
                                # trovato.

    #
f_rb_minore:
    dec    %eax
    push   %eax              # f_rb (lista, x, a, z) ==> EAX
    push   rb_a(%ebp)       # Si cerca il valore nell'array
    push   %edx              # ordinato «lista», tra gli indici
    push   %esi              # «a» e «z». Viene restituito l'indice
    call   f_rb              # dell'elemento trovato, oppure -1 se
    add    $16, %esp        # non è presente.
    jmp    f_rb_fine
    #
f_rb_maggiore:
    inc    %eax
    push   rb_z(%ebp)       # f_rb (lista, x, a, z) ==> EAX
    push   %eax              # Si cerca il valore nell'array
    push   %edx              # ordinato «lista», tra gli indici
    push   %esi              # «a» e «z». Viene restituito l'indice
    call   f_rb              # dell'elemento trovato, oppure -1 se
    add    $16, %esp        # non è presente.
    jmp    f_rb_fine
    #
f_rb_non_trovato:

```

```

    mov $-1, %eax    # Conclude la funzione con EAX = -1.
    jmp f_rb_fine
f_rb_fine:
    mov %eax, rb_m(%ebp) # Salva EAX nella variabile locale
    popa                # prevista. Conclude la funzione
    mov rb_m(%ebp), %eax # con EAX pari al valore salvato
    leave               # nella variabile locale.
    ret                 #

```

```

; Ricerca binaria.
;
section .data
lista:  dd 1, 3, 4, 7, 9, 10, 11, 22, 23, 44 ; Interi con
                                           ; segno.

a:      dd 0                ; Indice minimo.
z:      dd 9                ; Indice massimo.
;
section .text
global _start
;
; Main.
;
_start:
    push long [z] ; f_rb ($lista, $7, a, z) ==> EAX
    push long [a] ; Si cerca il valore 7 nell'array ordinato
    push long 7   ; «lista», tra gli indici «a» e «z».
    push lista   ; Viene restituito l'indice dell'elemento
    call f_rb    ; trovato, oppure -1 se non è presente.
    add esp, 16 ;
bp1:
    mov ebx, eax ; Restituisce l'indice trovato,
    mov eax, 1   ; ammesso che sia abbastanza piccolo
    int 0x80    ; da poter essere rappresentato come
                ; valore di uscita.

```

```
;
; Ricerca binaria all'interno di una lista ordinata di
; valori.
; f_rb (lista, x, a, z) ==> EAX
; Al termine EAX contiene l'indice del valore trovato,
; oppure -1 se questo non c'è.
;
f_rb:
    enter 4, 0
    pusha
    rb_m      equ -4           ; Gli si associa EAX.
    rb_lista  equ  8           ; Gli si associa ESI.
    rb_x      equ 12           ; Gli si associa EDX.
    rb_a      equ 16
    rb_z      equ 20
;
    mov esi, [rb_lista+ebp] ; ESI contiene l'indirizzo
                           ; dell'array.
    mov edx, [rb_x+ebp]     ; EDX contiene il valore
                           ; cercato.
                           ; EAX viene usato come
                           ; «elemento centrale».
;
    mov eax, [rb_a+ebp]     ; Calcola l'indice dell'elemento
    add eax, [rb_z+ebp]     ; centrale e lo mette in EAX. Lo
    sar eax, 1              ; scorrimento a destra serve a
                           ; dividere per due EAX.
;
bp2:
    cmp eax, [rb_a+ebp]     ; Se EAX, ovvero l'indice, è
    jb  f_rb_non_trovato   ; minore dell'indice minimo,
                           ; l'elemento non c'è.
;
    cmp edx, [esi+eax*4]    ; Se il valore cercato è minore
```



```
    jl  f_rb_minore           ; di quello trovato, cerca nella
    jg  f_rb_maggiore        ; parte inferiore; se è maggiore
    je  f_rb_fine            ; cerca in quella superiore; se
                               ; è uguale, l'elemento è stato
                               ; trovato.

;
f_rb_minore:
    dec  eax
    push eax                 ; f_rb (lista, x, a, z) ==> EAX
    push long [rb_a+ebp]    ; Si cerca il valore nell'array
    push edx                 ; ordinato «lista», tra gli
    push esi                 ; indici «a» e «z». Viene
    call f_rb                ; restituito l'indice
    add  esp, 16             ; dell'elemento trovato, oppure
    jmp  f_rb_fine          ; -1 se non è presente.

;
f_rb_maggiore:
    inc  eax
    push long [rb_z+ebp]    ; f_rb (lista, x, a, z) ==> EAX
    push eax                 ; Si cerca il valore nell'array
    push edx                 ; ordinato «lista», tra gli
    push esi                 ; indici «a» e «z». Viene
    call f_rb                ; restituito l'indice
    add  esp, 16             ; dell'elemento trovato, oppure
                               ; -1 se non è presente.

    jmp  f_rb_fine

;
f_rb_non_trovato:
    mov  eax, -1             ; Conclude la funzione con
    jmp  f_rb_fine          ; EAX = -1.
f_rb_fine:
    mov  [rb_m+ebp], eax    ; Salva EAX nella variabile
    popa                    ; locale prevista. Conclude la
    mov  eax, [rb_m+ebp]    ; funzione con EAX pari al
```

leave	; valore salvato nella variabile
ret	; locale.

### 64.14.3 Bubblesort

«

Viene mostrato un esempio di programma che esegue il riordino di array attraverso l'algoritmo Bubblesort. L'array in questione contiene numeri interi con segno. L'algoritmo è descritto in pseudocodifica nella sezione [62.5.1](#).

```
# Bubblesort
#
.section .data
lista: .int 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 # Interi con
                                           # segno.
a:     .int 0                               # Indice minimo.
z:     .int 9                               # Indice massimo.
#
.section .text
.globl _start
#
# Main.
#
_start:
    push z      # f_bs ($lista, a, z)
    push a     # Riordina l'array «lista» senza
    push $lista # restituire alcun valore.
    call f_bs  #
    add $12, %esp #
bp1:
    mov $0, %ebx # Restituisce sempre zero.
    mov $1, %eax #
    int $0x80   #
#
```

```
# Riordino con l'algoritmo «bubblesort».
# f_bs (lista, a, z)
#
f_bs:
    enter $0, $0
    pusha
    .equ bs_lista, 8      # EDX
    .equ bs_a, 12        #
    .equ bs_z, 16        #
                          #
    mov bs_lista(%ebp), %edx # EDX contiene il riferimento
                          # alla lista.
                          # ESI viene usato come indice
                          # di scansione.
                          # EDI viene usato come indice
                          # di scansione.
                          # EAX viene usato per
                          # scambiare i valori.
                          # EBX viene usato per
                          # scambiare i valori.
    mov bs_a(%ebp), %esi  # ESI parte dall'indice
                          # iniziale.
f_bs_loop_1:
    cmp bs_z(%ebp), %esi  # Se ESI >= z, termina.
    jae f_bs_end_loop_1  #
    #
    mov %esi, %edi        # EDI := ESI - 1
    inc %edi              #
f_bs_loop_2:
    cmp bs_z(%ebp), %edi  # Se EDI > z, termina.
    ja f_bs_end_loop_2   #
    #
    mov (%edx,%esi,4), %eax # Se EBX < EAX scambia
    mov (%edx,%edi,4), %ebx # i valori
```

```

    cmp    %eax, %ebx          #
    jl     f_bs_scambio      #
f_bs_loop_2_inc_edi:
    inc    %edi              # EDI++
    jmp    f_bs_loop_2      #
f_bs_scambio:
    mov    %eax, (%edx,%edi,4) # lista[ESI] :=: lista[EDI]
    mov    %ebx, (%edx,%esi,4) #
    jmp    f_bs_loop_2      #
f_bs_end_loop_2:
    inc    %esi              # ESI++
    jmp    f_bs_loop_1      #
f_bs_end_loop_1:
    popa
    leave
    ret

```

```

; Bubblesort
;
section .data
lista: dd 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 ; Interi con
                                           ; segno.
a:      dd 0                  ; Indice minimo.
z:      dd 9                  ; Indice massimo.
;
section .text
global _start
;
; Main.
;
_start:
    push  long [z]           ; f_bs ($lista, a, z)
    push  long [a]          ; Riordina l'array «lista» senza
    push  long lista        ; restituire alcun valore.

```

```
    call  f_bs      ;
    add   esp, 12   ;
bp1:
    mov   ebx, 0    ; Restituisce sempre zero.
    mov   eax, 1    ;
    int   0x80     ;
;
; Riordino con l'algoritmo «bubblesort».
; f_bs (lista, a, z)
;
f_bs:
    enter 0, 0
    pusha
    bs_lista equ 8      ; EDX
    bs_a      equ 12    ;
    bs_z      equ 16    ;
;
    mov   edx, [bs_lista+ebp] ; EDX contiene il riferimento
; alla lista.
; ESI viene usato come indice
; di scansione.
; EDI viene usato come indice
; di scansione.
; EAX viene usato per
; scambiare i valori.
; EBX viene usato per
; scambiare i valori.
    mov   esi, [bs_a+ebp] ; ESI parte dall'indice
; iniziale.
f_bs_loop_1:
    cmp   esi, [bs_z+ebp] ; Se ESI >= z, termina.
    jae   f_bs_end_loop_1 ;
;
    mov   edi, esi      ; EDI := ESI - 1
```

```

    inc    edi                                ;
f_bs_loop_2:
    cmp    edi, [bs_z+ebp]                   ; Se EDI > z, termina.
    ja     f_bs_end_loop_2                   ;
    ;
    mov    eax, [edx+esi*4]                  ; Se EBX < EAX scambia
    mov    ebx, [edx+edi*4]                  ; i valori.
    cmp    ebx, eax                           ;
    jl     f_bs_scambio                       ;
f_bs_loop_2_inc_edi:
    inc    edi                                ; EDI++
    jmp    f_bs_loop_2                       ;
f_bs_scambio:
    mov    [edx+edi*4], eax                   ; lista[ESI] ::= lista[EDI]
    mov    [edx+esi*4], ebx                   ;
    jmp    f_bs_loop_2                       ;
f_bs_end_loop_2:
    inc    esi                                ; ESI++
    jmp    f_bs_loop_1                       ;
f_bs_end_loop_1:
    popa
    leave
    ret

```

Per verificare il funzionamento del programma si deve usare necessariamente GDB. Inizialmente, prima di mettere in esecuzione il programma, si vede l'array nel suo stato originale:

```
(gdb) print (int[10])lista [Invio]
```

```
$1 = {1, 4, 3, 7, 9, 10, 22, 44, 11, 23}
```

Si fissa quindi uno stop e si avvia il programma:

```
(gdb) break bp1 [Invio]
```

```
(gdb) run [Invio]
```

Quando il programma viene sospeso in corrispondenza di **'bp1'**, l'array è ordinato:

```
(gdb) print (int[10])lista [Invio]
```

```
$1 = {1, 3, 4, 7, 9, 10, 11, 22, 23, 44}
```

## 64.15 Calcoli con gli indirizzi in fase di compilazione

Attraverso le funzionalità del compilatore è possibile calcolare la distanza tra due indirizzi, espressa in byte. È anche possibile fare riferimento all'indirizzo attuale, attraverso un simbolo predefinito. Nelle sezioni successive vengono mostrati alcuni esempi per dimostrare l'uso di queste funzionalità.

### 64.15.1 Distanza tra due indirizzi

L'esempio seguente serve a dimostrare come il compilatore possa calcolare la distanza tra due indirizzi, contrassegnati da dei simboli, inizializzando con tale valore calcolato una variabile globale. In pratica, viene calcolata la grandezza complessiva in byte di un array di numeri interi; grandezza che viene poi emessa come valore di uscita.

```
#
.section .data
lista:  .int    1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:    .int    (z - lista)
#
.section .text
.globl _start
#
_start:
```

```

mov    z,    %ebx
mov    $1,   %eax
int    $0x80

```

```

;
section .data
lista:  dd      1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      dd      (z - lista)
;
section .text
global _start
;
_start:
    mov    ebx, [z]
    mov    eax, 1
    int   0x80

```

Il significato dell'istruzione è intuitivo: alla variabile 'z' si assegna la differenza tra gli indirizzi utilizzati da 'lista' a 'z'. In questo caso, il risultato che si ottiene è 40, dal momento che si contano 10 valori da 4 byte ciascuno. Eventualmente, si può fare riferimento alla posizione attuale in modo differente, in questo caso significa sostituire il riferimento esplicito alla variabile 'z' con un riferimento implicito:

```

...
lista:  .int    1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      .int    (. - lista)
...

```

```

...
lista:  dd      1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      dd      ($ - lista)

```



```
...
```

Si possono anche fare dei calcoli più complessi, come nel caso dell'esempio seguente in cui si determina l'indice superiore dell'array. Il risultato che si ottiene è nove, dal momento che l'indice del primo elemento deve essere zero.

```
...
lista:  .int    1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      .int    (z - lista) / 4 - 1
...
```

```
...
lista:  dd     1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      dd     (z - lista) / 4 - 1
...
```

## 64.15.2 Riempimento di spazio inutilizzato

In certe situazioni è necessario riempire una certa area di memoria (o di codice) in modo che complessivamente occupi una dimensione data. Questo procedimento si usa specialmente quando si genera un file binario, privo di formato (come nel caso di un settore di avvio), che deve avere una dimensione stabilita e che in una certa posizione deve contenere un'impronta determinata. Qui viene dimostrato il concetto intervenendo solo nell'area della memoria che viene inizializzata.



```

#
.section .data
lista: .int    1, 4, 3, 7, 9, 10, 22, 44, 11, 23
        .skip (0x30 - (. - lista)), 0xFF

#
.section .text
.globl _start
#
_start:
    mov    $0, %ebx
    mov    $1, %eax
    int   $0x80

```

```

;
section .data
lista: dd      1, 4, 3, 7, 9, 10, 22, 44, 11, 23
times    (30h - ($ - lista)) db  0xFF

;
section .text
global _start
;
_start:
    mov    ebx, 0
    mov    eax, 1
    int   0x80

```

In questo caso, dopo la definizione dell'array, si richiede al compilatore di allocare altro spazio di memoria, in modo da occupare complessivamente 48 byte ( $30_{16}$ ), riempiendo lo spazio ulteriore con caratteri  $FF_{16}$ . Naturalmente, il valore complessivo dello spazio da utilizzare può essere espresso in qualunque base di numerazione; in questo caso, la scelta di rappresentare in base sedici è motivata dal fatto che con NASM non si può usare la forma consueta (non si può

scrivere ‘**0x30**’), perché si otterrebbe un risultato differente, a causa di un errore di interpretazione da parte del compilatore.

In quasi tutti gli esempi di questo capitolo, realizzati per il compilatore NASM, si usa la notazione ‘**0xnn...**’ per esprimere un numero in base sedici. Generalmente l’interpretazione da parte di NASM è corretta, ma nella situazione particolare mostrata, il compilatore si confonde e riconosce solo la forma ‘**nn...h**’.

Dal momento che l’array occupa già 40 byte, vengono aggiunti semplicemente 8 byte, pari a due gruppi da 32 bit:

```
(gdb) print /x (int[12])lista [Invio]
```

```
$1 = {0x1, 0x4, 0x3, 0x7, 0x9, 0xa, 0x16, 0x2c, 0xb, 0x17,
      0xffffffff, 0xffffffff}
```

## 64.16 Interazione con il sistema operativo

È possibile gestire un certo grado di comunicazione tra il programma in linguaggio assembler e il sistema GNU/Linux. In particolare si possono ottenere i parametri della chiamata del programma (gli argomenti della riga di comando) ed è possibile chiamare delle funzioni di sistema attraverso delle «interruzioni».

### 64.16.1 Parametri di chiamata del programma

All’avvio del programma, questo riceve una pila contenente il numero degli argomenti della riga di comando (nome del programma incluso), il nome del programma che è stato avviato e quindi gli argomenti stessi. Si può realizzare un sorgente molto semplice per l’indagine con GDB:

```
.section .text
.globl _start
_start:
    mov    %esp, %ebp
bp1:
    mov    $0, %ebx    # Restituisce zero.
    mov    $1, %eax    #
    int   $0x80       #
```

```
section .text
global _start
_start:
    mov    ebp, esp
bp1:
    mov    ebx, 0      ; Restituisce zero.
    mov    eax, 1      ;
    int   0x80        ;
```

Supponendo che il programma compilato si chiami **‘argomenti’**, lo si avvia sotto il controllo di GDB nello stesso modo di sempre:

```
$ gdb argomenti [Invio]
```

Gli argomenti del programma vanno passati necessariamente attraverso un comando di GDB:

```
(gdb) set args 1 2 3 4 "ciao amore" [Invio]
```

Si fissa il punto di sospensione del programma e quindi si avvia:

```
(gdb) break bp1 [Invio]
```

```
(gdb) run [Invio]
```

Il programma viene sospeso in corrispondenza del simbolo **‘bp1’** e si può consultare la pila, o più precisamente lo *stack frame* della pila:

```
(gdb) backtrace [Invio]
```

```
#0  bp1 () at argomenti.s:6
#1  0x00000006 in ?? ()
#2  0xbfeb1a89 in ?? ()
#3  0xbfeb1a98 in ?? ()
#4  0xbfeb1a9a in ?? ()
#5  0xbfeb1a9c in ?? ()
#6  0xbfeb1a9e in ?? ()
#7  0xbfeb1aa0 in ?? ()
#8  0x00000000 in ?? ()
```

In questa situazione non sono presenti variabili locali; quindi, nella pila, dopo l'indirizzo corrispondente al simbolo '**bp1**' (che si trova lì solo perché si sta usando GDB ed è stato sospeso il corso del programma), appare la quantità di argomenti (sei); gli elementi successivi contengono dei puntatori alle stringhe che rappresentano i vari argomenti ricevuti (si osservi che gli argomenti sono rappresentati tutti in forma di stringa), stringhe che sono tutte terminate con un byte a zero. Per leggere gli argomenti con GDB si devono fare dei tentativi; qui vengono indicate le dimensioni esatte, ma se si usano dimensioni maggiori si possono vedere delle porzioni degli argomenti successivi:

```
(gdb) print (char[33])*0xbfeb1a89 [Invio]
```

```
$1 = "/tmp/argomenti\0001\0002\0003\0004\000ciao amore"
```

In questo caso si legge il primo argomento, ma usando una dimensione eccessiva, si vedono di seguito anche gli altri, separati dai vari byte a zero, rappresentati con la sequenza '**\000**'. Il primo argomento da solo sarebbe:

```
(gdb) print (char[14])*0xbfeb1a89 [Invio]
```

```
$2 = "/tmp/argomenti"
```

## Gli altri argomenti:

```
(gdb) print (char[1])*0xbfef1a98 [Invio]
```

```
$3 = "1"
```

```
(gdb) print (char[1])*0xbfef1a9a [Invio]
```

```
$4 = "2"
```

```
(gdb) print (char[1])*0xbfef1a9c [Invio]
```

```
$5 = "3"
```

```
(gdb) print (char[1])*0xbfef1a9e [Invio]
```

```
$6 = "4"
```

```
(gdb) print (char[9])*0xbfef1aa0 [Invio]
```

```
$7 = "ciao amore"
```

```
(gdb) quit [Invio]
```

## 64.16.2 Funzioni del sistema operativo

«

Si accede alle funzioni offerte dal sistema operativo attraverso quella che è nota come «interruzione software» (*interrupt*) e si genera con l'istruzione '**INT**'. Per la precisione, in un sistema GNU/Linux occorre l'interruzione  $80_{16}$  ( $128_{10}$ ), come è stato mostrato in tutti gli esempi apparsi fino a questo punto. Per selezionare il tipo di funzione e per passarle degli argomenti si usano i registri in questo modo:

Regi- stro	Utilizzo
<b><i>EAX</i></b>	Contiene il numero che rappresenta la funzione di sistema.
<b><i>EBX</i></b>	Primo parametro della funzione.
<b><i>ECX</i></b>	Secondo parametro della funzione.
<b><i>EDX</i></b>	Terzo parametro della funzione.
<b><i>ESI</i></b>	Quarto parametro della funzione.
<b><i>EDI</i></b>	Quinto parametro della funzione.

Se la funzione deve restituire un valore, questo viene ottenuto attraverso il registro ***EAX***.

Per una mappa completa delle chiamate di sistema si può consultare [http://wayback.archive.org/web/\\*/www.lxhp.in-berlin.de/lhpsysc0.html](http://wayback.archive.org/web/*/www.lxhp.in-berlin.de/lhpsysc0.html), come annotato nei riferimenti alla fine del capitolo. Qui vengono mostrate delle tabelle riepilogative di alcune funzioni importanti.

Pagi- na di ma- nuale	Descrizione	<b><i>EAX</i></b>	<b><i>EBX</i></b>	<b><i>ECX</i></b>	<b><i>EDX</i></b>
<i>exit(2)</i>	Conclude il funziona- mento del programma restituendo un valore.	1	Valore da restituire: numero intero com- preso tra zero e 255.		

Pagina di manuale	Descrizione	<i>EAX</i>	<i>EBX</i>	<i>ECX</i>	<i>EDX</i>
<i>read(2)</i>	Legge da un descrittore di file e mette i dati letti in una memoria tampone. Attraverso <i>EAX</i> restituisce la quantità di byte letta effettivamente e aggiorna il puntatore del file per la lettura successiva.	3	Descrittore del file da leggere.	Indirizzo iniziale della memoria tampone.	Dimensione in byte della memoria tampone.
<i>write(2)</i>	Scriva il contenuto di una memoria tampone in un descrittore di file. Attraverso <i>EAX</i> restituisce la quantità di byte scritta effettivamente.	4	Descrittore del file da scrivere.	Indirizzo iniziale della memoria tampone.	Dimensione in byte della memoria tampone.

### 64.16.3 Esempi di lettura e scrittura con i flussi standard

«

Di solito, il primo programma che si scrive è quello che visualizza un messaggio e termina, ma in questo caso, un'operazione così semplice sul piano teorico, in pratica è già abbastanza complicata. Quello che segue è un programma che, avvalendosi di una chiamata di sistema, visualizza un messaggio attraverso lo standard output. Come si può osservare, si utilizza anche una tecnica per far calcolare al compilatore la lunghezza della stringa da visualizzare.

```
#
.equ SYS_EXIT, 1 # exit(2)
.equ SYS_WRITE, 4 # write(2)
.equ STDOUT, 1 # Descrittore di standard output.
#
```



```

.section .data                                # Qui si dichiara la stringa
msg: .ascii "Ciao a tutti!\n"                # da visualizzare,
size = . - msg                               # calcolandone la
                                                # dimensione.

#
.section .text
.globl _start
_start:
    mov    $SYS_WRITE, %eax                    # Scrive nello standard
    mov    $STDOUT,    %ebx                    # output.
    mov    $msg,       %ecx                    #
    mov    $size,     %edx                    #
    int   $0x80

exit:
    mov    $SYS_EXIT,  %eax                    # Conclude il funzionamento.
    mov    $0,         %ebx                    #
    int   $0x80

```

```

;
SYS_EXIT    equ    1    ; exit(2)
SYS_WRITE   equ    4    ; write(2)
STDOUT      equ    1    ; Descrittore di standard output.
;
section .data                                ; Qui si dichiara la stringa
msg: db "Ciao a tutti!", 0x0A                ; da visualizzare,
size equ $ - msg                               ; calcolandone la
                                                ; dimensione.

;
section text
global _start
_start:
    mov    eax, SYS_WRITE                      ; Scrive nello standard
    mov    ebx, STDOUT                          ; output.
    mov    ecx, msg                             ;

```

```
    mov    edx, size          ;
    int    0x80
exit:
    mov    eax, SYS_EXIT     ; Conclude il funzionamento.
    mov    ebx, 0           ;
    int    0x80
```

Segue un esempio di programma che legge dallo standard input e scrive ciò che ha letto attraverso lo standard output. Come già nell'esempio precedente, vengono dichiarate inizialmente delle costanti per semplificare la lettura del codice; inoltre vengono usate aree di memoria non inizializzate e delle funzioni banali senza parametri, per le quali non si utilizzano variabili locali. Si mostrano due listati, uno adatto per GNU AS e l'altro per NASM.

```
#
.equ SYS_EXIT, 1    # exit(2)
.equ SYS_READ, 3    # read(2)
.equ SYS_WRITE, 4   # write(2)
.equ STDIN, 0      # Descrittore di standard input.
.equ STDOUT, 1     # Descrittore di standard output.
.equ STDERR, 2     # Descrittore di standard error.
.equ MAX_SIZE, 1000 # Dimensione massima dei dati da
                    # leggere.

#
.section .data      # Non ci sono variabili già
                    # inizializzate.

#
.section .bss
.lcomm record, MAX_SIZE # Memoria tampone per la lettura dei
                        # dati.
.lcomm size, 4         # Quantità di byte letti
                        # effettivamente.
```

```
#
.section .text
.globl _start
_start:
read_write_begin:
    call    read                # Legge dallo standard input.
    cmp     $0, %eax            # Se sono stati letti zero byte,
    jz     read_write_end      # il ciclo termina.
    call    write               # Scrive i byte letti nello
                                # standard output.
    jmp    read_write_begin    # Ripete il ciclo.
read_write_end:
    jmp    exit
read:
    mov     $SYS_READ, %eax     # Legge dallo standard input.
    mov     $STDIN, %ebx       #
    mov     $record, %ecx      #
    mov     $MAX_SIZE, %edx    #
    int    $0x80
    mov     %eax, size        # Salva la dimensione letta
                                # effettivamente.
    ret
write:
    mov     $SYS_WRITE, %eax    # Scrive nello standard output.
    mov     $STDOUT, %ebx      #
    mov     $record, %ecx      #
    mov     size, %edx         #
    int    $0x80
    ret
exit:
    mov     $SYS_EXIT, %eax     # Conclude il funzionamento.
    mov     $0, %ebx           #
    int    $0x80
```

```
;
SYS_EXIT    equ    1 ; exit(2)
SYS_READ    equ    3 ; read(2)
SYS_WRITE   equ    4 ; write(2)
STDIN       equ    0 ; Descrittore di standard input.
STDOUT      equ    1 ; Descrittore di standard output.
STDERR      equ    2 ; Descrittore di standard error.
MAX_SIZE    equ    1000 ; Dimensione massima dei dati da
                        ; leggere.

;
section .data          ; Non ci sono variabili già
                        ; inizializzate.

;
section .bss
record resb MAX_SIZE ; Memoria tampone per la lettura dei
                        ; dati.
size    resd 1        ; Quantità di byte letti
                        ; effettivamente.

;
section .text
global _start
_start:
read_write_begin:
    call    read          ; Legge dallo standard input.
    cmp    eax, 0        ; Se sono stati letti zero byte,
    jz     read_write_end ; il ciclo termina.
    call    write         ; Scrive i byte letti nello
                        ; standard output.
    jmp    read_write_begin ; Ripete il ciclo.
read_write_end:
    jmp    exit
read:
    mov    eax, SYS_READ ; Legge dallo standard input.
    mov    ebx, STDIN    ;
```

```
    mov    ecx, record    ;
    mov    edx, MAX_SIZE  ;
    int    0x80
    mov    [size], eax    ; Salva la dimensione letta
                          ; effettivamente.

    ret
write:
    mov    eax, SYS_WRITE ; Scrive nello standard output.
    mov    ebx, STDOUT    ;
    mov    ecx, record    ;
    mov    edx, [size]    ;
    int    0x80
    ret
exit:
    mov    eax, SYS_EXIT  ; Conclude il funzionamento.
    mov    ebx, 0         ;
    int    0x80
```

## 64.17 Riferimenti



- *Intel 80386 Reference Programmer's Manual*, <http://pdos.csail.mit.edu/6.828/2006/readings/i386/toc.htm>
  - *80386 Instruction Set*, <http://pdos.csail.mit.edu/6.828/2006/readings/i386/c17.htm>
- *Using Assembly Language in Linux*, <http://asm.sourceforge.net/articles/linasm.html>
- Norman Matloff, *Introduction to Linux Intel Assembly Language*, <http://heather.cs.ucdavis.edu/~matloff/50/LinuxAssembly.html>

- H.-Peter Recktenwald, *i386-PC-Linux System Calls*, 2000, [http://wayback.archive.org/web/\\*/www.lxhp.in-berlin.de/lhpsysc0.html](http://wayback.archive.org/web/*/www.lxhp.in-berlin.de/lhpsysc0.html)

<sup>1</sup> **GNU Binutils** GNU GPL

<sup>2</sup> **NASM** GNU LGPL

<sup>3</sup> **GNU Binutils** GNU GPL

<sup>4</sup> **GDB** GNU GPL

<sup>5</sup> **DDD** GNU GPL

<sup>6</sup> Il formato ELF prevede altri tipi di sezione, ma quelle di uso più frequente sono rappresentate nel modello sintattico.

<sup>7</sup> Evidentemente, il valore di uscita viene espresso in base otto:  $24_8$  è uguale a  $20_{10}$ .

<sup>8</sup> Dipende dal compilatore se è possibile limitare effettivamente l'uso al solo registro **CX**.

<sup>9</sup> L'indirizzo di memoria da raggiungere con l'istruzione '**CALL**', può essere fornito in modo «immediato», attraverso l'indicazione di un simbolo, oppure con un registro o con un indirizzo di memoria. Nell'ipotesi di un registro o di un indirizzo di memoria, si intende che il contenuto del registro o della variabile in memoria vadano considerati come l'indirizzo di destinazione della chiamata.

<sup>10</sup> Nel caso di un valore in virgola mobile, il risultato potrebbe essere atteso dal registro **ST0**, ma la gestione della virgola mobile non viene affrontata in questo capitolo.

# Compilazione e formato binario eseguibile



65.1	Compilazione di programmi composti da più file sorgenti	410
65.1.1	Inclusione di file	410
65.1.2	Due file sorgenti da collegare assieme	411
65.1.3	Incorporazione di codice in linguaggio C	418
65.2	Librerie dinamiche e librerie statiche	419
65.2.1	Il processo di «collegamento» dinamico	420
65.2.2	Creazione di una libreria dinamica	421
65.2.3	Creare un programma che utilizza una libreria dinamica	426
65.2.4	Creare un file che utilizza una libreria dinamica standard	429
65.2.5	Librerie statiche	432
65.3	Dal sorgente all'immagine in memoria	432
65.3.1	File oggetto	433
65.3.2	File eseguibile	435
65.3.3	Immagine del processo nella memoria virtuale	439
65.3.4	Allineamento dei segmenti in memoria	442
65.3.5	Script per il collegamento	443
65.3.6	Osservazioni sui simboli	449
65.3.7	Formati dei file oggetto	452
65.4	Formato ELF	453

65.4.1	Sezioni e segmenti .....	453
65.4.2	Intestazione ELF .....	454
65.4.3	Descrizione dei segmenti .....	459
65.4.4	Definizione manuale di un formato ELF .....	462
65.4.5	Esempio più complesso .....	467
65.5	Programmi completamente autonomi .....	470
65.5.1	Le specifiche «multiboot» .....	471
65.5.2	Esempio di programma da avviare secondo le specifiche «multiboot» .....	478
65.5.3	Visualizzazione di messaggi .....	482
65.5.4	Colori dello schermo .....	493
65.6	Compilazione C dal basso in alto .....	495
65.6.1	Compilazione di un programma che non fa uso di librerie .....	495
65.6.2	Uso di GDB e di DDD .....	499
65.6.3	Da «_start» a «main» .....	504
65.6.4	Compilazione naturale di un programma in linguaggio C .....	507
65.7	Compilazione C dall'alto in basso .....	509
65.7.1	Le fasi della compilazione .....	510
65.7.2	Precompilatore .....	511
65.7.3	Compilazione dei file intermedi .....	512
65.7.4	L'uso di librerie .....	513
65.7.5	Librerie statiche e librerie dinamiche .....	514



65.7.6	L'ordine dei file e delle librerie nella compilazione	517
65.7.7	Prevenzione e ricerca degli errori	518
65.7.8	Problemi con l'ottimizzazione	520
65.8	Compilazione guidata con Make	523
65.8.1	Obiettivo, dipendenze e comandi	524
65.8.2	Obiettivi fittizi	527
65.8.3	Scelta dell'obiettivo	529
65.8.4	Interpretazione dei comandi che portano a un obiettivo	530
65.8.5	Variabili o «macro»	533
65.8.6	Utilizzo oculato delle variabili	538
65.8.7	Espansione e continuazione al di fuori dei comandi	539
65.8.8	Variabili automatiche	540
65.8.9	Regole implicite	542
65.8.10	Uno script per ogni sottodirectory	544
65.8.11	Una regola per più obiettivi	546
65.8.12	Regole fittizie tipiche	547
65.8.13	Variabili per l'installazione	549
65.8.14	Definizione della shell	550
65.8.15	Installazione dei programmi	551
65.9	Riferimenti	552

## 65.1 Compilazione di programmi composti da più file sorgenti

&lt;&lt;

Per poter compilare un programma distribuito tra più file sorgenti, all'interno di questi file occorre dichiarare quali simboli (riferiti a variabili e funzioni) devono essere pubblici e come tali accessibili anche dagli altri; inoltre, nei file in cui si fa riferimento a simboli esterni, occorre dichiarare questa dipendenza.

### 65.1.1 Inclusione di file

&lt;&lt;

Prima di affrontare il problema del collegamento di più file oggetto in un file eseguibile singolo, conviene considerare l'inclusione automatica del contenuto di un file. In altri termini, si può ottenere una funzione simile al «copia-incolla», dichiarando in un file che, in un certo punto, va incluso il contenuto di un altro. Per esempio, per incorporare in un certo punto, il contenuto del file 'funzioni.s', occorre scrivere la direttiva seguente:

```
...  
; GNU AS  
.include "funzioni.s"  
...
```

```
...  
; NASM  
%include "funzioni.s"  
...
```

Naturalmente, il file 'funzioni.s' contiene qualcosa che si può copiare e incollare, tale e quale, in quel certo punto del sorgente che



```

push  a           # Si cerca il valore 7 nell'array
push  $7          # «lista», tra gli indici «a» e «z».
push  $lista      # Viene restituito l'indice
call  f_rs        # dell'elemento trovato, oppure -1 se
add   $16, %esp   # non è presente.

bp1:
mov   %eax, %ebx  # Restituisce l'indice trovato,
mov   $1, %eax    # ammesso che sia abbastanza piccolo
int   $0x80       # da poter essere rappresentato come
                    # valore di uscita.

```

```

# rs-f.s
#
.section .data
#
.section .text
.globl f_rs
#
# Ricerca sequenziale all'interno di una lista di valori.
# f_rs (lista, x, a, z) ==> EAX
# Al termine EAX contiene l'indice del valore trovato,
# oppure -1 se questo non c'è.
#
f_rs:
    enter $4, $0
    pusha
    .equ  rs_i,      -4           # Gli si associa EAX.
    .equ  rs_lista,  8           # Gli si associa ESI.
    .equ  rs_x,     12          # Gli si associa EDX.
    .equ  rs_a,     16
    .equ  rs_z,     20
    #
    mov   rs_lista(%ebp), %esi  # ESI contiene l'indirizzo
                                # dell'array.

```

```

    mov    rs_x(%ebp),    %edx # EDX contiene il valore
                                # cercato.
    #
    mov    rs_a(%ebp),    %eax # EAX viene usato come indice
                                # di scansione.
f_rs_loop:
    cmp    rs_z(%ebp),    %eax # Se EAX è maggiore
    ja    f_rs_non_trovato    # dell'indice massimo,
                                # l'elemento cercato non c'è.
    #
    cmp    (%esi,%eax,4), %edx # Se il valore cercato
    je    f_rs_trovato        # corrisponde a quello
                                # dell'indice corrente,
                                # termina la scansione.
    #
    inc    %eax                # Incrementa l'indice di
    jmp    f_rs_loop          # scansione e salta
                                # all'inizio del ciclo.
    #
f_rs_non_trovato:
    popa                        # Conclude la funzione con EAX = -1.
    mov    $-1, %eax           #
    leave                       #
    ret                          #
f_rs_trovato:
    mov    %eax, rs_i(%ebp)    # Salva EAX nella variabile
                                # locale prevista.
    popa                        # Conclude la funzione con EAX
    mov    rs_i(%ebp), %eax    # pari al valore salvato nella
    leave                       # variabile locale.
    ret                          #

```

Nel primo dei due listati, corrispondente al file 'rs-main.s', si deve osservare la dichiarazione esterna del simbolo '**f\_rs**', cor-

rispondente al nome della funzione contenuta nel file 'rs-f.s'.

```
...  
.section .text  
.globl _start  
.extern f_rs  
...
```

Dal momento che i dati necessari all'elaborazione vengono passati alla funzione attraverso i parametri della chiamata, a parte '**\_start**', non ci sono altre dichiarazioni di simboli pubblici nel file 'f-main.s'. Nel secondo listato, corrispondente al file 'rs-f.s', il simbolo '**f\_rs**' viene reso pubblico, per consentire al file 'rs-main.s' di farvi riferimento.

```
...  
.section .text  
.globl f_rs  
...
```

Seguono gli stessi due listati, nella versione adatta a NASM:

```
; rs-main.s  
;  
section .data  
lista: dd 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 ; Interi senza  
                                           ; segno.  
a:      dd 0                               ; Indice minimo.  
z:      dd 9                               ; Indice massimo.  
;  
section .text  
global _start  
extern f_rs
```

```

;
_start:
    push    long [z] ; f_rs ($lista, $7, a, z) ==> EAX
    push    long [a] ; Si cerca il valore 7 nell'array
    push    long 7   ; «lista», tra gli indici «a» e «z».
    push    lista   ; Viene restituito l'indice dell'elemento
    call    f_rs    ; trovato, oppure -1 se non è presente.
    add     esp, 16 ;
bp1:
    mov     ebx, eax ; Restituisce l'indice trovato,
    mov     eax, 1   ; ammesso che sia abbastanza piccolo
    int     0x80    ; da poter essere rappresentato come
                    ; valore di uscita.

```

```

; rs-f.s
;
section .data
;
section .text
global f_rs
;
; Ricerca sequenziale all'interno di una lista di valori.
; f_rs (lista, x, a, z) ==> EAX
; Al termine EAX contiene l'indice del valore trovato,
; oppure -1 se questo non c'è.
;
f_rs:
    enter 4, 0
    pusha
    rs_i      equ -4      ; Gli si associa EAX.
    rs_lista  equ  8      ; Gli si associa ESI.
    rs_x      equ 12      ; Gli si associa EDX.
    rs_a      equ 16
    rs_z      equ 20

```

```
    ;
    mov     esi, [rs_lista+ebp] ; ESI contiene l'indirizzo
                                ; dell'array.
    mov     edx, [rs_x+ebp]     ; EDX contiene il valore
                                ; cercato.

    ;
    mov     eax, [rs_a+ebp]     ; EAX viene usato come indice
                                ; di scansione.
f_rs_loop:
    cmp     eax, [rs_z+ebp]     ; Se EAX è maggiore dell'indice
    ja     f_rs_non_trovato    ; massimo, l'elemento cercato
                                ; non c'è.

    ;
    cmp     edx, [esi+eax*4]    ; Se il valore cercato
    je     f_rs_trovato        ; corrisponde a quello
                                ; dell'indice corrente,
                                ; termina la scansione.

    ;
    inc     eax                 ; Incrementa l'indice di
    jmp     f_rs_loop          ; scansione e salta all'inizio
                                ; del ciclo.

    ;
f_rs_non_trovato:
    popa                        ; Conclude la funzione con EAX = -1.
    mov     eax, -1             ;
    leave   ;
    ret     ;
f_rs_trovato:
    mov     [rs_i+ebp], eax     ; Salva EAX nella variabile
                                ; locale prevista.
    popa                        ; Conclude la funzione con EAX pari
    mov     eax, [rs_i+ebp]    ; al valore salvato nella variabile
    leave   ; locale.
    ret     ;
```



In questo caso, le direttive salienti sono, rispettivamente:

```
...
section .text
global _start
extern f_rs
...
```

```
...
section .text
global f_rs
...
```

Per compilare il tutto in un solo file eseguibile, occorre procedere secondo i comandi seguenti. Nel caso di GNU AS:

```
$ as --gstabs -o rs-main.o rs-main.s [Invio]
```

```
$ as --gstabs -o rs-f.o rs-f.s [Invio]
```

```
$ ld -o rs rs-main.o rs-f.o [Invio]
```

Nel caso di NASM:

```
$ nasm -g -f elf -o rs-main.o rs-main.s [Invio]
```

```
$ nasm -g -f elf -o rs-f.o rs-f.s [Invio]
```

```
$ ld -o rs rs-main.o rs-f.o [Invio]
```

Questo programma restituisce l'indice dell'elemento cercato e trovato nell'array. In questo caso, si tratta del quarto elemento che corrisponde all'indice 3:

```
$ ./rs ; echo $? [Invio]
```

### 65.1.3 Incorporazione di codice in linguaggio C



Per collegare assieme sorgenti scritti in linguaggi differenti, si agisce in modo analogo a quanto già mostrato per il solo linguaggio assembler. C'è però da considerare che ogni compilatore ha le proprie caratteristiche, sia per ciò che riguarda le convenzioni di chiamata delle funzioni, sia per il modo di nominare i simboli associati alle funzioni stesse. Nel caso di GCC (*GNU compiler collection*), valgono le convenzioni di chiamata comuni e i nomi delle funzioni non vengono modificati.

Qui si mostra un listato, in linguaggio C, da usare in sostituzione del file 'rs-f.s' descritto nella sezione precedente:

```
/* f_rs (<lista>, <x>, <ele-inf>, <ele-sup>) */  
  
int f_rs (int lista[], int x, int a, int z)  
{  
    int i;  
  
    /* Scandisce l'array alla ricerca dell'elemento. */  
  
    for (i = a; i <= z; i++)  
        {  
            if (x == lista[i])  
                {  
                    return i;  
                }  
        }  
  
    /* La corrispondenza non è stata trovata. */
```

```
    return -1;
}
```

Per compilare questo file e generare un file oggetto, ammesso che il sorgente si chiami `rs-f.c`, si procede con il comando seguente:

```
$ cc -c -o rs-f.o rs-f.c [Invio]
```

Il collegamento con il file `rs-main.o` avviene nel modo già visto:

```
$ ld -o rs rs-main.o rs-f.o [Invio]
```

## 65.2 Librerie dinamiche e librerie statiche

La compilazione dei programmi, secondo quanto descritto in precedenza, genera sempre file eseguibili «completi», in quanto incorporano tutto il codice necessario al proprio funzionamento. Oltre che suddividere il sorgente in file separati, da riunire assieme in un file eseguibile unico, è possibile costruire una libreria di funzioni, a cui i programmi accedono dopo essere stati avviati, senza incorporarne il codice. Una libreria di questo genere è nota come *libreria dinamica*, in quanto richiede la creazione di un «collegamento» (*link*) istantaneo, mentre il programma che la richiede è in funzione.

Il concetto di libreria dinamica si contrappone a quello di *libreria statica*, la quale comporta l'inclusione del proprio codice nel file eseguibile, in fase di compilazione.

## 65.2.1 Il processo di «collegamento» dinamico

&lt;&lt;

Il programma eseguibile che ha bisogno di utilizzare una libreria dinamica, si avvale di un altro programma che a sua volta deve eseguire il «collegamento dinamico» (*dynamic link*). Il nome di questo collegatore dinamico viene definito in fase di compilazione del primo programma e in un sistema GNU/Linux è costituito generalmente dal file `/lib/ld-linux.so.2`. A sua volta, il collegatore dinamico cerca il file contenente la libreria richiesta dal programma in un gruppo di directory che solitamente sono `/lib/`, `/usr/lib/` e altre, secondo la configurazione contenuta nel file `/etc/ld.so.conf`.

Il file `/etc/ld.so.conf` deve essere elaborato attraverso il programma `ldconfig` che a sua volta produce il file `/etc/ld.so.cache`, il quale viene interpellato effettivamente da `/lib/ld-linux.so.2`. Pertanto, quando si modifica il file `/etc/ld.so.conf`, occorre ricordarsi di riavviare `ldconfig`.

Se esiste la variabile di ambiente `LD_LIBRARY_PATH`, i file delle librerie vengono cercati nei percorsi che questa contiene. Per esempio, per utilizzare i file contenuti nella directory corrente, continuando eventualmente in altre directory consuete, basta assegnare il percorso `.`, seguito dagli altri a cui si è interessati:

```
$ export LD_LIBRARY_PATH="./lib:/usr/lib:/usr/local/lib" [Invio]
```

## 65.2.2 Creazione di una libreria dinamica



Per compilare dei file sorgenti in modo che diventino una libreria dinamica, occorre usare delle opzioni particolari in fase di collegamento (*link*) e nei file sorgenti è necessario pubblicizzare le funzioni in modo particolare. A titolo di esempio si prendono due funzioni, rispettivamente per il calcolo della potenza e del fattoriale (sono già usate nella sezione [64.11](#) in programmi compilati in modo statico), contenute in due file separati. La coppia di listati è completa e vengono mostrate entrambe le versioni per GNU AS e NASM, evidenziando le direttive significative per ottenere una libreria dinamica.

```
# lib_pwr.s
.section .text
.globl f_pwr
.type f_pwr, @function
#
f_pwr:
    enter $4, $0
    pusha
    #
    mov     8(%ebp), %esi    # Base.
    mov     12(%ebp), %edi   # Esponente.
    #
    cmp     $0, %esi        # Se la base è pari a 0,
    jz     f_pwr_end_0     # restituisce 0.
    #
    cmp     $0, %edi        # Se l'esponente è pari a 0,
    jz     f_pwr_end_1     # restituisce 1.
    #
    dec     %edi            # Riduce l'esponente di una unità.
    push   %edi            # f_pwr (ESI, EDI) ==> EAX
    push   %esi            #
```

```
    call  f_pwr          #
    add   $8, %esp      #
    mul   %esi          # EDX:EAX = EAX*ESI
    mov   %eax, -4(%ebp) # Salva il risultato.
    jmp   f_pwr_end_X   # Conclude la funzione.
    #
f_pwr_end_0:
    popa                # Conclude la funzione con EAX = 0.
    mov   $0, %eax      #
    leave                #
    ret                 #
f_pwr_end_1:
    popa                # Conclude la funzione con EAX = 1.
    mov   $1, %eax      #
    leave                #
    ret                 #
f_pwr_end_X:
    popa                # Conclude la funzione con EAX pari
    mov   -4(%ebp), %eax # al valore salvato nella variabile
    leave                # locale.
    ret                 #
```

```
# lib_fact.s
.section .text
.globl f_fact
.type f_fact, @function
#
f_fact:
    enter $4, $0
    pusha
    #
    mov   8(%ebp), %edi # Valore di cui calcolare il
                        # fattoriale.
    #
```

```

    cmp    $1, %edi        # Il fattoriale di 1 è 1.
    jz     f_fact_end_1    #
    #
    mov    %edi, %esi      # ESI contiene il valore di cui si
    dec    %esi            # vuole il fattoriale, ridotto di
                                # una unità.

    #
    push  %esi            # f_fact (ESI) ==> EAX
    call  f_fact          #
    add   $4, %esp        #
    mul   %edi            # EDX:EAX = EAX*EDI
    mov   %eax, -4(%ebp)  # Salva il risultato.
    jmp   f_fact_end_X    # Conclude la funzione.
    #
f_fact_end_1:
    popa                    # Conclude la funzione con EAX = 1.
    mov   $1, %eax         #
    leave                    #
    ret                      #
f_fact_end_X:
    popa                    # Conclude la funzione con EAX pari
    mov  -4(%ebp), %eax    # al valore salvato nella variabile
    leave                    # locale.
    ret                      #

```

```

; lib_pwr.s
section .text
global f_pwr:function
;
f_pwr:
    enter 4,0
    pusha
    ;
    mov   esi, [ebp+8]    ; Base.

```

```
mov    edi, [ebp+12] ; Esponente.
;
cmp    esi, 0        ; Se la base è pari a 0,
jz     f_pwr_end_0  ; restituisce 0.
;
cmp    edi, 0        ; Se l'esponente è pari a 0,
jz     f_pwr_end_1  ; restituisce 1.
;
dec    edi           ; Riduce l'esponente di una unità.
push   edi           ; f_pwr (ESI, EDI) ==> EAX
push   esi           ;
call   f_pwr        ;
add    esp, 8        ;
mul    esi           ; EDX:EAX = EAX*ESI
mov    [ebp-4], eax  ; Salva il risultato.
jmp    f_pwr_end_X  ; Conclude la funzione.
;
f_pwr_end_0:
popa                    ; Conclude la funzione con EAX = 0.
mov    eax, 0          ;
leave                   ;
ret                     ;
f_pwr_end_1:
popa                    ; Conclude la funzione con EAX = 1.
mov    eax, 1          ;
leave                   ;
ret                     ;
f_pwr_end_X:
popa                    ; Conclude la funzione con EAX pari
mov    eax, [ebp-4]    ; al valore salvato nella variabile
leave                   ; locale.
ret                     ;
```

```
; lib_fact.s
```



```
section .text
global f_fact:function
;
f_fact:
    enter 4,0
    pusha
    ;
    mov     edi, [ebp+8] ; Valore di cui calcolare il
                        ; fattoriale.
    ;
    cmp     edi, 1      ; Il fattoriale di 1 è 1.
    jz     f_fact_end_1 ;
    ;
    mov     esi, edi    ; ESI contiene il valore di cui si
    dec     esi        ; vuole il fattoriale, ridotto di
                        ; una unità.
    ;
    push   esi         ; f_fact (ESI) ==> EAX
    call  f_fact       ;
    add    esp, 4      ;
    mul    edi         ; EDX:EAX = EAX*EDI
    mov    [ebp-4], eax ; Salva il risultato.
    jmp   f_fact_end_X ; Conclude la funzione.
    ;
f_fact_end_1:
    popa              ; Conclude la funzione con EAX = 1.
    mov eax, 1        ;
    leave            ;
    ret              ;
f_fact_end_X:
    popa              ; Conclude la funzione con EAX pari
    mov eax, [ebp-4] ; al valore salvato nella variabile
    leave            ; locale.
    ret              ;
```

Come si può osservare, non basta dichiarare come globale il simbolo della funzione: occorre anche specificare il suo ruolo di funzione.

Ammesso che i file si chiamino, rispettivamente, ‘lib\_pwr.s’ e ‘lib\_fact.s’, si compilano come di consueto per ottenere i file oggetto relativi:

```
$ as --gstabs -o lib_pwr.o lib_pwr.s [Invio]
```

```
$ as --gstabs -o lib_fact.o lib_fact.s [Invio]
```

Ovvero:

```
$ nasm -g -f elf -o lib_pwr.o lib_pwr.s [Invio]
```

```
$ nasm -g -f elf -o lib_fact.o lib_fact.s [Invio]
```

Poi, per ottenere la libreria vera e propria, si procede con ‘ld’ nel modo seguente (a questo punto non fa differenza l’origine dei file oggetto):

```
$ ld -shared -o libmate.so lib_pwr.o lib_fact.o [Invio]
```

Così facendo si ottiene il file ‘libmate.so’ che costituisce la libreria voluta (la sigla «so» sta per *Shared object*).

### 65.2.3 Creare un programma che utilizza una libreria dinamica

«

Seguendo l’esempio della sezione precedente, si può creare un programma che si avvale della funzione ‘f\_fact’, contenuta nella libreria dinamica ‘libmate.so’:

```
# op1!  
#  
.section .data
```

```

op1:    .int    5
#
.section .text
.globl _start
.extern f_fact
#
_start:
    push   op1        # f_fact (op1) ==> EAX
    call  f_fact      #
    add   $4, %esp    #
    #
    mov   %eax, %ebx  # Restituisce il valore del fattoriale,
    mov   $1, %eax    # ammesso che sia abbastanza piccolo
    int  $0x80        # da poter essere rappresentato come
                    # valore di uscita.

```

```

; op1!
;
section .data
op1:    dd    5
;
section .text
global _start
extern f_fact
;
_start:
    push  long [op1] ; f_fact (op1) ==> EAX
    call  f_fact     ;
    add   esp, 4     ;
    ;
    mov   ebx, eax   ; Restituisce il valore del fattoriale,
    mov   eax, 1     ; ammesso che sia abbastanza piccolo
    int  0x80        ; da poter essere rappresentato come
                    ; valore di uscita.

```

La compilazione per produrre il file oggetto avviene nel modo consueto:

```
$ as --gstabs -o fact.o fact.s [Invio]
```

Ovvero:

```
$ nasm -g -f elf -o fact.o fact.s [Invio]
```

Poi, la trasformazione in file eseguibile richiede l'uso di opzioni particolari per 'ld':

```
$ ld -L . ↵
↵ -dynamic-linker /lib/ld-linux.so.2 ↵
↵ -lmate ↵
↵ -o fact ↵
↵ fact.o [Invio]
```

Vanno osservate alcune opzioni:

Opzione	Descrizione
-L .	Indica di cercare la libreria nella directory corrente.
-dynamic-linker /lib/ld-linux.so.2	Indica di usare, al momento dell'avvio del programma che si sta creando, il «collegatore dinamico» costituito dal file '/lib/ld-linux.so.2'.
-lmate	Indica di instaurare il collegamento dinamico con la libreria «mate», ovvero il file 'libmate.so' (che secondo l'opzione '-L .' si trova nella directory corrente).

Dato il modo in cui viene usata l'opzione `-l`, si comprende che i file delle librerie devono avere sempre un nome che inizia per `lib...`.

Dall'ultimo comando mostrato si ottiene il file eseguibile `fact` nella directory corrente, il quale ha bisogno della libreria `libmate.so`. Se si vuole avviare questo programma, è necessario che il file della libreria si trovi in uno dei percorsi previsti. In questo caso si trova provvisoriamente nella directory corrente e si può utilizzare la variabile di ambiente `LD_LIBRARY_PATH` per istruire di conseguenza il collegatore dinamico:

```
$ LD_LIBRARY_PATH="." ./fact ; echo $? [Invio]
```

120

Con `ldd` si può verificare la dipendenza del programma dalle librerie, ma anche in questo caso va utilizzata la variabile di ambiente `LD_LIBRARY_PATH`:

```
$ LD_LIBRARY_PATH="." ldd fact [Invio]
```

```
linux-gate.so.1 => (0xffffe000)
libmate.so => ./libmate.so (0xb7f46000)
```

## 65.2.4 Creare un file che utilizza una libreria dinamica standard

Così come è possibile utilizzare le proprie librerie dinamiche, si possono sfruttare benissimo quelle scritte da altri autori. Per poter utilizzare le funzioni comuni del linguaggio C, ci si può avvalere della libreria omonima, `c`, ovvero `libc.so`, che di norma si trova nella directory `/lib/`. A titolo di esempio viene mostrato un programma che emette un messaggio attraverso lo standard output:



```
# hello.s
#
.section .data
msg:      .ascii  "Ciao mondo!\n\0"
#
.section .text
.globl _start
.extern printf
.extern exit
#
_start:
    push  $msg          # printf (msg)
    call  printf        #
    add   $4, %esp     #
    #
    push  $0            # exit (0)
    call  exit          #
```

```
; hello.s
;
section .data
msg:      db      "Ciao mondo!", 0x0A, 0x00
;
section .text
global _start
extern printf
extern exit
;
_start:
    push  long msg      ; printf (msg)
    call  printf        ;
    add   esp, 4        ;
    ;
    push  long 0        ; exit (0)
```

```
call exit ;
```

Il programma utilizza due funzioni, `printf` e `exit`, la prima per visualizzare un messaggio e la seconda per concluderne il funzionamento. La funzione `printf` richiede come primo argomento (in questo caso anche l'unico) l'indirizzo iniziale di una stringa terminata da un byte completamente a zero: nel sorgente per GNU AS il codice di interruzione di riga e lo zero vengono inseriti con le sequenze `\n\0`, mentre in quello per NASM i codici relativi sono messi direttamente in forma numerica.

Il sorgente si compila come di consueto per ottenere il file oggetto. Successivamente, il collegamento avviene con il comando seguente:

```
$ ld -dynamic-linker /lib/ld-linux.so.2 ↵  
↵ -lc ↵  
↵ -o hello ↵  
↵ hello.o [Invio]
```

Rispetto al caso descritto nella sezione precedente, si può osservare che manca l'opzione `-L`, in quanto la libreria va cercata nei percorsi standard previsti; inoltre, conformemente all'esempio già visto, per indicare la libreria è stato usato solo il nome `c`, da cui l'opzione `-lc`.

Dal momento che la libreria si trova nei percorsi standard, per avviare il programma non servono accorgimenti particolari:

```
$ ./hello [Invio]
```

Ciao mondo!

Con `ldd` si può verificare la dipendenza del programma dalle librerie:

```
$ ldd hello [Invio]
```

```
linux-gate.so.1 => (0xffffe000)
libc.so.6 => /lib/tls/libc.so.6 (0xb7e71000)
/lib/ld-linux.so.2 (0xb7fb3000)
```

## 65.2.5 Librerie statiche

«

È utile sapere come sono organizzate le «librerie statiche» in un sistema GNU. Di per sé sono semplicemente file oggetto, compilati in modo da rendere pubblici i simboli delle funzioni a cui si può essere interessati esternamente, ma raccolti in un archivio che costituisce la libreria.

```
$ ar -cvq libmate.a lib_pwr.o lib_fact.o [Invio]
```

Il comando appena mostrato crea la libreria «mate» nel file ‘libmate.a’, composta dai file oggetto ‘lib\_pwr.o’ e ‘lib\_fact.o’. Il file della libreria è un semplice archivio «ar», che non prevede la compressione.

Il modo più semplice per collegare un programma che utilizza una libreria statica di questo genere è quello di indicare il file della libreria come se fosse un file oggetto:

```
$ ld -o fact fact.o libmate.a [Invio]
```

## 65.3 Dal sorgente all’immagine in memoria

«

Ciò che succede a partire da un file sorgente fino al programma in esecuzione in memoria è definito da un procedimento molto complesso, anche se il compilatore e il sistema operativo consentono di ignorarlo quasi completamente. Qui si mostra un esempio banale,



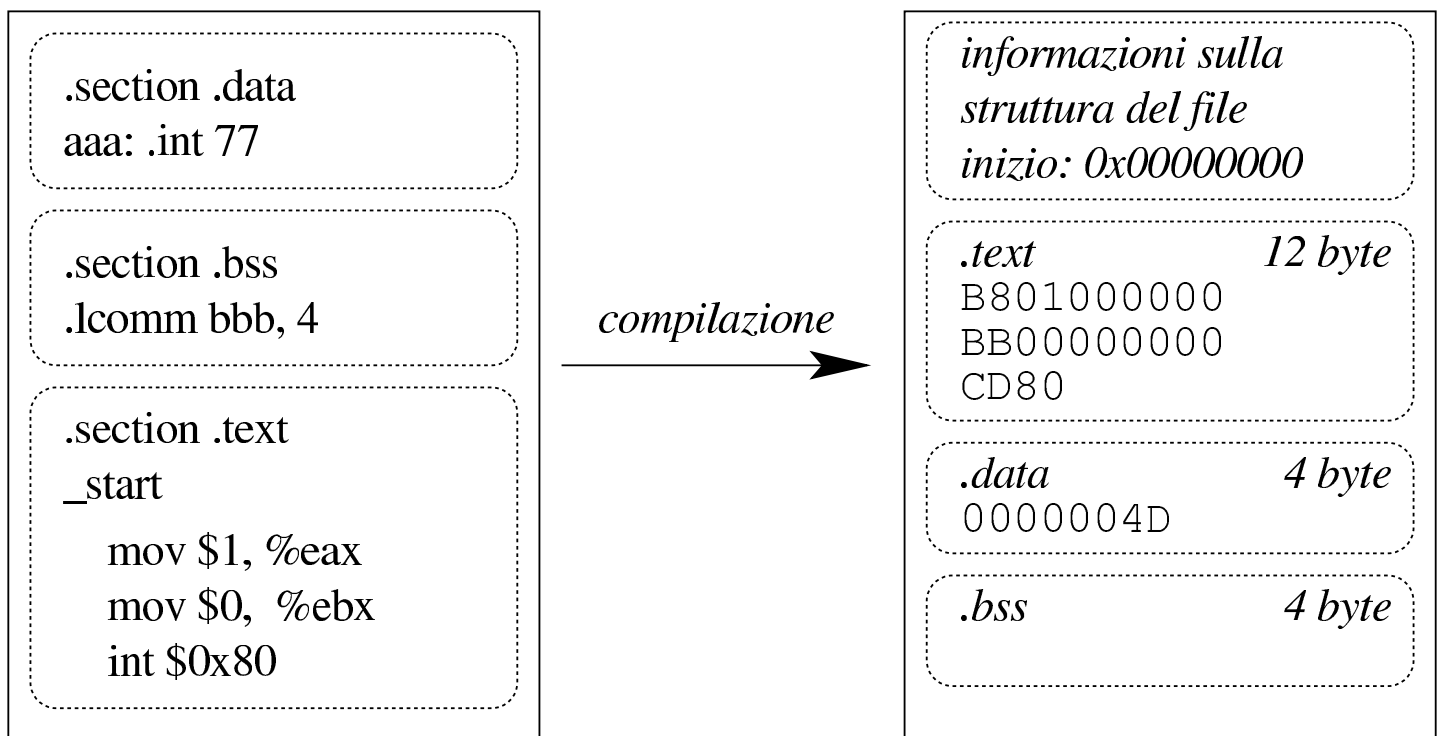
con tutti i passaggi fino ad arrivare all'immagine in memoria, senza però entrare nella questione dell'uso di librerie dinamiche.

La dimostrazione che appare si basa implicitamente sul formato ELF, sia per i file oggetto rilocabili, sia per i file eseguibili, senza entrare per ora nel dettaglio del formato stesso.

### 65.3.1 File oggetto

Di norma, la compilazione di un sorgente produce un file oggetto *rilocabile*, ma non eseguibile. Questo file oggetto contiene il codice ottenuto dall'interpretazione del sorgente, diviso in sezioni (come descritto nel sorgente stesso) che possono essere ricomposte, successivamente, con una certa libertà.

Figura 65.26. Dalle sezioni del file sorgente a quelle del file oggetto rilocabile.



A titolo di esempio si può prendere il file seguente (che riproduce quanto si vede nella figura), compilandolo nel modo consueto (qui

si mostra solo l'uso di GNU AS, per semplicità). Si suppone che il file sorgente si chiami 'prg.s':

```
.section .data
aaa:    .int 77
.section .bss
.lcomm bbb, 4
.section .text
.globl _start
_start:
    mov    $1, %eax
    mov    $0, %ebx
    int    $0x80
```

```
$ as -o prg.o prg.s [Invio]
```

Con Objdump si può analizzare il contenuto del file oggetto generato:

```
$ objdump -x prg.o [Invio]
```

```
prg.o:      file format elf32-i386
prg.o
architecture: i386, flags 0x00000010:
HAS_SYMS
start address 0x00000000
```

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	0000000c	00000000	00000000	00000034	2**2
			CONTENTS, ALLOC, LOAD, READONLY, CODE			
1	.data	00000004	00000000	00000000	00000040	2**2
			CONTENTS, ALLOC, LOAD, DATA			
2	.bss	00000004	00000000	00000000	00000044	2**2
			ALLOC			

SYMBOL TABLE:

```
00000000 1    d  .text  00000000  .text
00000000 1    d  .data  00000000  .data
00000000 1    d  .bss   00000000  .bss
00000000 1           .data  00000000  aaa
00000000 1    O  .bss   00000004  bbb
00000000 g           .text  00000000  _start
```

Anche solo intuitivamente, si comprende che il file oggetto riproduce le tre sezioni del sorgente, assegnando loro degli attributi. Per esempio, la sezione ‘**.text**’ deve essere caricata in memoria, usata in sola lettura e può essere eseguita; in modo analogo, la sezione ‘**.data**’ deve essere caricata in memoria in lettura-scrittura (l’informazione è implicita, in quanto non appare l’attributo ‘**READONLY**’), ma non può essere eseguita; la sezione ‘**.bss**’ viene allocata soltanto e non prevede limitazioni particolari, a parte il fatto di non poter essere eseguibile.

Per il momento, l’indirizzo iniziale di riferimento è  $00000000_{16}$ .

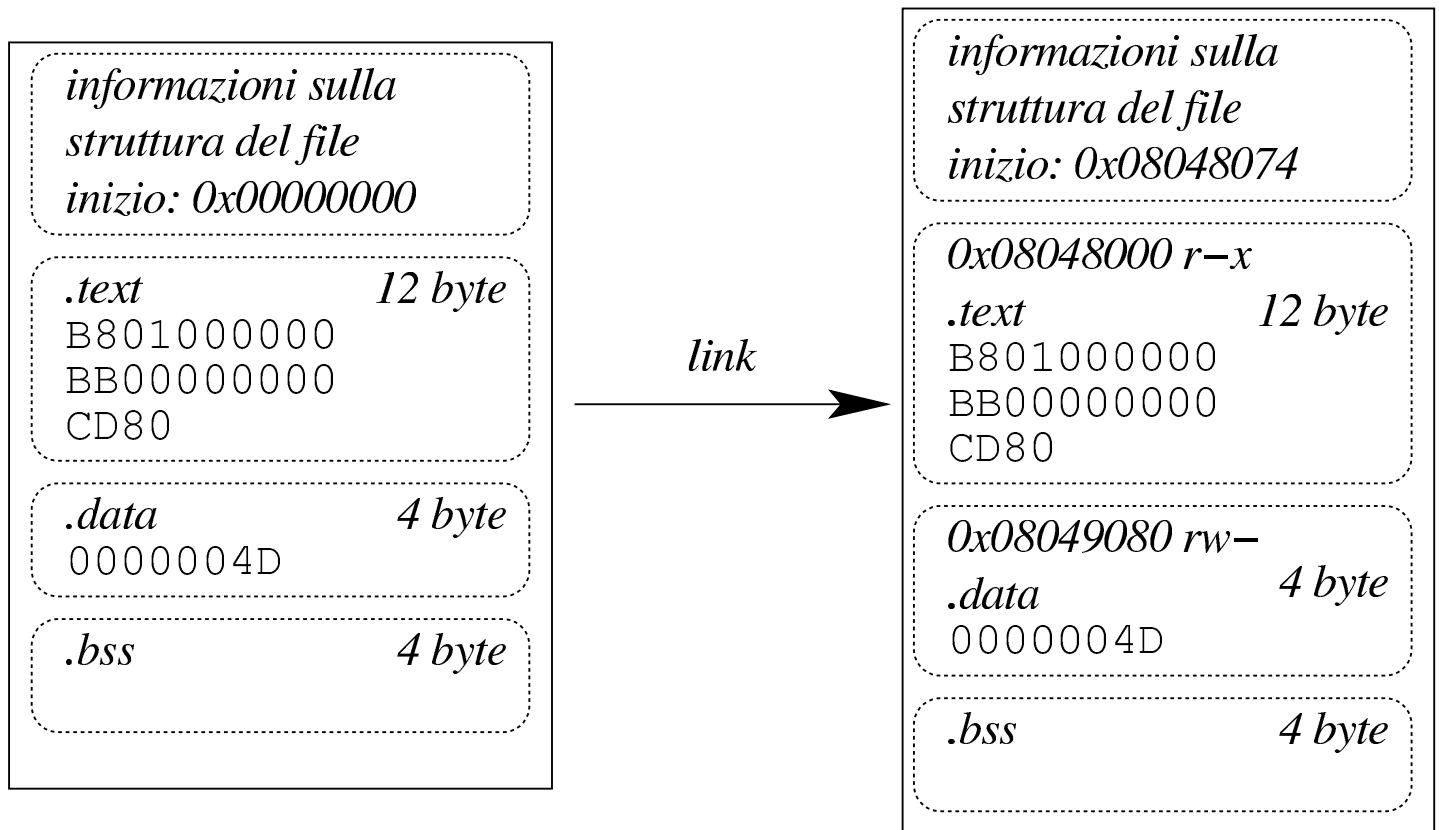
Un file oggetto di questo tipo non può essere eseguito perché non contiene le informazioni necessarie al caricamento in memoria.

### 65.3.2 File eseguibile

Per ottenere un file eseguibile, i file oggetto che servono vengono raccolti da un collegatore (*link editor*) che riordina i vari componenti e produce un file con le informazioni necessarie al caricamento in memoria.



Figura 65.29. Dalle sezioni del file oggetto rilocabile ai segmenti del file oggetto eseguibile.



Continuando nell'ipotesi della sezione precedente, si passa a generare un file eseguibile a partire dal file oggetto precedente:

```
$ ld -o prg prg.o [Invio]
```

Con `Objdump` si può analizzare il contenuto del file eseguibile generato:

```
$ objdump -x prg [Invio]
```

```
prg:      file format elf32-i386
prg
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048074
```

```
Program Header:
```

```

LOAD off      0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
      filesz 0x00000080 memsz 0x00000080 flags r-x
LOAD off      0x00000080 vaddr 0x08049080 paddr 0x08049080 align 2**12
      filesz 0x00000004 memsz 0x00000008 flags rw-

```

## Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	0000000c	08048074	08048074	00000074	2**2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
1	.data	00000004	08049080	08049080	00000080	2**2
	CONTENTS, ALLOC, LOAD, DATA					
2	.bss	00000004	08049084	08049084	00000084	2**2
	ALLOC					

## SYMBOL TABLE:

```

08048074 l    d  .text  00000000 .text
08049080 l    d  .data  00000000 .data
08049084 l    d  .bss   00000000 .bss
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
08049080 l      .data  00000000 aaa
08049084 l    O  .bss   00000004 bbb
08048074 g      .text  00000000 _start
08049084 g      *ABS*  00000000 __bss_start
08049084 g      *ABS*  00000000 _edata
08049088 g      *ABS*  00000000 _end

```

Il file eseguibile è organizzato in *segmenti* che possono riferiti, ognuno, a una o più sezioni; ma il file può contenere anche sezioni che non sono riconducibili a un segmento. Il segmento, a differenza della sezione pura e semplice, deve descrivere in che modo il contenuto deve essere caricato in memoria e quali caratteristiche deve avere durante il funzionamento.

Dal rapporto generato da Objdump, precisamente nel riepilogo intitolato **‘Program Header’**, si può vedere cosa deve essere caricato in memoria e in quale posizione (gli indirizzi si riferiscono alla me-

moria virtuale). Si notano solo due segmenti, riferiti rispettivamente alla sezione ‘**.text**’, contenente il codice da eseguire, e alla sezione ‘**.data**’.

Il segmento che riguarda la sezione ‘**.text**’ deve essere caricato in memoria a partire dall’indirizzo  $08048000_{16}$ , con permessi di lettura ed esecuzione; il segmento che si riferisce alla sezione ‘**.data**’ deve essere caricato in memoria a partire dall’indirizzo  $08049080_{16}$ , con permessi di lettura e scrittura. Non esiste un segmento per la sezione ‘**.bss**’ in quanto non contiene dati e si sa comunque che deve essere allocata a partire dall’indirizzo  $08049084_{16}$  (i permessi di lettura e scrittura sono impliciti in questo caso).

Quello che appare indicato come indirizzo iniziale è la posizione in cui si trova la prima istruzione da eseguire, pertanto è la posizione a cui deve passare il controllo il sistema di caricamento, dopo che è stata prodotta l’immagine del processo elaborativo in memoria. Questo indirizzo è interno al primo segmento, il quale è lungo  $128_{10}$ byte ( $80_{16}$ byte), pertanto, tra l’indirizzo iniziale e quello ci devono essere delle informazioni amministrative, mentre nello spazio rimanente (esattamente 12 byte) ci sono le istruzioni vere e proprie.

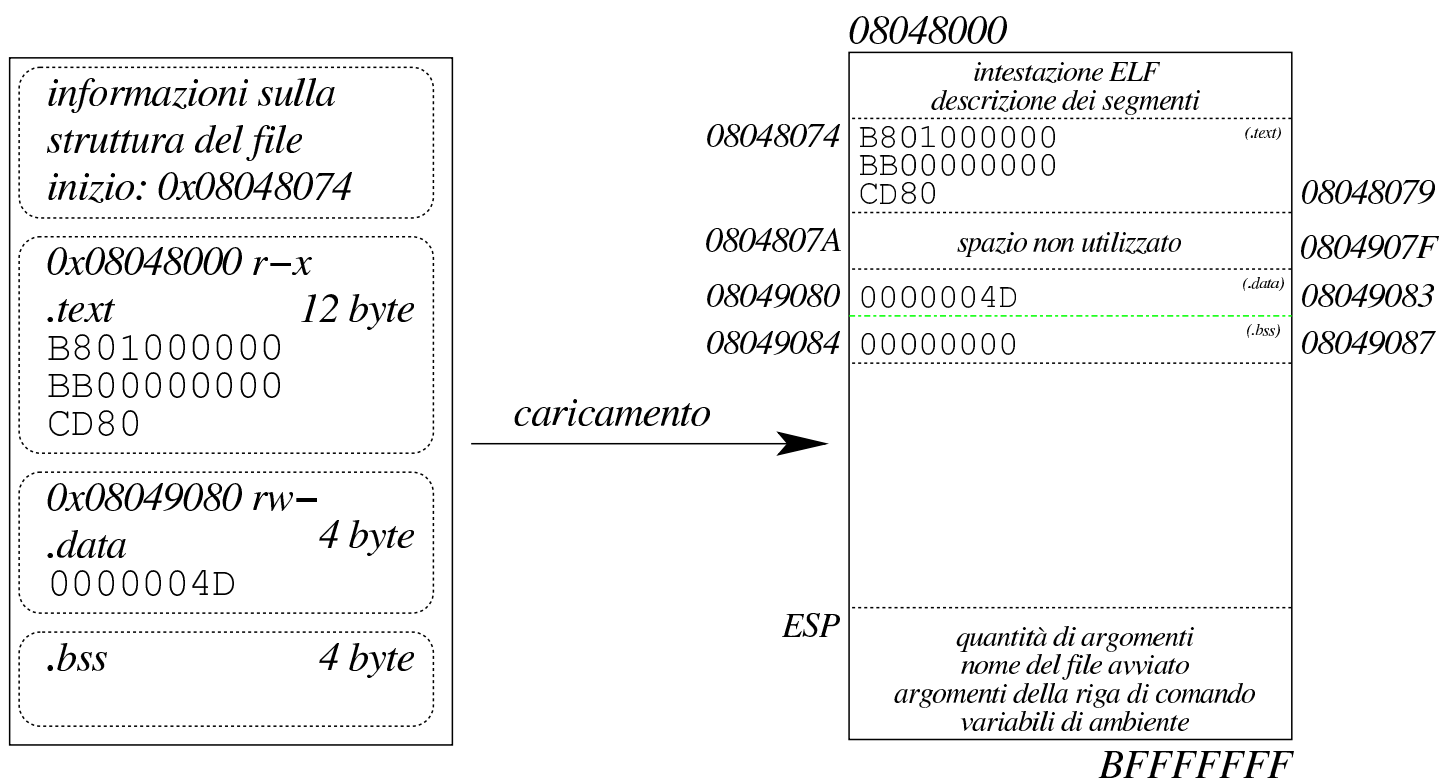
I 116 byte iniziali del primo segmento, di questo esempio, contengono precisamente l’intestazione ELF e la descrizione dei due segmenti del programma.

### 65.3.3 Immagine del processo nella memoria virtuale

Il sistema operativo legge il file eseguibile ed estrapola i segmenti, collocandoli in memoria, allocando anche lo spazio non inizializzato (privo pertanto di un segmento nel file eseguibile). Oltre a questo impila sul fondo le variabili di ambiente, gli argomenti della chiamata, il nome del file avviato effettivamente,... Per ultimo, su questa pila, mette la quantità di argomenti ricevuti nella riga di comando e li posiziona il registro *ESP*; successivamente, l'incremento di questo registro implica la crescita della pila dei dati.

In un sistema GNU/Linux i processi elaborativi utilizzano un'area della memoria virtuale che va da  $08048000_{16}$  a  $BFFFFFFF_{16}$ , come se ognuno di questi disponesse della stessa dotazione di memoria e fosse sempre tutta propria. È il sistema operativo che crea questa astrazione e alloca o libera la memoria quando serve. Si osservi che lo spazio non allocato non può essere utilizzato e se il programma vi volesse fare riferimento (senza seguire la procedura prevista per l'allocazione) si otterrebbe un *errore di segmentazione* (*segmentation fault*).

Figura 65.31. Dal file oggetto eseguibile all'immagine del processo nella memoria virtuale.



Continuando nell'ipotesi delle sezioni precedenti, si può eseguire il programma sotto il controllo di GDB:

```
$ gdb prg [Invio]
```

Per fissare uno stop occorre indicare un indirizzo che punti almeno all'inizio della seconda istruzione (se si pretende di puntare alla prima istruzione, GDB poi non si ferma). Sapendo che la prima istruzione è all'indirizzo  $08048074_{16}$  e che occupa cinque byte, si può usare l'indirizzo  $08048079_{16}$  per indicare l'inizio della seconda:

```
(gdb) break *0x08048079 [Invio]
```

```
(gdb) run [Invio]
```

Si vuole verificare che i dati siano dove previsto:



```
(gdb) print (int)*0x08049080 [Invio]
```

```
$1 = 77
```

```
(gdb) print (int)*0x08049084 [Invio]
```

```
$2 = 0
```

Il secondo indirizzo fa riferimento a una memoria non inizializzata che viene posta inizialmente a zero; pertanto il risultato coincide con le previsioni. Si può verificare la presenza dell'intestazione ELF e della descrizione dei segmenti:

```
(gdb) print /x (char[116])*0x08048000 [Invio]
```

```
$3 = {0x7f, 0x45, 0x4c, 0x46, 0x1, 0x1, 0x1, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x2, 0x0, 0x3, 0x0, 0x1, 0x0,
0x0, 0x0, 0x74, 0x80, 0x4, 0x8, 0x34, 0x0, 0x0, 0x0, 0xb0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x34, 0x0, 0x20, 0x0,
0x2, 0x0, 0x28, 0x0, 0x7, 0x0, 0x4, 0x0, 0x1, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x80, 0x4, 0x8, 0x0, 0x80, 0x4,
0x8, 0x80, 0x0, 0x0, 0x0, 0x80, 0x0, 0x0, 0x0, 0x5, 0x0,
0x0, 0x0, 0x0, 0x10, 0x0, 0x0, 0x1, 0x0, 0x0, 0x0, 0x80,
0x0, 0x0, 0x0, 0x80, 0x90, 0x4, 0x8, 0x80, 0x90, 0x4, 0x8,
0x4, 0x0, 0x0, 0x0, 0x8, 0x0, 0x0, 0x0, 0x6, 0x0, 0x0, 0x0,
0x0, 0x10, 0x0, 0x0}
```

```
(gdb) print (char[4])*0x08048000 [Invio]
```

```
$4 = "\177ELF"
```

Se si tenta di raggiungere un'area di memoria non allocata, si ottiene un errore:

```
(gdb) print /x (int)*0x0804A000 [Invio]
```

Cannot access memory at address 0x804a000

Il registro *ESP* si trova effettivamente in una zona abbastanza profonda della memoria virtuale:

```
(gdb) info registers [Invio]
```

```
...
esp                0xbf87a540          0xbf87a540
...
```

```
(gdb) quit [Invio]
```

### 65.3.4 Allineamento dei segmenti in memoria

«

Riprendendo il rapporto generato da Objdump, va osservato che i segmenti da caricare in memoria sono «allineati»:

```
Program Header:
```

```
LOAD off      0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
      filesz 0x00000080 memsz 0x00000080 flags r-x
LOAD off      0x00000080 vaddr 0x08049080 paddr 0x08049080 align 2**12
      filesz 0x00000004 memsz 0x00000008 flags rw-
```

È indicato che l'allineamento è da blocchi di 4096 byte ( $1000_{16}$  byte) ovvero  $2^{12}$  byte. Ciò comporta un allontanamento significativo del secondo segmento dal primo (da  $08048080_{16}$  che sarebbe il primo byte libero dopo il primo segmento, si salta a  $08049080_{16}$ ). Questo distacco lo produce il collegatore, o *link editor* (GNU LD), evidentemente per qualche motivo importante: in un sistema GNU/Linux la memoria virtuale è organizzata in pagine da 4 Kibyte (4096 byte) e non sarebbe possibile distinguere i permessi di accesso se i segmenti occupassero la stessa pagina.

È il caso di osservare che, nell'esempio mostrato, il distacco appare solo negli indirizzi che i segmenti devono prendere in memoria, perché nel file eseguibile, invece, sono collocati uno di seguito all'altro:

```
Program Header:
  LOAD off      0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
    filesz 0x00000080 memsz 0x00000080 flags r-x
  LOAD off      0x00000080 vaddr 0x08049080 paddr 0x08049080 align 2**12
    filesz 0x00000004 memsz 0x00000008 flags rw-
```

### 65.3.5 Script per il collegamento

GNU LD,<sup>1</sup> ovvero il programma che si usa per collegare i file oggetto rilocabili, consente di definire la struttura del file eseguibile da generare, con un certo grado di dettaglio, attraverso quello che viene definito uno script (precisamente *link script* o *linker script*). «

Esiste una configurazione predefinita di come deve essere realizzata la struttura del file eseguibile e la si può consultare con l'opzione **'--verbose'**:

```
$ ld --verbose [Invio]
```

```
...
OUTPUT_FORMAT("elf32-i386", "elf32-i386",
              "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)
SEARCH_DIR("/usr/i486-linux-gnu/lib32"); SEARCH_DIR("/usr/local/lib32");...
SECTIONS
{
  /* Read-only sections, merged into text segment: */
  PROVIDE (__executable_start = 0x08048000); . = 0x08048000 + SIZEOF_HEADERS;
  .interp          : { *(.interp) }
  .hash            : { *(.hash) }
  ...
  ...
  ...
```

```
.debug_weaknames 0 : { *(.debug_weaknames) }
.debug_funcnames 0 : { *(.debug_funcnames) }
.debug_typenames 0 : { *(.debug_typenames) }
.debug_varnames 0 : { *(.debug_varnames) }
/DISCARD/ : { *(.note.GNU-stack) }
}
```

Con l'opzione **'-T'** è possibile rimpiazzare completamente la configurazione predefinita, indicando lo script da caricare al suo posto. A titolo di esempio viene mostrato uno script molto semplificato che può essere usato con il programma apparso nelle sezioni precedenti, producendo un risultato simile:

```
ENTRY (_start)
SECTIONS {
    . = 0x08048000 + SIZEOF_HEADERS;
    .text . : { *(.text) }
    .data ALIGN (0x1000) : { *(.data) }
    .bss . : {
        *(.bss)
        *(COMMON)
    }
}
```

La direttiva che appare nella prima riga, dichiara il simbolo in corrispondenza del quale associare il punto di inizio; infatti, secondo le convenzioni comuni, il simbolo **'\_start'** è quello che in un sorgente in linguaggio assembleatore segnala l'inizio del programma:

```
ENTRY (_start)
```

Successivamente appare un blocco, all'interno del quale si dichiara la configurazione delle sezioni del programma. La prima direttiva di questo blocco definisce l'indirizzo iniziale di riferimento, ottenuto sommando a  $08048000_{16}$  la dimensione dell'intestazione (ov-

vero l'intestazione ELF vera e propria, assieme alla descrizione dei segmenti):

```
. = 0x08048000 + SIZEOF_HEADERS;
```

Di seguito appare la descrizione delle tre sezioni tipiche: `‘.text’`, `‘.data’` e `‘.bss’`. La prima è la più semplice, in quanto si limita a dichiarare di collocare la sezione `‘.text’` a partire dalla posizione corrente (quella raggiunta in quel punto), rappresentata da un punto singolo, `‘.’`, purché ci siano effettivamente sezioni con quel nome da collocare:

```
.text . : { *(.text) }
```

In questo tipo di direttiva, il punto che rappresenta la posizione corrente è facoltativo (nel senso che può essere omesso); ciò che appare tra parentesi graffe è il contenuto che la nuova sezione `‘.text’` deve avere e in questo caso rappresenta la somma delle sezioni `‘.text’` dei file oggetto rilocabili. In particolare, l'asterisco iniziale serve a precisare che in mancanza di tali sezioni nei file oggetto rilocabili, non si deve creare la sezione corrispondente nel file eseguibile.

La sezione `‘.data’` viene dichiarata in modo simile, con la differenza che, al posto del punto, viene indicato di spostare in avanti l'indirizzo in modo che sia un multiplo di  $1000_{16}$ , ovvero di  $4096_{10}$ . In questo modo si vuole ottenere che la sezione `‘.data’` sia distanziata nel file eseguibile e che così distante sia anche il segmento caricato in memoria. In pratica, l'espressione `‘ALIGN (0x1000)’` si traduce nel calcolo di un indirizzo adeguato all'allineamento che si intende ottenere, di 4096 byte:

```
.data ALIGN (0x1000) : { *(.data) }
```

L'ultima sezione, `‘.bss’`, è un po' più articolata, in quanto prevede l'inclusione delle sezioni con lo stesso nome provenienti dai file oggetto rilocabili (se ce ne sono), con l'aggiunta di tutto ciò che costituisce dati non inizializzati, rappresentato dall'espressione `‘*(COMMON)’`.

Riprendendo il programma di esempio già visto nelle sezioni precedenti, ammesso che lo script appena descritto sia contenuto nel file `‘config.ld’`, il file oggetto `‘prg.o’` potrebbe essere elaborato nel modo seguente:

```
$ ld -T config.ld -o prg prg.o [Invio]
```

Ecco cosa si può vedere con `Objdump`:

```
$ objdump -x prg [Invio]
```

```
prg:      file format elf32-i386
```

```
prg
```

```
architecture: i386, flags 0x00000112:
```

```
EXEC_P, HAS_SYMS, D_PAGED
```

```
start address 0x08048074
```

```
Program Header:
```

```
LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
```

```
filesz 0x00000080 memsz 0x00000080 flags r-x
```

```
LOAD off    0x00001000 vaddr 0x08049000 paddr 0x08049000 align 2**12
```

```
filesz 0x00000004 memsz 0x00000008 flags rw-
```

```
Sections:
```

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	0000000c	08048074	08048074	00000074	2**2
			CONTENTS, ALLOC, LOAD, READONLY, CODE			
1	.data	00000004	08049000	08049000	00001000	2**2
			CONTENTS, ALLOC, LOAD, DATA			
2	.bss	00000004	08049004	08049004	00001004	2**2
			ALLOC			

```
SYMBOL TABLE:
```

```
08048074 l d .text 00000000 .text
```

```

08049000 1    d  .data  00000000  .data
08049004 1    d  .bss   00000000  .bss
00000000 1    d  *ABS*  00000000  .shstrtab
00000000 1    d  *ABS*  00000000  .symtab
00000000 1    d  *ABS*  00000000  .strtab
08049000 1          .data  00000000  aaa
08049004 1    O  .bss   00000004  bbb
08048074 g          .text  00000000  _start

```

Come si vede, questa volta il segmento riferito alla sezione ‘**.data**’ parte esattamente da  $08049000_{16}$ , ma così vale anche per la posizione della stessa sezione ‘**.data**’ nel file eseguibile. In pratica, ciò comporta che il file eseguibile sia un po’ più grande rispetto a prima, mentre l’utilizzo della memoria non cambia in modo sostanziale.

Sempre a titolo di esempio, si può provare a vedere cosa succede se si evita di allineare la sezione ‘**.data**’:

```

ENTRY (_start)
SECTIONS {
    . = 0x08048000 + SIZEOF_HEADERS;
    .text . : { *(.text) }
    .data . : { *(.data) }
    .bss . : {
        *(.bss)
        *(COMMON)
    }
}

```

In questo modo, dato che il contenuto della sezione ‘**.text**’ è molto breve, succede che il contenuto di tutte le sezioni finisce nello stesso segmento, il quale, di conseguenza, deve avere tutti i permessi necessari:

```
$ ld -T config.ld -o prg prg.o [Invio]
```

```
$ objdump -x prg [Invio]
```

```
prg:      file format elf32-i386
prg
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048074
```

```
Program Header:
```

```
LOAD off      0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
      filesz 0x00000084 memsz 0x00000088 flags rwX
```

```
Sections:
```

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	0000000c	08048074	08048074	00000074	2**2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
1	.data	00000004	08048080	08048080	00000080	2**2
	CONTENTS, ALLOC, LOAD, DATA					
2	.bss	00000004	08048084	08048084	00000084	2**2
	ALLOC					

```
SYMBOL TABLE:
```

```
08048074 l d .text 00000000 .text
08048080 l d .data 00000000 .data
08048084 l d .bss 00000000 .bss
00000000 l d *ABS* 00000000 .shstrtab
00000000 l d *ABS* 00000000 .symtab
00000000 l d *ABS* 00000000 .strtab
08048080 l .data 00000000 aaa
08048084 l O .bss 00000004 bbb
08048074 g .text 00000000 _start
```

Naturalmente, anche il file eseguibile torna a essere di dimensioni più piccole.



Sia chiaro che gli esempi di script apparsi qui non possono essere validi in generale, ma servono solo per comprendere a grandi linee il meccanismo. Per utilizzare seriamente questo strumento occorre prima uno studio approfondito del manuale di GNU LD.

### 65.3.6 Osservazioni sui simboli

I file oggetto, rilocabili o eseguibili, contengono un elenco di simboli, che Objdump raccoglie in una tabella, denominata '**SYMBOL TABLE**'. Vale la pena di confrontare tale tabella nelle varie situazioni descritte qui, come riepilogato nelle figure successive.

Figura 65.49. La tabella dei simboli nel file oggetto rilocabile prodotto dalla compilazione del file sorgente.

```
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l          .data  00000000 aaa
00000000 l    O  .bss   00000004 bbb
00000000 g          .text  00000000 _start
```

Figura 65.50. La tabella dei simboli nel file oggetto eseguibile prodotto da GNU LD secondo la configurazione predefinita.

```

08048074 1      d  .text  00000000 .text
08049080 1      d  .data  00000000 .data
08049084 1      d  .bss   00000000 .bss
00000000 1      d  *ABS*  00000000 .shstrtab
00000000 1      d  *ABS*  00000000 .symtab
00000000 1      d  *ABS*  00000000 .strtab
08049080 1          .data  00000000 aaa
08049084 1      O  .bss   00000004 bbb
08048074 g          .text  00000000 _start
08049084 g          *ABS*  00000000 __bss_start
08049084 g          *ABS*  00000000 _edata
08049088 g          *ABS*  00000000 _end

```

Figura 65.51. La tabella dei simboli nel file oggetto eseguibile prodotto da GNU LD secondo la configurazione predisposta nella sezione precedente.

```

08048074 1      d  .text  00000000 .text
08048080 1      d  .data  00000000 .data
08048084 1      d  .bss   00000000 .bss
00000000 1      d  *ABS*  00000000 .shstrtab
00000000 1      d  *ABS*  00000000 .symtab
00000000 1      d  *ABS*  00000000 .strtab
08048080 1          .data  00000000 aaa
08048084 1      O  .bss   00000004 bbb
08048074 g          .text  00000000 _start

```

Si può osservare che, dopo la compilazione che produce un file oggetto rilocabile, appaiono gli stessi simboli previsti nel sorgente, con l'aggiunta di nomi corrispondenti a quelli delle sezioni. Nella trasformazione standard in file eseguibile, si vede la comparsa di altri simboli, in particolare: `‘.shstrtab’`, `‘.symtab’`,

‘**.strtab**’. Questi rappresentano la collocazione nel file di informazioni amministrative, relative al formato ELF. Inoltre, nel caso specifico dell’eseguibile generato secondo la configurazione predefinita di GNU LD, si vede la comparsa di simboli aggiuntivi che evidentemente dipendono dall’organizzazione della configurazione stessa.

Per comprendere come si possano inserire dei simboli aggiuntivi attraverso lo script per GNU LD, si può riprendere l’esempio già visto nella sezione precedente, ritoccando leggermente la definizione della sezione ‘**.bss**’:

```
ENTRY (_start)
SECTIONS {
    . = 0x08048000 + SIZEOF_HEADERS;
    .text . : { *(.text) }
    .data ALIGN (0x1000) : { *(.data) }
    .bss . : {
        _sbss = .;
        *(.bss)
        *(COMMON)
        _ebss = .;
    }
}
```

I simboli che si vogliono aggiungere sono ‘**\_sbss**’ e ‘**\_ebss**’, con lo scopo di individuare l’inizio e la fine della nuova sezione ‘**.bss**’.

Figura 65.53. La tabella dei simboli dopo l'introduzione forzata di `'_sbss'` e `'_ebss'`.

```

SYMBOL TABLE:
08048074 l      d  .text  00000000 .text
08049000 l      d  .data  00000000 .data
08049004 l      d  .bss   00000000 .bss
00000000 l      d  *ABS*  00000000 .shstrtab
00000000 l      d  *ABS*  00000000 .symtab
00000000 l      d  *ABS*  00000000 .strtab
08049000 l          .data  00000000 aaa
08049004 l      O  .bss   00000004 bbb
08049004 g          .bss   00000000 _sbss
08049008 g          .bss   00000000 _ebss
08048074 g          .text  00000000 _start

```

Eventualmente si può sperimentare cosa cambia nel contenuto dei file oggetto (rilocabili o eseguibili) quando si compila un sorgente con l'opzione `'--gstabs'` di GNU AS o con l'opzione `'-g'` di NASM.

### 65.3.7 Formati dei file oggetto

«

I file oggetto rilocabili e i file eseguibili possono essere realizzati secondo diversi formati, ma dipende dal sistema operativo qual è la scelta che si deve operare. Negli esempi mostrati, partendo dal presupposto di utilizzare un sistema GNU/Linux, si fa riferimento al formato ELF, in quanto è quello che deve essere usato e gli strumenti comuni sono già configurati per generare file conformi a tale standard.

Il formato del file che si deve produrre condiziona anche i tipi di sezioni che si possono dichiarare nel sorgente in linguaggio assembler. Il formato ELF dà molta libertà, comunque prevede una serie numerosa di sezioni con funzioni specifiche, in particolare

‘**.rodata**’ che comporta la creazione di un segmento di memoria con dati inizializzati, ma in sola lettura.

## 65.4 Formato ELF

Il formato ELF è il contenitore di un programma che non si trova necessariamente nello stato di poter essere eseguito. Il formato ELF si distingue per la presenza di un’intestazione che si trova obbligatoriamente all’inizio del file; quindi, il contenuto del file è affiancato da una serie di tabelle che lo descrivono in base a vari criteri.

### 65.4.1 Sezioni e segmenti

Per semplificare la descrizione di un formato ELF, lo si può immaginare composto da sezioni, il cui scopo è quello di descrivere tutto ciò che compone il programma, e da segmenti, con i quali si descrive in che modo il programma deve essere rappresentato in memoria ed eseguito. L’informazione relativa alle sezioni è indispensabile quando deve intervenire un «collegatore» (*linker*); l’informazione data dai segmenti riguarda l’avvio del programma.

Se si vuole abbandonare questo tipo di rappresentazione astratta, il formato ELF lo si può vedere come un involucro del codice eseguibile e dei dati inizializzati, contenente un’intestazione di riconoscimento (che si trova obbligatoriamente all’inizio del file) e da una serie di tabelle, più o meno concatenate tra di loro, alcune delle quali possono essere facoltative, in base al contesto per il quale il file oggetto è predisposto.

Tabella 65.54. Componenti principali che descrivono un formato ELF.

Tabella	Descrizione
<i>ELF header</i>	È l'intestazione del file e deve trovarsi necessariamente all'inizio dello stesso. Contiene poi i riferimenti alla tabella dei segmenti ( <i>program header table</i> ) e a quella delle sezioni ( <i>section header table</i> ).
<i>program header table</i>	È la tabella dei segmenti da caricare in memoria, con le informazioni necessarie a procedere in tal senso. La presenza di questa tabella è obbligatoria in un file oggetto eseguibile.
<i>section header table</i>	È la tabella delle sezioni.
<i>string table</i>	È la tabella delle stringhe, a cui fanno riferimento le altre tabelle quando devono indicare una stringa di qualunque tipo.
<i>symbol table</i>	È la tabella dei simboli associati a varie parti del contenuto. La tabella dei simboli, per indicare i nomi dei simboli, deve fare riferimento alla tabella delle stringhe.

## 65.4.2 Intestazione ELF



L'intestazione ELF è il componente più importante del formato, in quanto la sua presenza è obbligatoria. L'intestazione consente di identificare un file ELF come tale e di raggiungere le tabelle delle sezioni e dei segmenti, da cui poi si arriva al contenuto rimanente del file.

Tabella 65.55. Intestazione ELF secondo l'architettura x86, in particolare con le informazioni necessarie a produrre un file eseguibile.

Nome mnemonico	Dimensione x86-32	Dimensione x86-64	Descrizione
e_ident[0]	8 bit	8 bit	Impronta di identificazione del formato: deve corrispondere a $7F_{16}$ , 'E', 'L', 'F'.
e_ident[1]	8 bit	8 bit	
e_ident[2]	8 bit	8 bit	
e_ident[3]	8 bit	8 bit	
e_ident[4]	8 bit	8 bit	Definisce la classe del file: $01_{16}$ rappresenta un file oggetto a 32 bit; $02_{16}$ rappresenta invece un file a 64 bit.
e_ident[5]	8 bit	8 bit	Definisce la codifica dei dati: $01_{16}$ rappresenta un formato LSB, ovvero <i>little endian</i> ; $02_{16}$ formato MSB, ovvero <i>big endian</i> . Sia $01_{16}$ , sia $02_{16}$ , si riferiscono a una rappresentazione numerica dei valori negativi attraverso il complemento a due.
e_ident[6]	8 bit	8 bit	Definisce la versione dell'intestazione (inizialmente esiste solo la versione $01_{16}$ ).

Nome mnemonico	Dimensione x86-32	Dimensione x86-64	Descrizione
e_ident[7]			Questi byte definiscono informazioni di importanza minore e di solito vengono lasciati a 00 <sub>16</sub> .
e_ident[8]	8 bit	8 bit	
e_ident[9]	8 bit	8 bit	
e_ident[10]	8 bit	8 bit	
e_ident[11]	8 bit	8 bit	
e_ident[12]	8 bit	8 bit	
e_ident[13]	8 bit	8 bit	
e_ident[14]	8 bit	8 bit	
e_ident[15]	8 bit	8 bit	Dichiara la dimensione in byte della sequenza di identificazione. Il valore obbligato per questo byte è 10 <sub>16</sub> , ovvero 16 <sub>10</sub> .
e_type	16 bit	16 bit	Definisce il tipo di file oggetto. Un file oggetto rilocabile ha il codice 01 <sub>16</sub> ; un file oggetto eseguibile ha il codice 02 <sub>16</sub> .
e_machine	16 bit	16 bit	Definisce il tipo di architettura. Il codice 03 <sub>16</sub> si riferisce al tipo Intel.
e_version	32 bit	32 bit	Definisce la versione del file oggetto (inizialmente esiste solo la versione 00000001 <sub>16</sub> ).



Nome mnemonico	Dimensione x86-32	Dimensione x86-64	Descrizione
e_entry	32 bit	64 bit	Contiene l'indirizzo a cui occorre passare il controllo per l'esecuzione del programma.
e_phoff	32 bit	64 bit	<i>program header table offset</i> Contiene lo scostamento, rispetto all'inizio del file, necessario per raggiungere il primo byte della tabella che descrive i segmenti da caricare in memoria. Tale tabella è nota come <i>program header table</i> ed è obbligatoria la sua presenza in un file oggetto eseguibile.
e_shoff	32 bit	64 bit	<i>section header table offset</i> Contiene lo scostamento, rispetto all'inizio del file, necessario per raggiungere il primo byte della tabella che descrive le sezioni. Tale tabella è nota come <i>section header table</i> .
e_flags	32 bit	32 bit	Contiene degli indicatori specifici per il tipo di microprocessore. Nel caso dell'architettura x86-32 può contenere semplicemente valori a zero.
e_ehsize	16 bit	16 bit	<i>ELF header size</i> Contiene la dimensione dell'intestazione ELF.

Nome mnemonico	Dimensione x86-32	Dimensione x86-64	Descrizione
e_phentsize	16 bit	16 bit	<i>program header entry size</i> Definisce la dimensione di una voce descrittiva di un segmento, nella tabella dei segmenti. Tutte le voci di tale tabella hanno la stessa dimensione.
e_phnum	16 bit	16 bit	<i>program header number</i> Definisce la quantità di voci contenute nella tabella di descrizione dei segmenti.
e_shentsize	16 bit	16 bit	<i>section header entry size</i> Definisce la dimensione di una voce descrittiva di una sezione, nella tabella delle sezioni. Tutte le voci di tale tabella hanno la stessa dimensione.
e_shnum	16 bit	16 bit	<i>section header number</i> Definisce la quantità di voci contenute nella tabella di descrizione delle sezioni.
e_shstrndx	16 bit	16 bit	<i>section header string index</i> Definisce l'indice, all'interno della tabella delle sezioni, che identifica la voce che fa riferimento alla tabella delle stringhe.

### 65.4.3 Descrizione dei segmenti

La descrizione dei segmenti, necessaria per mettere in esecuzione un programma, è contenuta nella tabella *program header*, composta da un array di voci, di dimensione uniforme, ognuna delle quali descrive un segmento. Si raggiunge la prima voce di questo array con lo scostamento indicato nell'intestazione ELF (`'e_phoff'`), quindi, sapendo la dimensione di ogni voce (`'e_phentsize'`) e la quantità di queste (`'e_phnum'`), è possibile scandire anche le altre.

Tabella 65.56. Descrizione di una voce nella tabella dei segmenti, secondo l'architettura x86-32.

Nome mnemonico	Dimensione	Descrizione
<code>p_type</code>	32 bit	Definisce il tipo di operazione da compiere. La situazione più semplice è costituita da codice eseguibile e dati da caricare in memoria: $01_{16}$ .
<code>p_offset</code>	32 bit	Definisce lo scostamento, dall'inizio del file, necessario a raggiungere il primo byte del segmento.
<code>p_vaddr</code>	32 bit	Definisce l'indirizzo assoluto, nell'ambito della memoria virtuale, dove il primo byte del segmento deve trovarsi in memoria, una volta caricato.
<code>p_paddr</code>	32 bit	Equivale al campo <code>'p_vaddr'</code> , ma si riferisce alla «memoria fisica». In un sistema GNU/Linux questo valore è sempre uguale a <code>'p_vaddr'</code> .

Nome mnemonico	Dimensione	Descrizione
<code>p_filesz</code>	32 bit	Definisce la dimensione del segmento nel file e in casi particolari può essere pari a zero.
<code>p_memsz</code>	32 bit	Definisce la dimensione del segmento rappresentato in memoria e in casi particolari può essere pari a zero.
<code>p_flags</code>	32 bit	Definisce degli indicatori che descrivono i permessi del segmento: 1 = esecuzione; 2 = scrittura; 4 = lettura. Per avere permessi multipli si sommano i permessi elementari. Generalmente, il segmento di una porzione di codice dispone di permessi di accesso in lettura e in esecuzione, mentre quello di un'area di dati, consente normalmente la lettura e la scrittura.
<code>p_align</code>	32 bit	Definisce l'allineamento in memoria, a blocchi del valore indicato, il quale a sua volta deve essere una potenza di due.

Tabella 65.57. Descrizione di una voce nella tabella dei segmenti, secondo l'architettura x86-64.

Nome mnemonico	Dimensione	Descrizione
p_type	32 bit	Definisce il tipo di operazione da compiere. La situazione più semplice è costituita da codice eseguibile e dati da caricare in memoria: $01_{16}$ .
p_flags	32 bit	Definisce degli indicatori che descrivono i permessi del segmento: 1 = esecuzione; 2 = scrittura; 4 = lettura. Per avere permessi multipli si sommano i permessi elementari. Generalmente, il segmento di una porzione di codice dispone di permessi di accesso in lettura e in esecuzione, mentre quello di un'area di dati, consente normalmente la lettura e la scrittura.
p_offset	64 bit	Definisce lo scostamento, dall'inizio del file, necessario a raggiungere il primo byte del segmento.
p_vaddr	64 bit	Definisce l'indirizzo assoluto, nell'ambito della memoria virtuale, dove il primo byte del segmento deve trovarsi in memoria, una volta caricato.
p_paddr	64 bit	Equivale al campo ' <b>p_vaddr</b> ', ma si riferisce alla «memoria fisica». In un sistema GNU/Linux questo valore è sempre uguale a ' <b>p_vaddr</b> '.

Nome mnemonico	Dimensione	Descrizione
<code>p_filesz</code>	64 bit	Definisce la dimensione del segmento nel file e in casi particolari può essere pari a zero.
<code>p_memsz</code>	64 bit	Definisce la dimensione del segmento rappresentato in memoria e in casi particolari può essere pari a zero.
<code>p_align</code>	64 bit	Definisce l'allineamento in memoria, a blocchi del valore indicato, il quale a sua volta deve essere una potenza di due.

#### 65.4.4 Definizione manuale di un formato ELF

«

Un programma eseguibile in formato ELF deve contenere l'intestazione ELF e la descrizione dei segmenti; le sezioni possono anche mancare del tutto. Viene mostrato un esempio di programma banale (non fa altro che restituire il valore 77) in cui tutto, anche ciò che costituisce il formato ELF, viene definito nel sorgente. Il programma in sé trae spunto dal documento *A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux* di Brian Raiter, come annotato alla fine del capitolo. Si osservi che nel sorgente le sezioni non vengono indicate affatto.

```
.code32
.globl _start
#
file_begin:
#
# ELF header.
#
```

```
elf_header_begin:
    .byte  0x7F                # e_ident
    .byte  'E', 'L', 'F'      #
    .byte  1                   #
    .byte  1                   #
    .byte  1                   #
    .byte  0, 0, 0, 0         #
    .byte  0, 0, 0, 0         #
    .byte  16                  #
#
    .short 2                  # e_type      2 = executable file
    .short 3                  # e_machine 3 = 386
    .int   1                  # e_version 1 = current version
    .int   _start            # e_entry   start address
    .int   (program_header_begin - file_begin)
                                # e_phoff   program header offset
    .int   0                  # e_shoff   0 = no section header table
    .int   0                  # e_flags   no flags
    .short (elf_header_end - elf_header_begin)
                                # e_ehsize  ELF header size
    .short (program_header_end - program_header_begin)
                                # e_phentsize program header entry size
    .short 1                  # e_phnum   program header entries
    .short 0                  # e_shentsize 0 = no section header table
    .short 0                  # e_shnum   section header entries
    .short 0                  # e_shstrndx 0 = undefined
elf_header_end:
#
# Program header table, with just one entry.
#
program_header_begin:
    .int   1                  # p_type   1 = segment to be loaded
    .int   0                  # p_offset segment's offset
    .int   0x08048000        # p_vaddr  segment's virtual
```

```

                                #          address
.int    0x08048000 # p_paddr      segment's physical
                                #          address
.int    (file_end - file_begin)
                                # p_filesz  file image size
.int    (file_end - file_begin)
                                # p_memsz   memory image size
.int    5           # p_flags 5 = read + execute
.int    0x1000     # p_align  segment's memory
                                #          alignment

program_header_end:
#
# Program code.
#
_start:
    mov $77, %ebx
    mov $1, %eax
    int $0x80
#
file_end:

```

Il sorgente scritto nel formato adatto a GNU AS va compilato così:

```
$ as -o elf_test.o elf_test.s [Invio]
```

```
$ ld --oformat binary -o elf_test elf_test.o [Invio]
```

Come si vede, GNU LD viene usato in modo da produrre un formato binario, puro e semplice, ovvero un file privo di formato, dato che è già tutto incluso nella descrizione del programma stesso.

Si mostra anche il sorgente adatto a NASM, dove va annotata anche l'origine, ovvero l'indirizzo in cui tutto deve essere caricato in memoria:



```
bits 32
org 0x08048000
;
file_begin:
;
; ELF header.
;
elf_header_begin:
    db 0x7F          ; e_ident
    db 'E', 'L', 'F' ;
    db 1            ;
    db 1            ;
    db 1            ;
    db 0, 0, 0, 0   ;
    db 0, 0, 0, 0   ;
    db 16           ;
;
    dw 2            ; e_type      2 = executable file
    dw 3            ; e_machine  3 = 386
    dd 1            ; e_version  1 = current version
    dd _start       ; e_entry     start address
    dd (program_header_begin - file_begin)
                                ; e_phoff      program header offset
    dd 0            ; e_shoff     0 = no section header table
    dd 0            ; e_flags     no flags
    dw (elf_header_end - elf_header_begin)
                                ; e_ehsize     ELF header size
    dw (program_header_end - program_header_begin)
                                ; e_phentsize   program header entry
                                ;               size
    dw 1            ; e_phnum     program header entries
    dw 0            ; e_shentsize 0 = no section header table
    dw 0            ; e_shnum     section header entries
    dw 0            ; e_shstrndx  0 = undefined
```

```
elf_header_end:
;
; Program header table, with just one entry.
;
program_header_begin:
    dd 1          ; p_type      1 = segment to be loaded
    dd 0          ; p_offset    segment's offset
    dd 0x08048000 ; p_vaddr     segment's virtual
    ;           ;             address
    dd 0x08048000 ; p_paddr     segment's physical
    ;           ;             address
    dd (file_end - file_begin)
    ;           ;             p_filesz      file image size
    dd (file_end - file_begin)
    ;           ;             p_memsz      memory image size
    dd 5          ; p_flags     5 = 1 (execute) + 4 (read)
    dd 0x1000     ; p_align     segment's memory
    ;           ;             alignment
program_header_end:
;
; Program code.
;
_start:
    mov ebx, 77
    mov eax, 1
    int 0x80
;
file_end:
```

In questo caso, la compilazione non richiede altro che NASM, il quale produce direttamente il formato binario voluto:

```
$ nasm -f bin -o elf_test elf_test.s [Invio]
```

```
$ chmod +x elf_test [Invio]
```

I valori che rappresentano scostamenti e dimensioni del codice, sono calcolati attraverso il compilatore, facendo riferimento ai simboli rappresentati dalle etichette che delimitano le varie porzioni del sorgente. Ecco come si presenta il programma eseguibile dal punto di vista di Objdump:

```
$ objdump -x elf_test [Invio]
```

```
ELF_test:      file format elf32-i386
ELF_test
architecture: i386, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x08048054

Program Header:
   LOAD off      0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
           filesz 0x00000060 memsz 0x00000060 flags r-x

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
SYMBOL TABLE:
no symbols
```

## 65.4.5 Esempio più complesso

Viene mostrato un esempio più complesso, composto sempre da una sola voce nella tabella dei segmenti; in particolare viene definita una variabile inizializzata, incorporata nel segmento del codice, che nel sorgente appare in fondo. Viene mostrata solo la versione per GNU AS, trattandosi del programma per il calcolo del fattoriale, già descritto in precedenza.

```
.code32
.globl _start
#
```

```
file_begin:
#
# ELF header.
#
elf_header_begin:
    .byte 0x7F          # e_ident
    .byte 'E', 'L', 'F' #
    .byte 1            #
    .byte 1            #
    .byte 1            #
    .byte 0, 0, 0, 0   #
    .byte 0, 0, 0, 0   #
    .byte 16           #
#
    .short 2           # e_type      2 = executable file
    .short 3           # e_machine 3 = 386
    .int 1             # e_version 1 = current version
    .int _start        # e_entry      start address
    .int (program_header_begin - file_begin)
                        # e_phoff      program header offset
    .int 0             # e_shoff    0 = no section header table
    .int 0             # e_flags    no flags
    .short (elf_header_end - elf_header_begin)
                        # e_ehsize    ELF header size
    .short (program_header_end - program_header_begin)
                        # e_phentsize  program header entry
                        #                size
    .short 1           # e_phnum    program header entries
    .short 0           # e_shentsize 0 = no section header table
    .short 0           # e_shnum    section header entries
    .short 0           # e_shstrndx 0 = undefined
elf_header_end:
#
# Program header table, with just one entry.
```

```
#
program_header_begin:
    .int 1          # p_type  1 = segment to be loaded
    .int 0          # p_offset  segment's offset
    .int 0x08048000 # p_vaddr  segment's virtual address
    .int 0x08048000 # p_paddr  segment's physical address
    .int (file_end - file_begin)
                    # p_filesz  file image size
    .int (file_end - file_begin)
                    # p_memsz   memory image size
    .int 5          # p_flags 5 = read + execute
    .int 0x1000     # p_align  segment's memory alignment
program_header_end:
#
# Program code.
#
_start:
    mov  op1, %esi  # ESI contiene il valore di cui si vuole
                    # calcolare il fattoriale.
    push %esi      # f_fact (ESI) ==> EAX
    call f_fact    #
    add  $4, %esp  #
    mov  %eax, %ebx # Restituisce il valore del fattoriale,
    mov  $1, %eax  # ammesso che sia abbastanza piccolo
    int  $0x80     # da poter essere rappresentato come
                    # valore di uscita.

#
# Fattoriale di un numero senza segno.
# f_fatt (a) ==> EAX
# EAX = a!
#
f_fact:
    enter $4, $0
    pusha
```

```
#
mov  8(%ebp), %edi # Valore di cui calcolare il
                        # fattoriale.

#
cmp  $1, %edi      # Il fattoriale di 1 è 1.
jz   f_fact_end_1  #
#
mov  %edi, %esi    # ESI contiene il valore di cui si
dec  %esi          # vuole il fattoriale, ridotto di
                        # una unità.

#
push %esi          # f_fact (ESI) ==> EAX
call f_fact        #
add  $4, %esp      #
mul  %edi          # EDX:EAX = EAX*EDI
mov  %eax, -4(%ebp) # Salva il risultato.
jmp  f_fact_end_X  # Conclude la funzione.
#
f_fact_end_1:
  popa              # Conclude la funzione con EAX = 1.
  mov  $1, %eax     #
  leave            #
  ret              #
f_fact_end_X:
  popa              # Conclude la funzione con EAX pari
  mov  -4(%ebp), %eax # al valore salvato nella variabile
  leave            # locale.
  ret              #

#
# Initialized data.
#
op1:  .int    5
#
file_end:
```

## 65.5 Programmi completamente autonomi

A volte esiste la necessità di realizzare un programma funzionante in modo autonomo, ovvero *stand alone*, senza il sostegno del sistema operativo. Un lavoro di questo tipo richiede lo studio delle stesse problematiche che riguardano inizialmente la costruzione di un nuovo sistema operativo, ma di norma è meglio fermarsi alla produzione di un programma singolo.<sup>2</sup>

C'è da osservare che l'avvio di un programma «autonomo» in un elaboratore x86 può essere di una complessità mostruosa, a causa di problematiche ereditate dall'architettura originale del microprocessore 8088/8086. La prima difficoltà che si incontra a tale proposito sta nel far sì che il microprocessore si metta a lavorare in «modalità protetta», ovvero in una condizione che consenta di utilizzare in modo ragionevole la memoria centrale. Nel tempo, questa e altre questioni sono diventate di competenza dei programmi che si occupano di avviare un sistema operativo, come è il caso di GRUB 1 e di SYSLINUX, che così predispongono un contesto più confortevole al programma o al kernel da avviare successivamente.

Qui si mostrano esempi che utilizzano anche codice in linguaggio C, il quale viene descritto a partire dal capitolo 66.

### 65.5.1 Le specifiche «multiboot»

Le specifiche *multiboot* sono definite dal documento *Multiboot specification*, disponibile presso <http://www.gnu.org/software/grub/manual/multiboot/>. Si tratta della definizione di un'interfaccia tra sistema di avvio e sistema operativo (o programma autonomo), inizialmente per un'architettura x86. Il documento citato contiene sia

le specifiche, sia un esempio completo di programma che interagisce con il sistema di avvio secondo le specifiche stesse. Qui si riassumono i concetti principali.

### 65.5.1.1 Formato del file che deve essere avviato

«

Il file-immagine che contiene il programma da avviare (programma che potrebbe essere il kernel di un sistema operativo), deve contenere un'intestazione particolare, definita *multiboot header*, costituita in pratica da un'impronta di riconoscimento e da una serie di dati. Attraverso questa intestazione, il sistema di avvio è almeno in grado di riconoscere il file-immagine come qualcosa che deve essere avviato effettivamente e di recepirne le caratteristiche.

Questa intestazione deve trovarsi nella parte iniziale del file-immagine da caricare ed eseguire, ma non è necessario che sia esattamente all'inizio dello stesso, essendo sufficiente che sia contenuta completamente entro i primi 8 Kibyte.

Figura 65.62. La prima parte obbligatoria dell'intestazione.



Il primo campo da 32 bit, definito *magic*, contiene un'impronta di riconoscimento, costituita precisamente dal numero  $1BADB002_{16}$ . Questa serve al sistema di avvio a individuare la presenza e l'ini-

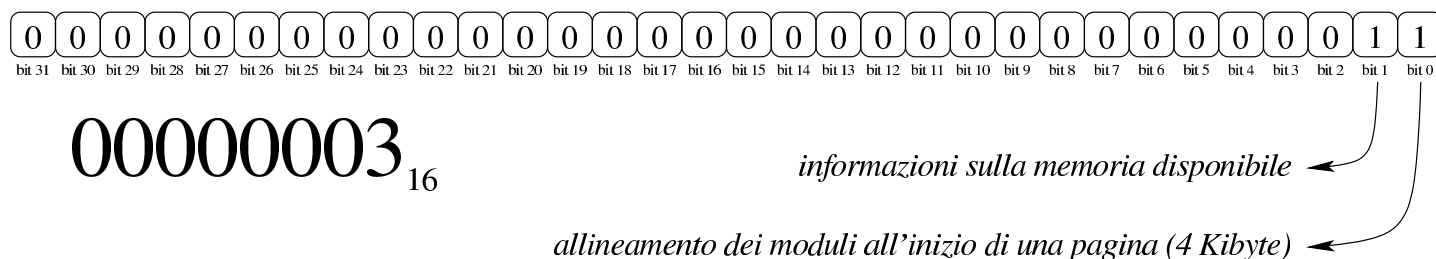


zio di una tale intestazione. Il secondo campo da 32 bit, definito *flags*, contiene degli indicatori con i quali si richiede un certo comportamento al sistema di avvio. Il terzo campo da 32 bit, definito *checksum*, contiene un numero calcolato in modo tale che la somma tra i numeri contenuti nei tre campi da 32 bit porti a ottenere zero, senza considerare i riporti.

I nomi indicati sono quelli definiti dallo standard e, come si vede, il campo *checksum* si ottiene calcolando  $-(magic + flags)$ , dove si deve intendere che i calcoli avvengono con valori interi senza segno e si ignorano i riporti.

Se il file-immagine da avviare è informato ELF, le informazioni che il sistema di avvio necessita per piazzarlo correttamente in memoria e per passare il controllo allo stesso, sono già disponibili e non c'è la necessità di occuparsi di altri campi facoltativi che possono seguire i tre già descritti. Stante questa semplificazione, per quanto riguarda il campo *flags* sono importanti i primi due bit, mentre gli altri vanno lasciati a zero.

Figura 65.63. Il campo *flags* e il suo utilizzo fondamentale.



Il bit meno significativo del campo *flags*, se impostato a uno, serve a richiedere il caricamento in memoria dei moduli eventuali (assieme al file-immagine principale) in modo che risultino allineati all'inizio di una «pagina» (ovvero all'inizio di un blocco da 4 Kibyte). Alcuni

sistemi operativi hanno la necessità di trovare i moduli allineati in questo modo e in generale l'attivazione di tale bit non può creare danno.

Il secondo bit del campo *flags* serve a richiedere al sistema di avvio di passare le informazioni disponibili sulla memoria. Queste informazioni vengono rese disponibili a partire da un'area a cui punta inizialmente il registro *EBX*. In generale si tratta di un'informazione utile (che al massimo può essere ignorata), pertanto conviene attivare anche questo bit.

Figura 65.64. Calcolo del campo *checksum*.

		complemento a due	
<i>magic</i>	1BADB002+	FFFFFFFFF-	1BADB005+
<i>flags</i>	00000003=	1BADB005=	E4524FFB=
	<u>1BADB005</u>	E4524FFA+	1 00000000
		<u>00000001=</u>	verifica della somma di controllo
<i>checksum</i>	E4524FFB		

### 65.5.1.2 Situazione dopo l'avvio del file-immagine

« Quando, dopo il trasferimento in memoria del programma, il sistema di avvio passa il controllo allo stesso, la situazione che questo programma si trova è sostanzialmente quella seguente, dove però sono stati omessi molti dettagli importanti:

- il microprocessore è in modalità protetta;
- il registro *EAX* contiene il numero  $2BADB002_{16}$  (si osservi che la prima cifra è cambiata, rispetto all'impronta che deve avere il file-immagine da avviare);

- il registro ***EBX*** deve contenere l'indirizzo fisico, a 32 bit, di una serie di campi contenenti informazioni passate dal sistema di avvio (*multiboot information structure*).

Di norma, la prima cosa che fa il programma che è stato avviato in questo modo è di azzerare il registro ***EFLAGS*** e di predisporre uno spazio per la pila dei dati, posizionando il registro ***ESP*** di conseguenza.

### 65.5.1.3 Informazioni passate dal sistema di avvio al programma

Il sistema di avvio conforme alle specifiche *multiboot* offre una serie di informazioni, collocate in una struttura che parte dall'indirizzo indicato nel registro ***EBX***. Questa struttura ha un certo grado di complessità, in quanto può fare riferimento ad altre strutture. Qui viene descritta brevemente solo una prima porzione di questa struttura; per l'approfondimento occorre consultare le specifiche, pubblicate presso <http://www.gnu.org/software/grub/manual/multiboot/> .

Figura 65.65. Inizio della struttura informativa offerta da un sistema di avvio aderente alle specifiche *multiboot*.

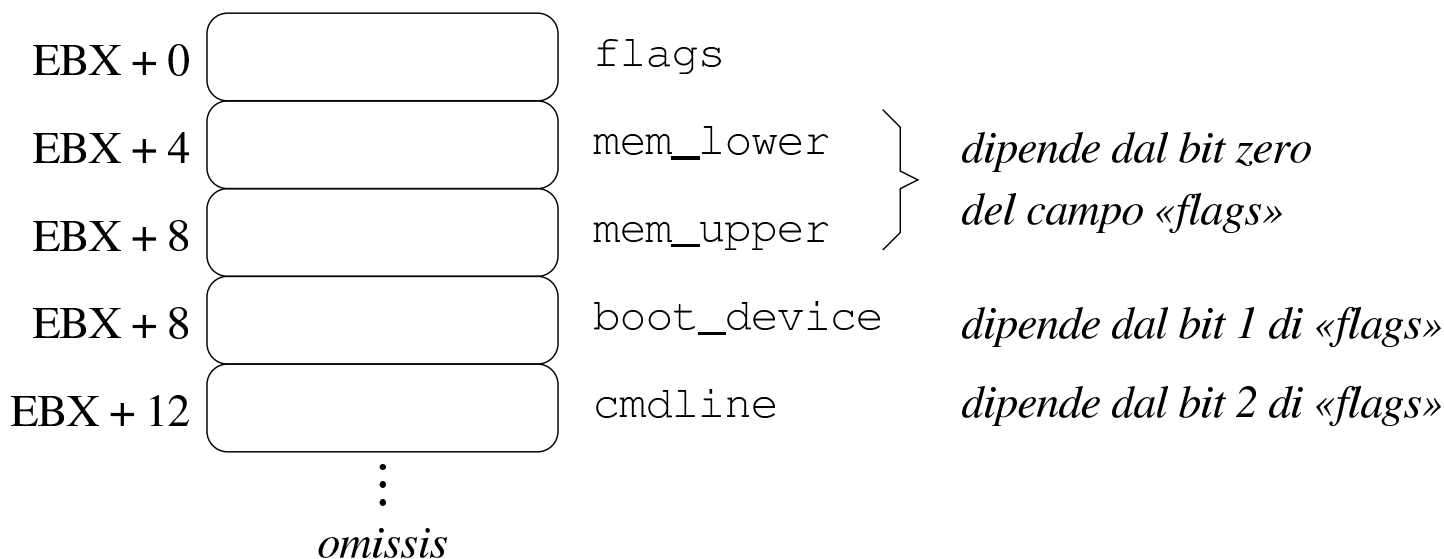


Tabella 65.66. Descrizione dei primi campi della struttura informativa fornita dal sistema di avvio *multiboot*.

Nome mnemonico del campo	bit del campo 'flags' da cui dipende	Descrizione
flags		Il primo campo definisce una serie di indicatori, con i quali si dichiara se una certa informazione, successiva, viene fornita ed è valida.

Nome mnemonico del campo	bit del campo 'flags' da cui dipende	Descrizione
<p>mem_lower</p> <p>mem_upper</p>	<p>0</p>	<p>Se è attivo il bit meno significativo del campo 'flags', i campi 'mem_lower' e 'mem_upper' contengono la dimensione della memoria bassa (da zero a un massimo di 640 Kibyte) e della memoria alta (quella che si trova a partire da un mebibyte). La dimensione è da intendersi in kibibyte (simbolo Kibyte) e, per quanto riguarda la memoria alta, viene indicata solo la dimensione continua fino al primo «buco».</p>
<p>boot_device</p>	<p>1</p>	<p>Se è attivo il secondo bit, partendo dal lato meno significativo, il campo 'boot_device' dà informazioni sull'unità di avvio. L'informazione è divisa in quattro byte, come descritto nelle specifiche <i>multiboot</i>.</p>
<p>cmdline</p>	<p>2</p>	<p>Se è attivo il terzo bit, partendo dal lato meno significativo, il campo 'cmdline' contiene l'indirizzo iniziale di una stringa che riproduce la riga di comando passata al kernel.</p>

Come si può intuire leggendo la tabella che descrive i primi cinque

campi, il significato dei bit del campo **'flags'** viene attribuito, mano a mano che l'aggiornamento delle specifiche prevede l'espansione della struttura informativa. Per esempio, un campo **'flags'** con il valore  $100_2$  sta a significare che esistono i campi fino a **'cmdline'** e il contenuto di quelli precedenti non è valido, ma i campi successivi, non esistono affatto. La comprensione di questo concetto dovrebbe rendere un po' più semplice la lettura delle specifiche.

### 65.5.2 Esempio di programma da avviare secondo le specifiche «multiboot»

«

I listati seguenti mostrano il contenuto dei file necessari a produrre un programma da avviare secondo le specifiche *multiboot*. Per la precisione, il programma non fa alcunché e serve solo come base di partenza per lo sviluppo di qualcosa di più complesso, con l'ausilio del linguaggio C. I file mostrati hanno, nell'ordine di apparizione, i nomi: `'loader.s'`, `'kernel.c'`, `'linker.ld'` e `'Makefile'`.

Listato 65.67. File `'loader.s'` usato per la prima parte del codice, contenente l'intestazione *multiboot* e la preparazione dell'ambiente minimo di funzionamento, compresa la collocazione della pila dei dati. Il programma chiama la funzione `'_kernel'` presente nel file `'kernel.c'`, passando come parametri il codice di riconoscimento del sistema di avvio e il puntatore alle altre informazioni che questo può fornire. Al ritorno dalla chiamata della funzione, il programma tenta di arrestare il microprocessore, ma se non ci riesce si mette in un ciclo senza fine che produce apparentemente lo stesso risultato.

```
.globl _loader
.extern _kernel
#
```

```
# Dimensione della pila interna al kernel. Qui vengono
# previsti 16384 elementi (0x4000) da 32 bit, pari a 65536
# byte.
#
.equ STACK_SIZE, 0x4000
#
# Si inizia subito con il codice che si mescola con i dati.
#
_loader:
    jmp boot      # Salta all'inizio del codice.
    .align 4      # Fa in modo di riempire lo spazio mancante
                  # al completamento di un blocco di 4 byte.
#
# Intestazione «multiboot», poco dopo l'inizio del
# file-immagine.
#
multiboot_header:
    .int 0x1BADB002          # magic
    .int 0x00000003          # flags
    .int -(0x1BADB002 + 0x00000003) # checksum
#
# Inizia il codice di avvio.
#
boot:
    #
    # Regola ESP alla base della pila.
    #
    movl $(stack_max + STACK_SIZE), %esp
    #
    # Azzera gli indicatori (e per questo usa la pila appena
    # sistemata).
    #
    pushl $0
    popf
```

```
#
# Chiama il kernel scritto in C, passandogli le
# informazioni ottenute dal sistema di avvio.
#
# void _kernel (unsigned int magic,
#               void *multiboot_info)
#
pushl %ebx    # Puntatore alla struttura contenente le
              # informazioni passate dal sistema di
              # avvio.
pushl %eax    # Codice di riconoscimento del sistema di
              # avvio.

#
call _kernel  # Chiama la funzione _kernel()
#
halt:
    hlt      # Se il kernel termina, ferma il
            # microprocessore.
    jmp halt # Se non si è fermato, crea un ciclo
            # senza fine.

#
# Alla fine del programma, viene collocato lo spazio per la
# pila dei dati, senza inizializzarlo. Per scrupolo si
# allinea ai 4 byte (32 bit).
#
.align 4
.comm stack_max, STACK_SIZE
#
```



Listato 65.68. File ‘kernel.c’ che potrebbe contenere idealmente il kernel di un piccolo sistema operativo. In questo caso il programma non fa alcunché e ignora anche la presenza di parametri nella chiamata.

```
void _kernel(void)
{
    ;
}
```

Listato 65.69. File ‘linker.ld’, da usare come script per GNU LD. Si può osservare che la sezione ‘.data’ viene distanziata da ‘.text’ e ‘.rodata’, in quanto si deve collocare in una pagina di memoria differente, per poter limitare i permessi di accesso in scrittura ai soli dati variabili.

```
ENTRY (_loader)
SECTIONS {
    . = 0x00100000;
    .text : { *(.text) }
    .rodata : { *(.rodata) }
    .data ALIGN (0x1000) : { *(.data) }
    .bss : {
        _sbss = .;
        *(.bss)
        *(COMMON)
        _ebss = .;
    }
}
```

## Listato 65.70. File 'Makefile' da usare per la compilazione.

```
all: loader kernel link
#
clean:
    rm *.o
    rm kernel
#
loader:
    as -o loader.o loader.s
#
kernel:
    gcc -Wall -Werror -o kernel.o -c kernel.c \
        -nostdlib -nostartfiles -nodefaultlibs
#
link:
    ld --script=linker.ld -o kernel loader.o kernel.o
```

Per avviare il programma che si ottiene, si può usare GRUB 1, utilizzando le direttive seguenti nel suo file di configurazione:

```
...
title mio kernel
kernel (fd0)/kernel
...
```

In questo caso si suppone di utilizzare un dischetto per l'avvio e che il file da avviare sia 'kernel', contenuto proprio nella radice.

### 65.5.3 Visualizzazione di messaggi



Quando si scrive un programma autonomo, come descritto sinteticamente nella sezione precedente, occorre considerare che il linguaggio C non può essere usato sfruttando le librerie consuete, pertan-

to occorre produrre tutto internamente, anche le funzioni per la visualizzazione dei messaggi. I listati seguenti vanno a sostituire il file 'kernel.c' della sezione precedente, allo scopo di visualizzare qualcosa sullo schermo (il contenuto dei primi campi della struttura informativa creata dal sistema di avvio), attraverso delle funzioni elementari definite internamente.

Per visualizzare un messaggio sullo schermo di un elaboratore x86 è necessario scrivere in una porzione di memoria che parte dall'indirizzo  $B8000_{16}$  (a partire da 736 Kibyte), utilizzando coppie di byte, dove il primo byte è un codice che descrive i colori da usare per il carattere e il suo sfondo, mentre il secondo contiene il carattere da visualizzare. Nel programma il codice in questione è  $07_{16}$  che mostra un carattere bianco su sfondo nero. Ciò che si deve osservare è che il programma tratta la coppia di byte come un numero a 16 bit, nel quale il carattere e il suo codice di visualizzazione sembrano invertiti, a causa del fatto che l'architettura è di tipo *little endian*.

Listato 65.72. File 'kernel.c' che include automaticamente i file 'display.c' e 'multiboot.c'.

```
#include "display.c"
#include "multiboot.c"
//
//
//
void
_kernel (unsigned long magic, type_multiboot_info *info)
{
    clear_screen ();
    //
    print_string ("Salve!\n\0");
    //
}
```

```
if (magic == 0x2BADB002)
{
    print_string ("Sono stato avviato attraverso un \0");
    print_string ("sistema di avvio aderente alle \0");
    print_string ("specifiche \n\0");
    print_string ("\\"multiboot\". Ecco solo alcune \0");
    print_string ("informazioni:\n\0");
    multiboot_information (info);
}
else
{
    print_string ("Sono stato avviato attraverso un \0");
    print_string ("sistema di avvio che non e' \0");
    print_string ("conforme alle specifiche\n\0");
    print_string ("\\"multiboot\".\n\0");
}
}
```

**Listato 65.73.** File 'display.c', contenente le funzioni necessarie a visualizzare stringhe e numeri in forma di stringa.

```
static unsigned short *Screen = (unsigned short *) 0xB8000;
//
static const unsigned int Rows = 25, Columns = 80;
static unsigned int Row = 0, Column = 0;
static unsigned char Attrib = 0x07;
//
//
//
static unsigned short
screen_cell (unsigned char c, unsigned char attrib)
{
    //
    // Assembla i due caratteri in un numero a 16 bit.
    //
```

```
    return (short) c | (((short) attrib) * 0x100);
}
//
//
//
static void
clear_screen (void)
{
    unsigned int i;
    //
    for (i = 0; i < (Rows * Columns) ; i++)
        {
            //
            // Scrive uno spazio nella posizione.
            //
            *(Screen + i) = screen_cell (0x20, Attrib);
        }
}
//
//
//
static void
new_line (void)
{
    int i, j;
    //
    Column = 0;
    Row++;
    //
    if (Row >= Rows)
        {
            //
            // Copia il testo in su.
            //
```

```
for (i = 0; i < (Rows - 1) * Columns ; i++)
{
    j = i + Columns;
    //
    // Trascrive la cella della riga successiva.
    //
    *(Screen + i) = *(Screen + j);
}
//
// Mette l'indice di riga nell'ultima posizione.
//
Row = Rows - 1;
//
// Pulisce la riga alla base dello schermo.
//
for (i = ((Rows - 1) * Columns) ;
     i < (Rows * Columns) ;
     i++)
{
    //
    // Cancella la cella dello schermo.
    //
    *(Screen + i) = screen_cell (0x20, Attrib);
}
}
//
//
//
static void
print_char (unsigned char c)
{
    //
    // Put the character.
```

```
//
if (c == '\n' || c == '\r')
{
    new_line ();
}
else
{
    *(Screen + (Row * Columns + Column))
    = screen_cell (c, Attrib);
    //
    // Move cursor.
    //
    Column++;
    if (Column >= Columns)
    {
        new_line ();
    }
}
}
//
//
//
static void
print_string (char *string)
{
    unsigned int i;
    //
    for (i = 0; i < 100000 ; i++)
    {
        if (string[i] != 0)
        {
            print_char (string[i]);
        }
        else

```

```
        {
            break;
        }
    }
}
//
//
//
static void
reverse_string (char *string)
{
    unsigned int i, j;
    unsigned char c;
    //
    // Scandisce la stringa alla ricerca del valore a zero.
    //
    for (i = 0; string[i] != 0; i++)
        {
            ;
        }
    //
    // L'indice "i" punta alla cella a zero.
    // Viene rimesso l'indice "i" in modo da puntare
    // all'ultimo carattere.
    //
    i--;
    //
    // Si inverte l'ordine delle cifre.
    //
    for (j = 0; j < i; j++, i--)
        {
            c = string[i];
            string[i] = string[j];
            string[j] = c;
        }
}
```



```
    }
    //
}
//
//
//
static void
num_to_string (unsigned long num, unsigned int base,
               char *string)
{
    unsigned int i;
    unsigned char remainder;
    //
    if (num == 0)
        {
            string[0] = '0';
            string[1] = 0;
            return;
        }
    //
    for (i = 0; num != 0; i++)
        {
            remainder = num % base;
            num = num / base;
            //
            if (remainder <= 9)
                {
                    string[i] = '0' + remainder;
                }
            else
                {
                    string[i] = 'A' + remainder - 10;
                }
        }
}
```

```
//
// Aggiunge la terminazione, tenendo conto che l'indice
// "i" è già posizionato dopo l'ultima cifra inserita.
//
string[i] = 0;
//
reverse_string (string);
}
//
//
//
static void
print_num (unsigned long num, char base)
{
    char string[100];
    //
    if (base == 'x')
        {
            num_to_string (num, 16, string);
            print_string ("0x\0");
            print_string (string);
        }
    else if (base == 'o')
        {
            num_to_string (num, 8, string);
            print_string ("0o\0");
            print_string (string);
        }
    else if (base == 'b')
        {
            num_to_string (num, 2, string);
            print_string ("0b\0");
            print_string (string);
        }
}
```

```
else
{
    num_to_string (num, 10, string);
    print_string (string);
}
}
```

Listato 65.74. File ‘multiboot.c’, contenente la definizione parziale della struttura delle informazioni *multiboot* e la funzione necessaria a visualizzarne il contenuto.

```
//
// The multiboot information.
//
typedef struct multiboot_info
{
    unsigned long flags;
    unsigned long mem_lower;
    unsigned long mem_upper;
    unsigned long boot_device;
        char *cmdline;
} type_multiboot_info;
//
//
//
static void
multiboot_information (type_multiboot_info *info)
{
    print_string ("flags:          \0");
    print_num (info->flags, 'b');
    print_string ("\n\0");
    //
    if ((info->flags & 1) > 0)
    {
        print_string ("mem_lower:      \0");
```

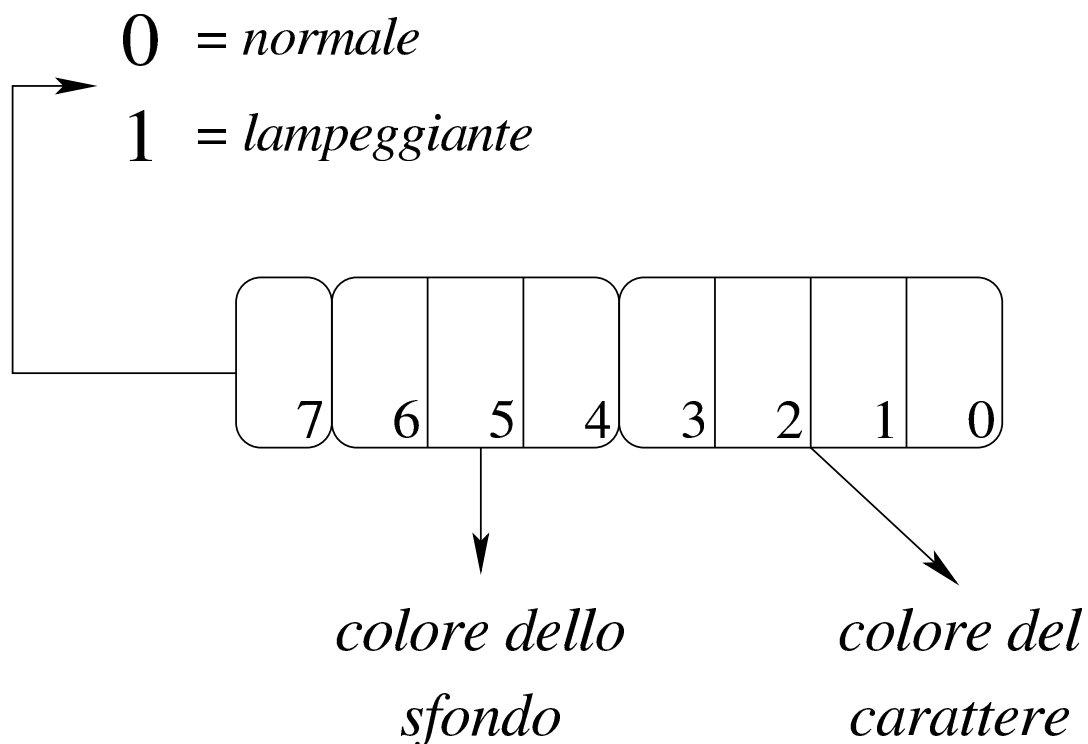
```
    print_num (info->mem_lower, 'x');
    print_string (" \0");
    print_num (info->mem_lower, 'd');
    print_string (" Kibyte\0");
    print_string ("\n\0");
    //
    print_string ("mem_upper:  \0");
    print_num (info->mem_upper, 'x');
    print_string (" \0");
    print_num (info->mem_upper, 'd');
    print_string (" Kibyte\0");
    print_string ("\n\0");
}
if ((info->flags & 2) > 0)
{
    print_string ("boot_device: \0");
    print_num (info->boot_device, 'x');
    print_string ("\n\0");
}
if ((info->flags & 4) > 0)
{
    print_string ("cmdline:      \0");
    print_string (info->cmdline);
    print_string ("\n\0");
}
}
```

Utilizzando questo programma si potrebbe visualizzare una schermata simile a quella seguente:

```
Salve!  
Sono stato avviato attraverso un sistema di avvio aderente  
alle specifiche "multiboot". Ecco solo alcune informazioni:  
flags:          0b11111100111  
mem_lower:     0x27F 639 Kibyte  
mem_upper:     0x7C00 31744 Kibyte  
boot_device:   0xFFFFFFFF  
cmdline:       (fd0)/kernel
```

## 65.5.4 Colori dello schermo

Nella sezione precedente si accenna al fatto che a partire dall'indirizzo di memoria  $B8000_{16}$ , ciò che si scrive serve a ottenere una rappresentazione sullo schermo. Ogni carattere utilizza due byte, in quanto uno dei due contiene il carattere vero e proprio e l'altro l'attributo che ne definisce il colore. Il byte del colore va usato suddividendo i bit nel modo seguente:



Come si vede, se è attivo il bit più significativo si ottiene un carattere lampeggiante, quindi i tre bit successivi descrivono lo sfondo e i quattro bit meno significativi descrivono invece il colore del carattere (in primo piano). Pertanto, i colori dello sfondo sono in quantità minore rispetto a quelli utilizzabili per il primo piano.

Tabella 65.77. Colore associato al primo piano o allo sfondo.

Codice	Sfondo	Primo piano
$0_{16}$	nero	nero
$1_{16}$	blu	blu
$2_{16}$	verde	verde
$3_{16}$	ciano (azzurro)	ciano (azzurro)
$4_{16}$	rosso	rosso
$5_{16}$	magenta (violetto)	magenta (violetto)
$6_{16}$	marrone	marrone
$7_{16}$	bianco	bianco
$8_{16}$	nero con lampeggio	grigio scuro
$9_{16}$	blu con lampeggio	blu chiaro
$A_{16}$	verde con lampeggio	verde chiaro
$B_{16}$	ciano con lampeggio	ciano chiaro
$C_{16}$	rosso con lampeggio	rosa
$D_{16}$	magenta con lampeggio	magenta chiaro
$E_{16}$	marrone con lampeggio	giallo
$F_{16}$	bianco con lampeggio	bianco luminoso

Per esempio, un colore indicato come  $28_{16}$  genera un testo di colore grigio scuro su sfondo verde, mentre  $A0_{16}$  genera un testo lampeggiante nero su sfondo verde.

## 65.6 Compilazione C dal basso in alto

Il valore del linguaggio C sta nel consentire una programmazione molto vicina a livello del linguaggio macchina, in modo relativamente indipendente dall'architettura. Ma ciò si può comprendere solo se si conosce il contesto operativo del linguaggio assembleatore, in modo particolare per quanto riguarda la gestione della memoria e tanto più per il modo in cui si utilizza la pila dei dati.

Per la compilazione dei programmi di esempio si fa riferimento a GCC<sup>3</sup> (*GNU compiler collection*) e precisamente al programma frontale 'gcc'.

### 65.6.1 Compilazione di un programma che non fa uso di librerie

Un programma in linguaggio C che non faccia uso di librerie di alcun tipo, deve seguire alcune regole che riguardano i programmi scritti in linguaggio assembleatore. Il listato seguente contiene la procedura per il calcolo del fattoriale, partendo da un valore già presente in memoria (si calcola precisamente il fattoriale di 5), ma il risultato non viene visualizzato in alcun modo, dal momento che questa sarebbe un'operazione che richiede proprio l'uso di librerie apposite:

```
int x = 5;
int i = 0;
void _start (void)
{
    i = (x - 1);
    while (i > 0)
    {
        x = x * i;
        i--;
    }
}
```

```
    }  
}
```

Se si conoscono i rudimenti del linguaggio C, si può osservare che, al posto della funzione *main()*, appare invece *\_start()*, come si fa in un programma scritto in linguaggio assembleatore.

Supponendo che il file che contiene quanto mostrato si chiami 'fact.c', la compilazione potrebbe iniziare dalla trasformazione in linguaggio assembleatore:

```
$ gcc -Wall -Werror -S -o fact.s fact.c ↵  
↵ -nostdlib -nostartfiles -nodefaultlibs [Invio]
```

Si otterrebbe il file 'fact.s', in linguaggio assembleatore, che, a seconda della versione di GCC, potrebbe essere molto simile al listato seguente:

```
    .file    "fact.c"  
.globl x  
    .data  
    .align 4  
    .type   x, @object  
    .size   x, 4  
x:  
    .long   5  
.globl i  
    .bss  
    .align 4  
    .type   i, @object  
    .size   i, 4  
i:  
    .zero   4  
    .text  
.globl _start
```



```

        .type    _start, @function
_start:
        pushl   %ebp
        movl    %esp, %ebp
        movl    x, %eax
        decl   %eax
        movl    %eax, i
        jmp    .L2
.L3:
        movl    x, %edx
        movl    i, %eax
        imull   %edx, %eax
        movl    %eax, x
        movl    i, %eax
        decl   %eax
        movl    %eax, i
.L2:
        movl    i, %eax
        testl   %eax, %eax
        jg     .L3
        popl   %ebp
        ret
        .size   _start, .-_start
        .ident  "GCC: (GNU) 4.1.2 20061115 ↵
↵(prerelease) (Debian 4.1.1-21)"
        .section .note.GNU-stack,"",@progbits

```

Tale file in linguaggio assembleatore può essere compilato con GNU AS e GNU LD nel modo consueto:

```
$ as -o fact.o fact.s [Invio]
```

```
$ ld -o fact fact.o [Invio]
```

Si può ispezionare il programma ottenuto con Objdump:

```
$ objdump -x fact [Invio]
```

```
fact:      file format elf32-i386
```

```
fact
```

```
architecture: i386, flags 0x00000112:
```

```
EXEC_P, HAS_SYMS, D_PAGED
```

```
start address 0x08048094
```

```
Program Header:
```

```
LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
      filesz 0x000000cd memsz 0x000000cd flags r-x
LOAD off    0x000000d0 vaddr 0x080490d0 paddr 0x080490d0 align 2**12
      filesz 0x00000004 memsz 0x00000008 flags rw-
STACK off   0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
      filesz 0x00000000 memsz 0x00000000 flags rw-
```

```
Sections:
```

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000039	08048094	08048094	00000094	2**2
			CONTENTS, ALLOC, LOAD, READONLY, CODE			
1	.data	00000004	080490d0	080490d0	000000d0	2**2
			CONTENTS, ALLOC, LOAD, DATA			
2	.bss	00000004	080490d4	080490d4	000000d4	2**2
			ALLOC			
3	.comment	0000003a	00000000	00000000	000000d4	2**0
			CONTENTS, READONLY			

```
SYMBOL TABLE:
```

```
08048094 l    d  .text  00000000 .text
080490d0 l    d  .data  00000000 .data
080490d4 l    d  .bss   00000000 .bss
00000000 l    d  .comment      00000000 .comment
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
00000000 l   df *ABS*  00000000 fact.c
080490d0 g    O  .data  00000004 x
080490d4 g    O  .bss   00000004 i
08048094 g    F  .text  00000039 _start
080490d4 g    *ABS* 00000000 __bss_start
080490d4 g    *ABS* 00000000 _edata
080490d8 g    *ABS* 00000000 _end
```

È possibile fare in modo che GCC interpelli automaticamente GNU AS, in modo da generare un file oggetto senza mostrare la creazione del file in linguaggio assembleatore (la trasformazione in linguaggio assembleatore avviene ugualmente, in un file temporaneo che poi viene cancellato in modo automatico). Pertanto, la compilazione si ridurrebbe ai due comandi seguenti:

```
$ gcc -Wall -Werror -c -o fact.o fact.c ↵  
↵ -nostdlib -nostartfiles -nodefaultlibs [Invio]
```

```
$ ld -o fact fact.o [Invio]
```

È il caso di osservare che il programma eseguibile ottenuto dal sorgente mostrato, produce un errore, dal momento che manca la chiamata della funzione del sistema operativo che ne conclude l'attività.

## 65.6.2 Uso di GDB e di DDD

Per poter sfruttare programmi come GDB, allo scopo di analizzare il funzionamento del programma, è necessario inserire delle informazioni aggiuntive durante la fase di trasformazione nel formato del linguaggio assembleatore. In pratica, si tratta di utilizzare l'opzione **'-gstabs'**, o altre simili, nella riga di comando di GCC. Riprendendo l'esempio della sezione precedente, la compilazione verrebbe eseguita con il comando seguente:

```
$ gcc -Wall -Werror -gstabs -S -o fact.s fact.c ↵  
↵ -nostdlib -nostartfiles -nodefaultlibs [Invio]
```

In questo caso, nel file in linguaggio assembler si troverebbero delle informazioni in più:

```
.file "fact.c"
.stabs "fact.c",100,0,2,.Ltext0
.text
.Ltext0:
.stabs "gcc2_compiled.",60,0,0,0
.stabs "int:t(0,1)=r(0,1);-2147483648;2147483647;",128,0,0,0
.stabs "char:t(0,2)=r(0,2);0;127;",128,0,0,0
.stabs "long int:t(0,3)=r(0,3);-2147483648;2147483647;",128,0,0,0
.stabs "unsigned int:t(0,4)=r(0,4);0;4294967295;",128,0,0,0
.stabs "long unsigned int:t(0,5)=r(0,5);0;4294967295;",128,0,0,0
.stabs "long long int:t(0,6)=r(0,6);-0;4294967295;",128,0,0,0
.stabs "long long unsigned int:t(0,7)=r(0,7);0;-1;",128,0,0,0
.stabs "short int:t(0,8)=r(0,8);-32768;32767;",128,0,0,0
.stabs "short unsigned int:t(0,9)=r(0,9);0;65535;",128,0,0,0
.stabs "signed char:t(0,10)=r(0,10);-128;127;",128,0,0,0
.stabs "unsigned char:t(0,11)=r(0,11);0;255;",128,0,0,0
.stabs "float:t(0,12)=r(0,1);4;0;",128,0,0,0
.stabs "double:t(0,13)=r(0,1);8;0;",128,0,0,0
.stabs "long double:t(0,14)=r(0,1);12;0;",128,0,0,0
.stabs "void:t(0,15)=(0,15)",128,0,0,0
.globl x
.data
.align 4
.type x, @object
.size x, 4
x:
.long 5
.globl i
.bss
.align 4
.type i, @object
.size i, 4
i:
.zero 4
.text
.stabs "_start:F(0,15)",36,0,0,_start
.globl _start
.type _start, @function
_start:
```

```
        .stabn 68,0,4,.LM0-_start
.LM0:
        pushl  %ebp
        movl   %esp, %ebp
        .stabn 68,0,5,.LM1-_start
.LM1:
        movl   x, %eax
        decl   %eax
        movl   %eax, i
        .stabn 68,0,6,.LM2-_start
.LM2:
        jmp    .L2
.L3:
        .stabn 68,0,8,.LM3-_start
.LM3:
        movl   x, %edx
        movl   i, %eax
        imull  %edx, %eax
        movl   %eax, x
        .stabn 68,0,9,.LM4-_start
.LM4:
        movl   i, %eax
        decl   %eax
        movl   %eax, i
.L2:
        .stabn 68,0,6,.LM5-_start
.LM5:
        movl   i, %eax
        testl  %eax, %eax
        jg     .L3
        .stabn 68,0,11,.LM6-_start
.LM6:
        popl   %ebp
        ret
        .size  _start, .-_start
.Lscope0:
        .stabs "x:G(0,1)",32,0,0,0
        .stabs "i:G(0,1)",32,0,0,0
        .stabs "\",100,0,0,.Letext0
.Letext0:
        .ident "GCC: (GNU) 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)"
```

```
.section .note.GNU-stack,"",@progbits
```

Per la compilazione successiva non ci sono cambiamenti; va quindi osservato che non è più compito di GNU AS l'inserimento di tali informazioni:

```
$ as -o fact.o fact.s [Invio]
```

```
$ ld -o fact fact.o [Invio]
```

Per utilizzare GDB o DDD si procede come nel caso di un programma che parte direttamente da un sorgente in linguaggio assembleatore:

```
$ gdb fact [Invio]
```

```
(gdb) break _start [Invio]
```

```
Breakpoint 1 at 0x8048097: file fact.c, line 5.
```

```
(gdb) run [Invio]
```

```
Breakpoint 1, _start () at fact.c:5
```

```
5          i = (x - 1);
```

```
(gdb) stepi [Invio]
```

```
0x0804809c      5          i = (x - 1);
```

```
(gdb) stepi [Invio]
```

```
0x0804809d in _start () at fact.c:5
```

```
5          i = (x - 1);
```

```
(gdb) stepi [Invio]
```

```
6           while (i > 0)
```

Come si può osservare, occorrono più comandi di avanzamento per passare alla riga successiva del codice originale, perché in realtà si fa riferimento alle istruzioni in linguaggio macchina.

```
(gdb) print i [Invio]
```

```
$1 = 4
```

```
(gdb) print x [Invio]
```

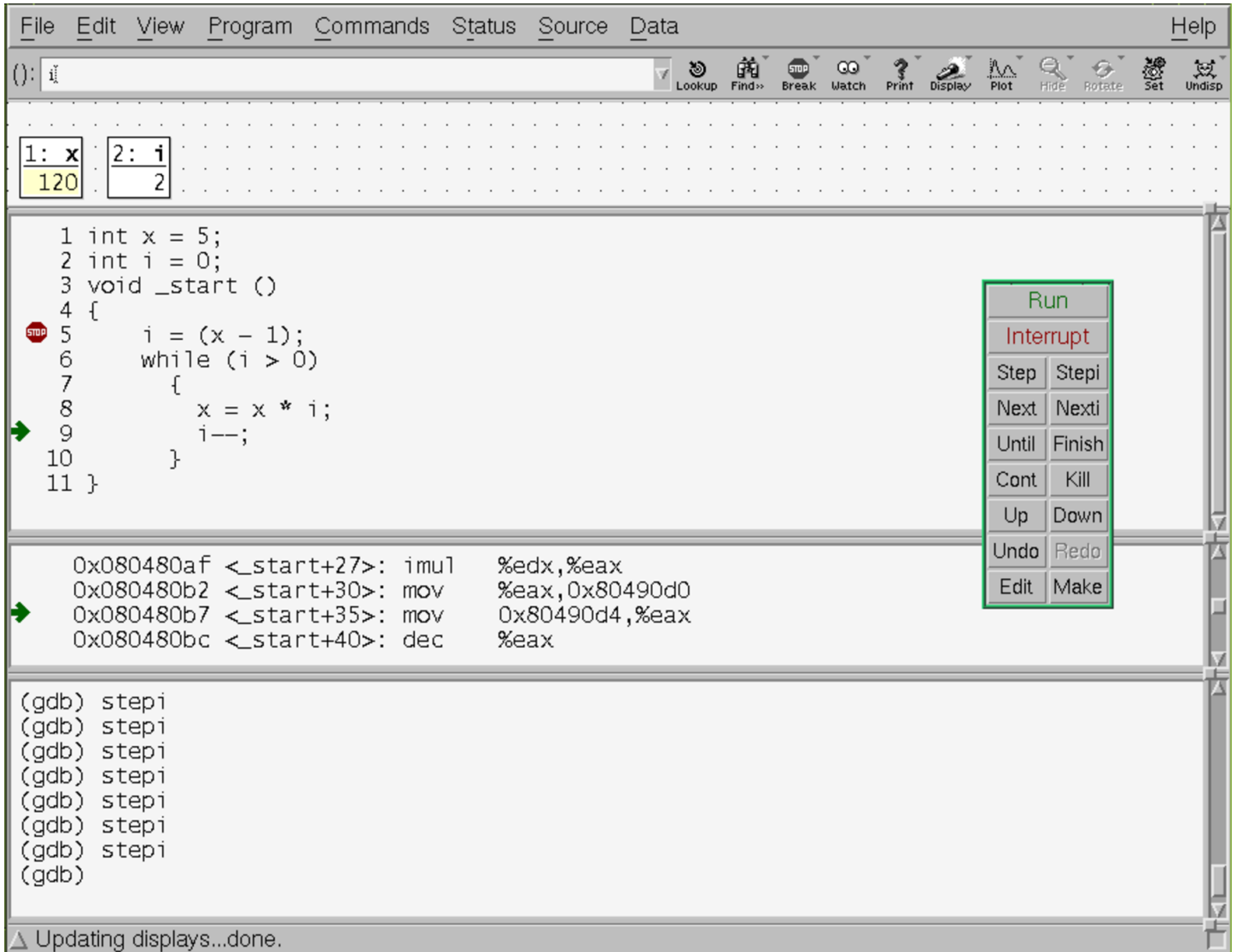
```
$2 = 5
```

```
(gdb) quit [Invio]
```

Naturalmente, se si può utilizzare DDD, tutto diventa più semplice:

```
$ ddd fact [Invio]
```

Figura 65.89. DDD che mette in evidenza lo stato di due variabili (si attiva la loro visualizzazione facendo un clic sul pulsante a icona denominato `DISPLAY`) durante il funzionamento, passo passo, del programma.



### 65.6.3 Da «\_start» a «main»

«

Per fare in modo che un programma in linguaggio C inizi dalla funzione *main()*, così come si prevede sia, si può istruire il collegatore (*linker*), attraverso uno script apposito che, in un sistema GNU/Linux, potrebbe essere come quello seguente:



```
ENTRY (main)
SECTIONS {
    . = 0x08048000 + SIZEOF_HEADERS;
    .text . : { *(.text) }
    .data ALIGN (0x1000) : { *(.data) }
    .bss . : {
        _sbss = .;
        *(.bss)
        *(COMMON)
        _ebss = .;
    }
}
```

## Il nuovo sorgente C:

```
int x = 5;
int i = 0;
int main ()
{
    i = (x - 1);
    while (i > 0)
    {
        x = x * i;
        i--;
    }
    return x;
}
```

Per la compilazione, i passaggi sarebbero quelli seguenti, supponendo che lo script per GNU LD sia contenuto nel file ‘config.ld’:

```
$ gcc -Wall -Werror -S -o fact.s fact.c ↵
↵ -nostdlib -nostartfiles -nodefaultlibs [Invio]
```

```
$ as -o fact.o fact.s [Invio]
```

```
$ ld -T config.ld -o fact fact.o [Invio]
```

Tuttavia, rimane ancora il problema della conclusione del programma che non avviene in modo grazioso. Se si osserva la nuova versione del programma, la funzione (che ora si chiama *main()*) restituisce un valore intero, corrispondente al risultato del calcolo eseguito, solo che non è stato chiarito in che modo quel valore debba essere acquisito dal sistema operativo. Si può quindi procedere in un modo diverso, creando un piccolo programma in linguaggio assembleatore, da associare a quello in linguaggio C:

```
.section .text
.globl _start
.extern main
_start:
    call main
    mov    %eax, %ebx
    mov    $1, %eax
    int    $0x80
```

Supponendo che questo file si chiami ‘**start.s**’, la compilazione complessiva potrebbe essere svolta nel modo seguente:

```
$ gcc -Wall -Werror -gstabs -S -o fact.s fact.c ↵
↵      -nostdlib -nostartfiles -nodefaultlibs [Invio]
```

```
$ as -o fact.o fact.s [Invio]
```

```
$ as --gstabs -o start.o start.s [Invio]
```

```
$ ld -o fact start.o fact.o [Invio]
```

Come si vede sono state aggiunte le opzioni ‘**-gstabs**’ e

`--gstabs`, dove appropriato; inoltre non serve più lo script per GNU LD. Se si avvia il programma, questo si arresta correttamente restituendo il fattoriale di 5:

```
$ ./fact ; echo $? [Invio]
```

120

#### 65.6.4 Compilazione naturale di un programma in linguaggio C

Quando non si utilizzano le opzioni `-nostdlibs`, `-nostartfiles` e `-nodefaultlibs`, la compilazione attraverso GCC avviene in modo più intuitivo, con l'inclusione automatica di tutto quello che è necessario per far sì che il programma parta dalla propria funzione *main()*; inoltre, se non si specifica il nome che si vuole produrre, si ottiene direttamente un file eseguibile con il nome `a.out`, secondo la tradizione.

In condizioni normali vengono inclusi nella compilazione alcuni file-oggetto che hanno un nome corrispondente al modello `crt*.o` e la libreria Libc. All'interno di uno di quei file-oggetto si trova la funzione `_start()`, dalla quale si arriva poi alla chiamata di *main()* in modo analogo a quanto mostrato nella sezione precedente, ma questi file potrebbero coinvolgere anche la libreria Libc.

L'opzione `-nostartfiles` serve a impedire che vengano incorporati automaticamente i file che contengono la funzione `_start()` e tutto ciò che altrimenti si prevede di far fare al programma prima di entrare nella funzione *main()*. L'opzione `-nodefaultlibs` serve a impedire l'inclusione automatica della libreria Libc. L'op-

zione `-nostdlibs` richiede entrambe le cose ed è stata usata negli esempi in modo ridondante.

Ecco la classica compilazione che produce direttamente il file eseguibile con il nome `a.out`:

```
$ gcc -Wall -Werror -gstabs fact.c [Invio]
```

Per buona abitudine è bene usare sempre l'opzione `-Wall` e possibilmente anche `-Werror`; inoltre, l'uso di `-gstabs` diventa essenziale per potersi avvalere di programmi come `GDB`.

Si può verificare che questo basta per arrivare al risultato voluto:

```
$ ./a.out ; echo $? [Invio]
```

120

Se poi si vogliono usare comandi tradizionali, da `gcc` occorre passare a `cc`, ma in un sistema GNU si tratta normalmente di un collegamento simbolico a `gcc` stesso.

Tabella 65.95. Riepilogo delle opzioni utilizzate con `gcc` nel corso del capitolo.

Opzione	Descrizione
<code>-s</code>	Genera un file in linguaggio assembler (prevale sull'opzione <code>-c</code> ).
<code>-o nome_file</code>	Dichiara il nome del file che si vuole ottenere.

Opzione	Descrizione
-c	Fa sì che la compilazione salti la fase di collegamento ( <i>link</i> ). In condizioni normali serve a generare solo i file-oggetto. Se si usa questa opzione, ma non si specifica l'opzione '-o', il file-oggetto ha un nome con la stessa radice del file sorgente e l'estensione '.o'.
-Wall	Richiede di mostrare tutti i messaggi che avvertono dell'uso imperfetto del linguaggio ( <i>warning</i> ).
-Werror	Fa sì che tutte le segnalazioni di avvertimento siano trattate come errori e portino al fallimento della compilazione.
-gstabs	Inserisce delle annotazioni, con le quali i programmi come GDB possono abbinare il sorgente originale all'esecuzione controllata del programma.

## 65.7 Compilazione C dall'alto in basso

Tradizionalmente, la compilazione di un programma scritto in linguaggio C avviene utilizzando il comando '**cc**' come nell'esempio seguente, sapendo che se non si usa l'opzione '-o' si ottiene il file 'a.out':

```
$ cc mio.c [Invio]
```

Tuttavia, l'elaborazione del file in linguaggio C richiede diversi passaggi, prima di arrivare al file eseguibile finale; passaggi che è bene tenere in considerazione.

In un sistema GNU il compilatore standard è GCC (*GNU compiler collection*) che si usa sia per il C, sia per altri linguaggi. Nel caso del linguaggio C, il programma frontale è precisamente ‘**gcc**’, al quale corrisponde comunque il collegamento ‘**cc**’.

Qui non si esaurisce il problema e si accenna soltanto alle situazioni più comuni. Per un approfondimento si vedano i documenti citati nella bibliografia che conclude il capitolo.

### 65.7.1 Le fasi della compilazione

«

La compilazione di un programma scritto in linguaggio C prevede diverse fasi: precompilazione, trasformazione in linguaggio assembler, trasformazione in file-oggetto, collegamento (*link*) di uno o più file-oggetto in un file eseguibile. Per conservare i file intermedi della compilazione si può usare l’opzione ‘**-save-temps**’ di ‘**gcc**’, come nell’esempio seguente:

```
$ gcc -save-temps mio.c [Invio]
```

In questo caso si ottengono i file ‘**mio.i**’, ‘**mio.s**’ e ‘**mio.o**’, contenenti rispettivamente il risultato elaborato dal precompilatore, la trasformazione in linguaggio assembler e il file-oggetto finale. Se poi il programma contenuto nel file sorgente è completo, si ottiene anche il file ‘**a.out**’ che costituisce il programma eseguibile.

Eventualmente, alcune opzioni di ‘**gcc**’ consentono di fermare l’elaborazione a uno stadio prestabilito: ‘**-E**’ serve a ottenere solo l’elaborazione da parte del precompilatore; ‘**-S**’ serve a ottenere il sorgente in linguaggio assembler; ‘**-c**’ serve a compilare, ma senza eseguire il collegamento finale (pertanto si ottiene il file-oggetto rilocabile).

## 65.7.2 Precompilatore

Ogni compilatore C «standard» prevede che il file sorgente venga elaborato, prima della compilazione vera e propria, attraverso un precompilatore, il quale elabora il sorgente e genera un altro sorgente ottenuto dall'interpretazione delle istruzioni di «precompilazione». Queste istruzioni di precompilazione costituiscono un linguaggio indipendente dal C vero e proprio. Il precompilatore di GCC è 'cpp' e di norma viene chiamato automaticamente da 'gcc' stesso, come già accennato nella sezione precedente.

```
#include <stdio.h>
int main (void)
{
    printf ("Ciao a tutti!\n");
    return 0;
}
```

Nell'esempio mostrato, l'istruzione '**#include <stdio.h>**' riguarda il precompilatore e richiede l'inclusione del file 'stdio.h' in quella posizione (il file si deve trovare all'interno di una directory prestabilita). Con l'opzione '**-E**' di 'gcc' (oppure anche con '**-save-temps**') si può vedere il risultato della precompilazione:

```
$ gcc -E -o mio.i mio.c [Invio]
```

Il file 'mio.i' che si genera dall'elaborazione ha un aspetto simile al pezzo che si vede nel listato successivo:

```
# 1 "mio_file.c"
# 1 "<built-in>"
# 1 "<command line>"
...
...
```

```
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
...
extern void funlockfile (FILE *__stream)
    __attribute__ ((__nothrow__));
# 834 "/usr/include/stdio.h" 3 4

# 2 "mio_file.c" 2
int main (void)
{
    printf ("Ciao a tutti!\n");
    return 0;
}
```

### 65.7.3 Compilazione dei file intermedi



Di norma, ogni compilatore tradizionale del linguaggio C si prende cura di tutte le fasi della compilazione, chiamando a sua volta i programmi necessari. Pertanto, con lo stesso programma frontale è possibile avviare manualmente la compilazione da fasi successive.

Per esempio:

1. `$ cc mio_file.i` [Invio]
2. `$ cc mio_file.s` [Invio]
3. `$ cc mio_file.o` [Invio]

Negli esempi si mostra l'uso del comando 'cc', ma 'gcc' è perfettamente conforme a questa convenzione tradizionale. Come si può



intuire, dall'estensione del nome del file il programma frontale determina quali azioni deve intraprendere: nel primo caso avvia la compilazione saltando solo la fase iniziale dell'analisi del precompilatore; nel secondo caso avvia l'assemblatore (e quindi continua con il collegatore); nell'ultimo caso avvia soltanto il collegatore (*linker*).

Naturalmente è possibile mescolare file differenti assieme, se la somma di questi deve portare a un solo file-eseguibile finale. Per esempio, si può compilare un programma composto dai file 'uno.c', 'due.i', 'tre.s' e 'quattro.o', dove ognuno viene elaborato in base alle proprie esigenze e alla fine il tutto viene collegato assieme:

```
$ cc uno.c due.i tre.s quattro.o [Invio]
```

#### 65.7.4 L'uso di librerie

In generale, la compilazione di un programma scritto secondo il linguaggio C implica automaticamente l'utilizzo della libreria Libc e il collegamento (*link*) con dei file-oggetto predefiniti, che contengono il codice necessario a preparare il programma prima di passare all'esecuzione della funzione *main()*.

Con 'gcc', per escludere l'utilizzo di qualunque libreria predefinita vanno usate le opzioni '**-nostartfiles**' e '**-nodefaultlibs**'; eventualmente l'opzione '**-nostdlibs**' dovrebbe valere per entrambe queste opzioni e può essere usata assieme a loro, benché sia ridondante.

Quando si fa uso di funzioni che non sono state dichiarate nel proprio programma, si tratta sempre di qualcosa che è contenuto in una libreria.

ria, di solito quella predefinita (Libc), ma per usarle correttamente è indispensabile che sia inserita all'inizio del file la dichiarazione del loro prototipo. Per questo, a seconda delle funzioni che si utilizzano, si includono i file che contengono i prototipi necessari; nel caso della funzione *printf()* si include comunemente il file 'stdio.h'.

Se si utilizza una funzione che appartiene a una libreria prevista nella compilazione, della quale però non si dichiara il prototipo, si può anche ottenere una compilazione «corretta», ma non è detto che, durante il funzionamento del programma, il passaggio degli argomenti attraverso i parametri della funzione avvenga in modo altrettanto corretto. In pratica, è molto probabile che la chiamata di tali funzioni produca risultati errati.

### 65.7.5 Librerie statiche e librerie dinamiche

«

Le librerie statiche sono file-oggetto raccolti in archivi generati con il programma 'ar', dove i nomi dei file di tali archivi hanno estensione '.a'. L'uso di queste librerie implica l'incorporazione del codice utilizzato nel programma finale.

Per compilare un programma che utilizza delle librerie statiche è sufficiente indicare i nomi dei file che le contengono, assieme agli altri file del programma:

```
$ gcc mio.c /usr/lib/libncurses.a [Invio]
```

In alternativa, secondo la modalità normale, quando i file di tali librerie si trovano nelle directory previste, si può usare l'opzione '-l', a cui si attacca il nome della libreria, ottenuto dal nome del file to-

gliendo l'estensione e il prefisso `'lib'`. Pertanto, l'esempio appena mostrato andrebbe trasformato così:

```
$ gcc -static mio.c -lncurses [Invio]
```

Le librerie dinamiche sono realizzate in modo differente rispetto a quelle statiche e sono contenute normalmente in file con estensione `'.so'`. La compilazione con l'uso di librerie dinamiche avviene in modo analogo a quanto visto per quelle statiche:

```
$ gcc mio.c /usr/lib/libncurses.so [Invio]
```

Oppure:

```
$ gcc -dynamic mio.c -lncurses [Invio]
```

Come si può intuire dagli esempi mostrati, se una stessa libreria è fornita sia in versione statica, sia in versione dinamica, le opzioni `'-static'` e `'-dynamic'` servono a precisare che tipo di compilazione si vuole. Se però si omette di specificarlo, in generale vengono utilizzate le librerie dinamiche.

L'opzione `'-l'` implica una ricerca dei file delle librerie all'interno di directory prestabilite, ma può succedere che sia necessario esplicitarlo nella riga di comando. In tal caso si può usare l'opzione `'-L'`:

```
$ gcc mio.c -L/opt/mia/lib -lmia [Invio]
```

Nell'esempio appena mostrato, la compilazione richiede l'uso della libreria `'mia'` (`'libmia.so'` o `'libmia.a'`) che va cercata prima nella directory `'/opt/mia/lib/'`.

Dal momento che l'uso delle librerie si affianca all'inclusione dei file che ne contengono il prototipo, conviene ricordare anche l'op-

zione `-I`, con la quale si richiede di cercare i file da includere a cominciare dalla directory specificata:

```
$ gcc mio.c -I/opt/mia/include -L/opt/mia/lib -lmia [Invio]
```

In questo nuovo esempio, si specifica anche che i file da includere vanno cercati a cominciare dalla directory `/opt/mia/include/`.

Naturalmente, il problema dei percorsi di ricerca per i file da includere riguarda solo quelli che nel sorgente si indicano tra parentesi angolari, come in questo esempio:

```
#include <stdio.h>
```

Diversamente, se il nome fosse messo tra apici doppi, il file verrebbe cercato nel percorso indicato esplicitamente nel sorgente stesso.

A ogni modo, quando la compilazione manifesta dei problemi che non sembrano dovuti a errori sintattici, conviene usare l'opzione `-v`, con la quale si vede esattamente cosa tenta di fare il programma frontale e dove si interrompe la compilazione. Ciò può essere molto utile per capire, per esempio, quando il problema deriva da file mancanti (librerie o altro).

Per il procedimento necessario alla produzione di una libreria, statica o dinamica, si veda la sezione [65.2](#).

## 65.7.6 L'ordine dei file e delle librerie nella compilazione

La compilazione corretta richiede che i file e le librerie siano indicati nella riga di comando secondo un ordine logico: prima il file che contiene la funzione *main()*, poi i file o le librerie contenenti le funzioni chiamate dal primo file, poi i file o le librerie contenenti le funzioni chiamate dai predecessori e così di seguito. Per esempio, se il file `uno.c` contiene la funzione *main()* e a sua volta chiama la funzione *due()* contenuta nel file `due.s`, la riga di comando per la compilazione deve avere l'aspetto seguente:

```
$ gcc uno.c due.s ...
```

Se poi la funzione *due()* si avvale della funzione *tre()*, contenuta nella libreria `libtre.a`, la riga di comando si sviluppa così:

```
$ gcc uno.c due.s -ltre ...
```

Naturalmente, anche la funzione *tre()* potrebbe avvalersi di una funzione contenuta in una seconda libreria. Per esempio potrebbe usare la funzione *quattro()* della libreria `libquattro.so`:

```
$ gcc uno.c due.s -ltre -lquattro ...
```

Questa è una regola generale da considerare in fase di collegamento (*link*). Si osservi che GNU LD (ovvero il programma usato automaticamente da `gcc` per questo scopo) non richiede necessariamente tale accorgimento, ma ugualmente è meglio curarsi di rispettare il principio.

## 65.7.7 Prevenzione e ricerca degli errori

«

Il linguaggio C può essere usato «bene» o «male», così come ogni altro linguaggio. Nel caso particolare del C, certi modi leciti di scrivere un programma possono essere facilmente motivo di errori banali, evitabili se si chiede al compilatore di segnalare anche le piccole mancanze. In pratica, con **'gcc'** è bene usare sempre l'opzione **'-Wall'** per ottenere la segnalazione di una serie numerosa di avvertimenti; eventualmente a questa opzione si può aggiungere **'-Werror'**, con la quale si trasformano gli avvertimenti in errori, così da evitare che in loro presenza la compilazione vada a buon fine.

Per analizzare il funzionamento del programma con GDB o altri analizzatori simili, conviene aggiungere l'opzione **'-gstabs'**, oppure un'altra opzione che inizi per **'-g...'**, in base alle caratteristiche del programma usato per l'analisi.

Infine, disponendo di un sistema GNU, o di un altro sistema compatibile con il modello di Unix, è bene abilitare lo scarico dell'immagine dei processi elaborativi in un file (*core dump*). Così facendo, quando durante il funzionamento un programma tenta di eseguire un'azione che il sistema impedisce, questo programma viene fermato e scaricato in un file **'core'** che può essere analizzato successivamente con GDB. A titolo di esempio viene mostrato un sorgente che produce un errore del genere:

```
int main (void)
{
    int a;
    a = 1 / 0;
    return a;
}
```

Se si compila il programma con l'accortezza di aggiungere l'opzione **'-Wall'** si viene avvisati del problema, ma in questo caso si preferisce ignorarlo:

```
$ gcc -Wall -gstabs errore.c [Invio]
```

```
errore.c: In function 'main':  
errore.c:4: warning: division by zero
```

Prima di proseguire, ci si assicura che lo scarico dell'immagine del processo elaborativo sia abilitata:<sup>4</sup>

```
$ ulimit -c unlimited [Invio]
```

Si avvia il programma difettoso:

```
$ ./a.out [Invio]
```

```
/bin/sh: line 1: 12134 Floating point exception↵  
↵(core dumped) ./a.out
```

Il messaggio della shell avvisa di avere «scaricato la memoria», ovvero di avere creato il file **'core'**. Con GDB si può procedere alla ricerca di cosa è stato a causare l'errore:

```
$ gdb a.out core [Invio]
```

```
...  
Core was generated by './a.out'.  
Program terminated with signal 8, Arithmetic exception.  
#0  0x08048344 in main () at errore.c:4  
4          a = 1 / 0;
```

## 65.7.8 Problemi con l'ottimizzazione

Il compilatore 'gcc' consente di utilizzare diverse opzioni per ottenere un risultato più o meno ottimizzato. L'ottimizzazione richiede una potenza elaborativa maggiore, al crescere del livello di ottimizzazione richiesto. In situazioni particolari, può succedere che la compilazione non vada a buon fine a causa di questo problema, interrompendosi con segnalazioni più o meno oscure, riferite alla scarsità di risorse. In particolare potrebbe essere rilevato un uso eccessivo della memoria virtuale, per arrivare fino allo scarico della memoria (*core dump*).

È evidente che in queste situazioni diventa necessario diminuire il livello di ottimizzazione richiesto, modificando opportunamente le opzioni relative. L'opzione in questione è '-On', come descritto nella tabella 65.102. In generale, l'assenza di tale opzione implica la compilazione normale senza ottimizzazione, mentre l'uso dell'opzione '-O0' può essere utile alla fine della serie di opzioni, per garantire l'azzeramento delle richieste di ottimizzazione precedenti.

Tabella 65.102. Opzioni di ottimizzazione per 'gcc'.

Opzione	Descrizione
-O	Ottimizzazione minima.
-O1	
-O2	Ottimizzazione media.
-O3	Ottimizzazione massima.
-O0	Annullamento delle richieste precedenti di ottimizzazione.



Alle volte, compilando un programma, può succedere che a causa del livello eccessivo di ottimizzazione prestabilito, non si riesca a produrre alcun risultato. In questi casi, può essere utile ritoccare lo script di Make, dopo l'uso del comando '**configure**'; per la precisione si deve ricercare un'opzione che inizia per '**-O**'. Purtroppo, il problema sta nel fatto che spesso si tratta di più di uno script, in base all'articolazione dei file che compongono il sorgente.

Ammesso che si tratti dei file 'Makefile', si potrebbe usare il comando seguente per attuare la ricerca:

```
$ find . -name Makefile ↵  
↵ -exec echo \{\} \; ↵  
↵ -exec grep \{-O \{\} \; [Invio]
```

Il risultato potrebbe essere simile a quello che si vede qui di seguito:

```
./doc/Makefile  
./backend/Makefile  
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072  
./frontend/Makefile  
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072  
./include/Makefile  
./japi/Makefile  
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072  
./lib/Makefile  
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072  
./sanei/Makefile  
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072  
./tools/Makefile  
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072  
./Makefile
```

In questo caso, si può osservare che i file './doc/Makefile', './include/Makefile' e 'Makefile', non contengono tale stringa.

Tabella 65.104. Riepilogo delle altre opzioni utilizzate con ‘gcc’ nel corso del capitolo.

Opzione	Descrizione
-E	Elabora il file solo con il precompilatore.
-S	Genera un file in linguaggio assembler (prevale sull’opzione ‘-c’).
-c	Fa sì che la compilazione salti la fase di collegamento ( <i>link</i> ). In condizioni normali serve a generare solo i file-oggetto. Se si usa questa opzione, ma non si specifica l’opzione ‘-o’, il file-oggetto ha un nome con la stessa radice del file sorgente e l’estensione ‘.o’.
-o <i>nome_file</i>	Dichiara il nome del file che si vuole ottenere.
-static -dynamic	Richiede espressamente di compilare utilizzando le librerie statiche o dinamiche.
-l <i>libreria</i>	Indica il nome di una libreria da utilizzare. Il nome del file che la contiene può essere ‘lib <i>libreria</i> .a’ o ‘lib <i>libreria</i> .so’, a seconda che si tratti di una libreria statica o dinamica.
-L <i>percorso</i>	Indica un percorso in cui ricercare i file delle librerie, che prende la precedenza sugli altri già considerati.
-I <i>percorso</i>	Indica un percorso in cui ricercare i file da includere, che prende la precedenza sugli altri già considerati.

Opzione	Descrizione
<code>-Wall</code>	Richiede di mostrare tutti i messaggi che avvertono dell'uso imperfetto del linguaggio ( <i>warning</i> ).
<code>-Werror</code>	Fa sì che tutte le segnalazioni di avvertimento siano trattate come errori e portino al fallimento della compilazione.
<code>-gstabs</code>	Inserisce delle annotazioni, con le quali i programmi come GDB possono abbinare il sorgente originale all'esecuzione controllata del programma.

## 65.8 Compilazione guidata con Make

La compilazione di un programma, in qualunque linguaggio sia scritto, può essere un'operazione molto laboriosa, soprattutto se si tratta di aggregare un sorgente suddiviso in più parti, o peggio, se si tratta di un progetto costituito da più programmi. Per semplificare la procedura si potrebbe predisporre uno script che esegue sequenzialmente tutte le operazioni necessarie, ma la tradizione richiede di utilizzare il programma Make.

Uno dei vantaggi più appariscenti nell'uso di Make sta nella possibilità di evitare che vengano rielaborati i file che non sono stati modificati, abbreviando quindi il tempo di compilazione necessario quando si procede a una serie di modifiche limitate.

Make viene usato normalmente assieme a uno script, denominato comunemente 'Makefile',<sup>5</sup> scritto in un modo che dovrebbe risultare molto semplice da interpretare; tuttavia, è comunque possibile fare il contrario, specialmente con le versioni più evolute di tale pro-

gramma. Evidentemente, Make è utile quando lo si utilizza con moderazione, ovvero con uno script semplice e lineare, altrimenti uno script di shell è sicuramente più appropriato al caso.

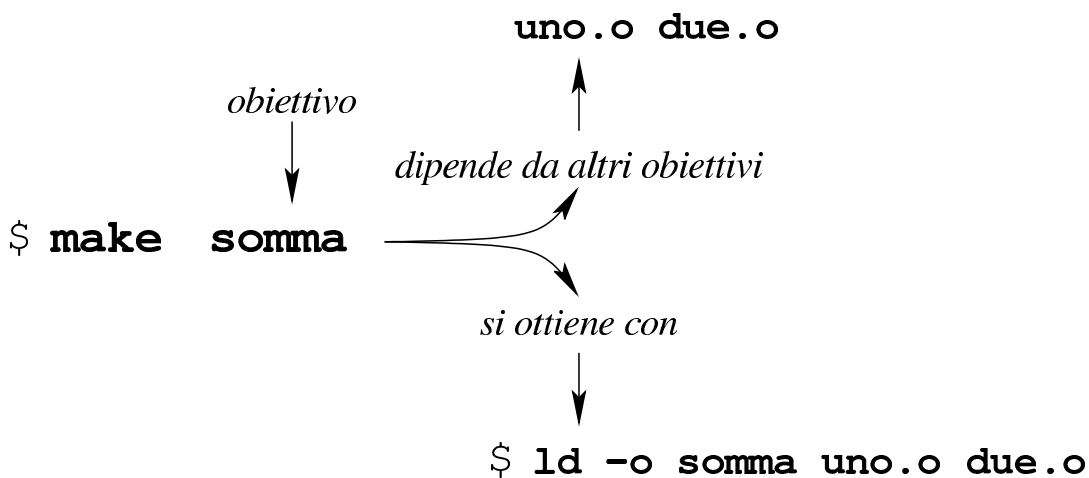
Gli esempi che qui mostrano script di Make non contengono commenti, pertanto è bene chiarire subito che le righe bianche o vuote vengono ignorate, così come si ignora il testo che appare alla destra del simbolo ‘#’.

### 65.8.1 Obiettivo, dipendenze e comandi

«

Make viene usato per realizzare un *obiettivo* attraverso uno o più comandi da impartire alla shell (precisamente ‘/bin/sh’), dopo che sono stati soddisfatti altri eventuali obiettivi da cui quello finale dipende. In linea di principio, **l’obiettivo è rappresentato dal nome di un file che deve essere generato.**

Per esempio, volendo produrre il programma ‘**somma**’ che si ottiene dalla compilazione dei file ‘uno.c’ e ‘due.c’, l’obiettivo «somma» che si ottiene con il comando ‘**ld -o somma uno.o due.o**’, dipende dagli obiettivi «uno.o» e «due.o», in quanto i file ‘uno.o’ e ‘due.o’ devono essere presenti per poter eseguire il collegamento con il programma ‘**ld**’.



Nello script di Make, l'obiettivo di esempio mostrato si descrive come si vede nella figura successiva, dove il tratteggio verticale a sinistra rappresenta l'inizio della prima colonna. Ciò che descrive un obiettivo è, nel suo complesso, una *regola*:

```
somma: uno.o due.o
      HT ld -o somma uno.o due.o
```

Si deve tenere a mente che la riga che definisce l'obiettivo e le dipendenze deve iniziare dalla prima colonna, mentre le righe contenenti dei comandi devono trovarsi rientrate con un carattere di tabulazione orizzontale (<HT>); al contrario, degli spazi veri e propri come rientro non sono ammissibili.

L'esempio introdotto è incompleto, perché non esplicita in che modo ottenere gli obiettivi 'uno.o' e 'due.o'. Ecco come potrebbe essere composto lo script completo delle regole che descrivono tutte le dipendenze:

```
somma: uno.o due.o
      ld -o somma uno.o due.o

uno.o: uno.c mate.h
      cc -c -o uno.o uno.c

due.o: due.c
      cc -c -o due.o due.c
```

Per comprendere l'esempio va chiarito che per ottenere il file 'uno.o' è necessario il file 'uno.c' che a sua volta include il file 'mate.h'.

h’.

Il vantaggio di usare Make sta nel fatto che questo tiene conto della data di modifica dei file, nel momento in cui valuta le dipendenze. Nel caso dell’esempio, per eseguire il collegamento (*link*) dei file oggetto nel file eseguibile ‘**somma**’, è necessario disporre di tali file oggetto, ma se il file eseguibile esiste e se questi file oggetto esistono e hanno una data di modifica antecedente a quella del file eseguibile, allora sarebbe da intendere che tale operazione non sia necessaria. Tuttavia, i file ‘uno.o’ e ‘due.o’ sono indicati come obiettivi da ottenere attraverso altri file: nel caso di ‘uno.o’ è stabilito che dipende dai file ‘uno.c’ e ‘mate.h’; nel caso di ‘due.o’ è stabilito che dipende solo dal file ‘due.c’ (si osservi che per i file ‘uno.c’, ‘mate.h’ e ‘due.c’ non sono state dichiarate altre dipendenze). A questo punto è logico attendersi che anche la data dei file di partenza conti. In pratica, le date di modifica di ‘uno.c’ e ‘mate.h’ devono essere antecedenti a quella di ‘uno.o’ e così deve essere antecedente anche quella di ‘due.c’ rispetto a quella di ‘due.o’. Se a un certo punto si modifica il file ‘mate.h’ (e quindi la data di modifica viene aggiornata dal sistema operativo), la dipendenza che riguarda il file ‘uno.o’ richiede la ripetizione dei comandi relativi; quindi viene ricompilato il file ‘uno.o’ e viene eseguito nuovamente il collegamento che genera il file eseguibile ‘**somma**’.

Figura 65.108. Modello sintattico di una regola, per la definizione di un obiettivo in uno script di Make.

```
obiettivo...: [dipendenza...]
```

```
<HT>comando
```

```
...
```

Per ottenere lo stesso risultato pratico dell'esempio mostrato, si può modificare il modo in cui si indica la dipendenza dovuta al file 'mate.h':

```
...
uno.o: uno.c
        cc -c -o uno.o uno.c

uno.c: mate.h
        touch uno.c mate.h
...
```

Ciò che appare nel pezzo mostrato indica che il file 'uno.o' dipende da 'uno.c' soltanto, ma il file 'uno.c' dipende dal file 'mate.h'. Se il file 'mate.h' si trova ad avere una data più recente di 'uno.c', le date vengono rese uguali e viene rifatta la compilazione.

## 65.8.2 Obiettivi fittizi

In generale, un obiettivo di Make viene raggiunto con la creazione o l'aggiornamento di un file che ha lo stesso nome dell'obiettivo, attraverso dei comandi stabiliti. In pratica, l'obiettivo è quel file da generare o aggiornare. Tuttavia, spesso si definiscono obiettivi che non implicano la creazione di un file con tale nome; pertanto servono per essere eseguiti sempre, assicurando che le dipendenze eventuali siano rispettate.

```
all: somma moltiplicazione

somma: ...
        ...
moltiplicazione: ...
        ...
...
```

L'esempio mostra una situazione tipica in cui si utilizza un obiettivo fittizio, in questo caso denominato **'all'**. Questo obiettivo ha il solo scopo di richiamare automaticamente gli obiettivi **'somma'** e **'moltiplicazione'** (ma nell'esempio, questi ulteriori obiettivi non vengono descritti). C'è da osservare però una cosa importante: **se per qualunque ragione dovesse esistere un file con lo stesso nome dell'obiettivo, avente una data di modifica successiva a quella dei file degli obiettivi da cui dipende, l'operazione non verrebbe eseguita**, salve naturalmente altre ipotesi riferite alle dipendenze degli obiettivi precedenti.<sup>6</sup>

Per ovviare all'inconveniente dovuto alla possibilità che esista un file con lo stesso nome di un obiettivo fittizio, non correlato a tale file, si può usare uno strattagemma consolidato:

```
clean: FORCE
      rm *.o core

FORCE:
```

In questo caso, l'obiettivo **'FORCE'** (usato comunemente per questo scopo), non ha dipendenze, non ha comandi, inoltre si dà per certo che non possa esistere un file con lo stesso nome; pertanto l'obiettivo risulta sempre da raggiungere. L'obiettivo **'clean'** che ha evidentemente lo scopo di eliminare alcuni file non più necessari, dipendendo dall'obiettivo **'FORCE'**, viene eseguito in ogni caso, anche se esistesse un file **'clean'**, perché la dipendenza non è mai soddisfatta.<sup>7</sup>



### 65.8.3 Scelta dell'obiettivo

Make è costituito generalmente dal programma eseguibile **'make'** e si usa solitamente secondo la sintassi seguente:

```
make [opzioni] [obiettivi]
```

Per esempio, il comando seguente richiede a Make di «raggiungere» l'obiettivo **'somma'**:

```
$ make somma [Invio]
```

Se però non si specifica l'obiettivo, questo viene determinato in modo predefinito:

```
$ make [Invio]
```

Ammesso che nella directory corrente sia presente lo script di Make (per convenzione deve trattarsi del file `'Makefile'`), l'obiettivo viene cercato al suo interno e se non è stato definito si intende il primo che appare nel file.<sup>8</sup>

È comunque possibile utilizzare Make anche senza script, ma in tal caso l'indicazione dell'obiettivo nella riga di comando è obbligatoria. L'utilizzo di Make senza uno script dipende da quelle che sono definite *regole implicite*. In pratica, quando si richiede un obiettivo non previsto espressamente, Make cerca di fare la cosa più logica, partendo dal presupposto che il contesto sia relativo alla compilazione di un programma. Si osservi l'esempio seguente:

```
$ make prova [Invio]
```

Se non è stato definito l'obiettivo **'prova'**, Make considera il contenuto della directory corrente e cerca qualcosa che sia ragionevol-

mente trasformabile nel file ‘prova’. Per esempio, se trova il file ‘prova.c’ esegue automaticamente il comando ‘**cc -o prova prova.c**’. Questa proprietà di Make consente di omettere la descrizione delle regole degli obiettivi «ovvi». Questo sistema di regole implicite serve anche per semplificare il lavoro di stesura di uno script di Make, quando si descrive un obiettivo finale e non si stabiliscono le regole per ottenere le dipendenze:

```
somma: uno.o due.o
        ld -o somma uno.o due.o

uno.c: mate.h
        touch mate.h uno.c
```

Questo esempio richiama quanto già mostrato in precedenza: dato che la costruzione dei file ‘uno.o’ e ‘due.o’ richiede dipendenze prevedibili, non è necessario descriverne le regole.

#### 65.8.4 Interpretazione dei comandi che portano a un obiettivo

«

In condizioni normali, i comandi che devono essere eseguiti per il raggiungimento di un certo obiettivo, vengono passati alla shell ‘/bin/sh’, indipendentemente dalla shell utilizzata dall’utente che avvia il programma ‘**make**’.

I comandi troppo lunghi possono essere spezzati e ripresi nella riga successiva, se alla fine della riga interrotta appare il simbolo ‘\’, esattamente come sarebbe in uno script per una shell Bourne. C’è però da osservare che, in questo caso, il comando passato alla shell comprende letteralmente sia ‘\’, sia il codice di interruzione di riga successivo, ma questo fatto, di norma, non ha conseguenze nel

risultato.

Sul problema dell'interruzione e proseguimento delle righe dei comandi occorre soffermarsi su un fatto: nella riga che viene ripresa, il carattere di tabulazione iniziale viene omesso automaticamente, nel momento in cui viene chiesto alla shell di eseguire il comando. L'esempio seguente rappresenta il contenuto di uno script che dovrebbe chiarire il meccanismo. Per ora si sorvoli sulla presenza della chiocciola all'inizio dei comandi:

```
esempio:
    @echo "supercalifragilisti\
    chespiralidoso"
    @echo "supercalifragilisti \
    chespiralidoso"
```

Ecco cosa succede:

```
$ make esempio [Invio]
```

```
supercalifragilistichespiralidoso
supercalifragilisti chespiralidoso
```

Si può osservare che nel primo caso la parola è rimasta unita, mentre nel secondo è separata perché uno spazio è stato inserito prima della segnalazione dell'interruzione.

I comandi di una regola sono eseguiti uno alla volta, ma Make tiene conto del risultato. Se il comando eseguito restituisce zero, ovvero se risulta eseguito correttamente, allora Make avvia il successivo, altrimenti interrompe l'operazione segnalando il fallimento dell'obiettivo e di quelli che da lui dipendono.<sup>9</sup> Pertanto, se i comandi possono restituire un errore anche se ciò non pregiudica il raggiungimento dell'obiettivo previsto, occorre provvedere in qualche modo.

Per esempio così:

```
obiettivo: ...
...
    mkdir ciao ; true
...
```

In questo caso, la regola che descrive l'obiettivo contiene un comando che serve a garantire la presenza di una certa directory. Il comando in questione potrebbe fallire se la directory esiste già, senza per questo pregiudicare il resto del procedimento, così si unisce al comando **'true'** che complessivamente fa sì che l'esito sia sempre «corretto». È comunque possibile usare un prefisso che informa Make di ignorare gli errori; si tratta del segno **'-'**, pertanto l'esempio appena apparso può essere modificato così:

```
obiettivo: ...
...
    -mkdir ciao
...
```

In generale, prima di avviare ogni comando, Make lo visualizza, in modo da far capire ciò che accade all'utente. In alcune situazioni, però, ciò può essere spiacevole, pertanto è possibile utilizzare il prefisso **'@'** che evita tale comportamento:

```
mio: ...
    @echo "sto per eseguire la compilazione, bla bla..."
    cc -o mio mio.c
```

Come si vede nell'esempio, si vuole fare in modo che il comando **'echo'** non sia «descritto», dato che già serve a mostrare qualcosa.

Tabella 65.118. Alcuni prefissi da usare nelle righe che contengono comandi.

Prefisso	Significato
-	fa in modo che gli errori vengano ignorati;
+	fa in modo che il comando venga eseguito sempre;
@	fa in modo che il testo del comando non venga mostrato.

### 65.8.5 Variabili o «macro»

All'interno di uno script di Make è possibile definire delle variabili, altrimenti note come «macro». Le variabili si dichiarano attraverso direttive espresse nella forma seguente:

```
nome = stringa
```

In particolare, la stringa non deve essere delimitata e l'ordine della dichiarazione delle variabili non viene tenuto in considerazione, come dimostrato poco più avanti. L'espansione di una variabile si indica attraverso due modi possibili:

```
$(nome)
```

Oppure:

```
${nome}
```

Si osservi l'esempio seguente, in particolare a proposito del fatto che l'ordine di dichiarazione delle variabili non è significativo:

```
bindir = $(exec_prefix)/bin
prefix = /usr/local
sbindir = $(exec_prefix)/sbin
exec_prefix = $(prefix)

all:
    @echo "prefix = $(prefix) "
    @echo "exec_prefix = $(exec_prefix) "
    @echo "bindir = $(bindir) "
    @echo "sbindir = $(sbindir) "
```

AmMESSo che questo sia lo script di Make contenuto nella directory corrente:

```
$ make [Invio]
```

```
prefix = /usr/local
exec_prefix = /usr/local
bindir = /usr/local/bin
sbindir = /usr/local/sbin
```

Il fatto che l'ordine nella dichiarazione nelle variabili non conti, implica che l'assegnamento a una variabile del proprio stesso contenuto produca un circolo vizioso. In pratica, una cosa come la dichiarazione seguente **non è ammissibile**:

```
opzioni = -c
opzioni = -gstabs $(opzioni)
```

Invece di agire così, per aggiungere qualcosa a una variabile occorre una direttiva differente:

```
nome += stringa
```

Si osservi l'esempio seguente e ciò che succede provando a usare **'make'**:

```
opzioni = -c
opzioni += -gstabs

all:
    @echo "opzioni = $(opzioni) "
```

```
$ make [Invio]
```

```
opzioni = -c -gstabs
```

Come si può intendere, le variabili di Make che appaiono all'interno dei comandi, vengono espansive prima dell'esecuzione dei comandi stessi; di conseguenza, se si vuole usare il dollaro ('\$') in modo che la shell lo recepisca, occorre raddoppiarlo:

```
obiettivo: ...
    ...
    NUM=3 ; echo $$NUM
    ...
```

GNU Make recepisce le variabili di ambiente e le assimila tra le proprie variabili, ma se nel proprio script vengono ridefinite, ciò prevale sul valore ottenuto dall'esterno.

Make prevede delle variabili predefinite, il cui scopo principale è controllare il funzionamento delle regole implicite, ma che spesso

vengono usate per coerenza anche nei comandi di obiettivi dichiarati esplicitamente. La tabella 65.127 ne elenca alcune e l'esempio successivo, riprendendone un altro già apparso, mostra in che modo potrebbero essere usate:

```
somma: uno.o due.o
        $(LD) $(LDFLAGS) -o somma uno.o due.o

uno.o: uno.c mate.h
        $(CC) -c $(CFLAGS) -o uno.o uno.c

due.o: due.c
        $(CC) -c $(CFLAGS) -o due.o due.c
```

Trattandosi di variabili conosciute, se utilizzate correttamente si facilita la lettura dello script, consentendo di precisare, se ce ne fosse bisogno, il nome del compilatore e le opzioni da dare:

```
LD = ld
CC = gcc
CFLAGS = -gstabs

somma: uno.o due.o
        $(LD) $(LDFLAGS) -o somma uno.o due.o

uno.o: uno.c mate.h
        $(CC) -c $(CPPFLAGS) $(CFLAGS) -o uno.o uno.c

due.o: due.c
        $(CC) -c $(CPPFLAGS) $(CFLAGS) -o due.o due.c
```



Tabella 65.127. Elenco di alcune variabili predefinite di Make.

Nome	Contenuto usuale	Annotazioni
MAKE	make	Il nome del programma stesso. Di solito viene usata questa informazione per l'avvio di altri Make in sottodirectory con un proprio script.
SHELL	/bin/sh	La shell che deve eseguire i comandi: è bene evitare di cambiare il valore di questa variabile, ovvero, se dichiarata, è bene confermarlo.
AR ARFLAGS	ar rw	Il programma di archiviazione usato per creare le librerie statiche e le sue opzioni consuete.
LD LDFLAGS	ld	Il programma usato per collegare i file-oggetto in un file eseguibile e le sue opzioni consuete. Di norma non sono previste opzioni particolari.
AS ASFLAGS	as	Il programma usato per compilare un file in linguaggio assembleatore e le sue opzioni consuete. Di norma non sono previste opzioni particolari.
CPP CPPFLAGS	cpp	Il precompilatore e le sue opzioni consuete. Di norma non sono previste opzioni particolari.

Nome	Contenuto usuale	Annotazioni
CC CFLAGS	cc	Il programma usato per compilare un file in linguaggio C e le sue opzioni consuete. Di norma non sono previste opzioni particolari.
PC PFLAGS	pc	Il programma usato per compilare un file in linguaggio Pascal e le sue opzioni consuete. Di norma non sono previste opzioni particolari.

### 65.8.6 Utilizzo oculato delle variabili

«

Dal momento che le variabili possono essere espansive in ogni posizione di uno script di Make, le definizioni ripetitive possono essere semplificate. Nell'esempio successivo si dichiara la variabile `'obj'`, contenente l'elenco dei file-oggetto coinvolti nella produzione di un certo file eseguibile:

```
obj = aaa.o bbb.o ccc.o ddd.o \  
      eee.o fff.o ggg.o  
...  
prog: $(obj)  
      ld -o prog $(obj)  
...
```

Generalmente, i nomi delle variabili sono scritti utilizzando solo lettere maiuscole, ma non c'è un obbligo in tal senso. Di solito, l'utilizzo di lettere maiuscole per le variabili vuole indicare la possibilità di modificarne il contenuto per qualunque adattamento possa essere necessario; per questo, se invece si utilizzano nomi di variabili con

lettere minuscole (come nell'esempio mostrato), di solito lo si fa per quelle cose che è bene non modificare.

Come già accennato, GNU Make eredita le variabili di ambiente come proprie variabili-macro, anche se poi queste possono essere ridefinite nello script. In ogni caso, il modo «normale» di assegnare un valore a una variabile, nel momento dell'avvio del programma **'make'**, è quello di usare la riga di comando, per esempio, così:

```
$ make "CFLAGS = -O" "LDFLAGS = -s" obiettivo [Invio]
```

Si supponga di avere uno script come quello seguente e si osservi cosa succede con il comando appena mostrato:

```
CFLAGS = -gstabs
LDFLAGS = -S

all:
    @echo "CFLAGS: $(CFLAGS) "
    @echo "LDFLAGS: $(LDFLAGS) "
```

```
$ make "CFLAGS = -O" "LDFLAGS = -s" obiettivo [Invio]
```

```
CFLAGS:  -O
LDFLAGS: -s
```

Pertanto, questo modo di passare il valore alle variabili di Make prevale sulla dichiarazione interna di uno script.

### 65.8.7 Espansione e continuazione al di fuori dei comandi

Il testo di uno script di Make, quando non costituisce un comando da passare alla shell e non si tratta nemmeno di un commento, può essere espanso, sia a causa dell'uso di variabili, sia per la presenza di



caratteri che si espandono in nomi di file, come si fa comunemente per le shell POSIX (quindi si possono usare l'asterisco, il punto interrogativo e le parentesi quadre, con lo stesso significato che hanno per una shell POSIX e si possono anche proteggere i simboli, contro l'espansione, facendoli precedere da una barra obliqua inversa: '\'). Inoltre, è possibile continuare il testo su più righe, usando il simbolo '\ ' alla fine della riga che deve continuare. L'esempio che appare sotto serve a mostrare l'effetto dell'espansione, ma non è un modello da seguire, perché in pratica si creerebbero delle complicazioni:

```
prog: *.o
      $(LD) $(LDFLAGS) -o prog *.o
```

In questo caso, la realizzazione del file 'prog' dipende da tutti i file-oggetto presenti nella directory corrente. L'esempio non è utile in generale, perché se tali file sono assenti viene meno la realizzazione dell'obiettivo. È comunque interessante osservare che l'espansione di '\*.o' nell'elenco delle dipendenze avviene per opera di Make, mentre ciò che appare nel comando viene espanso dalla shell.

### 65.8.8 Variabili automatiche



Alcune variabili non possono essere dichiarate e nemmeno modificate nel loro contenuto. Si tratta delle *variabili automatiche*, composte da un solo carattere. Per esempio, la variabile '@' rappresenta l'obiettivo attuale, ma per espandere il suo contenuto è sufficiente scrivere '\$@', senza bisogno di parentesi.

Per espandere una variabile si possono sempre evitare le parentesi se il nome di questa è composto da un solo carattere; tuttavia, si preferisce rinunciare alle parentesi solo quando si tratta precisamente di variabili automatiche.

Tabella 65.132. Alcune variabili automatiche.

Variabile automatica	Significato
*	Il nome dell'obiettivo attuale, ma senza suffisso; se l'obiettivo non ha suffisso, la variabile risulta vuota.
@	L'obiettivo attuale, completo.
<	La voce che costituisce la prima dipendenza (quella più a sinistra).
?	L'elenco delle dipendenze associate a file che sono più recenti di quello che rappresenta l'obiettivo.
^	L'elenco di tutte le dipendenze previste.
\$	Si espande semplicemente nel simbolo '\$' e, come per tutte le variabili automatiche, va usato aggiungendo un altro dollaro: '\$\$'. In pratica, nei comandi, le variabili di ambiente vanno annotate raddoppiando il simbolo dollaro; per esempio: '\$\$HOME'.

Per comprendere meglio il significato della descrizione fatta nella tabella precedente, si consideri di disporre dei file seguenti: 'uno.c', 'due.c', 'somma.o'. Inoltre, si suppone che solo 'due.c' abbia una data di modifica successiva a quella di 'somma.o'. A tale proposito, si consideri lo script seguente:

```
somma.o: uno.c due.c
    @echo \$$\* = \$*
    @echo \$$@ = @$
    @echo \$$\< = $<
    @echo \$$\? = $?
    @echo \$$\^ = $^
```

Se viene avviato, si può leggere lo stato delle variabili automatiche:

```
$ make [Invio]
```

```
$* = somma
$@ = somma.o
$< = uno.c
$? = due.c
$^ = uno.c due.c
```

Come si può vedere, in questo caso la variabile automatica ‘?’ consentirebbe di individuare le dipendenze per le quali si richiede una nuova compilazione.

È importante notare che le variabili automatiche possono essere usate solo all’interno di comandi, perché il loro contenuto si definisce dopo la dichiarazione dell’obiettivo e delle sue dipendenze.

## 65.8.9 Regole implicite



Le regole implicite sono quelle che descrivono degli obiettivi predefiniti, nel modo più logico possibile. Come già accennato altrove, queste regole definiscono i comandi attraverso delle variabili che è possibile controllare.

Tabella 65.135. Comandi comuni di regole implicite.

Comando	Condizione di utilizzo
<pre>\$(CC) -c \$(CPPFLAGS) \$(CFLAGS) ↵ ↪ nome.c</pre>	Se si richiede la realizzazione dell'obiettivo denominato <i>nome.o</i> ed esiste il file ' <i>nome.c</i> '.
<pre>\$(CXX) -c \$(CPPFLAGS) \$(CXXFLAGS) ↵ ↪ nome.cc \$(CXX) -c \$(CPPFLAGS) \$(CXXFLAGS) ↵ ↪ nome.cpp</pre>	Se si richiede la realizzazione dell'obiettivo denominato <i>nome.o</i> ed esiste il file ' <i>nome.cc</i> ' o ' <i>nome.cpp</i> '.
<pre>\$(AS) \$(ASFLAGS) nome.s</pre>	Se si richiede la realizzazione dell'obiettivo denominato <i>nome.o</i> ed esiste il file ' <i>nome.s</i> '. Se il file ' <i>nome.s</i> ' è assente ma al suo posto esiste ' <i>nome.s</i> ', allora il primo viene generato dal comando successivo.
<pre>\$(CPP) \$(CPPFLAGS) nome.S</pre>	Se si richiede la realizzazione dell'obiettivo denominato <i>nome.s</i> ed esiste il file ' <i>nome.S</i> '.
<pre>\$(CC) \$(LDFLAGS) nome.o ↵ ↪ \$(LOADLIBES) \$(LDLIBS)</pre>	Se si richiede la realizzazione dell'obiettivo denominato <i>nome</i> ed esiste il file ' <i>nome.o</i> '.

Naturalmente, le regole implicite si concatenano tra di loro. Per esempio, si suppone di disporre del file '*prova.c*' e di volerlo compilare utilizzando Make nel modo seguente:

```
$ make prova [Invio]
```

Si sta facendo riferimento all'obiettivo **'prova'** che si intende non sia stato dichiarato nello script di Make. Pertanto, per realizzare questo obiettivo, Make deve cercare una regola implicita appropriata e in questo caso è quella che serve a collegare un file oggetto `'prova.o'`:

```
prova: prova.o
      $(CC) $(LDFLAGS) prova.o $(LOADLIBES) $(LDLIBS)
```

Questa regola implicita, evidentemente, dipende da un'altra regola che descrive in che modo viene ottenuto il file `'prova.o'`. Dal momento che Make trova il file `'prova.c'`, la regola è questa:

```
prova.o: prova.c
      $(CC) -c $(CPPFLAGS) $(CFLAGS) prova.c
```

Di conseguenza viene eseguita la compilazione.

## 65.8.10 Uno script per ogni sottodirectory

«

Di solito si predispose uno script di Make per ogni sottodirectory che contenga qualcosa da costruire; poi, in una o in alcune directory si colloca uno script realizzato in modo da avviare lo stesso programma **'make'** nelle sottodirectory inferiori.

A titolo di esempio, si suppone di avere un progetto suddiviso in tre sottodirectory: `'mele/'`, `'arance/'` e `'limoni/'`. All'interno di ogni sottodirectory c'è un file `'Makefile'`. Nella directory che contiene queste sottodirectory c'è un file `'Makefile'` con una regola per avviare sequenzialmente gli altri file equivalenti delle sottodirectory:



```
sub = mele arance limoni

all:
    for d in $(sub) ; do cd $$d ; $(MAKE) ; cd .. ; done
```

Viene usato un ciclo per la scansione delle sottodirectory che la shell interpreta così:

```
for d in mele arance limoni
do
    cd $d
    make
    cd ..
done
```

Ogni volta che si usa Make in questo modo, si dovrebbe vedere un avvertimento come quello seguente:

```
make[1]: Entering directory `/home/tizio/mele'
...
make[1]: Leaving directory `/home/tizio/mele'
make[1]: Entering directory `/home/tizio/arance'
...
make[1]: Leaving directory `/home/tizio/arance'
make[1]: Entering directory `/home/tizio/limoni'
...
make[1]: Leaving directory `/home/tizio/limoni'
```

Per lasciare a Make il controllo del ciclo di avvii nelle sottodirectory, si può usare un meccanismo differente, come quello che si vede nel listato successivo:

```
sub = mele arance limoni

all: $(sub)

$(sub): FORCE
      cd $@ && $(MAKE)

FORCE:
```

Si può vedere che l'obiettivo **'all'** (evidentemente un obiettivo fittizio), dipende dai nomi delle sottodirectory. Successivamente è dichiarata una regola con obiettivo multiplo, ovvero una regola che vale indifferentemente per i tre obiettivi di ogni sottodirectory (la variabile automatica **'\$@'** si espande nel nome dell'obiettivo preso in considerazione effettivamente). Tale regola dipende però da un altro obiettivo, senza dipendenze e senza comandi, per il quale si è certi che non possa esistere un file con lo stesso nome.

Come mostrato in questi esempi, invece di scrivere il nome del programma eseguibile **'make'**, è stata usata la variabile **'MAKE'**, la quale riproduce il nome del comando usato per avviare l'interpretazione dello script. Per esempio, se per qualunque motivo il programma **'make'** fosse nominato in maniera differente o fosse usato al di fuori dei percorsi di ricerca per gli eseguibili, con la variabile **'MAKE'** si garantisce sempre di trovare lo stesso programma che risulta già in funzione.

### 65.8.11 Una regola per più obiettivi



La sezione precedente introduce il concetto di obiettivo multiplo, che però può risultare difficile da intendere con l'esempio mostrato, dal momento che si utilizza una variabile per esprimere l'elenco di

obiettivi. Lo stesso esempio può essere tradotto così, senza l'uso di variabili:

```
all: mele arance limoni

mele arance limoni: FORCE
    cd @$ && $(MAKE)

FORCE:
```

Senza usare una regola del genere, occorrerebbe suddividere la stessa in tre (una per ogni singolo obiettivo):

```
...
mele: FORCE
    cd mele && $(MAKE)

arance: FORCE
    cd arance && $(MAKE)

limoni: FORCE
    cd limoni && $(MAKE)
...
```

Questo dovrebbe chiarire anche l'utilità della variabile automatica '\$@', per individuare l'obiettivo preso effettivamente in considerazione in un dato momento.

## 65.8.12 Regole fittizie tipiche

Generalmente si organizza uno script di Make in modo da avere alcuni obiettivi fittizi, con cui eseguire le operazioni più comuni in modo complessivo. Tra questi, l'obiettivo più importante in assoluto è quello che deve essere eseguito in modo predefinito (ovvero il



primo) e generalmente viene chiamato ‘**all**’. Tale obiettivo serve di norma per indicare soltanto delle dipendenze da soddisfare.

Tabella 65.144. Obiettivi fittizi comuni.

Obiettivo	Significato comune
<code>all</code>	Le azioni da compiere quando non si indica alcun obiettivo in modo esplicito.
<code>clean</code>	I comandi da eseguire per cancellare i file oggetto, i binari già compilati ed eventualmente altri file temporanei.
<code>install</code>	I comandi necessari a installare i programmi dopo la compilazione.

Stando alla tabella appena mostrata, si può ricordare che le fasi tipiche di un’installazione di un programma distribuito in forma sorgente sono appunto quelle seguenti:

1. # **make** [*Invio*]

Richiama automaticamente l’obiettivo ‘**all**’, coincidente con i comandi necessari per la compilazione del programma.

2. # **make install** [*Invio*]

Provvede a installare gli eseguibili compilati nella loro destinazione prevista.

Supponendo di avere realizzato un programma, denominato ‘`mio_prog.c`’, il cui eseguibile debba essere installato nella directory ‘`/usr/local/bin/`’, si potrebbe utilizzare uno script composto come l’esempio seguente, dove l’obiettivo ‘**all**’ richiama la dipendenza dal programma ‘**mio\_prog**’ che viene soddisfatta in modo implicito:

```

all: mio_prog

clean:
    rm -f core *.o mio_prog

install:
    cp mio_prog /usr/local/bin

```

### 65.8.13 Variabili per l'installazione

In uno script di Make realizzato per la compilazione di un programma (composto eventualmente da uno o più file eseguibili) e per la sua installazione successiva, sono presenti normalmente alcune variabili, più o meno standardizzate, per descrivere la collocazione finale dei file. Alcune di queste variabili sono elencate nella tabella successiva.

Tabella 65.146. Alcune variabili usate comunemente per definire l'installazione.

Variabile	Utilizzo
<code>prefix</code>	La variabile ' <b>prefix</b> ' viene usata normalmente come punto di partenza da cui traggono origine altre variabili più specifiche. Di solito, il valore dato a questa variabile per la distribuzione di un pacchetto sorgente è '/usr/local/', dove poi chi compila e installa deve attribuire un valore che per sé possa essere più appropriato.
<code>exec_prefix</code>	La variabile ' <b>exec_prefix</b> ' rappresenta il punto di riferimento iniziale per l'installazione dei file eseguibili e di solito corrisponde al contenuto di ' <b>prefix</b> '.

Variabile	Utilizzo
<code>bindir</code>	Rappresenta la directory in cui vanno installati i file eseguibili a disposizione di tutti gli utenti e corrisponde normalmente alla directory <code>'bin/'</code> successiva al contenuto di <code>'exec_prefix'</code> . Pertanto, se <code>'prefix'</code> e <code>'exec_prefix'</code> indicano <code>'/usr/local/'</code> , <code>'bindir'</code> indica normalmente <code>'/usr/local/bin/'</code> .
<code>sbindir</code>	Rappresenta la directory in cui vanno installati i file eseguibili utili per l'amministrazione e corrisponde normalmente alla directory <code>'sbin/'</code> successiva al contenuto di <code>'exec_prefix'</code> .

Uno script che usa queste variabili potrebbe essere realizzato così:

```
prefix = /usr/local
exec_prefix = $(prefix)
bindir=$(exec_prefix)/bin

all: mio_prog

clean:
    rm -f core *.o mio_prog

install:
    cp mio_prog $(bindir)
```

## 65.8.14 Definizione della shell



In linea di principio, la shell usata per eseguire i comandi contenuti nelle regole di uno script di Make è `'/bin/sh'`. Tuttavia, il nome e il percorso esatto possono essere controllati attraverso la variabile `'SHELL'`. In generale può essere conveniente aggiungere nello script la dichiarazione seguente, in modo da non avere sorprese:

```
SHELL = /bin/sh
```

Per esempio, utilizzando GNU Make occorre considerare che le variabili di ambiente sono ereditate e la presenza eventuale di una variabile ‘**SHELL**’ potrebbe creare problemi: è per questo che la dichiarazione suggerita può essere conveniente.

Evidentemente, nello stesso modo descritto è possibile cambiare la shell che interpreta i comandi, ma ciò è sicuramente sconsigliabile, nell’ottica della creazione di script «standard».

### 65.8.15 Installazione dei programmi

È il caso di osservare che, normalmente, l’installazione dei programmi, ovvero la loro copia nella destinazione finale, dopo la compilazione, si esegue con il programma ‘**install**’ (eventualmente si veda la sezione [20.11.7](#)). Il motivo di questa scelta sta normalmente nella facilità con cui, assieme alla copia, si definiscono i permessi e la proprietà del file nella destinazione. Lo script seguente riprende gli ultimi esempi e concetti già visti:

```
SHELL = /bin/sh
prefix = /usr/local
exec_prefix = $(prefix)
bindir=$(exec_prefix)/bin

all: mio_prog

clean:
    rm -f core *.o mio_prog

install:
```

```
install -o root -g bin -m 755 mio_prog $(bindir)
```

## 65.9 Riferimenti

«

- *YoLinux Tutorial - Static, Shared Dynamic and Loadable Linux Libraries*, <http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html>
- Steve Chamberlain, Ian Lance Taylor, *Using LD, the GNU linker*, <http://www.zemris.fer.hr/~leonardo/oszur/tehnicki.dokumenti/gnu-linker.pdf>
- *a.out (file format)*, [http://en.wikipedia.org/wiki/A.out\\_\(file\\_format\)](http://en.wikipedia.org/wiki/A.out_(file_format))
- *COFF*, <http://en.wikipedia.org/wiki/COFF>
- *Portable Executable*, [http://en.wikipedia.org/wiki/Portable\\_Executable](http://en.wikipedia.org/wiki/Portable_Executable)
- *DWARF*, <http://en.wikipedia.org/wiki/DWARF>
- *Mach-O*, <http://en.wikipedia.org/wiki/Mach-O>
- *Executable and linkable format*  
[http://en.wikipedia.org/wiki/Executable\\_and\\_Linkable\\_Format](http://en.wikipedia.org/wiki/Executable_and_Linkable_Format)
- *format of executable and linking format (ELF) files*, pagina di manuale *elf(5)*
- Brian Raiter, *A Whirlwind Tutorial on Creating Really Teeny ELF Executables for Linux*, <http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html>



- *Multiboot specification*, <http://www.gnu.org/software/grub/manual/multiboot/>
- osdev.org wiki, *Tutorial: bare bones*, [http://wiki.osdev.org/Tutorial:Bare\\_bones](http://wiki.osdev.org/Tutorial:Bare_bones)
- Gergor Brunmar: *The booting process*, *The world of Protected mode*, *Mixing Assembly and C-code*, [http://www.osdever.net/tutorials/pdf/gb\\_booting.pdf](http://www.osdever.net/tutorials/pdf/gb_booting.pdf), [http://www.osdever.net/tutorials/pdf/gb\\_pmode.pdf](http://www.osdever.net/tutorials/pdf/gb_pmode.pdf), [http://www.osdever.net/tutorials/pdf/gb\\_asm\\_and\\_c.pdf](http://www.osdever.net/tutorials/pdf/gb_asm_and_c.pdf)
- mr. xsism, *Xosdev*, <http://www.osdever.net/tutorials/pdf/ch01.pdf>, <http://www.osdever.net/tutorials/pdf/ch02.pdf>
- Tim Robinson, *Writing a Kernel in C*, <http://www.osdever.net/tutorials/pdf/ckernel.pdf>
- Joachim Nock, K.J., *Making a Simple C kernel with Basic printf and clearscreen Functions*, <http://www.osdever.net/tutorials/view/writing-a-simple-c-kernel>
- OSDevWiki, *Category: Tutorials*, <http://wiki.osdev.org/Category:Tutorials>
- OSDevWiki, *Category: Babystep*, <http://wiki.osdev.org/Category:Babystep>
- Daniel Rowell Faulkner, *Hello World Boot Loader*, <http://www.osdever.net/tutorials/view/hello-world-boot-loader>
- OSRC, *the Operating System resource center*, <http://www.nondot.org/sabre/os/articles>
- SigOps, *How to Write an Operating System*, [http://www.acm.uiuc.edu/sigops/roll\\_your\\_own/](http://www.acm.uiuc.edu/sigops/roll_your_own/), <http://www.acm.uiuc.edu/>

[sigops/roll\\_your\\_own/hardware/kb.html](http://www.acm.uiuc.edu/sigops/roll_your_own/hardware/kb.html) , [http://www.acm.uiuc.edu/sigops/roll\\_your\\_own/hardware/text.html](http://www.acm.uiuc.edu/sigops/roll_your_own/hardware/text.html)

- *GNU Compiler Collection*, [http://en.wikipedia.org/wiki/GNU\\_Compiler\\_Collection](http://en.wikipedia.org/wiki/GNU_Compiler_Collection)
- *Using the GNU compiler collection (GCC)*, <http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/> , <http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc.pdf>
- Brian Gough, *An Introduction to GCC*, <http://www.network-theory.co.uk/docs/gccintro/>
- Richard M. Stallman, Roland McGrath and Paul D. Smith, *GNU Make: A Program for Directing Recompilation*, ISBN 1-882114-83-3, <http://www.gnu.org/software/make/manual/>

## <sup>1</sup> **GNU Binutils** GNU GPL

<sup>2</sup> Sono pochi i sistemi operativi affermati, mentre esistono una miriade di progetti più o meno abbandonati; quindi: prima di pensare di scrivere il proprio sistema converrebbe dare un'occhiata a quanti lavori del genere sono stati catalogati.

## <sup>3</sup> **GCC** GNU GPL

<sup>4</sup> Di norma, il comando `'ulimit'` è gestito internamente dalla shell; in questo esempio si fa riferimento a una shell POSIX.

<sup>5</sup> Lo script di Make potrebbe essere nominato anche in altri modi, per esempio senza l'iniziale maiuscola (quindi solo `'makefile'`) ma in generale conviene attenersi al suggerimento di usare il nome `'Makefile'` che, in un elenco ordinato per nome dei file di una directory, ha il vantaggio di apparire prima di altri a causa dell'iniziale maiuscola.

- <sup>6</sup> Nel caso di GNU Make esiste una direttiva apposita per dichiarare quali obiettivi sono fittizi (*phony*).
- <sup>7</sup> GNU Make offre un meccanismo più accurato per impedire che un obiettivo fittizio sia bloccato da un file, ma il metodo mostrato è valido in generale.
- <sup>8</sup> Nel caso di GNU Make, nella ricerca del primo obiettivo si escludono i nomi che iniziano con un punto, dal momento che a quelli viene dato un significato particolare.
- <sup>9</sup> Di solito il fallimento di un obiettivo può comportare la cancellazione contestuale del file corrispondente all'obiettivo mancato, anche se si tratta di una versione precedente.



## Introduzione al linguaggio C



66.1	Nozioni minime .....	563
66.1.1	Struttura fondamentale .....	564
66.1.2	Ciao mondo! .....	566
66.1.3	Variabili e tipi .....	569
66.1.4	Operatori ed espressioni 586	
66.1.5	Strutture di controllo di flusso .....	597
66.1.6	Funzioni .....	607
66.1.7	Vincoli nei nomi .....	611
66.1.8	I/O elementare .....	612
66.1.9	Restituzione di un valore .....	615
66.1.10	Attributi per GNU C .....	617
66.2	Istruzioni del precompilatore .....	619
66.2.1	Linguaggio a sé stante .....	620
66.2.2	Direttiva «#include» .....	621
66.2.3	Direttiva «#define» .....	623
66.2.4	Direttiva «#define» con parametri .....	627
66.2.5	Direttive «#if», «#else», «#elif» e «#endif» ..	634
66.2.6	Direttive «#if defined», «#if !defined», «#ifdef» e «#ifndef» .....	636
66.2.7	Direttiva «#undef» .....	640
66.2.8	Direttiva «#line» .....	640
66.2.9	Direttiva «#error» .....	645

66.2.10	Macro predefinite	646
66.2.11	Pragma	649
66.3	Dal campo di azione alla compilazione	650
66.3.1	Il punto di vista del «collegatore»	650
66.3.2	Campo di azione legato al file sorgente	651
66.3.3	Semplificazione dovuta all'uso comune dei file di intestazione	655
66.3.4	Campo di azione interno alle funzioni	657
66.3.5	Campo di azione interno ai raggruppamenti di istruzioni	661
66.3.6	Funzioni annidate	662
66.3.7	Visibilità, accessibilità, staticità	663
66.3.8	Compilazione di un progetto composto da più file	666
66.3.9	Osservazioni sulla vita delle costanti letterali	668
66.3.10	Libreria standard e file di intestazione	669
66.4	Annotazioni sulla terminologia	670
66.4.1	Parametri e argomenti	670
66.4.2	Byte e caratteri	671
66.4.3	Unità di traduzione	672
66.4.4	«Linkage»	672
66.4.5	Durata di memorizzazione	673
66.4.6	«Lvalue» e «rvalue»	674
66.4.7	«Digraph» e «Trigraph»	676
66.5	Puntatori, array, stringhe e allocazione dinamica della memoria	677

66.5.1	Espressioni a cui si assegnano dei valori	678
66.5.2	Puntatori	679
66.5.3	Array	687
66.5.4	Array multidimensionali	693
66.5.5	Natura dell'array	696
66.5.6	Puntatori costanti	701
66.5.7	Array e funzioni	702
66.5.8	Aritmetica dei puntatori	704
66.5.9	Osservazioni sui puntatori	709
66.5.10	Stringhe	710
66.5.11	Parametri della funzione main()	720
66.5.12	Puntatori a puntatori	723
66.5.13	Puntatori a più dimensioni	725
66.5.14	Puntatori e funzioni	730
66.5.15	Puntatori a variabili distrutte	734
66.5.16	Puntatore nullo	735
66.5.17	Utilizzo della memoria in modo dinamico	737
66.5.18	Puntatori «ristretti»	740
66.6	Le funzioni	743
66.6.1	Pila dei dati	743
66.6.2	Dichiarazione e chiamata di una funzione	745
66.6.3	Elenco indefinito di parametri	746
66.6.4	Annotazioni su «printf()» e altre funzioni simili	753
66.6.5	Costante predefinita «__func__»	754

66.7	Struttura, unione, campo, enumerazione, costante composta	755
66.7.1	Enumerazioni	755
66.7.2	Strutture	759
66.7.3	Assegnamento, inizializzazione, campo di azione e puntatori delle strutture	761
66.7.4	Scostamento all'interno delle strutture	767
66.7.5	Unioni	768
66.7.6	Campi	770
66.7.7	Istruzione «typedef»	772
66.7.8	Costanti letterali composte	773
66.8	Tipi di dati speciali, di uso comune	777
66.8.1	Tipo «_Bool»	778
66.8.2	Tipo «void»	779
66.8.3	Tipo «size_t»	781
66.8.4	Tipo «ptrdiff_t»	783
66.8.5	Tipo «va_list»	784
66.8.6	Tipo «wchar_t»	786
66.8.7	Tipo «wint_t»	787
66.8.8	Tipo «time_t»	787
66.8.9	Tipo «struct tm»	787
66.8.10	Tipo «FILE»	788
66.8.11	Tipo «fpos_t»	788
66.9	Configurazione locale	789



66.9.1	Configurazione locale nei sistemi Unix e simili ...	789									
66.9.2	Configurazione locale nel linguaggio C .....	790									
66.9.3	Caratteri multibyte e caratteri estesi .....	792									
66.9.4	Concatenamento eterogeneo .....	796									
66.9.5	Conversione tra caratteri multibyte e caratteri estesi	796									
66.10	Organizzazione dei file sorgenti .....	800									
66.10.1	File di intestazione .....	801									
66.10.2	Funzioni pubbliche .....	801									
66.10.3	Funzioni e variabili private .....	802									
66.10.4	Esempio di «stdlib.h» .....	802									
66.10.5	Parametri delle macroistruzioni .....	806									
66.10.6	Compilazione .....	807									
66.11	K&R .....	808									
66.11.1	Prototipi e chiamate delle funzioni .....	809									
66.11.2	Dichiarazione delle funzioni .....	811									
66.11.3	Operatori composti di assegnamento .....	812									
66.11.4	Tipi numerici .....	813									
66.11.5	Tipo «void *» .....	814									
66.11.6	Direttive del preprocessore .....	814									
66.11.7	Altre osservazioni su K&R .....	814									
66.11.8	Unproto .....	815									
66.12	Riferimenti .....	816									
!	586 590	!= 586 590	*	586 588 679	**	723 725	***	723			
*...const	701	*=	586 588	*&	709	+	586 588	++	586 588	+=	586

588 . 759 761 / 586 588 /\*...\*/ 564 // 564 /= 586 588 0... 575  
 0x... 575 ; 564 = 586 588 == 586 590 ? : 586 590 argc 720  
 argv 720 auto 657 bool 778 break 599 602 604 calloc()  
 737 case 599 *cast* 594 char 571 const 583 584 const...\* 701  
 const volatile 584 continue 602 604 cpp 619 default  
 599 do 603 double 571 else 598 enum 755 exit() 615  
 extern 651 657 extern const volatile 584 F 575 FILE  
 788 float 571 for 604 fpos\_t 788 free() 737 if 598 int  
 571 L 575 575 LL 575 locale.h 790 long 571 long long  
 571 L"... " 792 L'...' 792 main() 720 malloc() 737 NULL  
 735 offsetof 767 printf() 568 753 **prototipo di funzione**  
 608 ptrdiff\_t 783 realloc() 737 register 657  
 restrict 740 return 609 short 571 signed 571 sizeof  
 687 size\_t 781 static 651 657 stdarg.h 746 stdlib.h  
 737 strcat() 714 strchr() 714 strcmp() 714  
 strcoll() 714 strcpy() 714 strcspn() 714 string.h  
 714 strlen() 714 strncat() 714 strncmp() 714  
 strncpy() 714 strpbrk() 714 strrchr() 714 strspn()  
 714 struct 759 struct tm 787 switch 599 time\_t 787  
 typedef 772 U 575 UL 575 ULL 575 union 768 unsigned  
 571 va\_arg 746 va\_end 746 va\_list 746 784 va\_start  
 746 void 586 608 779 volatile 584 wchar\_t 786 792  
 while 602 wint\_t 787 # 564 #define 623 #define() 627  
 #define()...# 627 #define()...## 627  
 #define()...\_\_VA\_ARGS\_\_ 627 #define...## 623 #elif  
 634 #else 634 #endif 634 #error 645 #if 634 #ifdef 636  
 #ifndef 636 #if !defined 636 #if defined 636  
 #include 621 #line 640 #pragma 649 #undef 640 & 586  
 592 679 &\* 709 &= 586 592 && 586 590 ^ 586 592 ^= 586 592 ~

586 592 ~ = 586 592 \... 575 \0 575 \? 575 \a 575 \b 575 \f  
 575 \n 575 \r 575 \t 575 \v 575 \x... 575 \" 575 \\ 575 \  
 575 | 586 592 |= 586 592 || 586 590 {...} 564 \_Bool 778  
 \_\_Pragma 649 \_\_DATE\_\_ 646 \_\_FILE\_\_ 646 \_\_func\_\_ 754  
 \_\_LINE\_\_ 646 \_\_STDC\_HOSTED\_\_ 646  
 \_\_STDC\_IEC\_559\_\_ 646 \_\_STDC\_IEC\_COMPLEX\_\_ 646  
 \_\_STDC\_ISO\_10646\_\_ 646 \_\_STDC\_VERSION\_\_ 646  
 \_\_STDC\_\_ 646 \_\_TIME\_\_ 646 \_\_VA\_ARGS\_\_ 627 '...' 575 ,  
 596 - 586 588 -= 586 588 -- 586 588 -> 761 < 586 590 <= 586  
 590 << 586 592 <<= 586 592 > 586 590 >= 586 590 >> 586 592  
 >>= 586 592 % 586 588 %= 586 588

## 66.1 Nozioni minime

Il linguaggio C è il fondamento dei sistemi Unix. Un minimo di conoscenza di questo linguaggio è importante per districarsi tra i programmi distribuiti in forma sorgente, pur senza volerli modificare.

Il linguaggio C richiede la presenza di un compilatore per generare un file eseguibile (o interpretabile) dal kernel. Se si dispone di un sistema GNU con i cosiddetti «strumenti di sviluppo», intendendo con questo ciò che serve a ricompilare il kernel, si dovrebbe disporre di tutto quello che è necessario per provare gli esempi di questi capitoli.

Il linguaggio C standard deve disporre di una libreria applicabile a ogni tipo di architettura e di sistema operativo; pertanto le funzionalità di tale libreria è molto limitata. In questi capitoli dedicati al linguaggio C, quando si vuole fare riferimento a funzioni che sono definite al di fuori dello standard minimo, ciò viene annota-

to espressamente; in particolare, nel caso di estensioni che riguardano lo standard POSIX, può apparire anche una nota a margine simbolica.

### 66.1.1 Struttura fondamentale

«

Il contenuto di un sorgente in linguaggio C può essere suddiviso in tre parti: commenti, direttive del precompilatore e istruzioni C. I commenti vanno aperti e chiusi attraverso l'uso dei simboli `/*` e `*/`; se poi il compilatore è conforme a standard più recenti, è ammissibile anche l'uso di `//` per introdurre un commento che termina alla fine della riga.

```
/* Questo è un commento che continua  
   su più righe e finisce qui. */
```

```
// Qui inizia un altro commento che termina alla fine della  
// riga; pertanto, per ogni riga va ripetuta la sequenza  
// "/" di apertura.
```

Le direttive del precompilatore rappresentano un linguaggio che guida alla compilazione del codice vero e proprio. L'uso più comune di queste direttive viene fatto per includere porzioni di codice sorgente esterne al file. È importante fare attenzione a non confondersi, dal momento che tali istruzioni iniziano con il simbolo `#`: non si tratta di commenti.

Il programma C tipico richiede l'inclusione di codice esterno composto da file che terminano con l'estensione `.h`. La porzione di libreria che viene utilizzata più frequentemente è quella necessaria alla gestione dei flussi di standard input, standard output e standard error; si dichiara implicitamente il suo utilizzo includendo il file di intestazione `stdio.h` nel modo seguente:

```
#include <stdio.h>
```

Le istruzioni C terminano con un punto e virgola (‘;’) e i raggruppamenti di queste (noti come «istruzioni composte») si fanno utilizzando le parentesi graffe (‘{ }’).<sup>1</sup>

```
istruzione ;
```

```
{ istruzione ; istruzione ; istruzione ; }
```

Generalmente, un’istruzione può essere interrotta e ripresa nella riga successiva, dal momento che la sua conclusione è dichiarata chiaramente dal punto e virgola finale. L’istruzione nulla viene rappresentata utilizzando un punto e virgola da solo.

I nomi scelti per identificare ciò che si utilizza all’interno del programma devono seguire regole determinate, definite dal compilatore C a disposizione. Ma per cercare di scrivere codice portabile in altre piattaforme, conviene evitare di sfruttare caratteristiche speciali del proprio ambiente. In particolare:

- un nome può iniziare con una lettera alfabetica e continuare con altre lettere, cifre numeriche e il trattino basso;
- i nomi potrebbero iniziare anche con il trattino basso, ma ciò è preferibile evitarlo, se non ci sono motivi validi per questo;<sup>2</sup>
- nei nomi si distinguono le lettere minuscole da quelle maiuscole (pertanto, ‘**Nome**’ è diverso da ‘**nome**’ e da tante altre combinazioni di minuscole e maiuscole).

La lunghezza dei nomi può essere un elemento critico; generalmente la dimensione massima dovrebbe essere di 32 caratteri, ma ci sono versioni di C che ne possono accettare solo una quantità inferiore. In particolare, il compilatore GNU ne accetta molti di più di 32. In ogni caso, il compilatore non rifiuta i nomi troppo lunghi, semplicemente non ne distingue più la differenza oltre un certo punto.

Il codice di un programma C è scomposto in funzioni, dove normalmente l'esecuzione del programma corrisponde alla chiamata della funzione *main()*. Questa funzione può essere dichiarata senza parametri, `'int main (void)'`, oppure con due parametri precisi: `'int main (int argc, char *argv[])'`.<sup>3</sup>

### 66.1.2 Ciao mondo!

«

Come sempre, il modo migliore per introdurre a un linguaggio di programmazione è di proporre un esempio banale, ma funzionante. Al solito si tratta del programma che emette un messaggio e poi termina la sua esecuzione.

```
/*
 *      Ciao mondo!
 */

#include <stdio.h>

/* La funzione main() viene eseguita automaticamente
   all'avvio. */
int main (void)
{
    /* Si limita a emettere un messaggio. */
    printf ("Ciao mondo!\n");
}
```

Nel programma sono state inserite alcune righe di commento. In particolare, all'inizio, l'asterisco che si trova nella seconda riga ha soltanto un significato estetico, per guidare la vista verso la conclusione del commento stesso.

Il programma si limita a emettere la stringa «Ciao Mondo!» seguita da un codice di interruzione di riga, rappresentato dal simbolo '\n'.

### 66.1.2.1 Compilazione

Per compilare un programma scritto in C si utilizza generalmente il comando 'cc', anche se di solito si tratta di un collegamento simbolico al vero compilatore che si ha a disposizione. Supponendo di avere salvato il file dell'esempio con il nome 'ciao.c', il comando per la sua compilazione è il seguente:

```
$ cc ciao.c [Invio]
```

Quello che si ottiene è il file 'a.out' che dovrebbe già avere i permessi di esecuzione.

```
$ ./a.out [Invio]
```

```
Ciao mondo!
```

Se si desidera compilare il programma definendo un nome diverso per il codice eseguibile finale, si può utilizzare l'opzione standard '-o'.

```
$ cc -o ciao ciao.c [Invio]
```

Con questo comando, si ottiene l'eseguibile 'ciao'.

```
$ ./ciao [Invio]
```

```
Ciao mondo!
```

In generale, se ciò è possibile, conviene chiedere al compilatore di mostrare gli avvertimenti (*warning*), senza limitarsi ai soli errori. Pertanto, nel caso il compilatore sia GNU C, è bene usare l'opzione `'-Wall'`:

```
$ cc -Wall -o ciao ciao.c [Invio]
```

### 66.1.2.2 Emissione dati attraverso «printf()»

«

L'esempio di programma presentato sopra si avvale della funzione *printf()*<sup>4</sup> per emettere il messaggio attraverso lo standard output. Questa funzione è più sofisticata di quanto possa apparire dall'esempio, in quanto permette di comporre il risultato da emettere. Negli esempi più semplici di codice C appare immancabilmente questa funzione, per cui è necessario descrivere subito, almeno in parte, il suo funzionamento.

```
int printf (stringa_di_formato [, espressione] ...);
```

La funzione *printf()* emette attraverso lo standard output la stringa che costituisce il primo parametro, dopo averla rielaborata in base alla presenza di *specificatori di conversione* riferiti alle eventuali espressioni che compongono gli argomenti successivi; inoltre restituisce il numero di caratteri emessi.

L'utilizzo più semplice di *printf()* è quello che è già stato visto, cioè l'emissione di una stringa senza specificatori di conversione (il codice `'\n'` rappresenta un carattere preciso e non è uno specificatore, piuttosto si tratta di una cosiddetta sequenza di escape).

```
printf ("Ciao mondo!\n");
```



La stringa può contenere degli specificatori di conversione del tipo ‘%i’, ‘%c’, ‘%f’,... e questi fanno ordinatamente riferimento agli argomenti successivi. L’esempio seguente fa in modo che la stringa incorpori il valore del secondo argomento nella posizione in cui appare ‘%i’:

```
printf ("Totale fatturato: %i\n", 12345);
```

Lo specificatore di conversione ‘%i’ stabilisce anche che il valore in questione deve essere trasformato secondo una rappresentazione decimale intera. Per cui, il risultato diviene esattamente quello che ci si aspetta.

```
Totale fatturato: 12345
```

### 66.1.3 Variabili e tipi

I tipi di dati elementari gestiti dal linguaggio C dipendono dall’architettura dell’elaboratore sottostante. In questo senso, volendo fare un discorso generale, è difficile definire la dimensione delle variabili numeriche; si possono dare solo delle definizioni relative. Solitamente, il riferimento è dato dal tipo numerico intero (‘**int**’) la cui dimensione in bit corrisponde a quella della *parola*, ovvero dalla capacità dell’unità aritmetico-logica del microprocessore, oppure a qualunque altra entità che il microprocessore sia in grado di gestire con la massima efficienza. In pratica, con l’architettura x86 a 32 bit, la dimensione di un intero normale è di 32 bit, ma rimane la stessa anche con l’architettura x86 a 64 bit.

I documenti che descrivono lo standard del linguaggio C, definiscono la «dimensione» di una variabile come *rango* (*rank*).

### 66.1.3.1 Bit, byte e caratteri

«

A proposito della gestione delle variabili, esistono pochi concetti che sembrano rimanere stabili nel tempo. Il riferimento più importante in assoluto è il byte, che per il linguaggio C è almeno di 8 bit, ma potrebbe essere più grande.<sup>5</sup> Dal punto di vista del linguaggio C, il byte è l'elemento più piccolo che si possa indirizzare nella memoria centrale, questo anche quando la memoria fosse organizzata effettivamente a parole di dimensione maggiore del byte. Per esempio, in un elaboratore che suddivide la memoria in blocchi da 36 bit, si potrebbero avere byte da 9, 12, 18 bit o addirittura 36 bit.<sup>6</sup>

Una volta definito il byte, si considera che il linguaggio C rappresenti ogni variabile scalare come una sequenza continua di byte; pertanto, tutte le variabili scalari sono rappresentate come multipli di byte; di conseguenza anche le variabili strutturate lo sono, con la differenza che in tal caso potrebbero inserirsi dei «buchi» (in byte), dovuti alla necessità di allineare i dati in qualche modo.

Il tipo '**char**' (carattere), indifferentemente se si considera o meno il segno, rappresenta tradizionalmente una variabile numerica che occupa esattamente un byte, pertanto, spesso si confondono i termini «carattere» e «byte», nei documenti che descrivono il linguaggio C.

A causa della capacità limitata che può avere una variabile di tipo '**char**', il linguaggio C distingue tra un insieme di caratteri «minimo» e un insieme «esteso», da rappresentare però in altra forma.

## 66.1.3.2 Tipi primitivi

I tipi di dati primitivi rappresentano un valore **numerico** singolo, nel senso che anche il tipo ‘**char**’ viene trattato come un numero. Il loro elenco essenziale si trova nella tabella 66.9.

Tabella 66.9. Elenco dei tipi comuni di dati primitivi elementari in C.

Tipo	Descrizione
<code>char</code>	Carattere (generalmente di 8 bit).
<code>int</code>	Intero normale.
<code>float</code>	Virgola mobile a precisione singola.
<code>double</code>	Virgola mobile a precisione doppia.

Come già accennato, non si può stabilire in modo generale quali siano le dimensioni esatte in bit dei vari tipi di dati, ovvero il rango, in quanto l’elemento certo è solo la relazione tra loro.

$$\text{char} \leq \text{int} \leq \text{float} \leq \text{double}$$

Questi tipi primitivi possono essere estesi attraverso l’uso di alcuni qualificatori: ‘**short**’, ‘**long**’, ‘**long long**’, ‘**signed**’<sup>7</sup> e ‘**unsigned**’.<sup>8</sup> I primi tre si riferiscono al rango, mentre gli altri modificano il modo di valutare il contenuto di alcune variabili. La tabella 66.11 riassume i vari tipi primitivi con le combinazioni ammissibili dei qualificatori.

Tabella 66.11. Elenco dei tipi comuni di dati primitivi in C assieme ai qualificatori usuali.

Tipo	Abbreviazione	Descrizione
char		Tipo <b>'char'</b> per il quale non conta sapere se il segno viene considerato o meno.
signed char		Tipo <b>'char'</b> usato numericamente con segno.
unsigned char		Tipo <b>'char'</b> usato numericamente senza segno.
short int signed short int	short signed short	Intero più breve di <b>'int'</b> , con segno.
unsigned short int	unsigned short	Tipo <b>'short'</b> senza segno.
int signed int		Intero normale, con segno.
unsigned int	unsigned	Tipo <b>'int'</b> senza segno.
long int signed long int	long signed long	Intero più lungo di <b>'int'</b> , con segno.
unsigned long int	unsigned long	Tipo <b>'long'</b> senza segno.
long long int signed long long int	long long signed long long	Intero più lungo di <b>'long int'</b> , con segno.

Tipo	Abbreviazione	Descrizione
unsigned long long int	unsigned long long	Tipo ‘ <b>long long</b> ’ senza segno.
float		Tipo a virgola mobile a precisione singola.
double		Tipo a virgola mobile a precisione doppia.
long double		Tipo a virgola mobile «più lungo» di ‘ <b>double</b> ’.

Così, il problema di stabilire le relazioni di rango si complica:

$$\text{char} \leq \text{short} \leq \text{int} \leq \text{long} \leq \text{long long}$$

$$\text{float} \leq \text{double} \leq \text{long double}$$

I tipi ‘**long**’ e ‘**float**’ potrebbero avere un rango uguale, altrimenti non è detto quale dei due sia più grande.

Il programma seguente, potrebbe essere utile per determinare il rango dei vari tipi primitivi nella propria piattaforma.<sup>9</sup>

Listato 66.13. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/NxyS6KVy>, <http://ideone.com/uSVC3>.

```
#include <stdio.h>

int main (void)
{
    printf ("char          %i\n", (int) sizeof (char));
    printf ("short int       %i\n", (int) sizeof (short int));
    printf ("int              %i\n", (int) sizeof (int));
    printf ("long int         %i\n", (int) sizeof (long int));
    printf ("long long int    %i\n", (int) sizeof (long long int));
    printf ("float            %i\n", (int) sizeof (float));
    printf ("double           %i\n", (int) sizeof (double));
}
```

```
printf ("long double    %i\n", (int) sizeof (long double));  
return 0;  
}
```

Il risultato potrebbe essere simile a quello seguente:

char	1
short int	2
int	4
long int	4
long long int	8
float	4
double	8
long double	12

I numeri rappresentano la quantità di caratteri, nel senso di valori **'char'**, per cui il tipo **'char'** dovrebbe sempre avere una dimensione unitaria.<sup>10</sup>

I tipi primitivi di variabili mostrati sono tutti utili alla memorizzazione di valori numerici, a vario titolo. A seconda che il valore in questione sia trattato con segno o senza segno, varia lo spettro di valori che possono essere contenuti.

Nel caso di interi (**'char'**, **'short'**, **'int'**, **'long'** e **'long long'**), la variabile può essere utilizzata per tutta la sua estensione a contenere un numero binario. Pertanto, quando la rappresentazione è senza segno, il massimo valore ottenibile è  $(2^n)-1$ , dove  $n$  rappresenta il numero di bit a disposizione. Quando invece si vuole trattare il dato come un numero con segno, il valore numerico massimo ottenibile è circa la metà (se si usa la rappresentazione dei valori negativi in complemento a due, l'intervallo di valori va da  $(2^{n-1})-1$  a  $-(2^{n-1})$ )

Nel caso di variabili a virgola mobile non c'è più la possibilità di rappresentare esclusivamente valori senza segno; inoltre, più che esserci un limite nella grandezza rappresentabile (che comunque esiste), c'è soprattutto un limite nel grado di approssimazione.

Le variabili '**char**' sono fatte, in linea di principio, per contenere il codice di rappresentazione di un carattere, secondo la codifica utilizzata nel sistema. Generalmente si tratta di un dato di 8 bit, ma non è detto che debba sempre essere così. A ogni modo, il fatto che questa variabile possa essere gestita in modo numerico, permette una facile conversione da lettera a codice numerico corrispondente.

Un tipo di valore che non è stato ancora visto è quello logico: *Vero* è rappresentato da un qualsiasi valore numerico intero diverso da zero, mentre *Falso* corrisponde a zero.

### 66.1.3.3 Costanti letterali comuni

Quasi tutti i tipi di dati primitivi hanno la possibilità di essere rappresentati in forma di costante letterale. In particolare, si distingue tra:

- costanti carattere, rappresentate da un carattere alfanumerico racchiuso tra apici singoli, come '**A**', '**B**',...;
- costanti intere, rappresentate da un numero senza decimali, e a seconda delle dimensioni può trattarsi di uno dei vari tipi di interi (escluso '**char**');
- costanti con virgola, rappresentate da un numero con decimali (un punto seguito da altre cifre, anche se si tratta solo di zeri)

che, indipendentemente dalle dimensioni, di norma sono di tipo **'double'**.

Per esempio, 123 è generalmente una costante **'int'**, mentre 123.0 è una costante **'double'**.

Le costanti che esprimono valori interi possono essere rappresentate con diverse basi di numerazione, attraverso l'indicazione di un prefisso: **'0n'**, dove *n* contiene esclusivamente cifre da zero a sette, viene inteso come un numero in base otto; **'0xn'** o **'0Xn'**, dove *n* può contenere le cifre numeriche consuete, oltre alle lettere da «A» a «F» (minuscole o maiuscole, indifferentemente) viene trattato come un numero in base sedici; negli altri casi, un numero composto con cifre da zero a nove è interpretato in base dieci.

Per quanto riguarda le costanti che rappresentano numeri con virgola, oltre alla notazione **'intero .decimali'** si può usare la «notazione scientifica». Per esempio, **'7e+15'** rappresenta l'equivalente di  $7 \cdot (10^{15})$ , cioè un sette con 15 zeri. Nello stesso modo, **'7e-5'**, rappresenta l'equivalente di  $7 \cdot (10^{-5})$ , cioè 0,00007.

Il tipo di rappresentazione delle costanti numeriche, intere o con virgola, può essere specificato aggiungendo un suffisso, costituito da una o più lettere, come si vede nelle tabelle successive. Per esempio, **'123UL'** è un numero di tipo **'unsigned long int'**, mentre **'123.0F'** è un tipo **'float'**. Si osservi che il suffisso può essere composto, indifferentemente, con lettere minuscole o maiuscole.



Tabella 66.15. Suffissi per le costanti che esprimono valori interi.

Suffisso	Descrizione
assente	In tal caso si tratta di un intero «normale» o più grande, se necessario.
U	Tipo senza segno ( <b>'unsigned'</b> ).
L	Intero più grande della dimensione normale ( <b>'long'</b> ).
LL	Intero molto più grande della dimensione normale ( <b>'long long'</b> ).
UL	Intero senza segno, più grande della dimensione normale ( <b>'unsigned long'</b> ).
ULL	Intero senza segno, molto più grande della dimensione normale ( <b>'unsigned long long'</b> ).

Tabella 66.16. Suffissi per le costanti che esprimono valori con virgola.

Suffisso	Descrizione
assente	Tipo <b>'double'</b> .
F	Tipo <b>'float'</b> .
L	Tipo <b>'long double'</b> .

È possibile rappresentare anche le stringhe in forma di costante attraverso l'uso degli apici doppi, ma la stringa non è un tipo di dati primitivo, trattandosi piuttosto di un array di caratteri. Per il momento è importante fare attenzione a non confondere il tipo **'char'** con la stringa. Per esempio, **'F'** è un carattere (con un proprio valore numerico), mentre **"F"** è una stringa, ma la differenza tra i due è

notevole. Le stringhe vengono descritte nella sezione [66.5](#).

I caratteri privi di rappresentazione grafica possono essere indicati, principalmente, attraverso tre tipi di notazione: ottale, esadecimale e simbolica. In tutti i casi si utilizza la barra obliqua inversa (`\`) come carattere di escape, cioè come simbolo per annunciare che ciò che segue immediatamente deve essere interpretato in modo particolare.

La notazione ottale usa la forma `\ooo`, dove ogni lettera *o* rappresenta una cifra ottale. A questo proposito, è opportuno notare che se la dimensione di un carattere fosse superiore ai fatidici 8 bit, occorrerebbero probabilmente più cifre (una cifra ottale rappresenta un gruppo di 3 bit).

La notazione esadecimale usa la forma `\xhh`, dove *h* rappresenta una cifra esadecimale. Anche in questo caso vale la considerazione per cui ci vogliono più di due cifre esadecimali per rappresentare un carattere più lungo di 8 bit.

Dovrebbe essere logico, ma è il caso di osservare che la corrispondenza dei caratteri con i rispettivi codici numerici dipende dalla codifica utilizzata. Generalmente si utilizza la codifica ASCII, riportata anche nella sezione [47.7.5](#) (in questa fase introduttiva si omette di trattare la rappresentazione dell'insieme di caratteri universale).

La notazione simbolica permette di fare riferimento facilmente a codici di uso comune, quali `<CR>`, `<HT>`,... Inoltre, questa notazione permette anche di indicare caratteri che altrimenti verrebbero interpretati in maniera differente dal compilatore. La tabella 66.17 riporta i vari tipi di rappresentazione delle costanti carattere attraverso codici di escape.

Tabella 66.17. Elenco dei modi di rappresentazione delle costanti carattere attraverso codici di escape.

Codi- ce	ASCII	Altra codifica
<code>\ooo</code>	Notazione ottale in base alla codifica.	idem
<code>\xhh</code>	Notazione esadecimale in base alla codifica.	idem
<code>\\</code>	Una singola barra obliqua inversa ( <code>'\'</code> ).	idem
<code>\'</code>	Un apice singolo destro.	idem
<code>\"</code>	Un apice doppio.	idem
<code>\?</code>	Un punto interrogativo (per impedire che venga inteso come parte di una sequenza triplice, o <i>trigraph</i> ).	idem
<code>\0</code>	Il codice <code>&lt;NUL&gt;</code> .	Il carattere nullo (con tutti i bit a zero).
<code>\a</code>	Il codice <code>&lt;BEL&gt;</code> ( <i>bell</i> ).	Il codice che, rappresentato sullo schermo o sulla stampante, produce un segnale acustico ( <i>alert</i> ).
<code>\b</code>	Il codice <code>&lt;BS&gt;</code> ( <i>backspace</i> ).	Il codice che fa arretrare il cursore di una posizione nella riga ( <i>backspace</i> ).
<code>\f</code>	Il codice <code>&lt;FF&gt;</code> ( <i>form feed</i> ).	Il codice che fa avanzare il cursore all'inizio della prossima pagina logica ( <i>form feed</i> ).

Codi- ce	ASCII	Altra codifica
<code>\n</code>	Il codice <code>&lt;LF&gt;</code> ( <i>line feed</i> ).	Il codice che fa avanzare il cursore all'inizio della prossima riga logica ( <i>new line</i> ).
<code>\r</code>	Il codice <code>&lt;CR&gt;</code> ( <i>carriage return</i> ).	Il codice che porta il cursore all'inizio della riga attuale ( <i>carriage return</i> ).
<code>\t</code>	Una tabulazione orizzontale ( <code>&lt;HT&gt;</code> ).	Il codice che porta il cursore all'inizio della prossima tabulazione orizzontale ( <i>horizontal tab</i> ).
<code>\v</code>	Una tabulazione verticale ( <code>&lt;VT&gt;</code> ).	Il codice che porta il cursore all'inizio della prossima tabulazione verticale ( <i>vertical tab</i> ).

A parte i casi di `'\ooo'` e `'\xhh'`, le altre sequenze esprimono un concetto, piuttosto di un codice numerico preciso. All'origine del linguaggio C, tutte le altre sequenze corrispondono a un solo carattere non stampabile, ma attualmente non è più garantito che sia così. In particolare, la sequenza `'\n'`, nota come *new-line*, potrebbe essere espressa in modo molto diverso rispetto al codice `<LF>` tradizionale. Questo concetto viene comunque approfondito a proposito della gestione dei flussi di file.

In varie situazioni, il linguaggio C standard ammette l'uso di sequenze composte da due o tre caratteri, note come *digraph* e *trigraph* rispettivamente; ciò in sostituzione di simboli la cui rappresentazione, in quel contesto, può essere impossibile. In un sistema che ammetta almeno l'uso della codifica ASCII per scrivere il file sorgente, con l'ausilio di una tastiera comune, non c'è alcun bisogno di usare tali artifici, i quali, se usati, renderebbero estremamente complessa la lettura del sorgente. Pertanto, è bene sapere che esistono queste cose, ma è meglio non usarle mai. Tuttavia, siccome le sequenze a tre caratteri (*trigraph*) iniziano con una coppia di punti interrogativi, se in una stringa si vuole rappresentare una sequenza del genere, per evitare che il compilatore la traduca diversamente, è bene usare la sequenza '\?\?', come suggerisce la tabella 66.17.

Nell'esempio introduttivo appare già la notazione '\n' per rappresentare l'inserzione di un codice di interruzione di riga alla fine del messaggio di saluto:

```
...  
    printf ("Ciao mondo!\n");  
...
```

Senza di questo, il cursore resterebbe a destra del messaggio alla fine dell'esecuzione di quel programma, ponendo lì l'invito.

#### 66.1.3.4 Valore numerico delle costanti carattere

Il linguaggio C distingue tra i caratteri di un insieme fondamentale e ridotto, da quelli dell'insieme di caratteri universale (ISO 10646). Il gruppo di caratteri ridotto deve essere rappresentabile in una variabi-

le **'char'** (descritta nelle sezioni successive) e può essere gestito direttamente in forma numerica, se si conosce il codice corrispondente a ogni simbolo (di solito si tratta della codifica ASCII).

Se si può essere certi che nella codifica le lettere dell'alfabeto latino siano disposte esattamente in sequenza (come avviene proprio nella codifica ASCII), si potrebbe scrivere **'A'+1** e ottenere l'equivalente di **'B'**. Tuttavia, lo standard prescrive che sia garantito il funzionamento solo per le cifre numeriche. Pertanto, per esempio, **'0'+3** (zero espresso come carattere, sommato a un tre numerico) deve essere equivalente a **'3'** (ovvero un «tre» espresso come carattere).

Listato 66.19. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/yrc2S7Xv>, <http://ideone.com/HCvsD>.

```
#include <stdio.h>

int main (void)
{
    char c;
    for (c = '0'; c <= 'Z'; c++)
    {
        printf ("%c", c);
    }
    printf ("\n");
    return 0;
}
```

Il programma di esempio che si vede nel listato appena mostrato, se prodotto per un ambiente in cui si utilizza la codifica ASCII, genera il risultato seguente:

0123456789:;<=>?@ABCDEFGHIJKLMNPOQRSTUVWXYZ

### 66.1.3.5 Campo di azione delle variabili

Il campo di azione delle variabili in C viene determinato dalla posizione in cui queste vengono dichiarate e dall'uso di qualificatori particolari. Nella fase iniziale dello studio del linguaggio basta considerare, approssimativamente, che quanto dichiarato all'interno di una funzione ha valore locale per la funzione stessa, mentre quanto dichiarato al di fuori, ha valore globale per tutto il file. Pertanto, in questo capitolo si usano genericamente le definizioni di «variabile locale» e «variabile globale», senza affrontare altre questioni. Nella sezione [66.3](#) viene trattato questo argomento con maggiore dettaglio.

### 66.1.3.6 Dichiarazione delle variabili

La dichiarazione di una variabile avviene specificando il tipo e il nome della variabile, come nell'esempio seguente dove viene creata la variabile *numero* di tipo intero:

```
int numero;
```

La variabile può anche essere inizializzata contestualmente, assegnandole un valore, come nell'esempio seguente in cui viene dichiarata la stessa variabile *numero* con il valore iniziale di 1000:

```
int numero = 1000;
```

Una costante è qualcosa che non varia e generalmente si rappresenta attraverso una notazione che ne definisce il valore, ovvero attraverso una costante letterale. Tuttavia, a volte può essere più comodo definire una costante in modo simbolico, come se fosse una variabile,

per facilitarne l'utilizzo e la sua identificazione all'interno del programma. Si ottiene questo con il modificatore **'const'**. Ovviamente, è obbligatorio inizializzarla contestualmente alla sua dichiarazione. L'esempio seguente dichiara la costante simbolica *pi* con il valore del  $\pi$ :

```
const float pi = 3.14159265;
```

Le costanti simboliche di questo tipo, sono delle variabili per le quali il compilatore non concede che avvengano delle modifiche; pertanto, il programma eseguibile che si ottiene potrebbe essere organizzato in modo tale da caricare questi dati in segmenti di memoria a cui viene lasciato poi il solo permesso di lettura.

Tradizionalmente, l'uso di costanti simboliche di questo tipo è stato limitato, preferendo delle *macro-variabili* definite e gestite attraverso il precompilatore (come viene descritto nella sezione 66.2). Tuttavia, un compilatore ottimizzato è in grado di gestire al meglio le costanti definite nel modo illustrato dall'esempio, utilizzando anche dei valori costanti letterali nella trasformazione in linguaggio assembleatore, rendendo così indifferente, dal punto di vista del risultato, l'alternativa delle macro-variabili. Pertanto, la stessa guida *GNU coding standards* chiede di definire le costanti come variabili-costanti, attraverso il modificatore **'const'**.

### 66.1.3.7 Variabili costanti e variabili volatili

«

Come già descritto nella sezione precedente, una variabile può essere dichiarata con il modificatore **'const'** per sottolineare al compilatore che non deve essere modificata nel corso del programma, salva la possibilità di inizializzarla contestualmente alla sua dichiarazione.



```
const float pi = 3.14159265;
```

All'opposto della costante si può considerare un'area di memoria a cui accedono programmi differenti, in modo asincrono, ognuno con la facoltà di modificarla a proprio piacimento, oppure un'area che viene modificata direttamente dall'hardware. In questi casi, ovvero quando il compilatore non deve attuare delle semplificazioni che partano dalla presunzione del contenuto di una certa variabile, si usa il modificatore **'volatile'**. Si osservi l'esempio seguente:

```
...
volatile int i;
...
i = 1;
if (i > 0)
    {
        ...
    }
else
    {
        ...
    }
...
```

Anche se alla variabile *i* viene assegnato il valore uno, il compilatore non può escludere che nel momento della verifica della variabile questa abbia invece un valore differente. In altri termini, se la variabile *i* venisse dichiarata in modo normale, un compilatore ottimizzato potrebbe escludere le istruzioni sotto il controllo della parola chiave **'else'**.

Quando l'area di memoria che viene considerata «volatile», deve essere modificata da un processo estraneo, mentre il programma si

limita semplicemente a leggerne il contenuto prendendo atto del valore che ha, la variabile può essere dichiarata simultaneamente con i modificatori `'const'` e `'volatile'`, come nell'esempio seguente, dove, tra l'altro, si presume che la variabile in questione sia definita in un altro file-oggetto:

```
extern const volatile int variabile ;  
...
```

### 66.1.3.8 Il tipo indefinito: «void»

«

Lo standard del linguaggio C definisce un tipo particolare di variabile, individuato dalla parola chiave `'void'`: si tratta di un tipo che rappresenta una variabile di rango nullo; la quale, come tale, non può contenere alcun valore.

### 66.1.4 Operatori ed espressioni

«

L'operatore è qualcosa che esegue un qualche tipo di funzione, su uno o più operandi, restituendo un valore.<sup>11</sup> Il valore restituito è di tipo diverso a seconda degli operandi utilizzati. Per esempio, la somma di due interi genera un risultato intero. Gli operandi descritti di seguito sono quelli più comuni e importanti.

Le espressioni sono formate spesso dalla valutazione di sottoespressioni (espressioni più piccole). Va osservato che ci sono circostanze in cui il contesto non impone che ci sia un solo ordine possibile nella valutazione delle sottoespressioni, ma il programmatore deve tenere conto di questa possibilità, per evitare che il risultato dipenda dalle scelte non prevedibili del compilatore.

Tabella 66.27. Ordine di precedenza tra gli operatori previsti nel linguaggio C. Gli operatori sono raggruppati a livelli di priorità equivalente, partendo dall'alto con la priorità maggiore, scendendo progressivamente alla priorità minore. Le variabili *a*, *b* e *c* rappresentano la collocazione delle sottoespressioni da considerare ed esprimono l'ordine di associatività: prima *a*, poi *b*, poi *c*.

Operatori	Annotazioni
<p>(<i>a</i>)</p> <p>[<i>a</i>]</p> <p><i>a</i>-&gt;<i>b</i>   <i>a</i>.<i>b</i></p>	Le parentesi tonde usate per raggruppare una porzione di espressione hanno la precedenza su ogni altro operatore. Le parentesi quadre riguardano gli array; gli operatori '->' e '.', riguardano le strutture e le unioni.
<p>!<i>a</i>   ~<i>a</i>   ++<i>a</i>   --<i>a</i></p> <p>    +<i>a</i>   -<i>a</i></p> <p>*<i>a</i>   &amp;<i>a</i></p> <p>(<i>tipo</i>)   sizeof <i>a</i></p>	Gli operatori '+' e '-' di questo livello sono da intendersi come «unari», ovvero si riferiscono al segno di quanto appare alla loro destra. Gli operatori '*' e '&' di questo livello riguardano la gestione dei puntatori; le parentesi tonde si riferiscono al cast.
<i>a</i> * <i>b</i> <i>a</i> / <i>b</i> <i>a</i> % <i>b</i>	Moltiplicazione, divisione e resto della divisione intera.
<i>a</i> + <i>b</i> <i>a</i> - <i>b</i>	Somma e sottrazione.
<i>a</i> << <i>b</i> <i>a</i> >> <i>b</i>	Scorrimento binario.
<i>a</i> < <i>b</i> <i>a</i> <= <i>b</i> <i>a</i> > <i>b</i> <i>a</i> => <i>b</i>	Confronto.
<i>a</i> == <i>b</i> <i>a</i> != <i>b</i>	Confronto.
<i>a</i> & <i>b</i>	AND bit per bit.

Operatori	Annotazioni
$a \wedge b$	XOR bit per bit.
$a   b$	OR bit per bit.
$a \& b$	AND nelle espressioni logiche.
$a    b$	OR nelle espressioni logiche.
$c ? b : a$	Operatore condizionale
$b = a$ $b += a$ $b -= a$ $b * = a$ $b / = a$ $b \% = a$ $b \& = a$ $b \wedge = a$ $b   = a$ $b \ll = a$ $b \gg = a$	Operatori di assegnamento.
$a, b$	Sequenza di espressioni (espressione multipla).

### 66.1.4.1 Operatori aritmetici



Gli operatori che intervengono su valori numerici sono elencati nella tabella 66.28. Per dare un significato alle descrizioni della tabella, occorre tenere presente una caratteristica importante del linguaggio, per la quale, la maggior parte delle espressioni restituisce un valore. Per esempio, ‘ $\mathbf{b} = \mathbf{a} = \mathbf{1}$ ’ fa sì che la variabile  $\mathbf{a}$  ottenga il valore 1 e che, successivamente, la variabile  $\mathbf{b}$  ottenga il valore di  $\mathbf{a}$ . In questo senso, al problema dell’ordine di precedenza dei vari operatori si aggiunge anche l’ordine in cui le espressioni restituiscono un valo-

re. Per esempio, `d = e++` comporta l'incremento di una unità del contenuto della variabile `e`, ma ciò solo **dopo** averne restituito il valore che viene assegnato alla variabile `d`. Pertanto, se inizialmente la variabile `e` contiene il valore 1, dopo l'elaborazione dell'espressione completa, la variabile `d` contiene il valore 1, mentre la variabile `e` contiene il valore 2.

Tabella 66.28. Elenco degli operatori aritmetici e di quelli di assegnamento relativi a valori numerici.

Operatore e operandi	Descrizione
<code>++op</code>	Incrementa di un'unità l'operando prima che venga restituito il suo valore.
<code>op++</code>	Incrementa di un'unità l'operando dopo averne restituito il suo valore.
<code>--op</code>	Decrementa di un'unità l'operando prima che venga restituito il suo valore.
<code>op--</code>	Decrementa di un'unità l'operando dopo averne restituito il suo valore.
<code>+op</code>	Non ha alcun effetto.
<code>-op</code>	Inverte il segno dell'operando (prima di restituirne il valore).
<code>op1 + op2</code>	Somma i due operandi.
<code>op1 - op2</code>	Sottrae dal primo il secondo operando.
<code>op1 * op2</code>	Moltiplica i due operandi.
<code>op1 / op2</code>	Divide il primo operando per il secondo.
<code>op1 % op2</code>	Calcola il resto della divisione tra il primo e il secondo operando, i quali devono essere costituiti da valori interi.

Operatore e operandi	Descrizione
<i>var</i> = <i>valore</i>	Assegna alla variabile il valore alla destra.
<i>op1</i> += <i>op2</i>	<i>op1</i> = ( <i>op1</i> + <i>op2</i> )
<i>op1</i> -= <i>op2</i>	<i>op1</i> = ( <i>op1</i> - <i>op2</i> )
<i>op1</i> *= <i>op2</i>	<i>op1</i> = ( <i>op1</i> * <i>op2</i> )
<i>op1</i> /= <i>op2</i>	<i>op1</i> = ( <i>op1</i> / <i>op2</i> )
<i>op1</i> %= <i>op2</i>	<i>op1</i> = ( <i>op1</i> % <i>op2</i> )

## 66.1.4.2 Operatori di confronto e operatori logici

«

Gli operatori di confronto determinano la relazione tra due operandi. Il risultato dell'espressione composta da due operandi posti a confronto è un numero intero ('**int**') e precisamente si ottiene uno se il confronto è valido e zero in caso contrario. Gli operatori di confronto sono elencati nella tabella 66.29.

Il linguaggio C non ha una rappresentazione specifica per i valori booleani *Vero* e *Falso*,<sup>12</sup> ma si limita a interpretare un valore pari a zero come *Falso* e un valore diverso da zero come *Vero*. Va osservato, quindi, che il numero usato come valore booleano, può essere espresso anche in virgola mobile, benché sia preferibile di gran lunga un intero normale.

Tabella 66.29. Elenco degli operatori di confronto. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
$op1 == op2$	<i>Vero</i> se gli operandi si equivalgono.
$op1 != op2$	<i>Vero</i> se gli operandi sono differenti.
$op1 < op2$	<i>Vero</i> se il primo operando è minore del secondo.
$op1 > op2$	<i>Vero</i> se il primo operando è maggiore del secondo.
$op1 <= op2$	<i>Vero</i> se il primo operando è minore o uguale al secondo.
$op1 >= op2$	<i>Vero</i> se il primo operando è maggiore o uguale al secondo.

Quando si vogliono combinare assieme diverse espressioni logiche, comprendendo in queste anche delle variabili che contengono un valore booleano, si utilizzano gli operatori logici (noti normalmente come: AND, OR, NOT, ecc.). Il risultato di un'espressione logica complessa è quello dell'ultima espressione elementare valutata effettivamente, in quanto le sottoespressioni che non possono cambiare l'esito della condizione complessiva non vengono valutate. Gli operatori logici sono elencati nella tabella 66.30.

Tabella 66.30. Elenco degli operatori logici. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
<code>! op</code>	Inverte il risultato logico dell'operando.
<code>op1 &amp;&amp; op2</code>	Se il risultato del primo operando è <i>Falso</i> non valuta il secondo.
<code>op1    op2</code>	Se il risultato del primo operando è <i>Vero</i> non valuta il secondo.

Un tipo particolare di operatore logico è l'operatore condizionale, il quale permette di eseguire espressioni diverse in relazione al risultato di una condizione. La sua sintassi si esprime nel modo seguente:

```
condizione ? espressione1 : espressione2
```

In pratica, se l'espressione che rappresenta la condizione si avvera, viene eseguita la prima espressione che segue il punto interrogativo, altrimenti viene eseguita quella che segue i due punti.

### 66.1.4.3 Operatori binari

«

Il linguaggio C consente di eseguire alcune operazioni binarie, sui **valori interi**, come spesso è possibile fare con un linguaggio assembler, anche se non è possibile interrogare degli indicatori (*flag*) che informino sull'esito delle azioni eseguite. Sono disponibili le operazioni elencate nella tabella 66.31.



Tabella 66.31. Elenco degli operatori binari. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
$op1 \ \& \ op2$	AND bit per bit.
$op1 \   \ op2$	OR bit per bit.
$op1 \ \wedge \ op2$	XOR bit per bit (OR esclusivo).
$op1 \ \ll \ op2$	Scorrimento a sinistra di $op2$ bit (con $op2$ che rappresenta un valore positivo o senza segno). A destra vengono aggiunti bit a zero
$op1 \ \gg \ op2$	Scorrimento a destra di $op2$ bit (con $op2$ che rappresenta un valore positivo o senza segno). Il valore dei bit aggiunti a sinistra potrebbe tenere conto del segno.
$\sim op1$	Complemento a uno.
$op1 \ \&= \ op2$	$op1 = (op1 \ \& \ op2)$
$op1 \  = \ op2$	$op1 = (op1 \   \ op2)$
$op1 \ \wedge= \ op2$	$op1 = (op1 \ \wedge \ op2)$
$op1 \ \ll= \ op2$	$op1 = (op1 \ \ll \ op2)$
$op1 \ \gg= \ op2$	$op1 = (op1 \ \gg \ op2)$
$op1 \ \sim= \ op2$	$op1 = \sim op2$

A seconda del compilatore e della piattaforma, lo scorrimento a destra potrebbe essere di tipo aritmetico, ovvero potrebbe tenere conto

del segno del valore che viene fatto scorrere. Pertanto, non potendo fare affidamento su questa ipotesi, è bene che i valori di cui si fa lo scorrimento a destra siano sempre senza segno, o comunque positivi.

Per aiutare a comprendere l'uso degli operatori binari vengono mostrati alcuni esempi. In particolare si utilizzano due operandi di tipo `'char'` (a 8 bit) senza segno: *a* contenente il valore 42, pari a  $00101010_2$ ; *b* contenente il valore 51, pari a  $00110011_2$ .

$c = a \ \& \ b$	$c = a \   \ b$	$c = a \ ^ \ b$
$00101010_2 \ (42_{10}) \ \text{AND}$	$00101010_2 \ (42_{10}) \ \text{OR}$	$00101010_2 \ (42_{10}) \ \text{XOR}$
$00110011_2 \ (51_{10}) \ =$	$00110011_2 \ (51_{10}) \ =$	$00110011_2 \ (51_{10}) \ =$
<hr/>	<hr/>	<hr/>
$00100010_2 \ (34_{10})$	$00111011_2 \ (59_{10})$	$00011001_2 \ (25_{10})$

Lo scorrimento, invece, viene mostrato sempre solo per una singola unità: *a* contenente il valore 42; *b* contenente il valore 1.

$c = a \ \ll \ b$	$c = a \ \gg \ b$	$c = \sim a$
$00101010_2 \ (42_{10}) \ \ll$	$00101010_2 \ (42_{10}) \ \gg$	$00101010_2 \ (42_{10})$
$00000001_2 \ (1_{10}) \ =$	$00000001_2 \ (1_{10}) \ =$	$11010101_2 \ (213_{10})$
<hr/>	<hr/>	
$01010100_2 \ (84_{10})$	$00010101_2 \ (21_{10})$	

#### 66.1.4.4 Conversione di tipo

«

Quando si assegna un valore a una variabile, nella maggior parte dei casi, il contesto stabilisce il tipo di questo valore in modo corretto. Di fatto, è il tipo della variabile ricevente che stabilisce la conversione necessaria. Tuttavia, il problema si pone anche durante la valutazione di un'espressione.

Per esempio, `'5/4'` viene considerata la divisione di due interi e, di conseguenza, l'espressione restituisce un valore intero, cioè 1. Diverso sarebbe se si scrivesse `'5.0/4.0'`, perché in questo caso si tratterebbe della divisione tra due numeri a virgola mobile (per la precisione, di tipo `'double'`) e il risultato è un numero a virgola mobile.

Quando si pone il problema di risolvere l'ambiguità si utilizza esplicitamente la conversione del tipo, attraverso un *cast*:

```
(tipo) espressione
```

In pratica, si deve indicare tra parentesi tonde il nome del tipo di dati in cui deve essere convertita l'espressione che segue. Il problema sta nella precedenza che ha il cast nell'insieme degli altri operatori e in generale conviene utilizzare altre parentesi per chiarire la relazione che ci deve essere.

```
int x = 10;
double y;
...
y = (double) x/9;
```

In questo caso, la variabile intera `x` viene convertita nel tipo `'double'` (a virgola mobile) prima di eseguire la divisione. Dal momento che il cast ha precedenza sull'operazione di divisione, non si pongono problemi, inoltre, la divisione avviene trasformando implicitamente il 9 intero in un 9,0 di tipo `'double'`. In pratica, l'operazione avviene utilizzando valori `'double'` e restituendo un risultato `'double'`.

## 66.1.4.5 Espressioni multiple

«

Un'istruzione, cioè qualcosa che termina con un punto e virgola, può contenere diverse espressioni separate da una virgola. Tenendo presente che in C l'assegnamento di una variabile è anche un'espressione, la quale restituisce il valore assegnato, si veda l'esempio seguente:

```
int x;  
int y;  
...  
y = 10, x = 20, y = x*2;
```

L'esempio mostra un'istruzione contenente tre espressioni: la prima assegna a *y* il valore 10, la seconda assegna a *x* il valore 20 e la terza sovrascrive *y* assegnandole il risultato del prodotto  $x \cdot 2$ . In pratica, alla fine la variabile *y* contiene il valore 40 e *x* contiene 20.

Un'espressione multipla, come quella dell'esempio, restituisce il valore dell'ultima a essere eseguita. Tornando all'esempio appena visto, gli si può apportare una piccola modifica per comprendere il concetto:

```
int x;  
int y;  
int z;  
...  
z = (y = 10, x = 20, y = x*2);
```

La variabile *z* si trova a ricevere il valore dell'espressione '**y = x\*2**', perché è quella che viene eseguita per ultima nel gruppo raccolto tra parentesi.

A proposito di «espressioni multiple» vale la pena di ricordare ciò

che accade con gli assegnamenti multipli, con l'esempio seguente:

```
y = x = 10;
```

Qui si vede l'assegnamento alla variabile  $y$  dello stesso valore che viene assegnato alla variabile  $x$ . In pratica, sia  $x$ , sia  $y$ , contengono alla fine il numero 10, perché le precedenze sono tali che è come se fosse scritto: ' $y = (x = 10)$ '.

### 66.1.5 Strutture di controllo di flusso

Il linguaggio C gestisce praticamente tutte le strutture di controllo di flusso degli altri linguaggi di programmazione, compreso *go-to* che comunque è sempre meglio non utilizzare e qui, volutamente, non viene presentato.

Le strutture di controllo permettono di sottoporre l'esecuzione di una parte di codice alla verifica di una condizione, oppure permettono di eseguire dei cicli, sempre sotto il controllo di una condizione. La parte di codice che viene sottoposta a questo controllo, può essere una singola istruzione, oppure un gruppo di istruzioni (precisamente si chiamerebbe istruzione composta). Nel secondo caso, è necessario delimitare questo gruppo attraverso l'uso delle parentesi graffe.

Dal momento che è comunque consentito di realizzare un gruppo di istruzioni che in realtà ne contiene una sola, probabilmente è meglio utilizzare sempre le parentesi graffe, in modo da evitare equivoci nella lettura del codice. Dato che le parentesi graffe sono usate nel codice C, se queste appaiono nei modelli sintattici indicati, significa che fanno parte delle istruzioni e non della sintassi.

Negli esempi, i rientri delle parentesi graffe seguono le indicazioni della guida *GNU coding standards*.

### 66.1.5.1 Struttura condizionale: «if»

&lt;&lt;

La struttura condizionale è il sistema di controllo fondamentale dell'andamento del flusso delle istruzioni.

```
if (condizione) istruzione
```

```
if (condizione) istruzione else istruzione
```

Se la condizione si verifica, viene eseguita l'istruzione o il gruppo di istruzioni che segue; quindi il controllo passa alle istruzioni successive alla struttura. Se viene utilizzata la sotto-struttura che si articola a partire dalla parola chiave '**else**', nel caso non si verifichi la condizione, viene eseguita l'istruzione che ne dipende. Sotto vengono mostrati alcuni esempi.

```
int i_importo;  
...  
if (i_importo > 10000000) printf ("L'offerta è vantaggiosa\n");
```

```
int i_importo;  
int i_memorizza;  
...  
if (i_importo > 10000000)  
{  
    i_memorizza = i_importo;  
    printf ("L'offerta è vantaggiosa\n");  
}  
else  
{  
    printf ("Lascia perdere\n");  
}
```

L'esempio successivo, in particolare, mostra un modo grazioso per allineare le sottocondizioni, senza eccedere negli annidamenti:

```
int i_importo;
int i_memorizza;
...
if (i_importo > 10000000)
{
    i_memorizza = i_importo;
    printf ("L'offerta è vantaggiosa\n");
}
else if (i_importo > 5000000)
{
    i_memorizza = i_importo;
    printf ("L'offerta è accettabile\n");
}
else
{
    printf ("Lascia perdere\n");
}
```

### 66.1.5.2 Struttura di selezione: «switch»

La struttura di selezione che si attua con l'istruzione '**switch**', è un po' troppo complessa per essere rappresentata facilmente attraverso uno schema sintattico. In generale, questa struttura permette di **saltare** a una certa posizione della struttura, in base al risultato di un'espressione. L'esempio seguente mostra la visualizzazione del nome del mese, in base al valore di un intero.

```
int i_mese;
...
switch (i_mese)
{
```

```
case 1: printf ("gennaio\n"); break;
case 2: printf ("febbraio\n"); break;
case 3: printf ("marzo\n"); break;
case 4: printf ("aprile\n"); break;
case 5: printf ("maggio\n"); break;
case 6: printf ("giugno\n"); break;
case 7: printf ("luglio\n"); break;
case 8: printf ("agosto\n"); break;
case 9: printf ("settembre\n"); break;
case 10: printf ("ottobre\n"); break;
case 11: printf ("novembre\n"); break;
case 12: printf ("dicembre\n"); break;
}
```

Come si vede, dopo l'istruzione con cui si emette il nome del mese attraverso lo standard output, viene richiesta l'interruzione esplicita dell'analisi della struttura, attraverso l'istruzione **'break'**, perché altrimenti verrebbero eseguite le istruzioni del caso successivo, se presente. Infatti, un gruppo di casi può essere raggruppato assieme, quando si vuole che ognuno di questi esegua lo stesso insieme di istruzioni.

```
int i_anno;
int i_mese;
int i_giorni;
...
switch (i_mese)
{
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
```



```
    case 12:
        i_giorni = 31;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        i_giorni = 30;
        break;
    case 2:
        if (((i_anno % 4 == 0) && !(i_anno % 100 == 0)) ||
            (i_anno % 400 == 0))
            i_giorni = 29;
        else
            i_giorni = 28;
        break;
}
```

È anche possibile dichiarare un caso predefinito che si verifichi quando nessuno degli altri si avvera.

```
int i_mese;
...
switch (i_mese)
{
    case 1: printf ("gennaio\n"); break;
    case 2: printf ("febbraio\n"); break;
    ...
    case 11: printf ("novembre\n"); break;
    case 12: printf ("dicembre\n"); break;
    default: printf ("mese non corretto\n"); break;
}
```

La struttura di selezione che si ottiene con l'istruzione **'switch'** può

apparire disarmonica rispetto all'organizzazione del linguaggio C, per la presenza delle voci '**case valore** :'. Queste voci sono sostanzialmente delle «etichette» che individuano una posizione nel codice, da raggiungere in base al valore preso in considerazione per la selezione.

### 66.1.5.3 Iterazione con condizione di uscita iniziale: «while»

«

L'iterazione si ottiene normalmente in C attraverso l'istruzione '**while**', la quale esegue un'istruzione, o un gruppo di queste, finché la condizione continua a restituire il valore *Vero*. La condizione viene valutata prima di eseguire il gruppo di istruzioni e poi ogni volta che termina un ciclo, prima dell'esecuzione del successivo.

```
while (condizione) istruzione
```

L'esempio seguente fa apparire per 10 volte la lettera «x».

```
int i = 0;

while (i < 10)
{
    i++;
    printf ("x");
}
printf ("\n");
```

Nel blocco di istruzioni di un ciclo '**while**', ne possono apparire alcune particolari:

- '**break**', che serve a uscire definitivamente dalla struttura del ciclo;

- **'continue'**, che serve a interrompere l'esecuzione del gruppo di istruzioni, riprendendo immediatamente con il ciclo successivo (a partire dalla valutazione della condizione).

L'esempio seguente è una variante del calcolo di visualizzazione mostrato sopra, modificato in modo da vedere il funzionamento dell'istruzione **'break'**. All'inizio della struttura, **'while (1)'** equivale a stabilire che il ciclo è senza fine, perché la condizione è sempre vera. In questo modo, solo la richiesta esplicita di interruzione dell'esecuzione della struttura (attraverso l'istruzione **'break'**) permette l'uscita da questa.

```
int i = 0;

while (1)
{
    if (i >= 10)
    {
        break;
    }
    i++;
    printf ("x");
}
printf ("\n");
```

#### 66.1.5.4 Iterazione con condizione di uscita finale: «do-while»

Una variante del ciclo **'while'**, in cui l'analisi della condizione di uscita avviene dopo l'esecuzione del blocco di istruzioni che viene iterato, è definito dall'istruzione **'do'**.



```
do blocco_di_istruzioni while (condizione);
```

In questo caso, si esegue un gruppo di istruzioni una volta, poi se ne ripete l'esecuzione finché la condizione restituisce il valore *Vero*.

```
int i = 0;

do
{
    i++;
    printf ("x");
}
while (i < 10);
printf ("\n");
```

L'esempio mostrato è quello già usato nella sezione precedente, con l'adattamento necessario a utilizzare questa struttura di controllo.

La struttura di controllo '**do...while**' è in disuso, perché, generalmente, al suo posto si preferisce gestire i cicli di questo tipo attraverso una struttura '**while**', pura e semplice.

### 66.1.5.5 Ciclo enumerativo: «for»

«

In presenza di iterazioni in cui si deve incrementare o decrementare una variabile a ogni ciclo, si usa preferibilmente la struttura '**for**', che in C permetterebbe un utilizzo più ampio di quello comune:

```
for ( [espressione1] ; [espressione2] ; [espressione3] ) istruzione
```

La forma tipica di un'istruzione **for** è quella per cui la prima espressione corrisponde all'assegnamento iniziale di una variabile, la seconda a una condizione che deve verificarsi fino a che si vuole che sia eseguita l'istruzione (o il gruppo di istruzioni) e la terza all'incremento o decremento della variabile inizializzata con la prima espressione. In pratica, l'utilizzo normale del ciclo **for** potrebbe esprimersi nella sintassi seguente:

```
for (var = n; condizione; var++) istruzione
```

Il ciclo **for** potrebbe essere definito anche in maniera differente, più generale: la prima espressione viene eseguita una volta sola all'inizio del ciclo; la seconda viene valutata all'inizio di ogni ciclo e il gruppo di istruzioni viene eseguito solo se il risultato è *Vero*; l'ultima viene eseguita alla fine dell'esecuzione del gruppo di istruzioni, prima che si ricominci con l'analisi della condizione.

L'esempio già visto, in cui viene visualizzata per 10 volte una «x», potrebbe tradursi nel modo seguente, attraverso l'uso di un ciclo **for**:

```
int i;

for (i = 0; i < 10; i++)
{
    printf ("x");
}
printf ("\n");
```

Anche nelle istruzioni controllate da un ciclo **for** si possono collocare istruzioni **break** e **continue**, con lo stesso significato visto per il ciclo **while** e **do..while**.

Sfruttando la possibilità di inserire più espressioni in una singola istruzione, si possono realizzare dei cicli **for** molto più complessi, anche se questo è sconsigliabile per evitare di scrivere codice troppo difficile da interpretare. In questo modo, l'esempio precedente potrebbe essere ridotto a quello che segue, dove si usa un punto e virgola solitario per rappresentare un'istruzione nulla:

```
int i;

for (i = 0; i < 10; printf ("x"), i++)
{
    ;
}
printf ("\n");
```

Se si utilizzano istruzioni multiple, separate con la virgola, occorre tenere presente che **l'espressione che esprime la condizione deve rimanere singola** (se per la condizione si usasse un'espressione multipla, conterebbe solo la valutazione dell'ultima). Naturalmente, nel caso della condizione, si possono costruire condizioni complesse con l'ausilio degli operatori logici, ma rimane il fatto che l'operatore virgola (',') non dovrebbe avere senso lì.

Nel modello sintattico iniziale si vede che le tre espressioni sono opzionali e rimane solo l'obbligo di mettere i punti e virgola relativi. L'esempio seguente mostra un ciclo senza fine che viene interrotto attraverso un'istruzione **break**:

```
int i = 0;
for (;;)
{
    if (i >= 10)
    {
        break;
    }
    printf ("x");
    i++;
}
```

## 66.1.6 Funzioni

Il linguaggio C offre le funzioni come mezzo per realizzare la scomposizione del codice in subroutine. Prima di poter essere utilizzate attraverso una chiamata, le funzioni devono essere dichiarate, anche se non necessariamente descritte. In pratica, se si vuole indicare nel codice una chiamata a una funzione che viene descritta più avanti, occorre almeno dichiararne il prototipo.

Le funzioni del linguaggio C prevedono il passaggio di parametri solo **per valore**, con tutti i tipi di dati, esclusi gli array (che invece vanno passati per riferimento, attraverso il puntatore alla loro posizione iniziale in memoria).

Il linguaggio C, attraverso la libreria standard, offre un gran numero di funzioni comuni, i cui prototipi vengono incorporati nel codice attraverso l'inclusione di file di intestazione, con l'istruzione `#include` del precompilatore. Per esempio, come si è già visto, per poter utilizzare la funzione *printf()* si deve inserire la riga `#include <stdio.h>` nella parte iniziale del file sorgente.

### 66.1.6.1 Dichiarazione di un prototipo

«

Quando la descrizione di una funzione può essere fatta solo dopo l'apparizione di una sua chiamata, occorre dichiararne il prototipo all'inizio, secondo la sintassi seguente:

```
tipo nome ( [tipo [ nome ] [, ...] ] );
```

Il tipo, posto all'inizio, rappresenta il tipo di valore che la funzione restituisce. Se la funzione non deve restituire alcunché, si utilizza il tipo **'void'**. Se la funzione utilizza dei parametri, il tipo di questi deve essere elencato tra le parentesi tonde. L'istruzione con cui si dichiara il prototipo termina regolarmente con un punto e virgola.

Lo standard C stabilisce che una funzione che non richiede parametri deve utilizzare l'identificatore **'void'** in modo esplicito, all'interno delle parentesi.

Segue la descrizione di alcuni esempi.

- ```
int fattoriale (int);
```

In questo caso, viene dichiarato il prototipo della funzione **'fattoriale'**, che richiede un parametro di tipo **'int'** e restituisce anche un valore di tipo **'int'**.

- ```
int fattoriale (int n);
```

Come nell'esempio precedente, dove in più, per comodità si aggiunge il nome del parametro che comunque viene ignorato dal compilatore.



- ```
void elenca (void);
```

Si tratta della dichiarazione di una funzione che fa qualcosa senza bisogno di ricevere alcun parametro e senza restituire alcun valore (**void** è il tipo che rappresenta una variabile di rango nullo e, come tale, incapace di accogliere qualunque valore).

## 66.1.6.2 Descrizione di una funzione

La descrizione della funzione, rispetto alla dichiarazione del prototipo, richiede l'indicazione dei nomi da usare per identificare i parametri (mentre nel prototipo questi sono facoltativi) e naturalmente l'aggiunta delle istruzioni da eseguire. Le parentesi graffe che appaiono nello schema sintattico fanno parte delle istruzioni necessarie.

```
tipo nome ( [tipo parametro [, ...] ] )  
{  
    istruzione ;  
    ...  
}
```

Per esempio, la funzione seguente esegue il prodotto tra i due parametri forniti e ne restituisce il risultato:

```
int prodotto (int x, int y)  
{  
    return (x * y);  
}
```

I parametri indicati tra parentesi, rappresentano una dichiarazione di variabili locali<sup>13</sup> che contengono inizialmente i valori usati nella chiamata. Il valore restituito dalla funzione viene definito attraverso

l'istruzione `return`, come si può osservare dall'esempio. Naturalmente, nelle funzioni di tipo `void` l'istruzione `return` va usata senza specificare il valore da restituire, oppure si può fare a meno del tutto di tale istruzione.

Nei manuali tradizionale del linguaggio C si descrivono le funzioni nel modo visto nell'esempio precedente; al contrario, nella guida *GNU coding standards* si richiede di mettere il nome della funzione in corrispondenza della colonna uno, così:

```
int
prodotto (int x, int y)
{
    return (x * y);
}
```

Le variabili dichiarate all'interno di una funzione, oltre a quelle dichiarate implicitamente come mezzo di trasporto degli argomenti della chiamata, sono visibili solo al suo interno, mentre quelle dichiarate al di fuori di tutte le funzioni, sono variabili globali, accessibili potenzialmente da ogni parte del programma.<sup>14</sup> Se una variabile locale ha un nome coincidente con quello di una variabile globale, allora, all'interno della funzione, quella variabile globale non è accessibile.

Le regole da seguire, almeno in linea di principio, per scrivere programmi chiari e facilmente modificabili, prevedono che si debba fare in modo di rendere le funzioni indipendenti dalle variabili globali, fornendo loro tutte le informazioni necessarie attraverso i parametri. In questo modo diventa del tutto indifferente il fatto che una variabile locale vada a mascherare una variabile globale; inoltre, ciò permette di non dover tenere a mente il ruolo di queste variabili globali e (se

non si usano le variabili «statiche») fa sì che si ottenga una funzione completamente «rientrante».

### 66.1.6.3 File di intestazione e libreria

Una libreria di funzioni si compone almeno di due parti fondamentali: i prototipi delle funzioni e la descrizione delle funzioni stesse. Secondo la tradizione, l'inclusione di codice attraverso l'istruzione '**#include**' del precompilatore, si usa esclusivamente per includere «file di intestazione», contraddistinti convenzionalmente da un nome che termina con il suffisso '**.h**'. Questi file di intestazione devono essere costruiti con certi criteri, in modo che la loro inclusione multipla non possa creare problemi. Per quanto riguarda le funzioni, questi file possono contenerne esclusivamente i prototipi.

Per esempio, si potrebbe dire che per poter usare la funzione *printf()* si debba includere la «libreria» standard '`stdio.h`'. L'affermazione in sé può essere accettabile, ma non è precisa. Infatti, il file di intestazione '`stdio.h`' contiene prototipi e altre definizioni della porzione della libreria standard che consente di usare la funzione *printf()*, ma la descrizione effettiva di tale funzione si trova in un altro file.

### 66.1.7 Vincoli nei nomi

Quando si definiscono variabili e funzioni nel proprio programma, occorre avere la prudenza di non utilizzare nomi che coincidano con quelli delle librerie che si vogliono usare e che non possano andare in conflitto con l'evoluzione del linguaggio. A questo proposito va osservata una regola molto semplice: non si vanno usati nomi «esterni» che inizino con il trattino basso ('\_'); in tutti gli altri casi, inve-

ce, non si possono usare i nomi che iniziano con un trattino basso e continuano con una lettera maiuscola o un altro trattino basso.

Il concetto di nome esterno viene descritto a proposito della compilazione di un programma che si sviluppa in più file-oggetto da collegare assieme (sezione 66.3). L'altro vincolo serve a impedire, per esempio, la creazione di nomi come `'_Bool'` o `'__STDC_IEC_559__'`. Rimane quindi la possibilità di usare nomi che inizino con un trattino basso, purché continuino con un carattere minuscolo e siano visibili solo nell'ambito del file sorgente che si compone.

### 66.1.8 I/O elementare

«

L'input e l'output elementare che si usa nella prima fase di apprendimento del linguaggio C si ottiene attraverso l'uso di due funzioni fondamentali: *printf()* e *scanf()*. La prima si occupa di emettere una stringa dopo averla trasformata in base a dei codici di composizione determinati; la seconda si occupa di ricevere input (generalmente da tastiera) e di trasformarlo secondo codici di conversione simili alla prima. Infatti, il problema che si incontra inizialmente, quando si vogliono emettere informazioni attraverso lo standard output per visualizzarle sullo schermo, sta nella necessità di convertire in qualche modo tutti i dati che non siano già di tipo `'char'`. Dalla parte opposta, quando si inserisce un dato che non sia da intendere come un semplice carattere alfanumerico, serve una conversione adatta nel tipo di dati corretto.

Per utilizzare queste due funzioni, occorre includere il file di intestazione `'stdio.h'`, come è già stato visto più volte negli esempi.

Le due funzioni, *printf()* e *scanf()*, hanno in comune il fatto di disporre di una quantità variabile di parametri, dove solo il primo è stato precisato. Per questa ragione, la stringa che costituisce il primo argomento deve contenere tutte le informazioni necessarie a individuare quelli successivi; pertanto, si fa uso di *specificatori di conversione* che definiscono il tipo e l'ampiezza dei dati da trattare. A titolo di esempio, lo specificatore '**%i**' si riferisce a un valore intero di tipo '**int**', mentre '**%li**' si riferisce a un intero di tipo '**long int**'.

Vengono mostrati solo alcuni esempi, perché una descrizione più approfondita nell'uso delle funzioni *printf()* e *scanf()* appare in altre sezioni (67.3 e 69.17). Si comincia con l'uso di *printf()*:

```
...
double capitale = 1000.00;
double tasso    = 0.5;
int    interesse = (capitale * tasso) / 100;
...
printf ("%s: il capitale %f, ", "Ciao", capitale);
printf ("investito al tasso %f%% ", tasso);
printf ("ha prodotto un interesse pari a %i.\n", interesse);
...
```

Gli specificatori di conversione usati in questo esempio si possono considerare quelli più comuni: '**%s**' incorpora una stringa; '**%f**' traduce in testo un valore che originariamente è di tipo '**double**'; '**%i**' traduce in testo un valore '**int**'; inoltre, '**%%**' viene trasformato semplicemente in un carattere percentuale nel testo finale. Alla fine, l'esempio produce l'emissione del testo: «Ciao: il capitale 1000.00, investito al tasso 0.500000% ha prodotto un interesse pari a 5.»

La funzione *scanf()* è un po' più difficile da comprendere: la stringa che definisce il procedimento di interpretazione e conversione de-

ve confrontarsi con i dati provenienti dallo standard input. L'uso più semplice di questa funzione prevede l'individuazione di un solo dato:

```
...
int importo;
...
printf ("Inserisci l'importo: ");
scanf ("%i", &importo);
...
```

Il pezzo di codice mostrato emette la frase seguente e resta in attesa dell'inserimento di un valore numerico intero, seguito da [*Invio*]:

```
Inserisci l'importo: _
```

Questo valore viene inserito nella variabile *importo*. Si deve osservare il fatto che gli argomenti successivi alla stringa di conversione sono dei puntatori, per cui, avendo voluto inserire il dato nella variabile *importo*, questa è stata indicata preceduta dall'operatore '**&**' in modo da fornire alla funzione l'indirizzo corrispondente (si veda la sezione [66.5](#) sulla gestione dei puntatori).

Con una stessa funzione *scanf()* è possibile inserire dati per diverse variabili, come si può osservare dall'esempio seguente, ma in questo caso, per ogni dato viene richiesta la separazione con spazi orizzontali o anche con la pressione di [*Invio*].

```
printf ("Inserisci il capitale e il tasso:");
scanf ("%i%f", &capitale, &tasso);
```

## 66.1.9 Restituzione di un valore



In un sistema Unix e in tutti i sistemi che si rifanno a quel modello, i programmi, di qualunque tipo siano, al termine della loro esecuzione, restituiscono un valore che può essere utilizzato da uno script di shell per determinare se il programma ha fatto ciò che si voleva o se è intervenuto qualche tipo di evento che lo ha impedito.

Convenzionalmente si tratta di un valore numerico, con un intervallo di valori abbastanza ristretto, in cui zero rappresenta una conclusione normale, ovvero priva di eventi indesiderati, mentre qualsiasi altro valore rappresenta un'anomalia. A questo proposito si consideri quello «strano» atteggiamento degli script di shell, per cui zero equivale a *Vero*.

Lo standard del linguaggio C prescrive che la funzione *main()* debba restituire un tipo intero, contenente un valore compatibile con l'intervallo accettato dal sistema operativo: tale valore intero è ciò che dovrebbe lasciare di sé il programma, al termine del proprio funzionamento.

Se il programma deve terminare, per qualunque ragione, in una funzione diversa da *main()*, non potendo usare l'istruzione **return** per questo scopo, si può richiamare la funzione *exit()*:

```
exit (valore_restituito);
```

La funzione *exit()* provoca la conclusione del programma, dopo aver provveduto a scaricare i flussi di dati e a chiudere i file. Per questo motivo, non restituisce un valore all'interno del programma, al contrario, fa in modo che il programma restituisca il valore indicato

come argomento.

Per poterla utilizzare occorre includere il file di intestazione 'stdlib.h' che tra l'altro dichiara già due macro-variabili adatte a definire la conclusione corretta o errata del programma: *EXIT\_SUCCESS* e *EXIT\_FAILURE*.<sup>15</sup> L'esempio seguente mostra in che modo queste macro-variabili potrebbero essere usate:

```
#include <stdlib.h>
...
...
if (...)
{
    exit (EXIT_SUCCESS);
}
else
{
    exit (EXIT_FAILURE);
}
```

Naturalmente, se si può concludere il programma nella funzione *main()*, si può fare lo stesso con l'istruzione 'return':

```
#include <stdlib.h>
...
...
int main (...)
{
    ...
    if (...)
    {
        return (EXIT_SUCCESS);
    }
    else
    {
```



```

        return (EXIT_FAILURE);
    }
    ...
}

```

## 66.1.10 Attributi per GNU C

Il compilatore GNU C prevede l'uso di «attributi» nel proprio codice, come estensione del linguaggio. Dal momento che il compilatore GNU C è molto importante e diffuso, conviene sapere che forma possono avere tali attributi, almeno per non restare sbalorditi nella lettura del codice di altri autori:

```
__attribute__ ((tipo_di_attributo))
```

Frequentemente, questi attributi vanno collocati alla fine della dichiarazione di ciò a cui si riferiscono, come nell'esempio seguente, dove viene assegnato l'attributo '**deprecated**' al prototipo di una funzione:

```

...
mia_funzione (void) __attribute__ ((deprecated));
...

```

Se può servire, il nome dell'attributo può apparire anche preceduto e terminato da due trattini bassi; pertanto, l'esempio già visto può essere scritto anche così:

```

...
mia_funzione (void) __attribute__ ((__deprecated__));
...

```

Il fatto che siano previsti tali attributi dal compilatore GNU C, rende molto difficile l'individuazione di un errore frequente e banale: la mancanza del punto e virgola alla fine di un prototipo di funzione. Per esempio, si può supporre di avere realizzato un proprio file di intestazione con il contenuto seguente:

```
...  
mia_funzione_1 (int a, int b);  
mia_funzione_2 (int a, int b)  
mia_funzione_3 (int a, int b);  
mia_funzione_4 (int a, int b);  
...
```

Come si vede, il prototipo di *mia\_funzione\_2()* non è concluso con il punto e virgola. Durante la compilazione di un file che include questa porzione di codice, l'errore che viene evidenziato dal compilatore GNU C è incomprensibile, rispetto alla realtà effettiva:

```
In file included from ../lib/stdio.h:5,  
      from asctime.c:3:  
../lib/stdarg.h: In function 'asctime':  
../lib/stdarg.h:4: error: storage class specified for  
      parameter 'va_list'  
In file included from ../lib/stdio.h:10,  
      from asctime.c:3:  
../lib/sys/types.h:8: error: storage class specified for  
      parameter 'blkcnt_t'  
../lib/sys/types.h:9: error: storage class specified for  
      parameter 'blksize_t'  
../lib/sys/types.h:10: error: storage class specified for  
      parameter 'dev_t'  
...  
...  
../lib/stdio.h:94: error: expected declaration specifiers
```

```
or '...' before 'va_list'
.../lib/stdio.h:96: error: expected declaration specifiers
or '...' before 'va_list'
asctime.c:7: error: expected '=', ',', ';', 'asm' or
'__attribute__' before '{' token
asctime.c:99: error: old-style parameter declarations in
prototyped function definition
asctime.c:99: error: expected '{' at end of input
make: *** [asctime] Error 1
...
```

L'esempio mostrato si riferisce a un errore provocato volutamente nel file di intestazione `'time.h'`, a cui mai viene fatto riferimento nell'analisi del compilatore. Pertanto, di fronte a errori così incomprensibili, è determinante il controllo della conclusione corretta dei prototipi delle funzioni, all'interno dei file di intestazione prodotti per proprio conto.

## 66.2 Istruzioni del precompilatore

Il linguaggio C non può fare a meno del precompilatore e le sue direttive sono regolate dallo standard. Il precompilatore è un programma, o quella parte del compilatore, che si occupa di pre-elaborare un sorgente per generarne uno nuovo, il quale poi viene compilato con tutte le trasformazioni apportate. <<

Tradizionalmente, in un sistema operativo che si rifà al modello dei sistemi Unix, il precompilatore è costituito dal programma `'cpp'` che può essere utilizzato direttamente o in modo trasparente dal compilatore `'cc'`. Volendo simulare i passaggi iniziali della compilazione di un programma ipotetico denominato `'prg.c'`, evidenziando il ruolo del precompilatore, questi si potrebbero esprimere così:

```
$ cpp -E -o prg.i prg.c [Invio]
```

```
$ cc -o prg.o prg.i [Invio]
```

```
$ ...
```

In questo caso, il file ‘prg.i’ generato dal precompilatore è quello che viene chiamato dalla documentazione standard una *unità di traduzione*. Una unità di traduzione singola può essere il risultato della fusione di diversi file, incorporati attraverso le direttive ‘**#include**’, come viene descritto nel capitolo. Ciò che occorre osservare è che, quando si parla di campo di azione legato al «file», ci si riferisce al file generato dal precompilatore, ovvero all’unità di traduzione.

Va osservato che esistono programmi che utilizzano il precompilatore del linguaggio C per fini estranei al linguaggio stesso. Per esempio i file di configurazione delle risorse di X (il sistema grafico) vengono fatti elaborare da ‘**cpp**’ prima di essere interpretati.

### 66.2.1 Linguaggio a sé stante

«

Le direttive del precompilatore rappresentano un linguaggio a sé stante, con proprie regole. In generale:

- le direttive iniziano con il simbolo ‘**#**’, preferibilmente nella prima colonna;
- le direttive non utilizzano alcun simbolo di conclusione (non si usa il punto e virgola);
- ogni direttiva occupa una riga, la quale può essere spezzata e ripresa in righe successive, utilizzando il simbolo ‘**\**’ subito prima del codice di interruzione di riga;

- su una riga può essere inserita una sola direttiva, perché non c'è altro modo di distinguere la fine di una dall'inizio della successiva.

Se appare un simbolo '#' privo di altre indicazioni, questo viene semplicemente ignorato dal precompilatore. Di solito le direttive del precompilatore si scrivono senza annidamenti, ma questo fatto rischia di rendere particolarmente complicata la lettura del sorgente. A ogni modo, se si usano gli annidamenti, di solito questi riguardano solo le altre direttive e non il codice del linguaggio C puro e semplice.

I commenti del linguaggio C possono apparire solo alla fine delle direttive, ma non in tutte; pertanto vanno usati con prudenza. Vengono usati sicuramente alla fine delle direttive '#else' e '#endif' per ricordare a quale condizione si riferiscono.

### 66.2.2 Direttiva «#include»

La direttiva '#include' permette di includere un file. Generalmente si tratta di un cosiddetto *file di intestazione*, contenente una serie di definizioni necessarie al file sorgente in cui vengono incorporate. Il file da incorporare può essere indicato delimitandolo con le parentesi angolari, oppure con gli apici doppi; il modo in cui si delimita il nome del file serve a stabilire come questo deve essere cercato:<sup>16</sup>

```
#include <file>
```

```
#include "file"
```

I due esempi seguenti mostrano la richiesta di includere il file ‘`stdio.h`’ secondo le due forme possibili:

```
#include <stdio.h>
```

```
#include "stdio.h"
```

Delimitando il nome tra parentesi angolari si fa riferimento a un file che dovrebbe trovarsi in una posizione stabilita dalla configurazione del compilatore; per esempio, nel caso di GNU C in un sistema GNU/Linux, dovrebbe trattarsi della directory ‘`/usr/include/`’. Se invece si delimita il nome tra apici doppi, generalmente si fa riferimento a una posizione precisa nel file system, attraverso l’indicazione di un percorso (secondo la modalità prevista dal sistema operativo); pertanto, scrivendo il nome del file come nell’esempio, si dovrebbe intendere che la sua collocazione debba essere la directory corrente.

Di norma, quando si indica un file da includere delimitandolo con gli apici doppi e senza indicare alcun percorso, se questo file non si trova nella directory corrente, allora viene cercato nella directory predefinita, come se fosse stato indicato tra le parentesi angolari.

Un file incorporato attraverso la direttiva ‘`#include`’, può a sua volta fare lo stesso con altri; naturalmente, questa possibilità va considerata per evitare di includere più volte lo stesso file e di solito si usa un accorgimento che viene descritto più avanti nel capitolo.

### 66.2.3 Direttiva «#define»

La direttiva ‘**#define**’ serve a definire quelle che sono note come *macro*, ovvero delle variabili del precompilatore che, successivamente, il precompilatore stesso espande secondo regole determinate. Lo standard del linguaggio C distingue queste macro in due categorie: *object-like macro* e *function-like macro*. Nel corso di questi capitoli si usa la definizione di *macro-variabile* nel primo caso e di *macroistruzione* nel secondo.

Come sottoinsieme delle macro-variabili vengono considerate le *costanti manifeste*, per rappresentare dei valori semplici che si ripetono nel sorgente. Per esempio, *NULL* è la costante manifesta standard per rappresentare il puntatore nullo.

```
#define macro [sequenza_di_caratteri]
```

La direttiva ‘**#define**’ usata secondo la sintassi mostrata consente di definire delle macro-variabili, ovvero ciò che lo standard definisce *object-like macro*. Ciò che si ottiene è la sostituzione nel sorgente del nome indicato con la sequenza di caratteri che lo segue. Si osservi l’esempio seguente:

```
#define SALUTO Ciao! Come stai?
```

In questo caso viene dichiarata la macro-variabile *SALUTO* in modo tale che tutte le occorrenze di questo nome, successive alla sua dichiarazione, vengano sostituite con ‘**Ciao! Come stai?**’. È molto importante comprendere questo particolare: tutto ciò che appare dopo il nome della macro, a parte lo spazio che lo separa, viene utilizzato nella sostituzione. L’esempio seguente, invece rappresenta un

😊 programma completo.

Listato 66.68. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/8xkVUB59> , <http://ideone.com/HSVI2> .

```
#include <stdio.h>
#define SALUTO "Ciao! come stai?\n"
int main (void)
{
    printf (SALUTO);
    return 0;
}
```

In questo caso, la macro-variabile **SALUTO** può essere utilizzata in un contesto in cui ci si attende una stringa letterale, perché include gli apici doppi che sono necessari per questo scopo. Nell'esempio si vede l'uso della macro-variabile come argomento della funzione *printf()* e l'effetto del programma è quello di mostrare il messaggio seguente:

```
Ciao! come stai?
```

È bene precisare che la sostituzione delle macro-variabili non avviene se i loro nomi appaiono tra apici doppi, ovvero all'interno di stringhe letterali. Si osservi l'esempio seguente.





Listato 66.70. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/qfPSZZm0>, <http://ideone.com/CAAk3>.

```
#include <stdio.h>
#define SALUTO Ciao! come stai?
int main (void)
{
    printf ("SALUTO\n");
    return 0;
}
```

In questo caso, la funzione *printf()* emette effettivamente la parola ‘**SALUTO**’ e non avviene alcuna espansione di macro:

```
SALUTO
```

Una volta compreso il meccanismo basilare della direttiva ‘**#define**’ si può osservare che questa può essere utilizzata in modo più complesso, facendo anche riferimento ad altre macro già definite:

```
#define UNO 1
#define DUE UNO+UNO
#define TRE DUE+UNO
```

In presenza di una situazione come questa, utilizzando la macro **TRE**, si ottiene prima la sostituzione con ‘**DUE+UNO**’, quindi con ‘**UNO+UNO+1**’, infine con ‘**1+1+1**’ (dopo, tocca al compilatore).

Tradizionalmente i nomi delle macro-variabili vengono definiti utilizzando solo lettere maiuscole, in modo da poterli distinguere facilmente nel sorgente.

Come è possibile vedere meglio in seguito, è sensato anche dichiarare una macro senza alcuna corrispondenza. Ciò può servire per le direttive `#ifdef` e `#ifndef`.

Nella definizione di una macro-variabile può apparire l'operatore `##`, con lo scopo di attaccare ciò che si trova alle sue estremità. Si osservi l'esempio seguente.

Listato 66.73. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/XQ6Ns1AT>, <http://ideone.com/R5G3o>.

```
#include <stdio.h>
#define UNITO 1234 ## 5678
int main (void)
{
    printf ("%i\n", UNITO);
    return 0;
}
```

Eseguendo questo programma si ottiene semplicemente l'emissione del numero 12345678. Questo operatore può servire anche per unire assieme il nome di una macro-variabile, benché questo sia poco consigliabile.

Listato 66.74. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/lZ0QKzln>, <http://ideone.com/qOqC1>.

```
#include <stdio.h>
#define MIAMACRO 12345678
#define UNITO MI ## A ## MA ## CRO
int main (void)
{
    printf ("%i\n", UNITO);
    return 0;
}
```

## 66.2.4 Direttiva «#define» con parametri

La direttiva ‘**#define**’ può essere usata per creare una macroistruzione, ovvero una cosa che viene usata con l’apparenza di una funzione:

```
#define macro (parametro [, parametro] ...) sequenza_di_caratteri
```

Per comprendere il meccanismo è meglio avvalersi di esempi. In quello seguente, l’istruzione ‘**i = QUADRATO(i)**’ si traduce in ‘**i = (i) \* (i)**’:

```
#define QUADRATO(A)          (A) * (A)
...
...
i = QUADRATO (i);
...
```

Si osservi il fatto che, nella definizione, la stringa di sostituzione è stata composta utilizzando le parentesi: ciò permette di evitare problemi successivamente, nelle precedenze di valutazione del-

le espressioni, se l'argomento della funzione simulata attraverso la macroistruzione è composto:

```
...  
i = QUADRATO (123 * 34 + 3);  
...
```

In questo caso, la sostituzione genera `'i = (123 * 34 + 3) * (123 * 34 + 3)'` e si può vedere che le parentesi sono appropriate. L'esempio seguente, costituito da un programma completo, mostra l'uso di due parametri.

Listato 66.77. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/MAXBs8MQ>, <http://ideone.com/ML1LU>.

```
#include <stdio.h>  
#define MAX(X, Y) ((X) > (Y) ? (X) : (Y))  
int main (void)  
{  
    printf ("valore massimo tra %i e %i: %i\n",  
           3, 4, MAX (3, 4));  
    return 0;  
}
```

La macroistruzione `'MAX (3, 4)'` si traduce in `'((3) > (4) ? (3) : (4))'`.

È molto importante fare attenzione alla spaziatura nella dichiarazione di una macroistruzione: si può scrivere `#define MAX(x,y) ...`, `#define MAX( x,y) ...`, `#define MAX(x,y ) ...`, `#define MAX(x, y) ...`, ecc. Quello che invece non si può proprio è l’inserimento di uno spazio tra il nome della macroistruzione e la parentesi tonda aperta. Pertanto, se si scrive `#define MAX (x, y) ...` si commette un errore!

Al contrario, quando la macroistruzione viene richiamata, questo spazio può essere inserito senza problemi, come apparso già negli esempi.

Nella definizione di una macroistruzione può essere usato l’operatore `##` già descritto nella sezione precedente. Nell’esempio seguente si ottiene di visualizzare il numero 12345678.

Listato 66.78. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/GHXfnaHL> , <http://ideone.com/WRD5n> .

```
#include <stdio.h>
#define UNISCI(A, B) A ## B
int main (void)
{
    printf ("%i\n", UNISCI(1234, 5678));
    return 0;
}
```

Inoltre, è disponibile l’operatore `#` che ha lo scopo di racchiudere tra apici doppi la metavariable che lo segue immediatamente. Si osservi l’esempio seguente:

Listato 66.79. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/1nEG1ryz>, <http://ideone.com/czt2V>.

```
#include <stdio.h>
#define STRINGATO(a) # a
#define SALUTO STRINGATO (Ciao! come stai?\n)
int main (void)
{
    printf (SALUTO);
    return 0;
}
```

Prima viene definita la macroistruzione ***STRINGATO***, con la quale si vuole che il suo argomento sia raccolto tra apici doppi. Subito dopo viene definita la macro-variabile ***SALUTO*** che viene rimpiazzata da ‘***STRINGATO (Ciao! come stai?\n)***’ e quindi da “***Ciao! come stai?\n***”. Alla fine, il programma mostra regolarmente il messaggio già visto in un altro esempio precedente:

```
Ciao! come stai?
```

Si osservi cosa accadrebbe modificando l’esempio nel modo seguente, dove si vuole che la macroistruzione ***STRINGATO*** utilizzi due parametri:

```
...
#define STRINGATO(a, b) # a # b
#define SALUTO STRINGATO (Ciao!, come stai?\n)
...
```

Evidentemente si vuole che i due argomenti forniti alla macroistruzione ***STRINGATO*** siano raccolti ognuno tra apici doppi, pertanto la macro-variabile si trova a essere dichiarata, sostanzialmente come “***Ciao! "come stai?\n"***”. Alla fine il risultato mostrato dal

programma è differente, perché la sequenza delle due stringhe viene intesa come una sequenza sola, ma in tal caso manca lo spazio tra le due parti:

```
Ciao!come stai?
```

Si può complicare ulteriormente l'esempio per dimostrare fino a dove si estende la competenza dell'operatore '#', come si vede nel listato successivo.

Listato 66.83. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/idGYtl78>, <http://ideone.com/17IQ17c>.

```
#include <stdio.h>
#define STRINGATO(a, b) # a , b
#define SALUTO STRINGATO (%i un amore\n, 6)
int main (void)
{
    printf (SALUTO);
    return 0;
}
```

Qui gli spazi sono importanti, infatti, la macroistruzione **STRINGATO** si traduce in `"a" , b` e la virgola non avrebbe potuto essere unita alla lettera «a», altrimenti sarebbe stata inserita dentro la coppia di apici doppi. La macro-variabile **SALUTO** si traduce poi in `"%i un amore\n" , 6`, pertanto, alla fine, il programma mostra il messaggio seguente:

```
6 un amore
```

Per concludere viene mostrato un esempio ulteriore, con il quale si crea una sorta di funzione che il precompilatore deve trasformare in un blocco di istruzioni. Viene simulato il comportamento della

funzione standard *strncpy()*, senza però restituire un valore.

Listato 66.85. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/eCbQbPAR> , <http://ideone.com/A2Q2L> .

```
#include <stdio.h>

#define STRNCPY(DST, ORG, N) { \
    char *s1 = (DST); \
    const char *s2 = (ORG); \
    size_t n = (N); \
    int i; \
    for (i = 0; i < n && s2[i] != 0; i++) \
        { \
            s1[i] = s2[i]; \
        } \
    s1[i] = 0; }

int main (void)
{
    char stringa[100];
    STRNCPY (stringa, "Buon giorno a tutti!", 50)    // [1]
                                                    // [1] Si osservi che manca il
                                                    //      punto e virgola finale!

    printf ("%s\n", stringa);
    return 0;
}
```

Si può vedere che, per richiamare questa macroistruzione, non si richiede che le sia aggiunto il punto e virgola. Infatti, la macro in sé si espande in un raggruppamento tra parentesi graffe, che non ne ha bisogno; d'altra parte, volendoglielo aggiungere, non si può creare alcun problema.



La dichiarazione di una macroistruzione può prevedere una quantità variabile di parametri, come avviene già per le funzioni (sezione 66.6.3). Per ottenere questo si aggiungono dei puntini di sospensione alla fine dell'elenco dei parametri fissi, quindi, si utilizza la parola chiave '**\_\_VA\_ARGS\_\_**' per individuare gli argomenti opzionali. L'esempio seguente riproduce il funzionamento di *printf()*, richiamando la stessa funzione.

Listato 66.86. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/j98SG985Wo> , <http://ideone.com/cOKIA> .

```
#include <stdio.h>

#define PRINTF(A, ...) printf (A, __VA_ARGS__)

int main (void)
{
    PRINTF ("I primi numeri interi: %i, %i, %i\n", 1, 2, 3);
    return 0;
}
```

Questa volta il punto e virgola finale serve, perché non è stato incluso nella definizione della macroistruzione.

A proposito di '**\_\_VA\_ARGS\_\_**' va ancora osservato che individua sì gli argomenti opzionali, ma di questi ne deve essere specificato almeno uno. Pertanto, la macroistruzione *PRINTF()*, per come è stata dichiarata nell'esempio precedente, va usata sempre con almeno due argomenti. In questo caso, per poter usare la macroistruzione con un argomento solo, la sua definizione va modificata nel modo seguente:

```
...  
#define PRINTF(...) printf (__VA_ARGS__)  
...
```

## 66.2.5 Direttive «#if», «#else», «#elif» e «#endif»

&lt;&lt;

Le direttive '**#if**', '**#else**', '**#elif**' e '**#endif**', permettono di delimitare una porzione di codice che debba essere utilizzato o ignorato in relazione a una certa espressione che può essere calcolata solo attraverso definizioni precedenti.

```
#if espressione  
    espressione  
[#elif espressione  
    espressione ]  
...  
[#else  
    espressione ]  
#endif
```

Le espressioni che rappresentano le condizioni da valutare seguono regole equivalenti a quelle del linguaggio, tenendo conto che se si vogliono usare delle variabili, queste possono solo essere quelle del precompilatore. L'esempio seguente mostra la dichiarazione di una macro-variabile a cui si associa un numero, quindi si vede un confronto basato sul valore in cui si espande la macro-variabile stessa:

```
#define DIM_MAX 1000  
...  
...
```

```
int main (void)
{
...
#if DIM_MAX>100
    printf ("Dimensione enorme.\n");
    ...
#else
    printf ("Dimensione normale.\n");
    ...
#endif
...
}
```

L'esempio mostra il confronto tra la macro-variabile ***DIM\_MAX*** e il valore 100. Essendo stata dichiarata per tradursi in 1 000, il confronto è equivalente a  $1\,000 > 100$  che risulta vero, pertanto il compilatore include solo le istruzioni relative.

Gli operatori di confronto che si possono utilizzare per le espressioni logiche sono i soliti, in particolare, è bene ricordare che per valutare l'uguaglianza si usa l'operatore '==', come nell'esempio successivo:

```
#define NAZIONE ita
...
...
int main (void)
{
#if NAZIONE==ita
    char valuta[] = "EUR";
    ...
#elif NAZIONE==usa
    char valuta[] = "USD";
    ...
#endif
}
```

```
...  
}
```

Queste direttive condizionali possono essere annidate; inoltre possono contenere anche altri tipi di direttiva del precompilatore.

### 66.2.6 Direttive «`#if defined`», «`#if !defined`», «`#ifdef`» e «`#ifndef`»

«

Nelle espressioni che esprimono una condizione per la direttiva '`#if`' è possibile usare l'operatore '`defined`', seguito dal nome di una macro-variabile. La condizione '`defined macro`' si avvera se la macro indicata risulta definita, anche se dovesse essere priva di valore. Per converso, la condizione '`!defined macro`' si avvera quando la macro non risulta definita.

La direttiva '`#if defined`' può essere abbreviata come '`#ifdef`', mentre '`#if !defined`' si può esprimere come '`#ifndef`'.

```
#define DEBUG  
...  
int main (void)  
{  
...  
#if defined DEBUG  
    printf ("Punto di controllo n. 1\n");  
    ...  
#endif // DEBUG  
...  
}
```

```
#define DEBUG
...
int main (void)
{
...
#ifdef DEBUG
    printf ("Punto di controllo n. 1\n");
    ...
#endif // DEBUG
...
}
```

I due esempi equivalenti mostrano il caso in cui sia dichiarata una macro **DEBUG** (che non si traduce in alcunché) e in base alla sua esistenza viene incluso il codice che mostra un messaggio particolare.

```
#define OK
...
int main (void)
{
...
#if !defined OK
    printf ("Punto di controllo n. 1\n");
    ...
#endif // OK
...
}
```

```
#define OK
...
int main (void)
{
#ifdef OK
    printf ("Punto di controllo n. 1\n");
    ...
#endif // OK
...
}
```

Questi due esempi ulteriori sono analoghi a quanto già mostrato, con la differenza che le istruzioni controllate vengono incluse nella compilazione solo se la macro indicata non è stata dichiarata.

Quando si scrivono delle condizioni basate sull'esistenza o meno di una macro, può essere utile aggiungere alla conclusione un commento con cui si ricorda a quale macro si sta facendo riferimento, in modo da districarsi più facilmente in presenza di più livelli di annidamento. Ma occorre fare molta attenzione, perché se si commettono errori con questi commenti il compilatore non può dare alcuna segnalazione in merito e si rende incomprensibile il sorgente alla rilettura successiva.

Esiste una situazione ricorrente in cui viene utilizzata la direttiva `#if !defined` o `#ifndef` che è bene conoscere. Spesso i file di intestazione che vengono inclusi con direttive `#include` includono a loro volta tutto quello che serve loro, ma così facendo c'è la possibilità che lo stesso file venga incluso più volte. Per evitare di prendere in considerazione una seconda volta lo stesso file, si usa

un artificio molto semplice, come si vede nel listato successivo che riproduce il contenuto del file `'stdbool.h'` di una libreria standard ipotetica:

```
#ifndef _STDBOOL_H
#define _STDBOOL_H      1

#define bool      _Bool
#define true      1
#define false     0
#define __bool_true_false_are_defined  1

#endif // _STDBOOL_H
```

Come si vede, se il codice viene eseguito per la prima volta, la condizione `'ifndef _STDBOOL_H'` non si avvera e di conseguenza la macro-variabile `_STDBOOL_H` viene creata effettivamente e quindi viene considerato tutto il resto del codice fino alla direttiva `'#endif'`. Ma quando si tenta di eseguire lo stesso codice per la seconda volta, o per altre volte successive, dato che la macro-variabile `_STDBOOL_H` risulta già definita, questo codice viene ignorato semplicemente, senza altre conseguenze.

Le direttive che consentono di compilare selettivamente solo una porzione del codice, consentono di realizzare del codice molto sofisticato, ma rischiano di renderlo estremamente complesso da interpretare attraverso la lettura umana. Pertanto, è bene limitarne l'uso alle situazioni che sono utili effettivamente.

## 66.2.7 Direttiva «#undef»

&lt;&lt;

La direttiva ‘**#undef**’ permette di eliminare una macro a un certo punto del sorgente:

```
#undef macro
```

Si mostra un esempio molto semplice, nel quale prima si dichiara la macro-variabile *NAZIONE*, poi, quando non serve più, questa viene eliminata.

```
#define NAZIONE ita
...
/* In questa posizione, NAZIONE risulta definita */
...
#undef NAZIONE
...
/* In questa posizione, NAZIONE non è definita */
...
```

## 66.2.8 Direttiva «#line»

&lt;&lt;

Di norma, il compilatore abbastanza evoluto consente di inserire nel file eseguibile delle informazioni che consentano di abbinare il codice eseguibile alle righe del file sorgente originale. Per esempio, con GNU C si può usare l’opzione ‘**-gstabs**’ e altre simili. Naturalmente, in condizioni normali il compilatore conta da solo le righe e annota il nome del file sorgente originale.

Con la direttiva ‘**#line**’ è possibile istruire il compilatore in modo che tenga in considerazione un numero di riga differente, ma soprattutto consente di specificare a quale file sorgente ci si vuole riferire.



```
#line n_riga ["nome_file_sorgente"]
```

C'è da osservare che, per il programmatore, è poco probabile che sia necessario indicare una riga diversa nello stesso sorgente. In effetti, diventa più utile se si abbina il nome di un altro file. Per comprendere come possa essere utilizzata questa possibilità, occorre ipotizzare la costruzione di un altro compilatore per un linguaggio nuovo, con il quale si genera codice in linguaggio C. A titolo di esempio si suppone di volere tradurre il file 'hanoi.pseudo' che si vede nel listato 66.96 in un sorgente C, denominato 'hanoi.c', mantenendo il riferimento alle righe originali.

Listato 66.96. Il file 'hanoi.pseudo'.

```
1      HANOI (N, P1, P2)
2          IF N > 0
3              THEN
4                  HANOI (N-1, P1, 6-P1-P2)
5                  scrivi: "Muovi l'anello" N "dal piolo" P1 "al piolo" P2
6                  HANOI (N-1, 6-P1-P2, P2)
7          END IF
8      END HANOI
9
10     MAIN ()
11         HANOI (3, 1, 2)
12     END MAIN
```

Per ottenere il risultato atteso, il file 'hanoi.c' deve contenere diverse direttive '#line', come si vede nel listato 66.97, anche se alcune di quelle potrebbero essere omesse, contando sull'incremento automatico da parte del compilatore.

## Listato 66.97. Il file 'hanoi.c'.

```
#include <stdio.h>

#line 1 "hanoi.pseudo"
void hanoi (int N, int P1, int P2)
{
    #line 2 "hanoi.pseudo"
    if (N > 0)
        {
            #line 4 "hanoi.pseudo"
            hanoi (N-1, P1, 6-P1-P2);
            #line 5 "hanoi.pseudo"
            printf ("Muovi l'anello %i dal piolo %i al piolo %i\n", N, P1, P2);
            #line 6 "hanoi.pseudo"
            hanoi (N-1, 6-P1-P2, P2);
            #line 7 "hanoi.pseudo"
        }
    #line 8 "hanoi.pseudo"
}

#line 10 "hanoi.pseudo"
int main (void)
{
    #line 11 "hanoi.pseudo"
    hanoi (3, 1, 2);
    #line 12 "hanoi.pseudo"
    return 0;
    #line 12 "hanoi.pseudo"
}
```

La compilazione del file 'hanoi.c' potrebbe avvenire nel modo seguente:

```
$ cc -Wall -gstabs hanoi.c
```

Si dovrebbe ottenere il file eseguibile 'a.out' e si verifica sommariamente se funziona:

```
$ ./a.out
```

```
Muovi l'anello 1 dal piolo 1 al piolo 2
Muovi l'anello 2 dal piolo 1 al piolo 3
Muovi l'anello 1 dal piolo 2 al piolo 3
Muovi l'anello 3 dal piolo 1 al piolo 2
Muovi l'anello 1 dal piolo 3 al piolo 1
Muovi l'anello 2 dal piolo 3 al piolo 2
Muovi l'anello 1 dal piolo 1 al piolo 2
```

Il risultato è quello previsto. Se lo si esegue con l'ausilio di programmi come GDB, si può osservare che il riferimento al sorgente originale è quello del file `'hanoi.pseudo'`:

```
$ gdb a.out
```

```
(gdb) break main [Invio]
```

```
Breakpoint 1 at 0x80483d8: file hanoi.pseudo, line 11.
```

```
(gdb) run [Invio]
```

```
Starting program: /home/tizio/a.out
```

```
...
```

```
Breakpoint 1, main () at hanoi.pseudo:11
```

```
11          HANOI (3, 1, 2)
```

```
(gdb) stepi [Invio]
```

```
0x080483e0    11          HANOI (3, 1, 2)
```

```
(gdb) stepi [Invio]
```

```
0x080483e8    11          HANOI (3, 1, 2)
```

```
(gdb) stepi [Invio]
```

```
0x080483ef    11          HANOI (3, 1, 2)
```

```
(gdb) stepi [Invio]
```

```
hanoi (n=3, p1=1, p2=2) at hanoi.pseudo:2  
2          IF N > 0
```

```
(gdb) stepi [Invio]
```

```
0x08048355      2          IF N > 0
```

```
(gdb) stepi [Invio]
```

```
0x08048357      2          IF N > 0
```

```
(gdb) stepi [Invio]
```

```
2          IF N > 0
```

```
(gdb) stepi
```

```
0x0804835e      2          IF N > 0
```

```
(gdb) stepi [Invio]
```

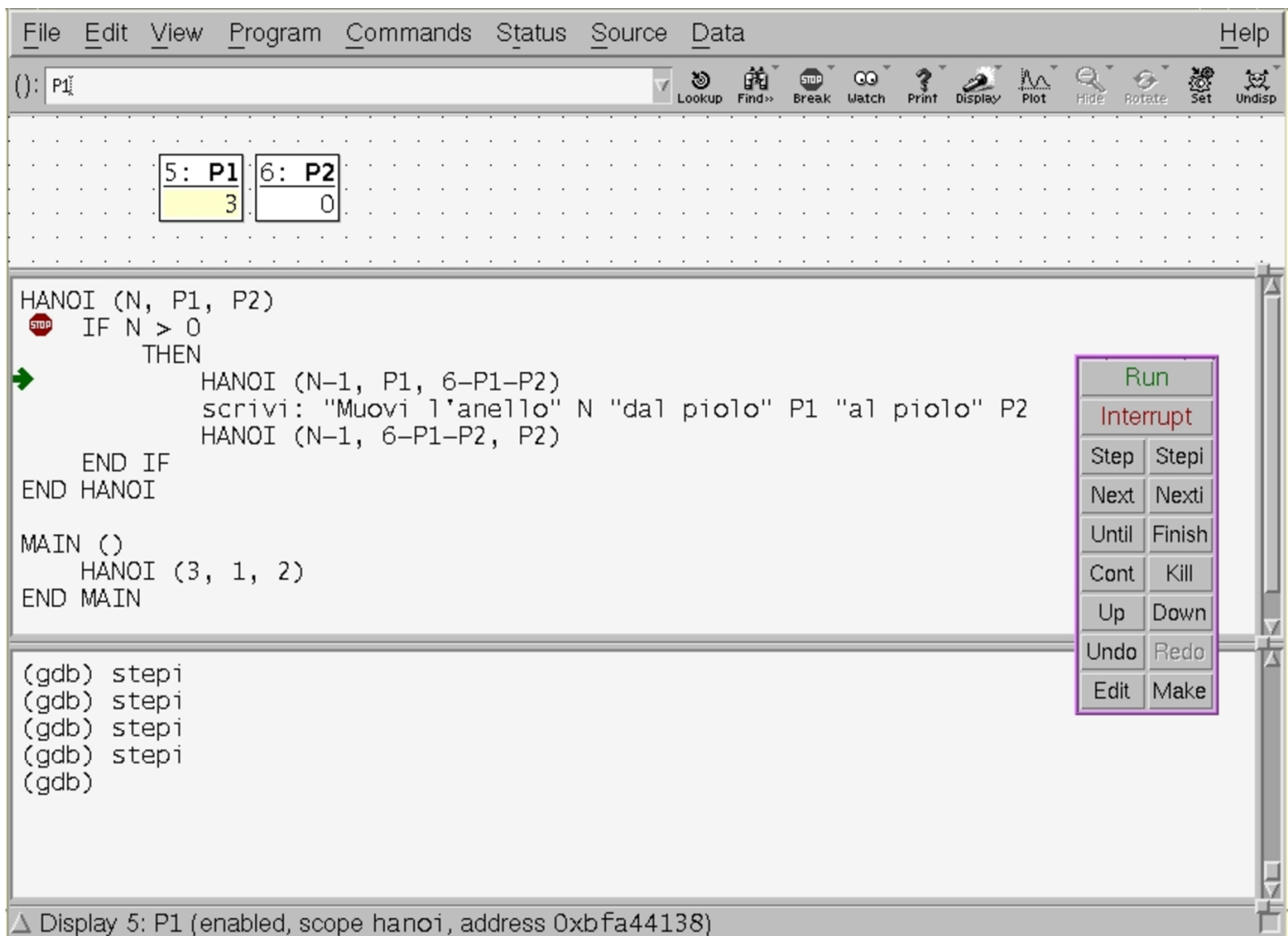
```
4          HANOI (N-1, P1, 6-P1-P2)
```

```
(gdb) stepi [Invio]
```

```
0x08048363      4          HANOI (N-1, P1, 6-P1-P2)
```

```
(gdb) quit [Invio]
```

Figura 66.110. Esecuzione controllata del programma attraverso DDD.



## 66.2.9 Direttiva «#error»

La direttiva '**#error**' serve a generare un messaggio diagnostico in fase di compilazione, normalmente con lo scopo di interrompere lì il procedimento. In pratica è un modo per interrompere la compilazione già in fase di elaborazione da parte del precompilatore, al verificarsi di certe condizioni.

```
#error messaggio
```

Il messaggio viene trattato in modo letterale, senza l'espansione delle macro.

```
#if ! __STDC_IEC_559__
#error compilatore non conforme alle specifiche IEC 60599!
#endif
```

L'esempio mostra una situazione verosimile per l'utilizzo della direttiva '**#error**', dove si controlla che il valore in cui si espande la macro-variabile `__STDC_IEC_559__` sia diverso da zero, ma se non è così viene visualizzato il messaggio di errore e la compilazione dovrebbe venire interrotta.

## 66.2.10 Macro predefinite

<<

Lo standard del C prevede che il compilatore disponga di alcune macro-variabili predefinite, elencate sinteticamente nella tabella successiva.

Tabella 66.112. Macro-variabili predefinite secondo lo standard.

| Macro-variabile              | Descrizione                                                                                                                                                                                                                                                                                                                                                 |
|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>__DATE__ __TIME__</pre> | <p>La data e l'ora della compilazione sono accessibili attraverso le macro-variabili <code>__DATE__</code> e <code>__TIME__</code>. Il formato della prima macro-variabile è "<i>Mmm gg aaaa</i>" e quello della seconda è "<i>hh:mm:ss</i>". Come si vede, le due macro-variabili si espandono in una stringa delimitata correttamente da apici doppi.</p> |

| Macro-variabile                                                | Descrizione                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|----------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>__FILE__</p> <p>__LINE__</p>                                | <p>Attraverso le macro-variabili <b>__FILE__</b> e <b>__LINE__</b> il programma può accedere all'informazione sul nome del file sorgente e della riga originale. Il nome del file e il numero della riga possono essere alterati attraverso la direttiva '<b>#line</b>'.</p>                                                                                                                                                                                                                                                                                                   |
| <p>__STDC__</p> <p>__STDC_HOSTED__</p> <p>__STDC_VERSION__</p> | <p>La macro-variabile <b>__STDC__</b> che si espande nel valore '1' sta a indicare che si tratta di un compilatore conforme allo standard; la macro <b>__STDC_HOSTED__</b>, se si espande nel valore '1', indica una conformità stretta, definita come <i>hosted implementation</i>; la macro <b>__STDC_VERSION__</b> si espande nella versione dello standard. Il valore in cui si espande la terza macro-variabile contiene l'anno e il mese, come per esempio '<b>199901L</b>', con la specificazione che si tratta di una costante numerica di tipo '<b>long int</b>'.</p> |

| Macro-variabile          | Descrizione                                                                                                                                                                                                                                                                                                                                             |
|--------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| __STDC_IEC_559__         | Se esiste la macro-variabile <b>__STDC_IEC_599__</b> che si espande nel valore '1', si intende indicare la conformità alle specifiche dello standard IEC 60559, inerenti l'aritmetica a virgola mobile.                                                                                                                                                 |
| __STDC_IEC_559_COMPLEX__ | Se esiste la macro-variabile <b>__STDC_IEC_599_COMPLEX__</b> che si espande nel valore '1', si intende indicare la conformità alle specifiche dello standard IEC 60559, inerenti l'aritmetica «complessa».                                                                                                                                              |
| __STDC_ISO_10646__       | Se esiste la macro-variabile <b>__STDC_ISO_10646__</b> , questa dovrebbe espandersi nella versione dello standard ISO/IEC 10646 che riguarda la codifica universale dei caratteri. La versione che si ottiene è un numero contenente l'anno e il mese, seguito dalla lettera «L», a indicare che si tratta di una costante numerica di tipo 'long int'. |

A parte il caso di **\_\_FILE\_\_** e **\_\_LINE\_\_**, le macro-variabili si espandono in un valore fisso.



## 66.2.11 Pragma

Attraverso i «pragma» è possibile dare al compilatore delle istruzioni che sono al di fuori dello standard. Il pragma, in sé, è un messaggio testuale che viene passato al compilatore, il quale può interpretarlo in fase di precompilazione o in quella successiva. Lo standard prevede due forme per esprimere un pragma al compilatore:

```
#pragma messaggio
```

```
_Pragma ("messaggio")
```

Il testo che compone il pragma nella sua prima forma viene trattato letteralmente, mentre quello del secondo modello richiede la protezione di alcuni caratteri: ‘\”’ e ‘\\’ corrispondono rispettivamente a ‘”’ e ‘\’. I due esempi seguenti sono equivalenti:

```
#pragma GCC dependency "parse.y"
```

```
_Pragma ("GCC dependency \"parse.y\"")
```

Lo standard prevede anche che sia possibile creare delle macroistruzioni che incorporino un pragma, come nell’esempio seguente:

```
#define DO_PRAGMA(x) _Pragma (#x)  
DO_PRAGMA (GCC dependency "parse.y")
```

## 66.3 Dal campo di azione alla compilazione

«

Il problema del campo di azione di variabili e funzioni va visto assieme a quello della compilazione di un programma composto da più file sorgenti, attraverso la produzione di file-oggetto distinti. Leggendo questo capitolo occorre tenere presente che la descrizione della questione è semplificata, omettendo alcuni dettagli. D'altra parte, per poter comprendere il problema la semplificazione è necessaria, tenendo conto che nel linguaggio C, per controllare il campo di azione delle variabili e delle funzioni, si utilizzano parole chiave non proprio «azzeccate» e in certi casi con significati diversi in base al contesto.

Per una descrizione più precisa e dettagliata, dopo la lettura di questo capitolo è necessario rivolgersi ai documenti che definiscono lo standard del linguaggio.

### 66.3.1 Il punto di vista del «collegatore»

«

Il programma che raccoglie assieme diversi file-oggetto per creare un file eseguibile (ovvero il *linker*), deve «collegare» i riferimenti incrociati a simboli di variabili e funzioni. In pratica, se nel file 'uno.o' si fa riferimento alla funzione  $f()$  dichiarata nel file 'due.o', nel programma risultante tale riferimento deve essere risolto con degli indirizzi appropriati. Naturalmente, lo stesso vale per le variabili globali, dichiarate da una parte e utilizzate anche dall'altra.

Per realizzare questi riferimenti incrociati, occorre che le variabili e le funzioni utilizzate al di fuori del file-oggetto in cui sono dichiarate, siano pubblicizzate in modo da consentire il richiamo da altri file-oggetto. Per quanto riguarda invece le variabili e le funzioni

dichiarate e utilizzate esclusivamente nello stesso file-oggetto, non serve questa forma di pubblicità.

Nei documenti che descrivono il linguaggio C standard si usa una terminologia specifica per distinguere le due situazioni: quando una variabile o una funzione viene dichiarata e usata solo internamente al file-oggetto rilocabile che si ottiene, è sufficiente che abbia una «collegabilità interna», ovvero un *linkage interno*; quando invece la si usa anche al di fuori del file-oggetto in cui viene dichiarata, richiede una «collegabilità esterna», ovvero un *linkage esterno*.

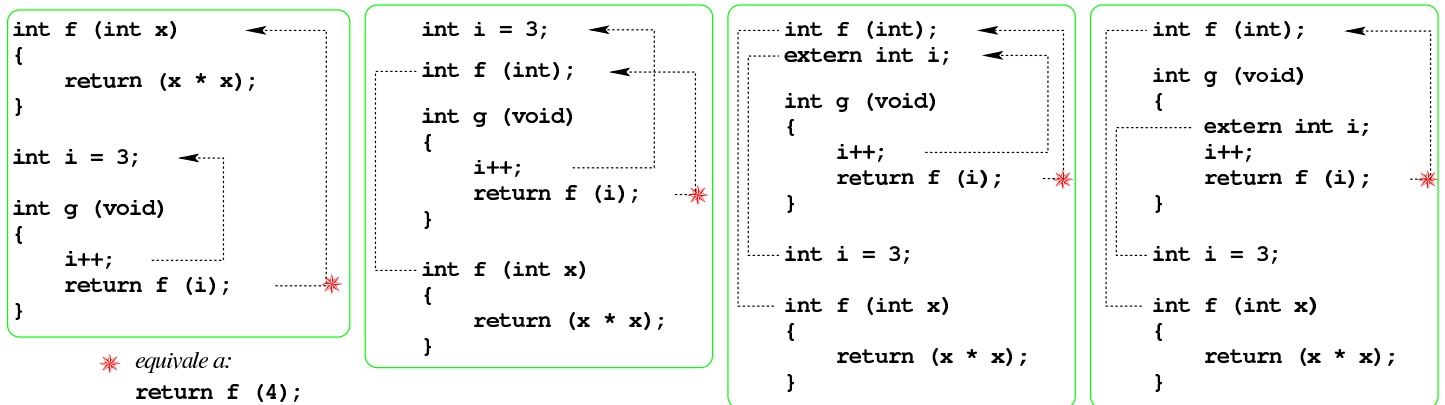
Nel linguaggio C, il fatto che una variabile o una funzione sia accessibile al di fuori del file-oggetto rilocabile che si ottiene, viene determinato in modo implicito, in base al contesto, nel senso che non esiste una classe di memorizzazione esplicita per definire questa cosa.

### 66.3.2 Campo di azione legato al file sorgente

Il file sorgente che si ottiene dopo l'elaborazione da parte del pre-compilatore, è suddiviso in componenti costituite essenzialmente dalla dichiarazione di variabili e di funzioni (prototipi inclusi). L'ordine in cui appaiono queste componenti determina la *visibilità* reciproca: in linea di massima si può accedere solo a quello che è già stato dichiarato. Inoltre, in modo predefinito, dopo la trasformazione in file-oggetto, queste componenti sono accessibili anche da altri file, per i quali, l'ordine di dichiarazione nel file originale non è più importante.<sup>17</sup>



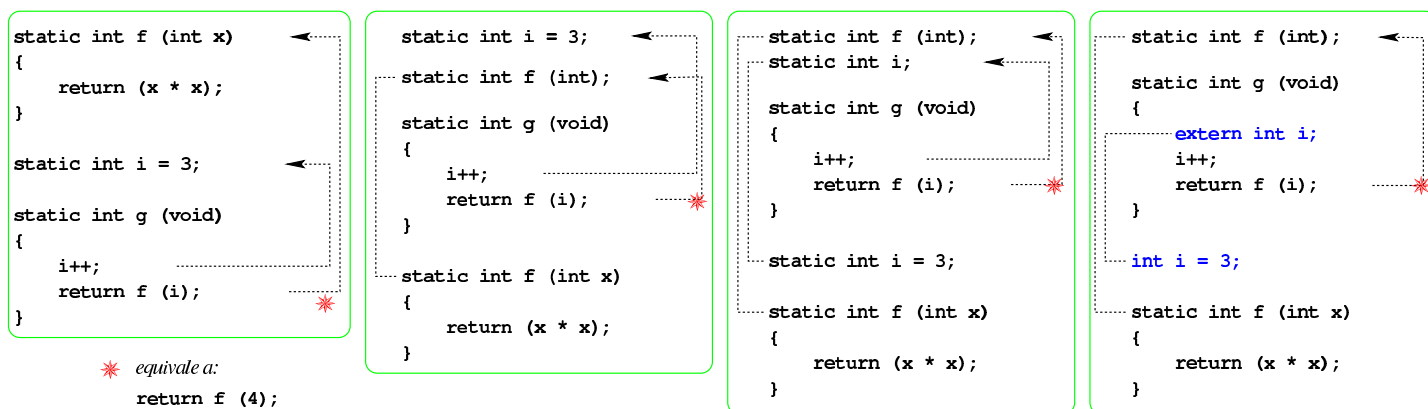
Figura 66.116. Quattro file sorgenti equivalenti, a confronto. La variabile  $i$ , la funzione  $f()$  e la funzione  $g()$  sarebbero accessibili anche da altri file. La funzione  $g()$  utilizza la variabile  $i$ , dichiarata esternamente a lei.



Nell'esempio della figura precedente, la funzione  $g()$  accede direttamente alla variabile  $i$  che risulta dichiarata al di fuori della funzione stessa. Il campo di azione di questa variabile inizia dalla sua dichiarazione e termina alla fine del file; quando la variabile viene definita in una posizione successiva al suo utilizzo, questa deve essere dichiarata preventivamente come «esterna», attraverso lo specificatore di classe di memorizzazione '**extern**'.

Per isolare le funzioni e la variabile degli esempi mostrati, in modo che non siano disponibili per il collegamento con altri file, si dichiarano per il solo uso locale attraverso lo specificatore di classe di memorizzazione '**static**', come si vede nella figura successiva. Va osservato che, nell'ultimo caso, la variabile  $i$  non può essere isolata dall'esterno, perché si trova in una posizione successiva al suo utilizzo, pertanto vi si accede come se fosse dichiarata in un altro file.

Figura 66.117. Quattro file sorgenti equivalenti a confronto, in cui, dove è stato possibile, le variabili e le funzioni sono state isolate dal collegamento con l'esterno.



Per accedere a una funzione o a una variabile definita in un altro file<sup>18</sup> si deve dichiarare localmente la funzione o la variabile con lo specificatore di classe di memorizzazione **'extern'**. La figura successiva mostra l'esempio già apparso, ma diviso in due file.

Figura 66.118. Due file collegati tra di loro: il primo file («a») viene proposto in due versioni equivalenti.

*file a*

```

➤ extern int f (int)
➤ extern int i;

int g (void)
{
    i++;
    return f (i); *
}

```

*file b*

```

int f (int x)
{
    return (x * x);
}

int i = 3;

```

\* *equivale a:*  
**return f (4);**

*file a*

```

➤ extern int f (int)

int g (void)
{
    extern int i;
    i++;
    return f (i); *
}

```

*file b*

```

int f (int x)
{
    return (x * x);
}

int i = 3;

```

Questi esempi mostrano che è possibile dichiarare la variabile «esterna» direttamente all'interno della funzione che ne fa uso; tuttavia, per la scrittura di un programma ordinato, è più grazioso se questa dichiarazione appare al di fuori delle funzioni.

Negli esempi mostrati non appare la funzione *main()* che, invece, in un programma comune deve esistere. È da osservare che la funzione *main()* non può essere dichiarata con lo specificatore di classe di memorizzazione 'static', anche se tutto è incluso in un file unico, perché dopo la produzione del file-oggetto rilocabile, per produrre un file eseguibile si associano normalmente delle librerie che contengono il codice iniziale del programma, il quale va a chiamare poi

la funzione *main()*. In altre parole, la compilazione prevede quasi sempre l'associazione con un file-oggetto fantasma contenente il codice responsabile della chiamata della funzione *main()*, la quale, così, deve essere accessibile all'esterno del proprio file.

Tabella 66.119. Specificatori di classe di memorizzazione utilizzabili nella dichiarazione delle funzioni e delle variabili al di fuori delle funzioni.

| Parola chiave       | Descrizione                                                                                                                                                                 |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                     | L'assenza dello specificatore di classe implica la dichiarazione di una variabile o di una funzione accessibile anche da altri file.                                        |
| <code>static</code> | Lo specificatore di classe ' <b>static</b> ' definisce una variabile o una funzione che può essere utilizzata solo all'interno del file in cui appare.                      |
| <code>extern</code> | Indica il riferimento a una variabile o a una funzione dichiarata in un altro file, oppure, nel caso delle variabili, anche nel file stesso ma in una posizione successiva. |

### 66.3.3 Semplificazione dovuta all'uso comune dei file di intestazione

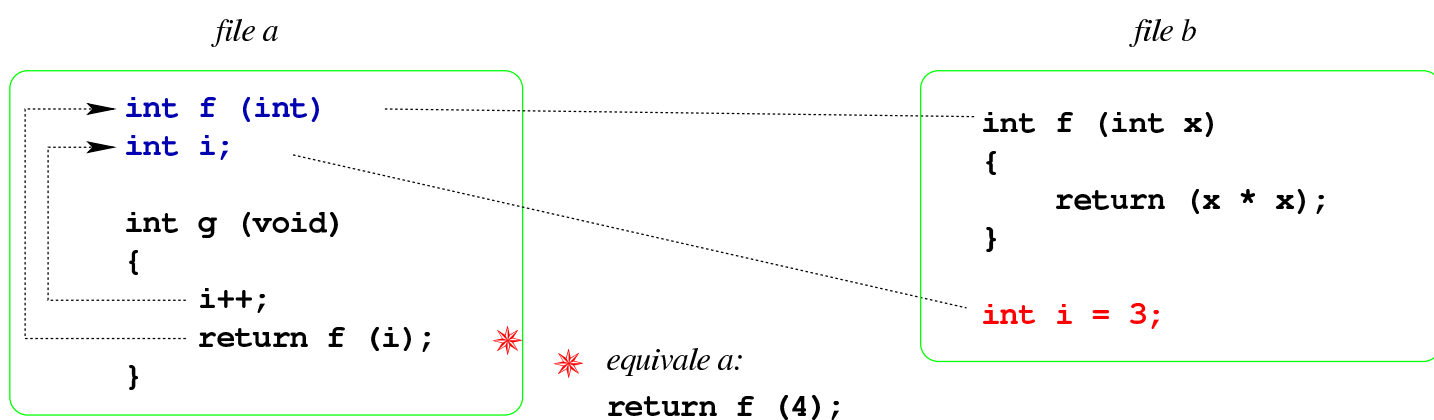
Nella tradizione del linguaggio C si fa uso di file di intestazione, ovvero porzioni di codice, in cui, tra le altre cose, si vanno a mettere i prototipi delle funzioni e le dichiarazioni delle variabili globali, a cui tutto il programma deve poter accedere.

Per semplificare questo lavoro di fusione, spesso un file incluso ne include automaticamente altri, da cui il proprio codice può dipendere. Così facendo, può anche succedere che lo stesso prototipo o la

stessa variabile appaiano dichiarati più volte nello stesso file finale (quello generato dal precompilatore).

Oltre a questo fatto, se il proprio programma è suddiviso in più file, i quali devono includere questo o quel file di intestazione, diventa impossibile precisare da quale parte i prototipi e le variabili vengono dichiarate e da quale altra parte vengono richiamate. Pertanto, di norma si lascia fare al compilatore. L'esempio di compilazione di due file, presentato alla fine della sezione precedente, va rivisto secondo quanto si vede nella figura successiva.

Figura 66.120. Due file collegati tra di loro senza dichiarare espressamente la classe di memorizzazione **'extern'**.



Naturalmente, è bene che le funzioni e le variabili pubbliche siano dichiarate sempre nello stesso modo; inoltre, se le variabili pubbliche devono essere inizializzate, ciò può avvenire una volta sola, in un solo file.



La classe di memorizzazione ‘**extern**’ è predefinita per i prototipi di funzione (purché non siano incorporati all’interno di altre funzioni) e per la dichiarazione delle variabili, purché assieme alla dichiarazione non ci sia anche un’inizializzazione. In pratica, nell’esempio non si può dichiarare espressamente con la parola chiave ‘**extern**’ la variabile *i* nel file ‘b’, dove viene anche inizializzata. Se si tenta di farlo, il compilatore dovrebbe segnalare un errore.

#### 66.3.4 Campo di azione interno alle funzioni

All’interno delle funzioni sono accessibili le variabili globali dichiarate esternamente a loro (come descritto nella sezione precedente), inoltre sono dichiarate implicitamente le variabili che costituiscono i parametri, dai quali si ricevono gli argomenti della chiamata, e si possono aggiungere altre variabili «locali». I parametri e le altre variabili che si dichiarano nella funzione sono visibili solo nell’ambito della funzione stessa; inoltre, se i nomi delle variabili e dei parametri sono gli stessi di variabili dichiarate esternamente, ciò rende temporaneamente inaccessibili quelle variabili esterne.

In condizioni normali, sia le variabili che costituiscono i parametri, sia le altre variabili dichiarate localmente all’interno di una funzione, vengono eliminate all’uscita dalla funzione stessa. Di norma ciò avviene utilizzando la pila dei dati che di solito ogni processo elaborativo dispone (si veda eventualmente la sezione [64.10](#)).

Figura 66.121. Variabili «automatiche» dichiarate implicitamente come tali.

```

int h = 1;
int i = 2;
int j = 3;
int k = 4;

float f (int x)
{
    int h;
    float i = 1.5;
    h = ((2 * j) * i);
    float m;
    m = (k / 2.0);
    return (m + i);
}

```

viene creata la variabile *x* contenente l'argomento della chiamata della funzione

viene creata una nuova variabile *h* che nasconde quella dichiarata esternamente

viene creata una nuova variabile *i* che nasconde quella dichiarata esternamente

la variabile *j* proviene dall'esterno della funzione

la variabile *h* ottiene il valore 9

viene creata la variabile *m*

la variabile *k* proviene dall'esterno della funzione

la variabile *m* riceve il valore 2,0

viene restituita la somma di 2,0 e 1,5

vengono eliminate le variabili dichiarate internamente alla funzione, compreso il parametro *x*

Le variabili create all'interno di una funzione, nel modo descritto dalla figura precedente, sono *variabili automatiche* ed è possibile esplicitare questa loro caratteristica con lo specificatore di classe di memorizzazione '**auto**'. Pertanto, la stessa cosa sarebbe stata ottenuta scrivendo l'esempio come nella figura successiva.

Figura 66.122. Variabili «automatiche» dichiarate espressamente attraverso lo specificatore di classe di memorizzazione **'auto'**.

```
int h = 1;
int i = 2;
int j = 3;
int k = 4;

float f (int x)
{
    auto int h;
    auto float i = 1.5;
    h = ((2 * j) * i);
    auto float m;
    m = (k / 2.0);
    return (m + i);
}
```

All'interno di una funzione è possibile utilizzare variabili che facciano riferimento a porzioni di memoria che non vengono rilasciate all'uscita della funzione stessa, pur isolandole rispetto alle variabili dichiarate esternamente. Si ottiene questo con lo specificatore di classe di memorizzazione **'static'** che non va confuso con lo stesso specificatore usato per le variabili dichiarate esternamente alle funzioni. In altre parole, quando in una funzione si dichiara una variabile con lo specificatore di classe di memorizzazione **'static'**, si ottiene di conservare il contenuto di quella variabile che torna a essere accessibile nelle chiamate successive della funzione.

Di norma, la dichiarazione di una variabile di questo tipo coinci-

de con la sua inizializzazione; in tal caso, l'inizializzazione avviene solo quando si chiama la funzione la prima volta.

Figura 66.123. Variabili «statiche» (da intendersi come variabili private) dichiarate all'interno delle funzioni.

```
int i = 2;           viene creata la variabile i che nasconde quella dichiarata esternamente
                   l'inizializzazione della variabile avviene solo la prima volta
int f (void)
{
    static int i = 0;
    i++;           viene incrementata di una unità la variabile i
    return i;     viene restituito il valore della variabile i
}                il contenuto della variabile i viene preservato per la prossima chiamata
```

All'interno delle funzioni possono essere usati anche gli specificatori di classe di memorizzazione '**register**' e '**extern**', come descritto nella tabella successiva.

Tabella 66.124. Specificatori di classe di memorizzazione utilizzabili nella dichiarazione delle variabili all'interno delle funzioni.

| Parola chiave | Descrizione                                                                                                                                                                                                                                            |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| auto          | È lo specificatore di classe di memorizzazione predefinito e indica che la variabile viene creata in corrispondenza della dichiarazione e viene eliminata all'uscita della funzione.                                                                   |
| register      | Con lo specificatore di classe di memorizzazione ' <b>register</b> ' si chiede di creare una variabile automatica che, se possibile, utilizzi un registro del microprocessore o qualunque altra risorsa limitata che possa ridurre i tempi di accesso. |

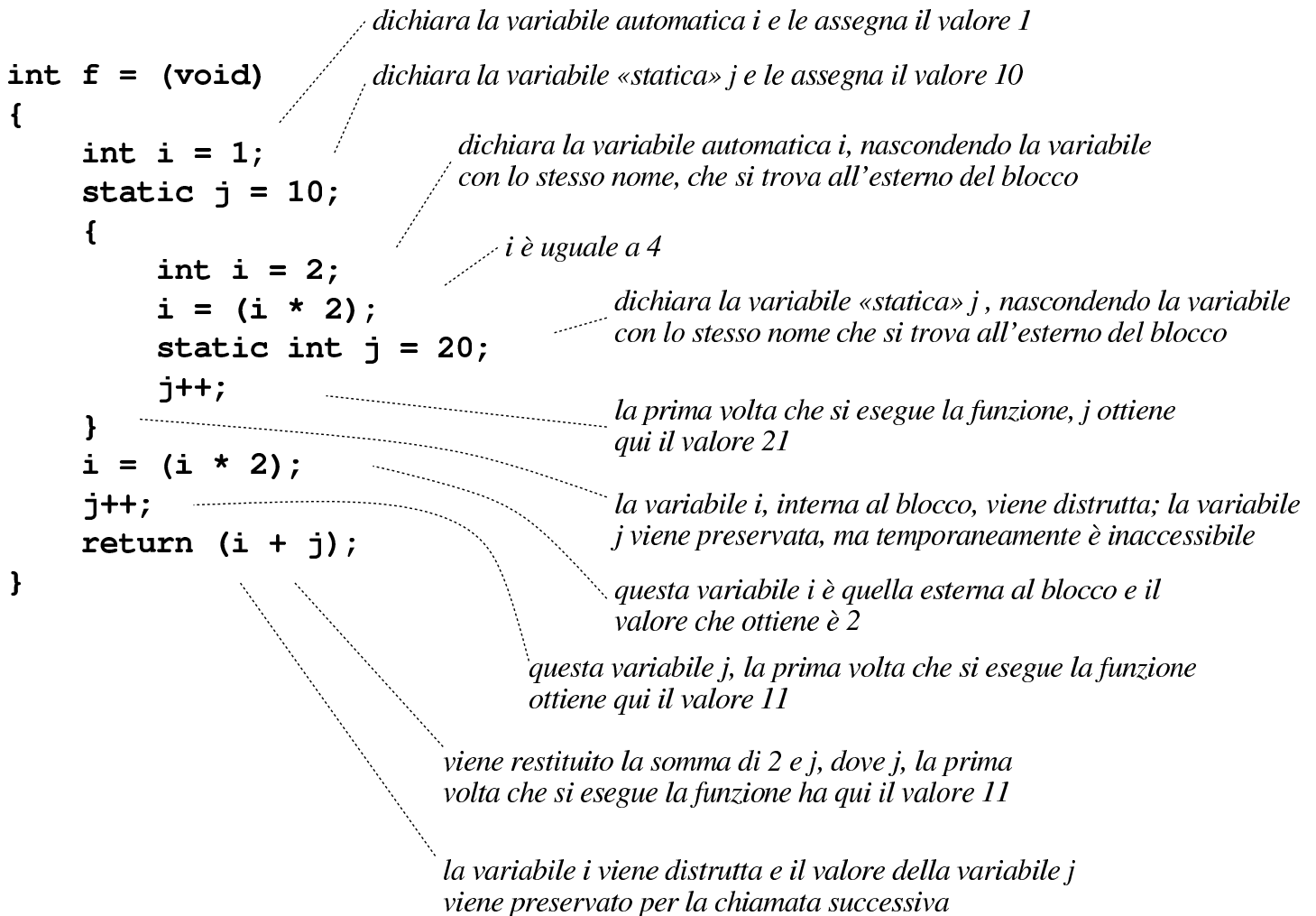
| Parola chiave | Descrizione                                                                                                                                                                                                                                                                                                              |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| static        | Definisce una variabile «privata» allocando della memoria che non viene rilasciata alla conclusione dell'attività della funzione, conservando il valore memorizzato per la chiamata successiva della stessa funzione. Si tratta comunque di una variabile a cui può accedere solo la funzione in cui è dichiarata.       |
| extern        | Indica il riferimento a una variabile dichiarata «esternamente» (come già mostrato nella sezione precedente). In generale, sarebbe meglio dichiarare in questo modo solo le variabili che sono definite al di fuori delle funzioni, lasciando che le funzioni vi accedano semplicemente in qualità di variabili globali. |

### 66.3.5 Campo di azione interno ai raggruppamenti di istruzioni

Le variabili dichiarate all'interno di raggruppamenti di istruzioni, ovvero all'interno di parentesi graffe, si comportano esattamente come quelle dichiarate all'interno delle funzioni: il loro campo di azione termina all'uscita dal blocco. L'esempio della figura successiva mostra un raggruppamento di istruzioni contenente la dichiarazione di una variabile automatica e di una «statica», con la descrizione dettagliata di ciò che accade, dentro e fuori dal raggruppamento.



Figura 66.125. Vita delle variabili all'interno dei raggruppamenti di istruzioni.



La dimostrazione serve a comprendere che, all'interno di una funzione, la posizione in cui si dichiara una variabile non è indifferente: in generale, per migliorare la leggibilità del codice, sarebbe bene dichiarare le variabili all'inizio delle funzioni, evitando accuratamente di farlo all'interno di raggruppamenti annidati.

### 66.3.6 Funzioni annidate

«

Così come esistono i raggruppamenti di istruzioni, all'interno dei quali la dichiarazione delle variabili ha un proprio campo di azione limitato, è possibile anche dichiarare delle sottofunzioni, accessibili

solo all'interno delle funzioni stesse, dopo che sono state dichiarate. Queste sottofunzioni non possono avere uno specificatore di classe di memorizzazione e appartengono esclusivamente alla funzione che le contiene.

In generale, l'uso di sottofunzioni è sconsigliabile e, d'altra parte, originariamente non era permesso.

### 66.3.7 Visibilità, accessibilità, staticità

Va chiarita la distinzione che c'è tra la visibilità di una variabile e l'accessibilità al suo contenuto. Quando una funzione dichiara delle variabili automatiche o statiche con un certo nome, se questa funzione chiama a sua volta un'altra funzione che al suo interno fa uso di variabili con lo stesso nome, queste ultime non si riferiscono alla prima funzione. Si osservi l'esempio del listato seguente.

Listato 66.126. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/bZoOp7vp>, <http://ideone.com/ZpYU4>.

```
#include <stdio.h>

int x = 100;

int f (void)
{
    return x;
}

int main (int argc, char *argv[])
{
    int x = 7;
    printf ("x == %i\n", x);
}
```

```
printf ("f() == %i\n", f());  
return 0;  
}
```

Avviando questo programma si ottiene il testo seguente:

```
x == 7  
f() == 100
```

In pratica, la funzione *f()* che utilizza la variabile *x*, si riferisce alla variabile con quel nome, dichiarata esternamente alle funzioni, che risulta inizializzata con il valore 100, ignorando perfettamente che la funzione *main()* la sta chiamando mentre gestisce una propria variabile automatica con lo stesso nome. Pertanto, la variabile automatica *x* della funzione *main()* non è visibile alle funzioni che questa chiama a sua volta.

D'altra parte, anche se la variabile automatica *x* non risulta visibile, il suo contenuto può essere accessibile, dal momento della sua dichiarazione fino alla fine della funzione (ma questo richiede l'uso di puntatori, come descritto nella sezione [66.5](#)). Alla fine dell'esecuzione della funzione, tutte le sue variabili automatiche perdono la propria identità, in quanto scaricate dalla pila dei dati, e il loro spazio di memoria può essere utilizzato per altri dati (per altre variabili automatiche di altre funzioni).

Si osservi che lo stesso risultato si otterrebbe anche se la variabile *x* della funzione *main()* fosse dichiarata come statica:



```
...
int main (int argc, char *argv[])
{
    static int x = 7;
    printf ("x == %i\n", x);

    printf ("f() == %i\n", f());
    return 0;
}
```

Le variabili statiche, siano esse dichiarate al di fuori o all'interno delle funzioni, hanno in comune il fatto che utilizzano la memoria dal principio alla fine del funzionamento del programma, anche se dal punto di vista del programma stesso non sono sempre visibili. Pertanto, il loro spazio di memoria sarebbe sempre accessibile, anche se sono oscurate temporaneamente o se ci si trova fuori dal loro campo di azione, attraverso l'uso di puntatori. Naturalmente, il buon senso richiede di mettere la dichiarazione di variabili statiche al di fuori delle funzioni, se queste devono essere manipolate da più di una di queste.

Le variabili che utilizzano memoria dal principio alla fine dell'esecuzione del programma, ma non sono statiche, sono quelle variabili dichiarate all'esterno delle funzioni, per le quali il compilatore predispone un simbolo che consenta la loro identificazione nel file-oggetto. Il fatto di non essere statiche (ovvero il fatto di guadagnare un simbolo di riconoscimento nel file-oggetto) consente loro di essere condivise tra più file (intesi come unità di traduzione), ma per il resto valgono sostanzialmente le stesse regole di visibilità. Il buon senso stesso fa capire che tali variabili possano essere dichiarate solo esternamente alle funzioni, perché dentro le funzioni si usa preva-

lentamente la pila dei dati e perché comunque, ciò che è dichiarato dentro la funzione deve avere una visibilità limitata.

### 66.3.8 Compilazione di un progetto composto da più file

«

Viene riproposto l'esempio utilizzato più volte in questo capitolo, nella sua versione per due file, completandolo con una funzione *main()*, in modo da poterlo compilare e dimostrare i passaggi necessari in situazioni del genere.

Listato 66.129. File 'a.c'.

```
#include <stdio.h>

int f (int);
int i;

int g (void)
{
    i++;
    return f (i);
}

int main (void)
{
    printf ("valore originale di i = %i, ", i);
    printf ("valore restituito da g() = %i\n", g());
    printf ("valore originale di i = %i, ", i);
    printf ("valore restituito da g() = %i\n", g());
    printf ("valore originale di i = %i, ", i);
    printf ("valore restituito da g() = %i\n", g());
    printf ("valore originale di i = %i, ", i);
    printf ("valore restituito da g() = %i\n", g());
    return 0;
}
```

## Listato 66.130. File 'b.c'.

```
int f (int x)
{
    return (x * x);
}

int i = 1;
```

Disponendo di più file sorgenti separati, la compilazione avviene in due fasi: la generazione dei file oggetto e il «collegamento» (*link*) di questi in modo da ottenere un file eseguibile. Fortunatamente, tutto questo può essere gestito tramite lo stesso compilatore 'cc'.

Per generare i file oggetto si utilizza 'cc' con l'opzione '-c'; se si può disporre del compilatore GNU C, è meglio aggiungere anche l'opzione '-Wall'. Si suppone che il primo file sia stato nominato 'a.c' e il secondo 'b.c'. Si inizia dalla compilazione dei singoli file in modo da generare i file oggetto 'a.o' e 'b.o'.

```
$ cc -Wall -c a.c [Invio]
```

```
$ cc -Wall -c b.c [Invio]
```

Quindi si passa all'unione dei due risolvendo i riferimenti incrociati, generando il file eseguibile 'prova'.

```
$ cc -o prova a.o b.o [Invio]
```

Ecco cosa si dovrebbe vedere eseguendo il file che si ottiene dalla compilazione:

```
$ ./prova [Invio]
```

```
valore originale di i = 1, valore restituito da g() = 4  
valore originale di i = 2, valore restituito da g() = 9  
valore originale di i = 3, valore restituito da g() = 16  
valore originale di i = 4, valore restituito da g() = 25
```

Per un uso migliore del compilatore si veda la parte [65](#).

### 66.3.9 Osservazioni sulla vita delle costanti letterali

«

Una costante letterale può essere gestita dal compilatore come meglio crede, ma quando si tratta di un'informazione che non può risiedere completamente in una parola del microprocessore e non si può collocare in un'istruzione del linguaggio macchina, è evidente che debba essere conservata nella memoria usata dal programma. Si osservi l'esempio seguente:

```
void f (void)  
{  
    char x[] = "ciao amore";  
    printf ("%s\n", x);  
}
```

L'array  $x[]$ , o meglio, il puntatore che lo rappresenta, viene creato ogni volta alla chiamata della funzione  $f()$  e anche distrutto alla sua conclusione. Ma questo array viene inizializzato ogni volta con una stringa prestabilita, la quale deve essere disponibile per tutto il tempo di funzionamento del programma. In altri termini, quella stringa è un array senza nome allocato in memoria dal principio dell'esecuzione del programma, pertanto al di fuori della pila dei dati.

## 66.3.10 Libreria standard e file di intestazione



La libreria standard del linguaggio C prevede la disponibilità di una serie di funzioni, macro del precompilatore e tipi di dati per usi specifici.

Dal punto di vista del programmatore, si ha la percezione della presenza di questa libreria attraverso l'inclusione dei «file di intestazione», ovvero di quei file che per tradizione hanno un nome che finisce per `.h` e si incorporano attraverso le direttive `#include` del precompilatore. Tuttavia, di norma le funzioni della libreria standard sono contenute in un file-oggetto già compilato (che può essere realizzato in forma differente, a seconda che serva per l'accesso dinamico alle funzioni, oppure che debba essere incorporato nel file eseguibile finale, come spiegato nella sezione [65.7](#)), noto come libreria C, o solo `Libc`, che viene incluso automaticamente nella compilazione di un progetto, a meno di escluderlo espressamente.

Con il compilatore GNU C, per escludere l'utilizzo di qualunque libreria predefinita vanno usate le opzioni `-nostartfiles` e `-nodefaultlibs`; eventualmente l'opzione `-nostdlibs` dovrebbe valere per entrambe queste opzioni e può essere usata assieme a loro, benché sia ridondante.

Anche se la libreria C viene realizzata nel modo descritto, il concetto di libreria standard non si esaurisce nei file-oggetto che contengono le sue funzioni, perché rimane la necessità di dichiarare le macro del precompilatore, i tipi di dati che fanno parte dello standard complessivo, ma soprattutto i prototipi delle funzioni che compongono

la libreria. Pertanto, i file di intestazione rimangono indispensabili e fanno parte integrante della libreria.

A titolo dimostrativo, si può osservare il programma seguente che, pur facendo uso della libreria standard, in quanto si sfrutta la funzione *printf()*, non incorpora alcun file di intestazione. In tal caso, però, è indispensabile dichiarare il prototipo della funzione utilizzata:

```
extern int printf (const char *format, ...);

int main (void)
{
    printf ("Ciao a tutti!\n");
    return 0;
}
```

## 66.4 Annotazioni sulla terminologia

«

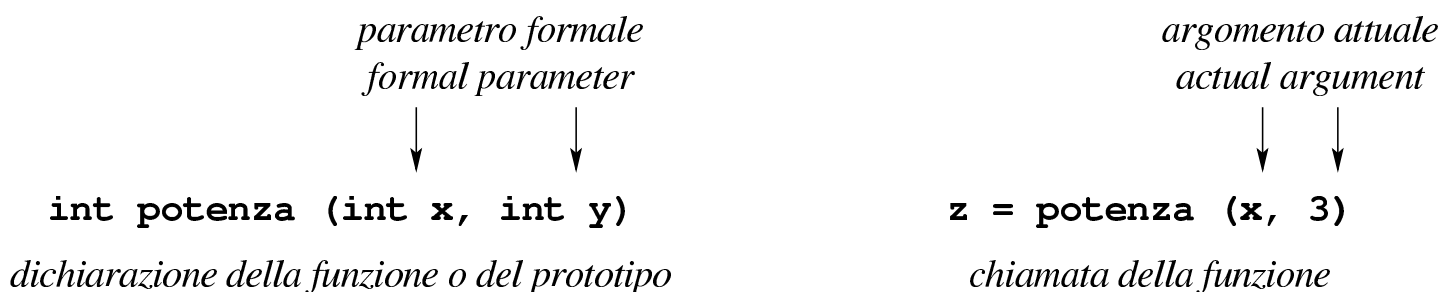
I documenti che descrivono lo standard del linguaggio C utilizzano una terminologia specifica. Qui si descrivono alcuni di quei termini con delle annotazioni riguardo al contesto a cui si riferiscono.

### 66.4.1 Parametri e argomenti

«

Generalmente, i termini «argomento» e «parametro», riferiti alle funzioni o alle procedure dei linguaggi di programmazione, vengono usati in modo intercambiabile, benché si intuisca una differenza tra i due. Lo standard C chiarisce l'ambito corretto di utilizzo per entrambi: i valori annotati in una chiamata di funzione sono gli argomenti attuali; le variabili che descrivono formalmente ciò che una funzione deve ricevere dall'esterno sono i parametri formali.

Figura 66.134. Distinzione tra parametri e argomenti.



## 66.4.2 Byte e caratteri

Secondo il linguaggio C, il byte è l'unità di memorizzazione più piccola che possa essere utilizzata per contenere un carattere, tra quelli dell'insieme minimo. Pertanto, per definizione, il tipo '**char**' (indifferentemente se con o senza segno) occupa esattamente un byte.

In pratica, per il linguaggio C il byte non è necessariamente un insieme di otto bit, anche se di norma questa corrispondenza è valida.

Va considerato anche che il tipo '**char**', senza altre indicazioni, può essere inteso come valore con segno o senza segno, a seconda della piattaforma. Tuttavia, come punto fermo, l'insieme di caratteri minimo deve essere rappresentabile con valori positivi. In pratica, di norma questo insieme minimo di caratteri corrisponde alla codifica ASCII, la quale si rappresenta completamente con 7 bit, pertanto l'ottavo bit di un byte standard potrebbe essere usato come segno, senza interferire con l'interpretazione corretta dei caratteri. In altri termini, per utilizzare il tipo '**char**' in modo compatibile da una piattaforma all'altra, questo va considerato solo per i valori utili alla rappresentazione dell'insieme di caratteri minimo, con i quali si ha la certezza di avere a che fare sempre solo con valori positivi.

### 66.4.3 Unità di traduzione

«

Il file generato dal precompilatore, formato normalmente dall'incorporazione di diversi file, viene definito una *unità di traduzione*. Il concetto di «traduzione» deriva dal fatto che il precompilatore, oltre a incorporare altri file, traduce le macro-variabili e le macroistruzioni espandendole secondo la loro dichiarazione; pertanto, i file sorgenti originali subiscono una prima trasformazione che produce il codice C vero e proprio.

Quando si fa riferimento al campo di azione delle variabili definite al di fuori delle funzioni, si afferma che questo riguarda l'ambito del file. In tal caso, per file si intende l'unità di traduzione.

### 66.4.4 «Linkage»

«

Quando si fa riferimento a variabili o funzioni che sono dichiarate esternamente a tutte le funzioni, il campo di azione è legato al file (nel senso di unità di traduzione), essendo accessibili solo a partire dalla dichiarazione stessa. Quando si combinano assieme più file attraverso il meccanismo del «collegamento» (*link*), il programma che esegue questo compito tratta i nomi uguali di variabili e di funzioni nel senso di un riferimento alla stessa cosa (la stessa variabile o la stessa funzione). Quando una variabile o una funzione è dichiarata in modo tale da consentire questo collegamento, si ha un *linkage esterno*; quando la dichiarazione è tale da impedirlo (con lo specificatore di classe di memorizzazione '**static**'), si ha un *linkage interno*.

Si può rendere esplicito che una variabile o una funzione sono da cercarsi al di fuori del proprio file, oppure in una posizione più avanzata dello stesso file, richiedendo un *linkage esterno* con lo specificatore



di classe di memorizzazione **'extern'**. In tal caso, si può collegare esternamente anche una variabile indicata all'interno di una funzione o di un altro tipo di blocco, sempre con lo specificatore **'extern'**.

Le variabili che, diversamente, sono dichiarate all'interno di un blocco di qualunque genere, non sono collegabili, soprattutto nel caso delle variabili automatiche, la cui vita dipende dal blocco in cui sono contenute.

#### 66.4.5 Durata di memorizzazione

Nella documentazione standard si usa spesso il termine *storage duration*, ovvero ***durata di memorizzazione***, per fare riferimento al tempo di vita di una certa informazione contenuta in memoria.

Di norma si possono distinguere due casi fondamentali: ciò che viene memorizzato in un'area di memoria sempre disponibile (anche se non è detto che a ogni parte del programma sia consentito di accedervi) e ciò che si mette nella pila dei dati. Nel primo caso si parla di *static storage duration*, in quanto i dati stanno lì e non si muovono; nel secondo si parla di *automatic storage duration*, in quanto la memoria della pila viene liberata e riutilizzata in modo dinamico.

È per questa ragione che, nella dichiarazione delle variabili all'interno delle funzioni, esiste lo specificatore di classe **'static'**, a indicare una variabile che, pur essendo accessibile solo all'interno della funzione, va collocata al di fuori della pila dei dati, in modo da conservare il proprio contenuto durante le chiamate successive della stessa funzione.

## 66.4.6 «Lvalue» e «rvalue»

«

Nello standard del linguaggio C, il termine *lvalue* indica, approssimativamente, ciò che appare a sinistra di un operatore di assegnamento, nelle condizioni per cui ciò è ammissibile. Per esempio, nell'espressione seguente, la variabile *x* rappresenta un *lvalue*:

```
x = 3;
```

L'espressione seguente, invece, **non è valida**, perché la costante '3' non può essere un *lvalue*:

```
3 = x; // Non è valida, perché «3» non è un «lvalue».
```

Il termine poteva significare, originariamente, *left-value*, da contrapporsi a un possibile *right-value*, costituito da ciò che in un'espressione si trova alla destra dell'operatore di assegnamento. Tuttavia, lo standard attuale definisce la sigla in questione un *location value*, ovvero un'espressione che si riferisce a un'area di memorizzazione.

Un'espressione che sia un *lvalue* deve anche consentire la lettura dell'area di memorizzazione a cui si riferisce; pertanto, ciò che è un *lvalue* deve poter essere usato alla destra di un operatore di assegnamento (in qualità di *rvalue*). D'altra parte, non è garantito che un *lvalue* individui sempre un'area di memorizzazione modificabile, dal momento che esistono variabili qualificate come costanti, alle quali si assegna un valore in fase di dichiarazione, ma successivamente non è più consentita la modifica. Per distinguere anche questa situazione, volendo escludere il caso delle costanti, si specifica che l'espressione *lvalue* deve anche essere modificabile.

Tabella 66.137. Operatori che richiedono un operando di tipo *lvalue*. In tutti i casi, escluso ‘&*lvalue*’, deve trattarsi di un *lvalue* modificabile in quel contesto.

| Parola chiave                                                            | Descrizione                  |
|--------------------------------------------------------------------------|------------------------------|
| & <i>lvalue</i>                                                          | Indirizzo di <i>lvalue</i> . |
| <i>++lvalue</i><br><i>lvalue++</i><br><i>--lvalue</i><br><i>lvalue--</i> | Incremento e decremento.     |

| Parola chiave                | Descrizione   |
|------------------------------|---------------|
| <i>lvalue=rvalue</i>         | Assegnamenti. |
| <i>lvalue+=rvalue</i>        |               |
| <i>lvalue-=rvalue</i>        |               |
| <i>lvalue*=rvalue</i>        |               |
| <i>lvalue%=rvalue</i>        |               |
| <i>lvalue&lt;&lt;=rvalue</i> |               |
| <i>lvalue&gt;&gt;=rvalue</i> |               |
| <i>lvalue&amp;=rvalue</i>    |               |
| <i>lvalue^=rvalue</i>        |               |
| <i>lvalue =rvalue</i>        |               |
| <i>lvalue~=rvalue</i>        |               |

Attualmente, lo standard C, al posto di *rvalue*, preferisce esprimere il concetto come «valore di un'espressione».

#### 66.4.7 «Digraph» e «Trigraph»

«

In varie situazioni lo standard C consente l'utilizzo di sequenze speciali di caratteri, in sostituzione di simboli che in certi contesti potrebbero mancare, essendo invece indispensabili. In generale, quando per la scrittura dei file sorgenti si può contare su un insieme di

caratteri pari a quello della codifica ASCII, queste sequenze speciali non vanno usate assolutamente, perché complicano terribilmente la lettura dei file. A ogni modo, conviene essere a conoscenza della loro esistenza e del significato che assumono.

| <i>Digraph</i> | <i>Trigraph</i> | Carattere corrispondente |
|----------------|-----------------|--------------------------|
| <:             | ?? (            | [                        |
| :>             | ??)             | ]                        |
| <%             | ??<             | {                        |
| %>             | ??>             | }                        |
| %:             | ??=             | #                        |
| %:%:           | ??=??=          | ##                       |
|                | ??!             |                          |
|                | ??'             | ^                        |
|                | ??/             | \                        |
|                | ??-             | ~                        |

## 66.5 Puntatori, array, stringhe e allocazione dinamica della memoria

All'inizio del capitolo sono stati mostrati solo i tipi di dati più semplici. Per poter utilizzare gli array si gestiscono dei puntatori alle zone di memoria contenenti tali strutture.



Quando si ha a che fare con i puntatori è importante considerare che il modello di memoria che si ha di fronte è un'astrazione, nel senso che una struttura di dati appare idealmente continua, mentre nella realtà il compilatore potrebbe anche provvedere a scomporla in blocchi separati.

Nella spiegazione che si fa qui, come nelle altre sezioni del capitolo, l'esposizione è semplificata rispetto alle definizioni dello standard; pertanto, per un approccio più preciso ci si deve rivolgere ai documenti ufficiali sul linguaggio C.

### 66.5.1 Espressioni a cui si assegnano dei valori

«

Quando si utilizza un operatore di assegnamento, come '=' o altri operatori composti, ciò che si mette alla sinistra rappresenta la «variabile ricevente» del risultato dell'espressione che si trova alla destra dell'operatore (nel caso di operatori di assegnamento composti, l'espressione alla destra va considerata come quella che si ottiene scomponendo l'operatore). Ma il linguaggio C consente di rappresentare quella «variabile ricevente» attraverso un'espressione, come nel caso dei puntatori che vengono descritti in questo capitolo. Pertanto, per evitare confusione, la documentazione dello standard chiama l'espressione a sinistra dell'operatore di assegnamento un *lvalue* (*Left value* o *Location value*).

Nel capitolo si evita questa terminologia, tuttavia è importante comprendere che un'espressione può rappresentare una «variabile», pur senza averle dato un nome (nella sezione [66.4.6](#) il concetto di *lvalue* e di *rvalue* viene descritto con migliore dettaglio).

## 66.5.2 Puntatori

Una variabile, di qualunque tipo sia, rappresenta normalmente un valore posto da qualche parte nella memoria del sistema.<sup>19</sup> Quando si usano i tipi di dati normali, è il compilatore a prendersi cura di tradurre i riferimenti agli spazi di memoria rappresentati simbolicamente attraverso dei nomi.

Attraverso l'operatore di indirizzamento e-commerciale ('&'), è possibile ottenere il puntatore (riferito alla rappresentazione ideale di memoria del linguaggio C) a una variabile «normale». Tale valore può essere inserito in una variabile particolare, adatta a contenerlo: una *variabile puntatore*.

Per esempio, se *p* è una variabile puntatore adatta a contenere l'indirizzo di un intero, l'esempio mostra in che modo assegnare a tale variabile il puntatore alla variabile *i*:

```
int i = 10;
...
p = &i; // L'indirizzo di «i» viene assegnato al
        // puntatore «p».
```

La dichiarazione di una variabile puntatore avviene in modo simile a quello delle variabili normali, con l'aggiunta di un asterisco prima del nome. L'esempio seguente dichiara la variabile *p* come puntatore a un tipo 'int'. Si osservi che va indicato il tipo di dati a cui si punta, perché questa informazione è parte integrante del puntatore.

```
int *p;
```

Non deve essere interesse del programmatore il modo esatto in cui si rappresentano i puntatori dei vari tipi di dati, diversamente non

ci sarebbe l'utilità di usare un linguaggio come il C invece di un semplice assembler di linguaggio macchina.

Una volta dichiarata la variabile puntatore, questa viene utilizzata normalmente, senza asterisco, finché si intende fare riferimento al puntatore stesso.

L'asterisco usato nella dichiarazione serve a definire il tipo di dati, quindi, `int *p` rappresenta la dichiarazione della variabile `p` di tipo `int *`. Tuttavia si può fare un ragionamento leggermente differente, con l'aiuto delle parentesi: `int (*p)` è la dichiarazione di una zona di memoria senza nome, di tipo `int`, a cui punta la variabile `p` attraverso la dereferenziazione `*p`. Le due cose sono equivalenti, in quanto portano comunque alla creazione della variabile `p` di tipo puntatore a intero, ma la seconda forma consente di comprendere, successivamente, la sintassi per la creazione di un puntatore a funzione.

È importante chiarire subito in che modo si dichiarano più variabili puntatore con una sola istruzione; si osservi l'esempio seguente in cui si creano le variabili `p` e `p2`, in particolare per il fatto che l'asterisco va ripetuto:

```
int *p, *p2;
```

Attraverso l'operatore di «dereferenziazione», l'asterisco (`*`), è possibile accedere alla zona di memoria a cui la variabile punta. Per «dereferenziazione» si intende quindi l'azione con cui si toglie il riferimento e si raggiungono i dati a cui un puntatore si riferisce.<sup>20</sup>



Attenzione a non fare confusione con gli asterischi: una cosa è quello usato per dichiarare o per dereferenziare un puntatore e un'altra è l'operatore con cui invece si ottiene la moltiplicazione.

L'esempio già accennato potrebbe essere chiarito nel modo seguente, dove si mostra anche la dichiarazione della variabile puntatore:

```
int i = 10;
int *p;
...
p = &i;
```

A questo punto, dopo aver assegnato a  $p$  il puntatore alla variabile  $i$ , è possibile accedere alla stessa area di memoria in due modi diversi: attraverso la variabile  $i$ , oppure attraverso la dereferenziazione di  $p$ , ovvero la traduzione  $*p$ .

```
int i = 10;
int *p;
...
p = &i;
...
*p = 20;
```

Nell'esempio, l'istruzione ' $*p=20$ ' è tecnicamente equivalente a ' $i=20$ '. Per chiarire un po' meglio il ruolo delle variabili puntatore, si può complicare l'esempio nel modo seguente:

```
int i = 10;
int *p;
int *p2;
...
p = &i;
...
p2 = p;
...
*p2 = 20;
```

In particolare è stata aggiunta una seconda variabile puntatore, *p2*, solo per fare vedere che è possibile passare un puntatore anche ad altre variabili senza dover usare l'asterisco. Comunque, in questo caso, '*\*p2=20*' è tecnicamente equivalente sia a '*\*p=20*', sia a '*i=20*'.

Si osservi che l'asterisco è un operatore che, evidentemente, ha la precedenza rispetto a quelli di assegnamento. Eventualmente, anche in questo caso si possono usare le parentesi per togliere ambiguità al codice:

```
int i = 10;
int *p;
...
p = &i;
...
(*p2) = 20;
```

Come accennato inizialmente, il tipo di dati a cui un puntatore si rivolge, fa parte integrante del puntatore stesso. Ciò è importante perché quando si dereferenzia un puntatore occorre sapere quanto è grande l'area di memoria a cui si deve accedere a partire dal puntatore. Per questa ragione, quando si assegna a una variabile puntatore un altro puntatore, questo deve essere compatibile, nel senso che de-

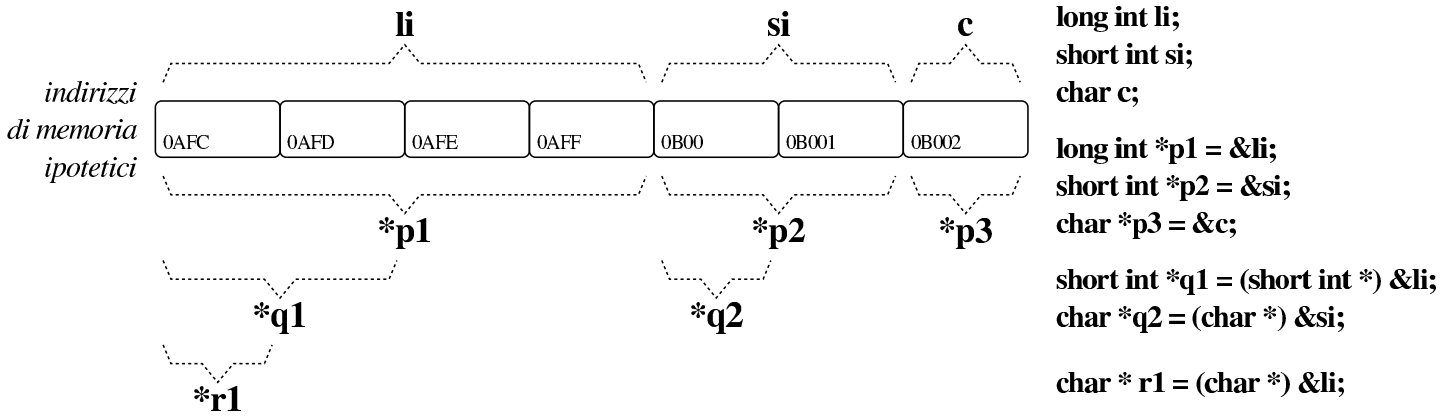
ve riferirsi allo stesso tipo di dati, altrimenti si rischia di ottenere un risultato inatteso. A questo proposito, l'esempio seguente contiene probabilmente un errore:

```
char *pc;
int *pi;
...
pi = pc; // I due puntatori si riferiscono a dati di tipo
         // differente!
...
```

Quando invece si vuole trasformare realmente un puntatore in modo che si riferisca a un tipo di dati differente, si può usare un cast, come si farebbe per convertire i valori numerici:

```
char *pc;
int *pi;
...
pi = (int *) pc; // Il programmatore dimostra di essere
                 // consapevole di ciò che sta facendo
                 // attraverso un cast!
...
...
```

Nello schema seguente appare un esempio che dovrebbe consentire di comprendere la differenza che c'è tra i puntatori, in base al tipo di dati a cui fanno riferimento. In particolare, *p1*, *q1* e *r1* fanno tutti riferimento all'indirizzo ipotetico  $0AFC_{16}$ , ma l'area di memoria che considerano è diversa, pertanto *\*p1*, *\*q1* e *\*r1* sono tra loro «variabili» differenti, anche se si sovrappongono parzialmente.

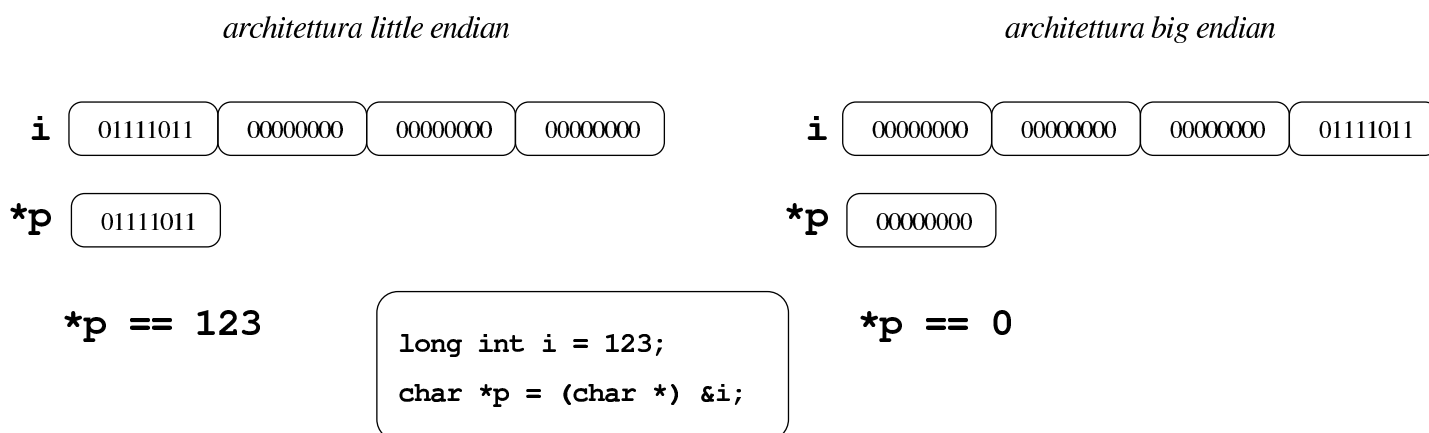


L'esempio seguente rappresenta un programma completo che ha lo scopo di determinare se l'architettura dell'elaboratore è di tipo *big endian* o di tipo *little endian*. Per capirlo si dichiara una variabile di tipo '**long int**' che si intende debba essere di rango superiore rispetto al tipo '**char**', assegnandole un valore abbastanza basso da poter essere rappresentato anche in un tipo '**char**' senza segno. Con un puntatore di tipo '**char \***' si vuole accedere all'inizio della variabile contenente il numero intero '**long int**': se già nella porzione letta attraverso il puntatore al primo «carattere» si trova il valore assegnato alla variabile di tipo intero, vuol dire che i byte sono invertiti e si ha un'architettura *little endian*, mentre diversamente si presume che sia un'architettura *big endian*.

Listato 66.149. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/Abe4VgIo> , <http://ideone.com/WVpmK> .

```
#include <stdio.h>
int main (void)
{
    long int i = 123;
    char *p = (char *) &i;
    if (*p == 123)
        {
            printf ("little endian\n");
        }
    else
        {
            printf ("big endian\n");
        }
    return 0;
}
```

Figura 66.150. Schematizzazione dell'operato del programma di esempio, per determinare l'ordine dei byte usato nella propria architettura.



Il linguaggio C utilizza il passaggio degli argomenti alle funzioni per valore; per ottenere il passaggio per riferimento occorre utilizzare

dei puntatori. Si immagini di volere realizzare una funzione banale che modifica la variabile utilizzata nella chiamata, sommandovi una quantità fissa. Invece di passare il valore della variabile da modificare, si può passare il suo puntatore; in questo modo la funzione (che comunque deve essere stata realizzata appositamente per questo scopo) agisce nell'area di memoria a cui punta questo puntatore.

```
...
void funzione_stupida (int *x)
{
    (*x)++;
}
...
int main (void)
{
    int y = 10;
    ...
    funzione_stupida (&y);
    ...
    return 0;
}
```

L'esempio mostra la dichiarazione e descrizione di una funzione che non restituisce alcun valore e ha un parametro costituito da un puntatore a un intero. Il lavoro della funzione è solo quello di incrementare il valore contenuto nell'area di memoria a cui si riferisce tale puntatore.

Poco dopo, nella funzione *main()* inizia il programma vero e proprio; viene dichiarata la variabile *y* corrispondente a un intero normale inizializzato a 10, poi, a un certo punto viene chiamata la funzione vista prima, passando il puntatore a *y*.

Il risultato è che dopo la chiamata, la variabile *y* contiene il valore

precedente incrementato di un'unità.

Quando si usano i puntatori, invece delle variabili comuni, occorre considerare che se la vita della variabile a cui un puntatore fa riferimento si è esaurita, il puntatore relativo diventa privo di valore. Questo significa che il fatto di avere conservato il puntatore a una certa area di memoria **non** implica automaticamente la garanzia che tale zona contenga dati validi o che sia ancora raggiungibile.

### 66.5.3 Array

Nel linguaggio C, l'array è una sequenza ordinata di elementi dello stesso tipo nella rappresentazione ideale di memoria di cui si dispone. In questo senso, quando si dichiara un array, quello che il programmatore ottiene in pratica è il riferimento alla posizione iniziale di questo, mentre gli elementi successivi si raggiungono tenendo conto della lunghezza di ogni elemento.

Questo ragionamento vale in senso generale ed è un po' approssimativo. In contesti particolari, il riferimento a un array restituisce qualcosa di diverso dal puntatore al primo elemento.

Visto in questi termini, si può intendere che l'array in C è sempre a una sola dimensione, tutti gli elementi devono essere dello stesso tipo in modo da avere la stessa lunghezza e la quantità degli elementi, una volta definita, è fissa.

È compito del programmatore ricordare la quantità di elementi che compone l'array, perché determinarlo diversamente è complicato e

a volte non è possibile. Inoltre, quando un programma tenta di accedere a una posizione oltre il limite degli elementi esistenti, c'è il rischio che non si verifichi alcun errore, arrivando però a dei risultati imprevedibili.

Lo standard prescrive che sia consentito raggiungere l'indirizzo successivo all'ultimo elemento, anche se tale contenuto diventa privo di significato. Ciò serve a garantire che non si provochino errori nell'accesso alla memoria, se l'indice va oltre il limite di un array, ma per una sola posizione, per leggere un contenuto privo di utilità. In pratica, ciò significa che dopo un array ci deve essere qualunque altra variabile, o al limite uno spazio inutilizzato. Ma questo è compito del compilatore.

La dichiarazione di un array avviene in modo intuitivo, definendo il tipo degli elementi e la loro quantità. L'esempio seguente mostra la dichiarazione dell'array *a* di sette elementi di tipo `int`:

```
int a[7];
```

Per accedere agli elementi dell'array si utilizza un indice, il cui valore iniziale è sempre zero e, di conseguenza, quello con cui si raggiunge l'elemento *n*-esimo deve avere il valore *n*-1. L'esempio seguente mostra l'assegnamento del valore 123 al **secondo** elemento:

```
a[1] = 123;
```

In presenza di array monodimensionali che hanno una quantità ridotta di elementi, può essere sensato attribuire un insieme di valori iniziale all'atto della dichiarazione.




Alcuni compilatori consentono l'inizializzazione degli array solo quando questi sono dichiarati all'esterno delle funzioni, con un campo di azione globale, oppure all'interno delle funzioni, ma dichiarati come «statici», nel senso che continuano a esistere all'uscita della funzione.

```
int a[] = {123, 453, 2, 67};
```

L'esempio mostrato dovrebbe chiarire in che modo si possono dichiarare gli elementi dell'array, tra parentesi graffe, togliendo così la necessità di specificare la quantità di elementi. Tuttavia, le due cose possono coesistere:

```
int a[10] = {123, 453, 2, 67};
```

In tal caso, l'array si compone di 10 elementi, di cui i primi quattro con valori prestabiliti, mentre gli altri ottengono il valore zero. Si osservi però che il contrario non può essere fatto: 

```
int a[5] = {123, 453, 2, 67, 32, 56, 78}; // Non si può!
```

Gli standard recenti del linguaggio C consentono anche la dichiarazione di array per i quali il compilatore non può sapere subito la quantità di elementi da predisporre, **purché ciò avvenga nel campo di azione delle funzioni** (o di blocchi inferiori). In pratica, in questi casi è possibile indicare la quantità di elementi attraverso un'espressione che si traduca in un numero intero, come nell'esempio seguente, dove la quantità di elementi è data dal prodotto tra la variabile *s* e la costante 3:

```
int s = 33;  
...  
int a[s * 3];
```

Gli array dichiarati al di fuori delle funzioni (quelli il cui campo di azione è legato al file) e quelli che, pur essendo dichiarati nelle funzioni, continuano a esistere per tutto il tempo di esecuzione del programma (in quanto «statici»), possono avere soltanto una quantità di elementi già stabilita in fase di compilazione. Per fare riferimento a array definiti in altri file, oppure in posizioni più avanzate dello stesso file, è possibile usare una dichiarazione «esterna», nella quale è bene specificare la quantità di elementi, ma questa deve essere coerente con quella della dichiarazione a cui si fa riferimento:

```
extern int i[3];  
...  
int i[3];
```

In alternativa si può fare una dichiarazione esterna di un array senza specificarne la quantità di elementi, ma questo implica che, fino a quando non appare la dichiarazione completa, l'array sia di tipo incompleto e non si possa determinare la sua dimensione con l'aiuto dell'operatore '**sizeof**':

```
extern int i[];           // Tipo incompleto.  
...  
int i[3];
```

La scansione di un array avviene generalmente attraverso un'iterazione enumerativa, in pratica con un ciclo '**for**' che si presta particolarmente per questo scopo. Si osservi l'esempio seguente:

```
int a[7];
int i;
...
for (i = 0; i < 7; i++)
{
    ...
    a[i] = ...;
    ...
}
```

L'indice *i* viene inizializzato a zero, in modo da cominciare dal primo elemento dell'array; il ciclo può continuare fino a che *i* continua a essere inferiore a sette, infatti l'ultimo elemento dell'array ha indice sei; alla fine di ogni ciclo, prima che riprenda il successivo, viene incrementato l'indice di un'unità.

Per scandire un array in senso opposto, si può agire in modo analogo, come nell'esempio seguente:

```
int a[7];
int i;
...
for (i = 6; i >= 0; i--)
{
    ...
    a[i] = ...;
    ...
}
```

Questa volta l'indice viene inizializzato in modo da puntare alla posizione finale; il ciclo viene ripetuto fino a che l'indice è maggiore o uguale a zero; alla fine di ogni ciclo, l'indice viene decrementato di un'unità.

Se non si può conoscere la dimensione dell'array, questa deve essere calcolata con l'ausilio dell'operatore **'sizeof'**, come nell'esempio seguente, ammesso che il contesto sia tale da consentire all'operatore di restituire un valore valido:

```
// Da qualche parte si dichiara il valore di «x» come numero  
// intero.  
  
...  
int a[7 * x];  
int i;  
  
...  
int s = (sizeof a) / (sizeof (a[0]));  
for (i = 0; i < s; i++)  
    {  
        ...  
        a[i] = ...;  
        ...  
    }
```

Il calcolo della quantità di elementi è ottenuto determinando la dimensione dell'array in byte e dividendo tale valore per la dimensione in byte di un intero, ovvero per la dimensione di ogni elemento dell'array stesso.

Quando un array è argomento dell'operatore **'sizeof'**, si ottiene la dimensione complessiva dell'array stesso (nell'unità gestita da **'sizeof'**). Tuttavia occorre considerare che, se l'array non è ancora stato definito nella sua dimensione, non si può avere il risultato atteso.

## 66.5.4 Array multidimensionali

Gli array in C sono monodimensionali, però nulla vieta di creare un array i cui elementi siano array tutti uguali. Per esempio, nel modo seguente, si dichiara un array di cinque elementi che a loro volta sono insiemi di sette elementi di tipo `int`. Nello stesso modo si possono definire array con più di due dimensioni.

```
int a[5][7];
```

L'esempio seguente mostra il modo normale di scandire un array a due dimensioni:

```
int a[5][7];
int i;
int j;
...
for (i = 0; i < 5; i++)
{
    ...
    for (j = 0; j < 7; j++)
    {
        ...
        a[i][j] = ...;
        ...
    }
    ...
}
```

Anche se in pratica un array a più dimensioni è solo un array «normale» in cui si individuano dei sottogruppi di elementi, la scansione deve avvenire sempre indicando formalmente lo stesso numero di elementi prestabiliti per le dimensioni rispettive, anche se dovrebbe essere possibile attuare qualche trucco. Per esempio, tornando al

listato mostrato, se si vuole scandire in modo continuo l'array, ma usando un solo indice, bisogna farlo gestendo l'ultimo:

```
int a[5][7][9];
int j;
...
for (j = 0; j < (5 * 7 * 9); j++)
{
    ...
    a[0][0][j] = ...;
    ...
}
```

Rimane comunque da osservare il fatto che questo non sia un bel modo di programmare.

Anche gli array a più dimensioni possono essere inizializzati, secondo una modalità analoga a quella usata per una sola dimensione, con la differenza che l'informazione sulla quantità di elementi per dimensione non può essere omessa. L'esempio seguente è un programma completo, in cui si dichiara e inizializza un array a due dimensioni, per poi mostrarne il contenuto:



Listato 66.166. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/Ht9r6QwN>, <http://ideone.com/xzD7f>.

```
#include <stdio.h>

int main (int argc, char *argv[])
{
    int a[3][4] = {{1,  2,  3,  4},
                  {5,  6,  7,  8},
                  {9, 10, 11, 12}};

    int i, j;

    for (i = 0; i < 3; i++)
        {
            for (j = 0; j < 4; j++)
                {
                    printf ("a[%i][%i]=%i\t", i, j, a[i][j]);
                }
            printf ("\n");
        }

    return 0;
}
```

Il programma dovrebbe mostrare il testo seguente:

|           |            |            |            |
|-----------|------------|------------|------------|
| a[0][0]=1 | a[0][1]=2  | a[0][2]=3  | a[0][3]=4  |
| a[1][0]=5 | a[1][1]=6  | a[1][2]=7  | a[1][3]=8  |
| a[2][0]=9 | a[2][1]=10 | a[2][2]=11 | a[2][3]=12 |

Anche nell'inizializzazione di un array a più dimensioni si possono omettere degli elementi, come nell'estratto seguente:

```
...
int a[3][4] = {{1, 2},
               {5, 6, 7, 8}};
...
```

In tal caso, il programma si mostrerebbe così:

```
a[0][0]=1      a[0][1]=2      a[0][2]=0      a[0][3]=0
a[1][0]=5      a[1][1]=6      a[1][2]=7      a[1][3]=8
a[2][0]=0      a[2][1]=0      a[2][2]=0      a[2][3]=0
```

Di certo, pur sapendo di voler utilizzare un array a più dimensioni, si potrebbe pretendere di inizializzarlo come se fosse a una sola, come nell'esempio seguente, ma il compilatore dovrebbe avvisare del fatto:

```
...
int a[3][4] = {1, 2, 3, 4, 5, 6,           // Così non è
               7, 8, 9, 10, 11, 12};      // grazioso.
...
```

### 66.5.5 Natura dell'array




Inizialmente si è accennato al fatto che quando si crea un array, quello che viene restituito in pratica è un puntatore alla sua posizione iniziale, ovvero all'indirizzo del primo elemento di questo. Si può intuire che non sia possibile assegnare a un array un altro array, anche se ciò potrebbe avere significato. Al massimo si può assegnare elemento per elemento.

Per evitare errori del programmatore, la variabile che contiene l'indirizzo iniziale dell'array, quella che in pratica rappresenta l'array stesso, è in **sola lettura**. Quindi, nel caso dell'array già visto, la va-



riabile  $a$  non può essere modificata, mentre i singoli elementi  $a[i]$  sì:

```
int a[7];
```

Data la filosofia del linguaggio C, se fosse possibile assegnare un valore alla variabile  $a$ , si modificherebbe il puntatore, facendo in modo che questo punti a un array differente. Ma per raggiungere questo risultato vanno usati i puntatori in modo esplicito. Si osservi l'esempio seguente. 

Listato 66.172. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/SLI8GS82>, <http://ideone.com/RImyk>.

```
#include <stdio.h>

int main (void)
{
    int a[3];
    int *p;

    p = a;          // «p» diventa un alias dell'array «a».

    p[0] = 10;     // Si può fare solo con gli array
    p[1] = 100;   // a una sola dimensione.
    p[2] = 1000;  //

    printf ("%i %i %i \n", a[0], a[1], a[2]);

    return 0;
}
```

Viene creato un array,  $a$ , di tre elementi di tipo 'int', e subito dopo una variabile puntatore,  $p$ , al tipo 'int'. Si assegna quindi alla

variabile  $p$  il puntatore rappresentato da  $a$ ; da quel momento si può fare riferimento all'array indifferentemente con il nome  $a$  o  $p$ .

Si può osservare anche che l'operatore '&', seguito dal nome di un array, produce ugualmente l'indirizzo dell'array che è equivalente a quello fornito senza l'operatore stesso, con la differenza che riguarda l'array nel suo complesso:

```
...  
p = &a;      // I due puntatori non sono dello stesso tipo!  
...
```

Pertanto, in questo caso si pone il problema di compatibilità del tipo di puntatore che si può risolvere con un cast esplicito:

```
...  
p = (int *) &a; // «p» diventa un alias dell'array «a».  
...
```

In modo analogo, si può estrapolare l'indice che rappresenta l'array dal primo elemento, cosa che si ottiene senza incorrere in problemi di compatibilità tra i puntatori. Si veda la trasformazione dell'esempio nel modo seguente.

Listato 66.175. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/91Df91s1dq>, <http://ideone.com/6rPpE>.

```
#include <stdio.h>

int main (void)
{
    int a[3];
    int *p;

    p = &a[0];    // «p» diventa un alias dell'array «a».

    p[0] = 10;    // Si può fare solo con gli array
    p[1] = 100;   // a una sola dimensione.
    p[2] = 1000; //

    printf ("%i %i %i \n", a[0], a[1], a[2]);

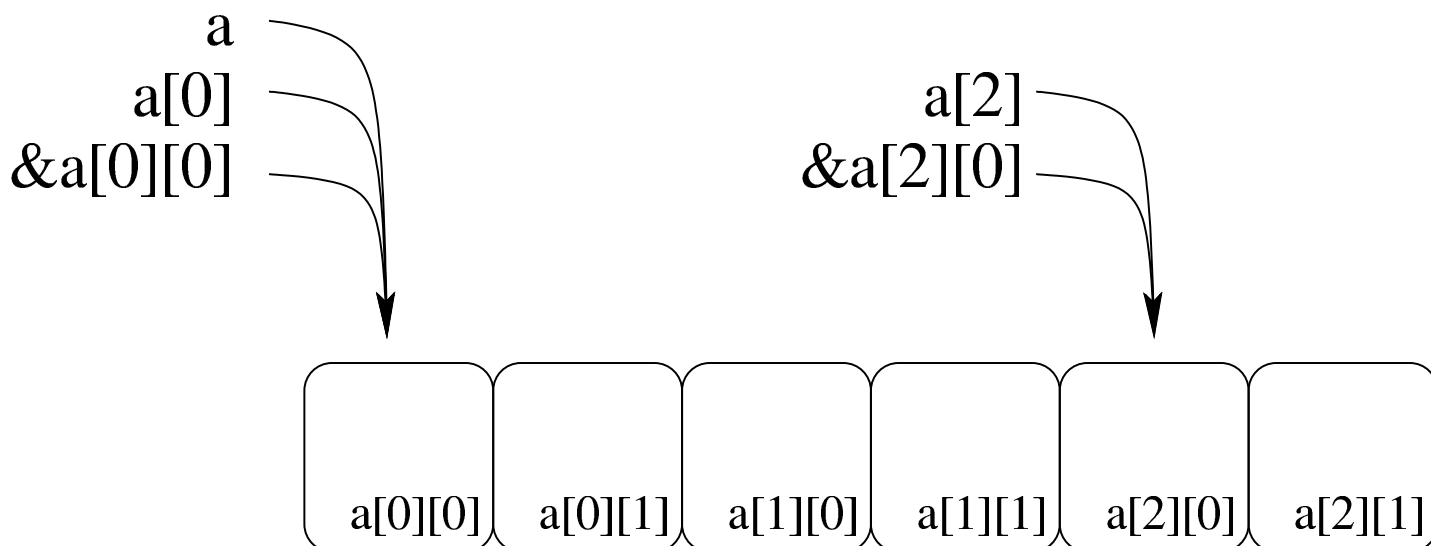
    return 0;
}
```

Anche se si può usare un puntatore come se fosse un array, va osservato che la variabile *p*, in quanto dichiarata come puntatore, viene considerata in modo differente dal compilatore; per esempio non è possibile determinare la dimensione dell'array a cui punta attraverso l'operatore '**sizeof**', perché si otterrebbe semplicemente la quantità di byte che costituisce la variabile puntatore.

Quando si opera con array a più dimensioni, il riferimento a una porzione di array restituisce l'indirizzo della porzione considerata. Per esempio, si supponga di avere dichiarato un array a due dimensioni, nel modo seguente:

```
int a[3][2];
```

Se a un certo punto, in riferimento allo stesso array, si scrivesse ‘**a[2]**’, si otterrebbe l’indirizzo del terzo gruppo di due interi:



Tenendo d’occhio lo schema appena mostrato, considerato che si sta facendo riferimento all’array **a** di 3×2 elementi di tipo ‘**int**’, va osservato che:

- in condizioni normali ‘**a**’ si traduce nel puntatore a un array di due elementi di tipo ‘**int**’;
- ‘**a[0]**’ e ‘**&a[0][0]**’ si traducono nel puntatore a un elemento di tipo ‘**int**’ (precisamente il primo);
- ‘**&a**’ si traduce nel puntatore a un array composto da 3×2 elementi di tipo ‘**int**’.

Pertanto, se questa volta si volesse assegnare a una variabile puntatore di tipo ‘**int \***’ l’indirizzo iniziale dell’array, nell’esempio seguente si creerebbe un problema di compatibilità:

```
...
int a[3][2];
int *p;
p = a;    // I due puntatori non sono dello stesso tipo!
...
```

Pertanto, occorrerebbe riferirsi all'inizio dell'array in modo differente oppure attraverso un cast.

### 66.5.6 Puntatori costanti

Si può far sì che un puntatore funzioni in modo più simile a quello di un array a una sola dimensione, dichiarando il puntatore come costante, nel senso che il puntatore in sé non può essere cambiato: «

```
...
int a[3];
int *const p = a;    // Puntatore in sola lettura.
p[1] = 9;
p = a;              // Questo non si può!
...
```

L'esempio seguente, invece, fa sì che la memoria a cui si vuole accedere tramite il puntatore sia protetta in sola lettura:

```
...
int a[3];
const int *p = a;    // Qui è la memoria a essere
                    // in sola lettura.
p[1] = 9;            // Questo non si può!
p = a;
...
```

Anche se si può bloccare il puntatore, così da farlo funzionare in modo equivalente a un array vero e proprio, rimane però il fatto che

'**sizeof**', usato per «misurare» un puntatore, restituisce comunque la grandezza della variabile che costituisce il puntatore stesso. Inoltre ci sono altre questioni che riguardano i puntatori, affrontate in una sezione separata, a proposito dell'aritmetica dei puntatori.

### 66.5.7 Array e funzioni

«

Si è visto che le funzioni possono accettare solo parametri composti da tipi di dati elementari, compresi i puntatori. In questa situazione, l'unico modo per trasmettere a una funzione un array attraverso i parametri, è quello di inviarne il puntatore iniziale. Di conseguenza, le modifiche che vengono poi apportate da parte della funzione si riflettono nell'array di origine. Si osservi l'esempio seguente.

Listato 66.181. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/2Zibjb5j>, <http://ideone.com/geaQV>.

```
#include <stdio.h>

void elabora (int *p)
{
    p[0] = 10;
    p[1] = 100;
    p[2] = 1000;
}

int main (void)
{
    int a[3];

    elabora (a);
    printf ("%i %i %i \n", a[0], a[1], a[2]);
}
```

```
    return 0;
}
```

La funzione *elabora()* utilizza un solo parametro, rappresentato da un puntatore a un tipo `'int'`. La funzione **presume** che il puntatore si riferisca all'inizio di un array di interi e così assegna alcuni valori ai primi tre elementi (anche il numero degli elementi non può essere determinato dalla funzione).

All'interno della funzione *main()* viene dichiarato l'array *a* di tre elementi interi e subito dopo viene passato come argomento alla funzione *elabora()*. Così facendo, in realtà si passa il puntatore al primo elemento dell'array.

Infine, la funzione altera gli elementi come è già stato descritto e gli effetti si possono osservare così:

```
10 100 1000
```

L'esempio potrebbe essere modificato per presentare la gestione dell'array in modo più elegante. Per la precisione si tratta di ritoccare la funzione `'elabora'`:



```
void elabora (int a[])
{
    a[0] = 10;
    a[1] = 100;
    a[2] = 1000;
}
```

Si tratta sostanzialmente della stessa cosa, solo che si pone l'accento sul fatto che l'argomento è un array di interi, benché di tipo incompleto.

In entrambi i casi, se all'interno della funzione si tenta di misurare la dimensione dell'array con l'operatore `'sizeof'`, si ottiene solo la grandezza della variabile usata per contenere il puntatore relativo. Sarebbe anche possibile specificare la dimensione dell'array, senza però che questo fatto abbia delle conseguenze significative e senza che `'sizeof'` la consideri:

```
void elabora (int a[3]) // Anche così sizeof restituisce
{                       // solo la grandezza del puntatore.
    a[0] = 10;
    a[1] = 100;
    a[2] = 1000;
}
```

## 66.5.8 Aritmetica dei puntatori

Con le variabili puntatore è possibile eseguire delle operazioni elementari: possono essere incrementate e decrementate. Il risultato che si ottiene è il riferimento a una zona di memoria adiacente, in funzione della dimensione del tipo di dati per il quale è stato creato il puntatore. Si osservi l'esempio seguente:

```
int i = 10;
int j;
int *p = &i;
p++;
j = *p; // Attenzione!
```

In questo caso viene creato un puntatore al tipo `'int'` che inizialmente contiene l'indirizzo della variabile `i`. Subito dopo questo puntatore viene incrementato di una unità e ciò comporta che si riferisca a un'area di memoria adiacente, immediatamente successiva a quella occupata dalla variabile `i` (molto probabilmente si tratta dell'area



occupata dalla variabile  $j$ ). Quindi si tenta di copiare il valore di tale area di memoria, interpretato come `int`, all'interno della variabile  $j$ .

Se un programma del genere funziona nell'ambito di un sistema operativo che controlla l'utilizzo della memoria, se l'area che si tenta di raggiungere incrementando il puntatore non è stata allocata, si ottiene un «errore di segmentazione» e l'arresto del programma stesso. L'errore si verifica quando si tenta l'accesso, mentre la modifica del puntatore è sempre lecita.

Lo stesso meccanismo riguarda tutti i tipi di dati che non sono array, perché per gli array, l'incremento o il decremento di un puntatore riguarda i componenti dell'array stesso. In pratica, quando si gestiscono tramite puntatori, gli array sono da intendere come una serie di elementi dello stesso tipo e dimensione, dove, nella maggior parte dei casi, il nome dell'array si traduce nell'indirizzo del primo elemento:

```
int i[3] = { 1, 3, 5 };
int *p;
...
p = i;
```

Nell'esempio si vede che il puntatore  $p$  punta all'inizio dell'array di interi  $i[]$ .

```
*p = 10; // Equivale a: i[0] = 10.
p++;
*p = 30; // Equivale a: i[1] = 30.
p++;
*p = 50; // Equivale a: i[2] = 50.
```

Ecco che, incrementando il puntatore, si accede all'elemento adia-

cente successivo, in funzione della dimensione del tipo di dati. Decrementando il puntatore si ottiene l'effetto opposto, di accedere all'elemento precedente. La stessa cosa avrebbe potuto essere ottenuta così, senza alterare il valore contenuto nella variabile  $p$ :

```
* (p + 0) = 10; // Equivale a: i[0] = 10.
* (p + 1) = 30; // Equivale a: i[1] = 30.
* (p + 2) = 50; // Equivale a: i[2] = 50.
```

Inoltre, come già visto in altre sezioni, si potrebbe usare il puntatore con la stessa notazione propria dell'array, ma ciò solo perché si opera a una sola dimensione:

```
p[0] = 10; // Equivale a: i[0] = 10.
p[1] = 30; // Equivale a: i[1] = 30.
p[2] = 50; // Equivale a: i[2] = 50.
```

Questo lascia intuire che ' $i[n]$ ' corrisponda in pratica a ' $*(i + n)$ ', cosa che è vera per lo standard del linguaggio, ma potrebbe non essere accettabile dal compilatore che si usa effettivamente:


```
* (i + 0) = 10; // Equivale a: i[0] = 10.
* (i + 1) = 30; // Equivale a: i[1] = 30.
* (i + 2) = 50; // Equivale a: i[2] = 50.
```

In presenza di più dimensioni, il ragionamento è analogo. Nel modello seguente, le lettere  $i$  e  $j$  rappresentano gli indici usati per la scansione, mentre le lettere  $I$  e  $J$  sono la quantità di elementi della dimensione corrispondente. Per esempio, secondo il modello seguente, in un array  $x[10][30]$ , la lettera  $J$  corrisponde a 30.

```
x[i][j] == *(x + (i * J) + j)
```

In modo analogo si dovrebbe procedere per dimensioni maggiori:

$$x[i][j][k] == *(x + (i*J*K) + (j*K) + k)$$

Se il compilatore non accetta questo modo di gestire un array, il meccanismo vale per un puntatore dello stesso tipo degli elementi dell'array (che punti all'inizio dell'array stesso). L'esempio seguente mette in evidenza l'uso di un puntatore per scandire un array a due dimensioni. 

Listato 66.191. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/tnGTykMU> , <http://ideone.com/ogdeH> .

```
#include <stdio.h>

int main (int argc, char *argv[])
{

    int a[3][4] = {{1,  2,  3,  4},
                  {5,  6,  7,  8},
                  {9, 10, 11, 12}};

    int i, j;
    const int *p = (int *) a;
    int x;

    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 4; j++)
        {
            x = *(p + i * 4 + j);
            //
            printf ("a[%i][%i]=%i\t", i, j, x);
            //
        }
    }
}
```

```

    }
    printf ("\n");
}

return 0;
}

```

I punti più importanti dell'esempio appaiono evidenziati: trattandosi di un array a più di una dimensione, la copia del puntatore avviene con l'ausilio di un cast; la scansione degli indirizzi, a partire dal puntatore *p* avviene attraverso una formula, mentre la forma seguente ha un significato diverso, descritto in un'altra sezione, a proposito dei puntatori a puntatori:

```

...
    x = p[i][j];           // Non è la stessa cosa!
...

```

La versione funzionante dell'esempio mostrato deve fare apparire il testo seguente:


```

a[0][0]=1      a[0][1]=2      a[0][2]=3      a[0][3]=4
a[1][0]=5      a[1][1]=6      a[1][2]=7      a[1][3]=8
a[2][0]=9      a[2][1]=10     a[2][2]=11     a[2][3]=12

```

Naturalmente, quando si usano direttamente i puntatori, è compito esclusivo del programmatore sapere quando l'incremento o il decremento di un puntatore ha significato. Diversamente si rischia di accedere a zone di memoria estranee al contesto di proprio interesse, con risultati imprevedibili.

Prima di concludere l'argomento, vale la pena di tradurre il problema dell'aritmetica dei puntatori in modo opposto, ovvero co-

me indirizzi. Per esempio, dato l'array  $a[]$ , a una sola dimensione, si può considerare equivalente la notazione ' $\&(a[i])$ ' rispetto a ' $(a + i)$ '. 

### 66.5.9 Osservazioni sui puntatori

«  
Ammesso che la variabile  $p$  sia un puntatore a qualcosa, la notazione  $*p$  equivale a ' $p[0]$ ', così come ' $*(p+n)$ ' corrisponde a ' $p[n]$ '. Pertanto, l'uso delle parentesi quadre contenenti un indice, poste dopo il nome di una variabile puntatore, corrisponde alla dereferenziazione che si fa con l'asterisco.

Ammesso che la variabile  $p$  sia un puntatore a qualcosa, la notazione ' $\&*p$ ' corrisponde sempre a ' $p$ ', anche se si tratta di un puntatore nullo.

Ammesso che la variabile  $x$  sia tale da potervi assegnare un valore e che possa essere operando di ' $\&$ ', la notazione ' $*\&x$ ' corrisponde sempre a ' $x$ '.

Ammesso che la variabile  $p$  sia un puntatore a qualcosa, la notazione ' $*(\text{tipo})p$ ' individua un'area di memoria che parte dalla posizione indicata dal puntatore e si estende per la dimensione del tipo indicato. In altre parole, si tratta di un cast con il quale si trasforma il tipo di puntatore al volo, ma per questo occorre mostrare un esempio.

Listato 66.194. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/89Ev895Myz>, <http://ideone.com/2A0nj>.

```
#include <stdio.h>
int main (int argc, char *argv[])
{
    int x = 10;
```

```
void *p = &x;
printf ("%i\n", *(int *) p);
return 0;
}
```

In questo caso, il puntatore *p* è di tipo indefinito (**void**) e riceve l'indirizzo della variabile *x*. Successivamente, il valore a cui punta *p* viene usato all'interno della funzione *printf()*, ma prima di essere dereferenziato, viene convertito in un puntatore di tipo **int \***.

### 66.5.10 Stringhe



Le stringhe, nel linguaggio C, non sono un tipo di dati a sé stante; si tratta solo di array di caratteri con una particolarità: l'ultimo carattere è sempre zero, ovvero una sequenza di bit a zero, che si rappresenta simbolicamente come carattere con **'\0'**. In questo modo, si evita di dover accompagnare le stringhe con l'informazione della loro lunghezza.

Pertanto, va osservato che una stringa è sempre un array di caratteri, ma un array di caratteri non è necessariamente una stringa, in quanto per esserlo occorre che l'ultimo elemento sia il carattere **'\0'**. Seguono alcuni esempi che servono a comprendere questa distinzione.


```
char c[20];
```

L'esempio mostra la dichiarazione di un array di caratteri, senza specificare il suo contenuto. Per il momento non si può parlare di stringa, soprattutto perché per essere tale, la stringa deve contenere dei caratteri.

```
char c[] = {'c', 'i', 'a', 'o'};
```

Questo esempio mostra la dichiarazione di un array di quattro caratteri. All'interno delle parentesi quadre non è stata specificata la dimensione perché questa si determina dall'inizializzazione. Anche in questo caso non si può ancora parlare di stringa, perché manca la terminazione.

```
char z[] = {'c', 'i', 'a', 'o', '\0'};
```

Questo esempio mostra la dichiarazione di un array di cinque caratteri corrispondente a una stringa vera e propria. L'esempio seguente è tecnicamente equivalente, solo che utilizza una rappresentazione più semplice: 

```
char z[] = "ciao";
```

Pertanto, la stringa rappresentata dalla costante `"ciao"` è un array di cinque caratteri, perché, pur senza mostrarlo, include implicitamente anche la terminazione.

L'indicazione letterale di una stringa può avvenire attraverso sequenze separate, senza l'indicazione di alcun operatore di concatenamento. Per esempio, `"ciao amore\n"` è perfettamente uguale a `"ciao " "amore" "\n"` che viene inteso come una costante unica.

In un sorgente C ci sono varie occasioni di utilizzare delle stringhe letterali (delimitate attraverso gli apici doppi), senza la necessità di dichiarare l'array corrispondente. Però è importante tenere presente la natura delle stringhe per sapere come comportarsi con loro. Per prima cosa, bisogna rammentare che la stringa, anche se espressa in forma letterale, è un array di caratteri; come tale restituisce sem-

😊 plicemente il puntatore del primo di questi caratteri (salvo le stesse eccezioni che riguardano tutti i tipi di array).

```
char *p;
...
p = "ciao";
...
```

☹️ L'esempio mostra il senso di quanto affermato: non esistendo un tipo di dati «stringa», si può assegnare una stringa solo a un puntatore al tipo '**char**' (ovvero a una variabile di tipo '**char \***'). L'esempio seguente non è valido, perché non si può assegnare un valore alla variabile che rappresenta un array, dal momento che il puntatore relativo è un valore costante:

```
char z[];
...
z = "ciao";      // Non si può.
...
```

Quando si utilizza una stringa tra gli argomenti della chiamata di una funzione, questa riceve il puntatore all'inizio della stringa. In pratica, si ripete la stessa situazione già vista per gli array in generale.

Listato 66.201. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/qCgdnWnE>, <http://ideone.com/kkzaT>.

```
#include <stdio.h>

void elabora (char *z)
{
    printf (z);
}

int main (void)
```



```
{
    elabora ("ciao\n");
    return 0;
}
```

L'esempio mostra una funzione banale che si occupa semplicemente di emettere la stringa ricevuta come parametro, utilizzando *printf()*. La variabile utilizzata per ricevere la stringa è stata dichiarata come puntatore al tipo 'char' (ovvero come puntatore di tipo 'char \*'), poi tale puntatore è stato utilizzato come argomento per la chiamata della funzione *printf()*. Volendo scrivere il codice in modo più elegante si potrebbe dichiarare apertamente la variabile ricevente come array di caratteri di dimensione indefinita. Il risultato è lo stesso.

Listato 66.202. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/83oK83pP4x>, <http://ideone.com/uwq6D>.

```
#include <stdio.h>

void elabora (char z[])
{
    printf (z);
}

int main (void)
{
    elabora ("ciao\n");
    return 0;
}
```

Tabella 66.203. Funzioni comuni per la gestione delle stringhe, definite nel file ‘string.h’ (il modificatore ‘restrict’ viene descritto in una sezione apposita).

| Funzione                                                                                                                                                                                                                     | Descrizione                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>char *strcpy (char *restrict <i>dst</i>,               const char *restrict <i>org</i>); char *strncpy (char *restrict <i>dst</i>,               const char *restrict <i>org</i>,               size_t <i>n</i>);</pre> | <p>La funzione <i>strcpy()</i> copia il contenuto della stringa <i>org</i> nella stringa <i>dst</i>, compreso il carattere di terminazione &lt;NUL&gt;. Perché l’operazione possa avvenire è necessario che le due stringhe non si sovrappongano e che per la stringa di destinazione ci sia abbastanza spazio per i caratteri da copiare. La funzione restituisce il puntatore all’inizio della stringa di destinazione.</p> <p>La funzione <i>strncpy()</i> si comporta sostanzialmente come <i>strcpy()</i>, con la differenza che copia al massimo <i>n</i> caratteri, aggiungendo comunque il carattere di terminazione &lt;NUL&gt;.</p> |

| Funzione                                                                                                                                                                                                                     | Descrizione                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>char *strcat (char *restrict <i>dst</i>,               const char *restrict <i>org</i>); char *strncat (char *restrict <i>dst</i>,               const char *restrict <i>org</i>,               size_t <i>n</i>);</pre> | <p>La funzione <i>strcat()</i> accoda alla stringa <i>dst</i> il contenuto della stringa <i>org</i>, sovrascrivendo il carattere <i>&lt;NUL&gt;</i> che concludeva la prima stringa e aggiungendolo comunque alla fine della copia. Perché l'operazione possa avvenire è necessario che le due stringhe non si sovrappongano e, soprattutto, che ci sia abbastanza spazio disponibile dopo la prima stringa da estendere. La funzione restituisce il puntatore alla prima stringa. La funzione <i>strncat()</i> si comporta sostanzialmente come <i>strcat()</i>, con la differenza che copia al massimo <i>n</i> caratteri dalla seconda stringa, aggiungendo comunque il carattere di terminazione <i>&lt;NUL&gt;</i>.</p> |

| Funzione                                                                                                                                                                                                                                                                         | Descrizione                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>int strcmp (const char *<i>str_1</i>,             const char *<i>str_2</i>); int strcoll (const char *<i>str_1</i>,             const char *<i>str_2</i>); int strncmp (const char *<i>str_1</i>,             const char *<i>str_2</i>,             size_t <i>n</i>);</pre> | <p>La funzione <i>strcmp()</i> confronta due stringhe e restituisce zero nel caso siano uguali, oppure un valore minore di zero se la prima stringa è minore della seconda, oppure un valore maggiore di zero se la prima stringa è maggiore della seconda.</p> <p>La funzione <i>strcoll()</i> funziona sostanzialmente come <i>strcmp()</i>, con la differenza che il confronto ha luogo tenendo conto della configurazione locale (precisamente la categoria 'LC_COLLATE').</p> <p>La funzione <i>strncmp()</i> si comporta sostanzialmente come <i>strcmp()</i>, con la differenza che confronta al massimo <i>n</i> caratteri.</p> |

| Funzione                                                                                                                                          | Descrizione                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>char *strchr (const char *<i>str</i>,               int <i>c</i>); char *strrchr (const char *<i>str</i>,               int <i>c</i>);</pre> | <p>La funzione <i>strchr()</i> cerca nella stringa <i>str</i> il carattere <i>c</i> (il carattere che si ottiene riducendo il valore di <i>c</i> a quello di un tipo 'char'), includendo nella ricerca anche il carattere di terminazione &lt;NUL&gt;. La funzione restituisce un puntatore al carattere trovato, oppure restituisce il puntatore nullo se questo non c'è.</p> <p>La funzione <i>strrchr()</i> si comporta sostanzialmente come <i>strchr()</i>, con la differenza che cerca l'ultima corrispondenza disponibile nella stringa.</p> |

| Funzione                                                                                                                                                                        | Descrizione                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>char *strpbrk (const char *<i>str_1</i>,               const char *<i>str_2</i>);</pre>                                                                                    | <p>La funzione <i>strpbrk()</i> cerca nella stringa <i>str_1</i> la prima corrispondenza con uno qualsiasi dei caratteri contenuti nella stringa <i>str_2</i>. Restituisce il puntatore al carattere trovato nella stringa <i>str_1</i> che soddisfi la condizione; se non trova alcuna corrispondenza restituisce il puntatore nullo.</p>                                                                                            |
| <pre>size_t strspn (const char *<i>str_1</i>,               const char *<i>str_2</i>); size_t strcspn (const char *<i>str_1</i>,               const char *<i>str_2</i>);</pre> | <p>La funzione <i>strspn()</i> conta la lunghezza massima della sottostringa iniziale di <i>str_1</i> che contiene soltanto caratteri dell'insieme contenuto nella stringa <i>str_2</i>.</p> <p>La funzione <i>strcspn()</i> svolge il compito opposto, di contare la lunghezza massima della sottostringa iniziale di <i>str_2</i>, contenente solo caratteri che <b>non</b> fanno parte dell'insieme contenuto in <i>str_2</i>.</p> |

| Funzione                                           | Descrizione                                                                                                                                         |
|----------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>size_t strlen (const char *<i>str</i>);</pre> | La funzione <i>strlen()</i> restituisce la quantità di caratteri contenuta nella stringa, escluso il carattere di terminazione <i>&lt;NUL&gt;</i> . |

All'inizio del capitolo, in occasione della descrizione delle costanti letterali per i tipi di dati primitivi, è già descritto il modo con cui si possono rappresentare alcuni caratteri speciali attraverso delle sequenze di escape che vengono annotate qui, nuovamente, per maggiore comodità del lettore, in quanto quelle sequenze sono valide anche nelle stringhe letterali.

Tabella 66.204. Elenco dei modi di rappresentazione delle costanti carattere attraverso codici di escape.

| Codice di escape | Descrizione                                                                                                                  |
|------------------|------------------------------------------------------------------------------------------------------------------------------|
| \ooo             | Notazione ottale.                                                                                                            |
| \xhh             | Notazione esadecimale.                                                                                                       |
| \\               | Una singola barra obliqua inversa ('\ <i>\</i> ').                                                                           |
| \'               | Un apice singolo destro.                                                                                                     |
| \"               | Un apice doppio.                                                                                                             |
| \?               | Un punto interrogativo. Si usa in quanto le sequenze <i>trigraph</i> sono formate da un prefisso di due punti interrogativi. |

| Codice di escape | Descrizione                                                   |
|------------------|---------------------------------------------------------------|
| <code>\0</code>  | Il codice <code>&lt;NUL&gt;</code> .                          |
| <code>\a</code>  | Il codice <code>&lt;BEL&gt;</code> ( <i>bell</i> ).           |
| <code>\b</code>  | Il codice <code>&lt;BS&gt;</code> ( <i>backspace</i> ).       |
| <code>\f</code>  | Il codice <code>&lt;FF&gt;</code> ( <i>formfeed</i> ).        |
| <code>\n</code>  | Il codice <code>&lt;LF&gt;</code> ( <i>linefeed</i> ).        |
| <code>\r</code>  | Il codice <code>&lt;CR&gt;</code> ( <i>carriage return</i> ). |
| <code>\t</code>  | Una tabulazione orizzontale ( <code>&lt;HT&gt;</code> ).      |
| <code>\v</code>  | Una tabulazione verticale ( <code>&lt;VT&gt;</code> ).        |

### 66.5.11 Parametri della funzione `main()`



La funzione ***main()***, se viene dichiarata con i suoi parametri tradizionali, permette di acquisire la riga di comando utilizzata per avviare il programma. La dichiarazione completa è la seguente:

```
int main (int argc, char *argv[])
{
    ...
}
```

Gli argomenti della riga di comando vengono convertiti in un array di stringhe (cioè di puntatori a `'char'`), in cui il primo elemento è il nome utilizzato per avviare il programma e gli elementi successivi



sono gli altri argomenti. Il primo parametro, *argc*, serve a contenere la quantità di elementi del secondo, *argv[]*, il quale è l'array di stringhe da scandire. È il caso di annotare che questo array dovrebbe avere sempre almeno un elemento: il nome utilizzato per avviare il programma e, di conseguenza, *argc* è sempre maggiore o uguale a uno.<sup>21</sup>

L'esempio seguente mostra in che modo gestire tale array, con la semplice riemissione degli argomenti attraverso lo standard output. 😊

```
#include <stdio.h>

int main (int argc, char *argv[])
{
    int i;

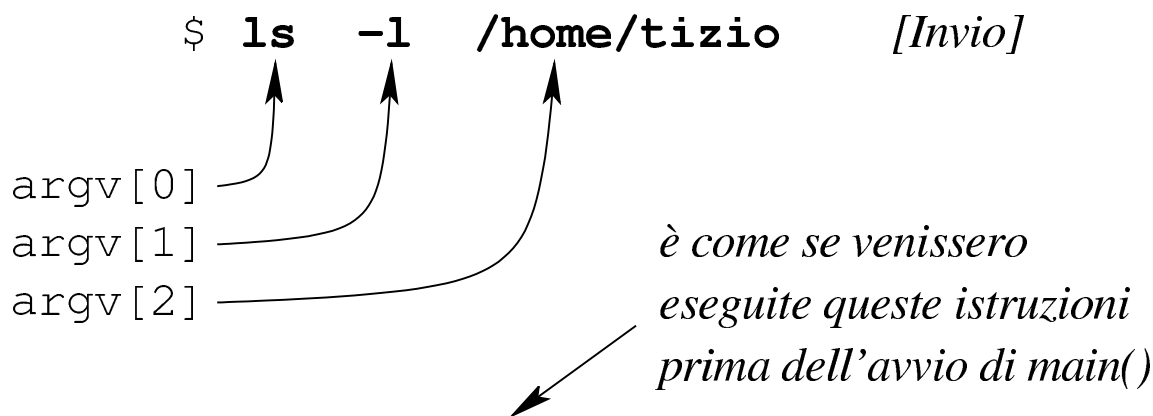
    printf ("Il programma si chiama %s\n", argv[0]);

    for (i = 1; i < argc; i++)
    {
        printf ("argomento n. %i: %s\n", i, argv[i]);
    }
}
```

In alternativa, ma con lo stesso effetto, l'array di puntatori a stringhe può essere definito nel modo seguente, come puntatore di puntatori a caratteri: 😊

```
int main (int argc, char **argv)
{
    ...
}
```

Figura 66.208. Schematizzazione di ciò che accade alla chiamata della funzione *main()*, con un esempio.



```

int argc = 3;
char *argv[argc];
argv[0] = "/bin/ls";
argv[1] = "-l";
argv[2] = "/home/tizio";
main (argc, argv);
  
```

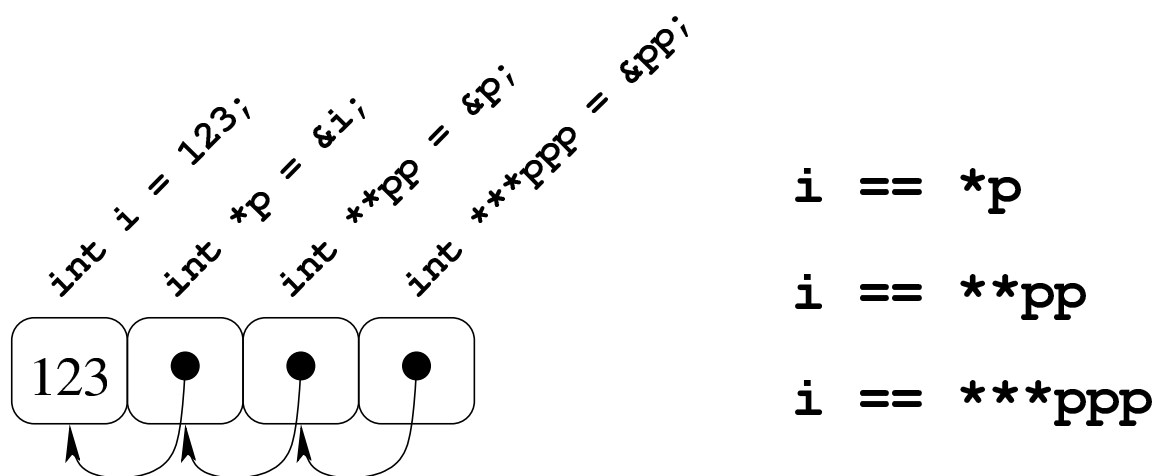
Chi è abituato a utilizzare linguaggi di programmazione più evoluti del C, può trovare strano che non si possa scrivere `main (int argc, char argv[][])` e usare di conseguenza l'array. Il motivo per cui ciò non è possibile dipende dal fatto che gli array a più dimensioni sono ottenuti attraverso sottoinsiemi uniformi del tipo dichiarato, così, in questo caso le stringhe dovrebbero essere della stessa dimensione, ma evidentemente ciò non corrisponde alla realtà. Inoltre, la dichiarazione della funzione dovrebbe contenere le dimensioni dell'array che non possono essere note. Pertanto, un array formato da stringhe diseguali, può essere ottenuto solo come array di puntatori al tipo `char`.

## 66.5.12 Puntatori a puntatori

Una variabile puntatore potrebbe fare riferimento a un'area di memoria contenente a sua volta un puntatore per un'altra area. Per dichiarare una cosa del genere, si possono usare più asterischi, come nell'esempio seguente:

```
int i = 123;
int *p = &i;           // Puntatore al tipo "int".
int **pp = &p;        // Puntatore di puntatore al tipo
                       // "int".
int ***ppp = &pp;     // Puntatore di puntatore di
                       // puntatore al tipo "int".
```

Il risultato si potrebbe rappresentare graficamente come nello schema seguente:



Per dimostrare in pratica il funzionamento di questo meccanismo di riferimenti successivi, si può provare con il programma seguente.

Listato 66.211. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/cettwOsS>, <http://ideone.com/pzBoW>.

```
#include <stdio.h>
int
```

```
main (void)
{
    int i = 123;
    int *p = &i;           // Puntatore al tipo "int".
    int **pp = &p;        // Puntatore di puntatore al tipo
                          // "int".
    int ***ppp = &pp;     // Puntatore di puntatore di puntatore
                          // al tipo "int".

    printf ("i, p, pp, ppp: %i, %u, %u, %u\n",
            i, (unsigned int) p, (unsigned int) pp,
            (unsigned int) ppp);

    printf ("i, p, pp, *ppp: %i, %u, %u, %u\n",
            i, (unsigned int) p, (unsigned int) pp,
            (unsigned int) *ppp);

    printf ("i, p, *pp, **ppp: %i, %u, %u, %u\n",
            i, (unsigned int) p, (unsigned int) *pp,
            (unsigned int) **ppp);

    printf ("i, *p, **pp, ***ppp: %i, %i, %i, %i\n",
            i, *p, **pp, ***ppp);

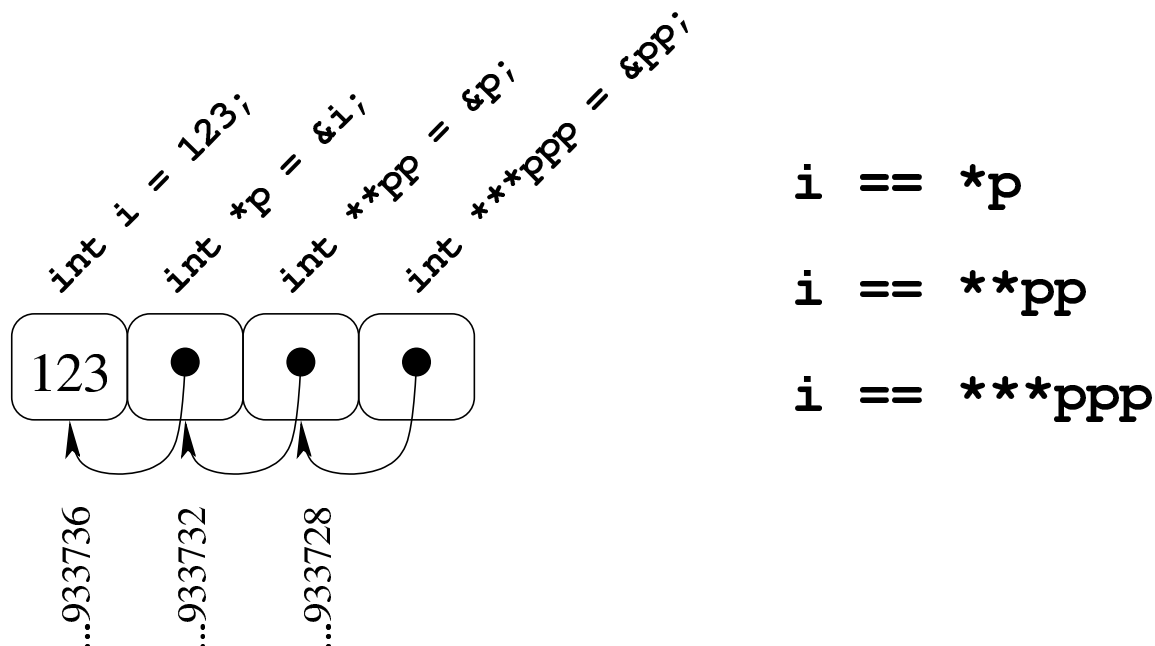
    return 0;
}
```

Eseguendo il programma si dovrebbe ottenere un risultato simile a quello seguente, dove si può verificare l'effetto delle dereferenziazioni applicate alle variabili puntatore:

```
i, p, pp, ppp: 123, 3217933736, 3217933732, 3217933728
i, p, pp, *ppp: 123, 3217933736, 3217933732, 3217933732
i, p, *pp, **ppp: 123, 3217933736, 3217933736, 3217933736
```

```
i, *p, **pp, ***ppp: 123, 123, 123, 123
```

Pertanto si può ricostruire la disposizione in memoria delle variabili:



Come si può comprendere facilmente, la gestione di puntatori a puntatore è difficile e va usata con prudenza e solo quando ne esiste effettivamente l'utilità. Va notato anche che si ottiene la dereferenziazione (la traduzione di un puntatore nel contenuto di ciò a cui punta) usando la notazione tipica degli array, ma questo fatto viene descritto nella sezione successiva.

### 66.5.13 Puntatori a più dimensioni

Un array di puntatori consente di realizzare delle strutture di dati ad albero, non più uniformi come invece devono essere gli array a più dimensioni consueti. L'esempio seguente mostra la dichiarazione di tre array di interi, con una quantità di elementi disomogenea, e la successiva dichiarazione di un array di puntatori di tipo '`int *`', a cui si assegnano i riferimenti ai tre array precedenti. Nell'esempio

appare poi un tipo di notazione per accedere ai dati terminali che dovrebbe risultare intuitiva, ma se ne possono usare delle altre.

Listato 66.214. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/MPES5c6X>, <http://ideone.com/5bP39>.

```
#include <stdio.h>

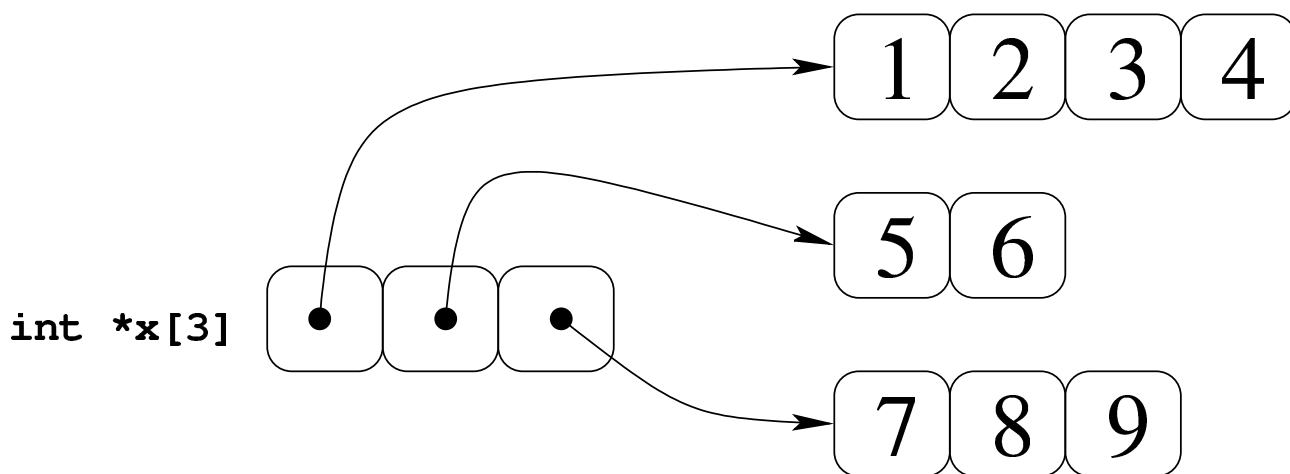
int main (void)
{
    int a[] = {1, 2, 3, 4};
    int b[] = {5, 6,};
    int c[] = {7, 8, 9};
    int *x[] = {a, b, c};

    printf ("*x[0] = {%i, %i, %i, %i}\n",
           *x[0], *(x[0]+1), *(x[0]+2), *(x[0]+3));
    printf ("*x[1] = {%i, %i}\n", *x[1], *(x[1]+1));
    printf ("*x[2] = {%i, %i, %i}\n",
           *x[2], *(x[2]+1), *(x[2]+2));

    return 0;
}
```

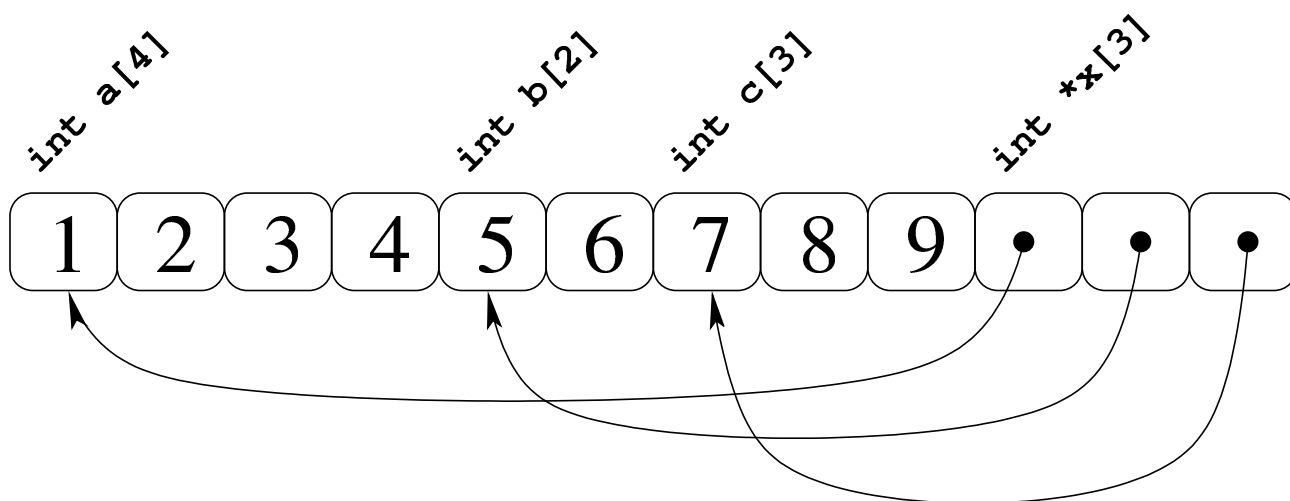
La figura successiva dovrebbe facilitare la comprensione del senso dell'array di puntatori. Come si può osservare, per accedere agli elementi degli array a cui puntano quelli di *x* è necessario dereferenziare gli elementi. Pertanto, `*x[0]` corrisponde al contenuto del primo elemento del primo sotto-array, `*(x[0]+1)` corrisponde al contenuto del secondo elemento del primo sotto-array e così di seguito. Dal momento che i sotto-array non hanno una quantità uniforme di elementi, non è semplice la loro scansione.

Figura 66.215. Schematizzazione semplificata del significato dell'array di puntatori definito nell'esempio.



Si potrebbe obiettare che la scansione di questo array di puntatori a array può avvenire ugualmente in modo sequenziale, come se fosse un array «normale» a una sola dimensione. Molto probabilmente ciò è possibile effettivamente, dal momento che è probabile che il compilatore disponga le variabili in memoria in sequenza, come si vede nella figura successiva, ma ciò non può essere garantito.

Figura 66.216. La disposizione più probabile delle variabili dell'esempio.



Se invece di un array di puntatori si ha un puntatore di puntatori, il meccanismo per l'accesso agli elementi terminali è lo stesso. L'e-

sempio seguente contiene la dichiarazione di un puntatore a puntatori di tipo intero, a cui viene assegnato l'indirizzo dell'array già descritto. La scansione può avvenire nello stesso modo, ma ne viene proposto uno alternativo e più chiaro, con il quale si comprende cosa si intende per puntatore a più dimensioni.

Listato 66.217. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/kDRp85cQ85> , <http://ideone.com/JaqDH>.

```
#include <stdio.h>

int main (void)
{
    int a[] = {1, 2, 3, 4};
    int b[] = {5, 6,};
    int c[] = {7, 8, 9};
    int *x[] = {a, b, c};
    int **y = x;

    printf ("*x[0] = {%i, %i, %i, %i}\n", y[0][0], y[0][1],
                                                y[0][2], y[0][3]);
    printf ("*x[1] = {%i, %i}\n", y[1][0], y[1][1]);
    printf ("*x[2] = {%i, %i, %i}\n", y[2][0], y[2][1],
                                                y[2][2]);

    return 0;
}
```

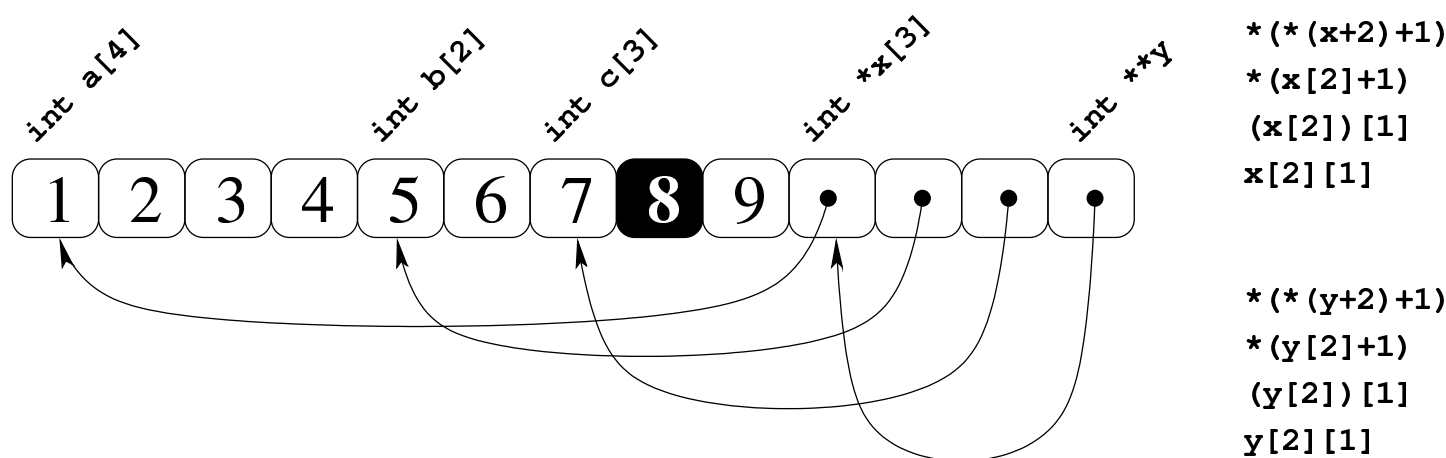
Come si vede, la variabile *y* viene usata come se fosse un array a due dimensioni, ma lo stesso sarebbe valso per la variabile *x*, in qualità di array di puntatori.

Per capire cosa succede, occorre fare mente locale al fatto che il



nome di una variabile puntatore seguito da un numero tra parentesi quadre corrisponde alla dereferenziazione dell' $n$ -esimo elemento successivo alla posizione a cui punta tale variabile, mentre il valore puntato in sé corrisponde all'elemento zero (ciò è come dire che  $*p$  equivale a  $p[0]$ ). Quindi, scrivere  $*(p+n)$  è esattamente uguale a scrivere  $p[n]$ . Se il valore a cui punta una variabile puntatore è a sua volta un puntatore, per dereferenziarlo occorrono due fasi: per esempio  $**p$  è il valore che si ottiene dereferenziano il primo puntatore e quello che si trova nella prima destinazione (quindi  $**p$  equivale a  $*p[0]$  e a  $p[0][0]$ ). Volendo gestire gli indici si possono considerare equivalenti i puntatori:  $*(*(p+m)+n)$ ,  $*(p[m]+n)$ ,  $(p[m])[n]$  e  $p[m][n]$ .

Figura 66.218. Tanti modi alternativi per raggiungere lo stesso elemento.



Seguendo lo stesso ragionamento si possono gestire strutture ad albero più complesse, con più livelli di puntatori, ma qui non vengono proposti esempi di questo tipo.

Sia l'array di puntatori, sia il puntatore a puntatori, possono essere gestiti con gli indici come se si trattasse di un array a più dimensioni. Pertanto, la notazione  $a[m][n]$  può rappresentare l'elemento  $m, n$  di un array  $a$  ottenuto secondo la rappresentazione «norma-

le» a matrice, oppure secondo uno schema ad albero attraverso dei puntatori: la differenza sta solo nella presenza o meno di elementi costituiti da puntatori.

#### 66.5.14 Puntatori e funzioni

«

Nello standard del linguaggio C, la dichiarazione di una funzione è in pratica la definizione di un puntatore al codice della stessa, un po' come accade con gli array.<sup>22</sup> In generale, è possibile dichiarare dei puntatori a un tipo di funzione definito in base al valore restituito e ai tipi di parametri richiesti, attraverso una forma che richiama quella del prototipo di funzione. Il modello seguente è quello della dichiarazione del prototipo:

```
tipo nome_funzione (tipo_parametro [ nome_parametro ] [ , ... ] ) ;
```

Questo è invece il modello della dichiarazione del puntatore:

```
tipo (*nome_puntatore) (tipo_parametro [ nome_parametro ] [ , ... ] ) ;
```

L'esempio seguente mostra la dichiarazione di un puntatore a una funzione che restituisce un valore di tipo '**int**' e utilizza due parametri di tipo '**int**':

```
int (*f) (int, int);
```

L'esempio seguente è equivalente, con la differenza che si nominano i parametri, anche se ciò è perfettamente inutile, esattamente come nei prototipi delle funzioni:

```
int (*f) (int i, int j);
```

L'assegnamento del puntatore avviene nel modo più semplice possibile, trattando il nome della funzione nello stesso modo in cui si fa con gli array: come un puntatore.

```
int (*f) (int, int);      // Puntatore a funzione.  
int prodotto (int, int); // Prototipo di funzione descritta  
                        // più avanti.  
...  
f = prodotto; // Il puntatore «f» contiene il riferimento  
              // alla funzione.
```

Una volta assegnato il puntatore, si può eseguire una chiamata di funzione semplicemente utilizzando il puntatore, per cui, i due esempi seguenti sono equivalenti:

```
i = f (2, 3);
```

```
i = prodotto (2, 3);
```

Nel linguaggio C precedente allo standard ANSI, perché il puntatore potesse essere utilizzato in una chiamata di funzione, occorreva indicare l'asterisco, in modo da dereferenziarlo:

```
i = (*f) (2, 3);      // Non serve più.
```

Per concludere viene mostrato un esempio completo, anche se banalizzato: la funzione *f()* restituisce un numero intero ottenuto incrementando di una unità l'argomento ricevuto. Questa funzione viene chiamata attraverso un puntatore denominato *pf*.

Listato 66.225. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/rbXNWbOh>, <http://ideone.com/L6ooG>.

```
#include <stdio.h>

int f (int i)
{
    return (i + 1);
}

int main (void)
{
    int x = 4;
    int y;
    int (*pf) (int i);
    pf = f;
    y = pf (x);
    printf ("%i + 1 = %i\n", x, y);
    return 0;
}
```

### Riquadro 66.226. Confusione tra le dichiarazioni.

L'interpretazione umana del linguaggio, a proposito dei puntatori, può essere complicata, pertanto l'uso dei puntatori deve essere fatto con criterio, senza abusarne. Gli esempi seguenti sono solo i più semplici:

```
int f (...); /* dichiarazione della funzione f() che restituisce un valore intero; */
```

```
int *f (...); /* dichiarazione della funzione f() che restituisce un puntatore a un intero; */
```

```
int (*f) (...); /* dichiarazione del puntatore f a una funzione che restituisce un intero; */
```

```
int *(*f) (...); /* dichiarazione del puntatore f a una funzione che restituisce un puntatore a un intero. */
```

Ancora più difficile sarebbe dichiarare una funzione che restituisce un array, o peggio, un puntatore a un array.

#### 66.5.14.1 Puntatori a funzione, membri di una struttura

Le strutture sono descritte in un'altra sezione (66.7), tuttavia è opportuno annotare qui in che modo possa essere utilizzato un puntatore a una funzione, quando è un membro di una struttura: «

```
struttura . membro (argomenti);
```

```
(*struttura . membro) (argomenti);
```

I due modelli sono equivalenti e si riferiscono alla chiamata di una funzione, il cui puntatore è costituito dalla variabile *struttura . membro*. È evidente che risulta più comprensibile la prima delle due modalità. A titolo di esempio, ipotizzando la struttura

*totale* e il membro *sottrai*, per una funzione che riceve un argomento di tipo intero (precisamente il numero 7), la chiamata potrebbe essere scritta indifferentemente nei due modi successivi:

```
...
totale.sottrai (7);
...
```

```
...
(*totale.sottrai) (7);
...
```

### 66.5.15 Puntatori a variabili distrutte

<<

L'esempio seguente potrebbe funzionare, ma contiene un errore di principio. 

Listato 66.229. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/DRtSxmiS>, <http://ideone.com/egses>.

```
#include <stdio.h>

double *f (void)
{
    double x = 1234.5678;
    return &x;                // Orrore!
}

int main (int argc, char *argv[])
{
    double *p;
    p = f ();
    printf ("x = %f\n", *p);
    return 0;
}
```

```
}
```

La funzione  $f()$  dichiara localmente una variabile che inizializza al valore 1234,5678, quindi restituisce il puntatore a questa variabile. A parte il fatto che il compilatore possa segnalare o meno la cosa, non si può utilizzare un puntatore rivolto a un'area di memoria che, almeno teoricamente, non è più allocata. In altri termini, se si costruisce un puntatore a qualcosa, occorre tenere sempre presente il ciclo di vita della sua destinazione e non solo della variabile che contiene tale riferimento.

Purtroppo questa attenzione non viene imposta e, generalmente, il compilatore consente di usare un puntatore a variabili che, formalmente, sono già state distrutte.

### 66.5.16 Puntatore nullo

Il linguaggio C prescrive che si possa assegnare a una variabile puntatore il valore zero, in qualità di numero intero:

```
...  
double *p = 0;  
...
```

Il puntatore che contiene il valore zero è indefinito, nel senso che punta a un'area di memoria irraggiungibile. Un puntatore di questo tipo è noto come **puntatore nullo** o *null pointer*; inoltre, due puntatori nulli, qualunque sia il tipo di dati a cui si riferiscono, sono uguali in una comparazione. Pertanto si potrebbe verificare la validità di un puntatore nel modo seguente:

```
...
char *p = 0;
...
if (p == 0)
{
    // Null pointer.
    ...
}
...
```

A ogni modo, lo standard prescrive che nel file ‘`stddef.h`’ sia definita la macro-variabile *NULL*, a rappresentare formalmente un puntatore nullo:

```
#include <stddef.h>
...
char *p = NULL;
...
if (p == NULL)
{
    // Null pointer.
    ...
}
...
```

Va osservato che la variabili puntatore, quando acquisiscono un indirizzo in base al verificarsi di certe condizioni, vanno inizializzate opportunamente al valore nullo (come già apparso negli esempi), in modo da poter poi verificare se hanno ottenuto o meno un tale indirizzo.



## 66.5.17 Utilizzo della memoria in modo dinamico



L'allocazione dinamica della memoria avviene generalmente attraverso la funzione *malloc()*, oppure *calloc()*, definite nella libreria standard, secondo i prototipi contenuti nel file 'stdlib.h'. Se queste riescono a eseguire l'operazione, restituiscono il puntatore alla memoria allocata, altrimenti restituiscono il valore 'NULL'.

```
void *malloc (size_t dimensione);
```

```
void *calloc (size_t quantità, size_t dimensione);
```

La differenza tra le due funzioni sta nel fatto che la prima, *malloc()*, viene utilizzata per allocare un'area di una certa dimensione, espressa generalmente in byte, mentre la seconda, *calloc()*, permette di indicare una quantità di elementi e si presta per l'allocazione di array.

Dovendo utilizzare queste funzioni per allocare della memoria, è necessario conoscere la dimensione dei tipi primitivi di dati, ma per evitare incompatibilità conviene farsi aiutare dall'operatore '**sizeof**'.

Il valore restituito da queste funzioni è di tipo '**void \***' cioè una specie di puntatore neutro, indipendente dal tipo di dati da utilizzare (in quanto il tipo '**void**', in sé, rappresenta una variabile di rango nullo, la quale non può contenere alcun dato). Per questo, in linea di principio, prima di assegnare a un puntatore il risultato dell'esecuzione di queste funzioni di allocazione, è opportuno eseguire un cast.

```
int *pi = NULL;
...
pi = (int *) malloc (sizeof (int));

if (pi != NULL)
{
    // Il puntatore è valido e allora procede.
    ...
}
else
{
    // La memoria non è stata allocata e si fa qualcosa
    // di alternativo.
    ...
}
```

Come si può osservare dall'esempio, il cast viene eseguito con la notazione '**(int \*)**' che richiede la conversione esplicita in un puntatore a '**int**'. Lo standard C non richiede l'utilizzo di questo cast, quindi l'esempio si può ridurre al modo seguente:

```
...
pi = malloc (sizeof (int));
...
```

La memoria allocata dinamicamente deve essere liberata in modo esplicito quando non serve più. Infatti, il linguaggio C non offre alcun meccanismo di *raccolta della spazzatura* o *garbage collector*. Per questo si utilizza la funzione *free()* che richiede semplicemente il puntatore e non restituisce alcunché.

```
void free (void *puntatore);
```

È necessario evitare di deallocare più di una volta la stessa area di memoria, perché ciò potrebbe provocare effetti imprevedibili.

```
int *pi = NULL;
...
pi = (int *) malloc (sizeof (int));

if (pi != NULL)
{
    // Il puntatore è valido e allora procede.
    ...
    free (pi); // Libera la memoria
    pi = NULL; // e per sicurezza azzerà il puntatore.
    ...
}
else
{
    // La memoria non è stata allocata e si fa qualcosa
    // di alternativo.
    ...
}
```

Lo standard prevede una funzione ulteriore, per la riallocazione di memoria: *realloc()*. Questa funzione si usa per ridefinire l'area di memoria con una dimensione differente:

```
void *realloc (void *puntatore, size_t dimensione);
```

In pratica, la riallocazione deve rendere disponibili gli stessi contenuti già utilizzati, salvo la possibilità che questi siano stati ridotti nella parte terminale. Se invece la dimensione richiesta nella rial-

locazione è maggiore di quella precedente, lo spazio aggiunto può contenere dati casuali. Va osservato che la collocazione in memoria, successiva alla riallocazione, può essere differente da quella precedente. Il funzionamento di *realloc()* non è garantito, pertanto occorre verificare nuovamente, dopo il suo utilizzo, che il puntatore ottenuto sia ancora valido.

### 66.5.18 Puntatori «ristretti»

«

Lo standard del linguaggio C prevede il modificatore ‘**restrict**’ per le variabili puntatore, da usare come nell’esempio seguente:

```
...  
int *restrict p;  
...
```

L’utilizzo di tale modificatore equivale a una dichiarazione di intenti (ovvero una promessa) che il programmatore fa al compilatore, nei riguardi del puntatore. Precisamente si dichiara che il puntatore viene usato per accedere ad aree di memoria in modo esclusivo, nel senso che nell’ambito del contesto a cui si fa riferimento, non esistono altri accessi alle stesse aree per mezzo di altri puntatori o di altre variabili. Partendo da questo presupposto, il compilatore può ottimizzare il risultato della compilazione semplificando il codice finale.

La definizione formale del significato di questo modificatore è molto complessa e il compilatore non è in grado di segnalarne un uso improprio. Ciò significa che va usata questa possibilità con prudenza, solo quando si ritiene di averne capito il senso e l’utilità.

Come esempio iniziale si può osservare il prototipo della funzione standard *strcpy()*:

```
char *strcpy (char *restrict dst, const char *restrict org);
```

Ci sono due parametri costituiti da stringhe che non devono risultare sovrapposte e in questo caso, il vincolo **'restrict'** è appropriato per esprimere il concetto: se entrambi i puntatori delle stringhe sono dichiarati con il modificatore **'restrict'**, è evidente che le stringhe rispettive non devono sovrapporsi.

L'impegno che il programmatore prende utilizzando il modificatore **'restrict'** è finalizzato solo al favorire l'ottimizzazione della compilazione.

La promessa che un programmatore fa dichiarando un puntatore **'restrict'** è limitata al campo di azione del puntatore stesso. Per esempio, tornando all'esempio del prototipo della funzione *strcpy()*, lì si intende che i parametri vengono usati nella funzione senza sovrapposizioni, ma, dato il contesto, rimane il fatto che le stringhe fornite come argomento della chiamata debbano già rispettare il vincolo di non essere sovrapposte.

Esempio 66.237. Viene allocata un'area di memoria composta da 100 elementi della grandezza di un intero normale. I primi 50 elementi vengono scanditi con il puntatore *r1* mentre quelli restanti con il puntatore *r2*. Nell'esempio, agli elementi **'r1[i]'** viene assegnato il valore di **'r2[i]+1'**, anche se il fatto in sé non ha una grande importanza.

```
int *restrict r1, *restrict r2;  
int *m = malloc (100 * sizeof (int));  
int i;
```

```
r1 = m;           // r1 viene usato per i primi 50 elementi.
r2 = m + 50;     // r2 viene usato per i 50 elementi successivi.

for (i = 0; i < 50; i++)
{
    r1[i] = r2[i] + 1;
}
```

Esempio 66.238. Viene allocata un'area di memoria composta da 100 elementi della grandezza di un intero normale. Gli elementi pari vengono scanditi con il puntatore *r1* mentre quelli dispari con il puntatore *r2*. Nell'esempio, agli elementi '*r1* [*j*]' viene assegnato il valore di '*r2* [*j*]+1', anche se il fatto in sé non ha una grande importanza.

```
int *restrict r1, *restrict r2;
int *m = malloc (100 * sizeof (int));
int i;
int j;

r1 = m;           // r1 viene usato per gli elementi con
                  // indice pari.
r2 = m + 1;      // r2 viene usato per gli elementi con
                  // indice dispari.

for (i = 0; i < 50; i++)
{
    j = i * 2;
    r1[j] = r2[j] + 1;
}
```

Se il compilatore non riconosce il modificatore '**restrict**' significa solo che non è in grado di ottimizzare il codice in un certo modo,

ma non è necessario modificare il proprio programma per togliere la parola chiave relativa, perché è sufficiente sfruttare una macrovariabile del precompilatore, a cui non si assegna alcun valore:

```
...  
#define restrict  
...
```

Oppure, se ciò non è possibile, la si dichiara come un commento privo di contenuto:

```
...  
#define restrict /**/  
...
```

## 66.6 Le funzioni

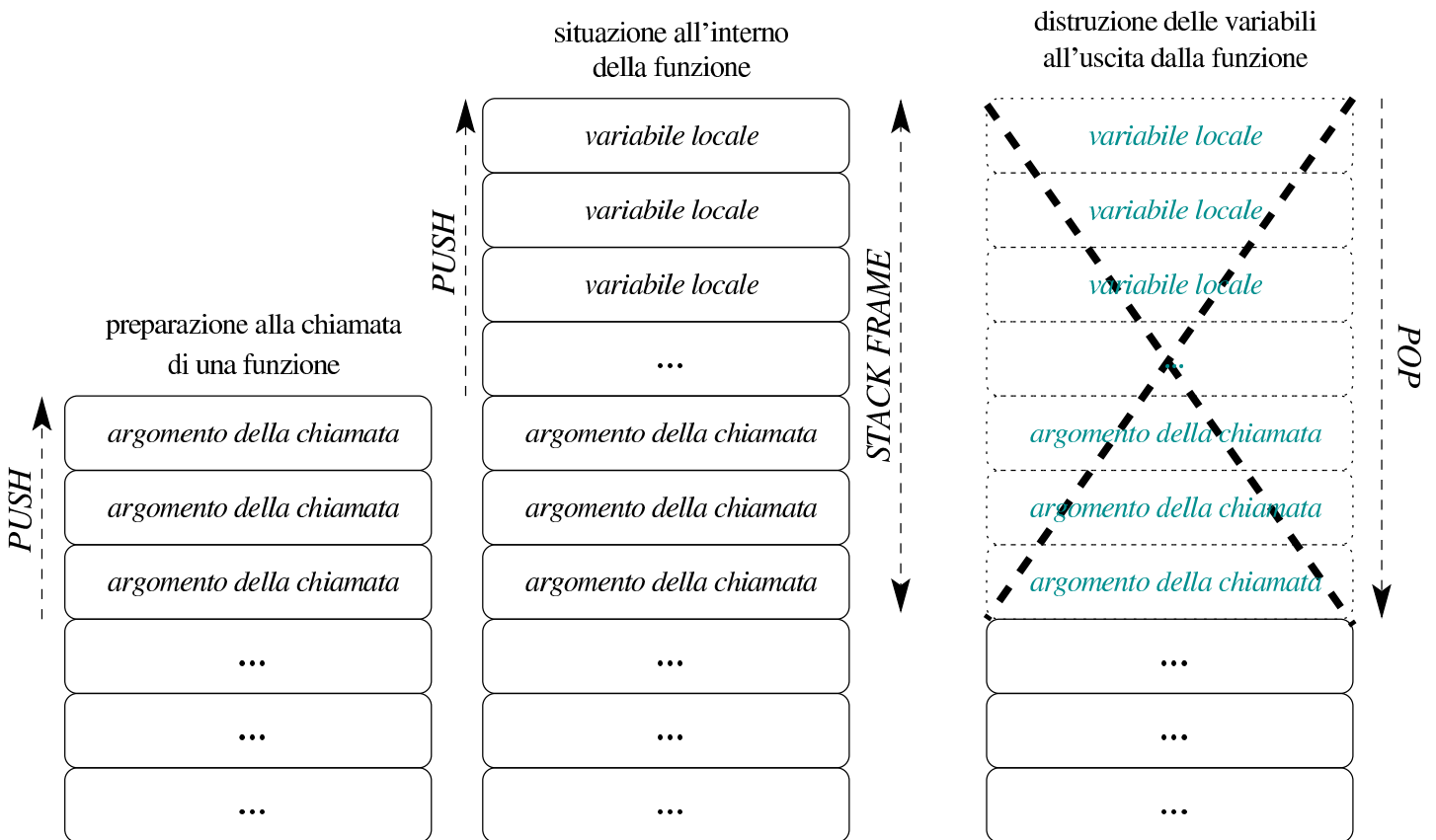
Per comprendere come «funzionano» le funzioni nel linguaggio C, occorre fare mente locale all'uso della pila dei dati con il linguaggio macchina. Qui si chiariscono alcuni concetti, partendo dal ripasso della pila dei dati.

### 66.6.1 Pila dei dati

Dal punto di vista del linguaggio macchina, generalmente si dispone di una pila di dati che si sviluppa a partire da un certo indirizzo di memoria, utilizzando di volta in volta indirizzi inferiori della stessa. Attraverso la pila dei dati, prima della chiamata di una funzione, gli argomenti vengono passati alla stessa aggiungendoli alla pila; successivamente, all'interno della funzione, tutte le variabili locali vengono ottenute facendo crescere ulteriormente la pila. Al termine dell'esecuzione della funzione, la pila viene ridotta allo stato prece-

dente alla chiamata, espellendo le variabili locali e i parametri della chiamata.

Figura 66.241. Semplificazione del meccanismo attraverso cui si passano gli argomenti a una funzione e si gestiscono le variabili locali.



Naturalmente, dal momento che la pila di dati viene gestita attraverso la memoria centrale, la quale consente un accesso diretto ai dati, tramite un indirizzo, nella pila si possono gestire dati di tutti i tipi, volendo anche degli array. A proposito degli array, quando questi sono creati all'interno delle funzioni, pertanto attraverso l'uso della pila dei dati, al compilatore non è necessario sapere preventivamente le dimensioni di questi, perché lo spazio che usano nella memoria è allocato dinamicamente, tramite la pila.



## 66.6.2 Dichiarazione e chiamata di una funzione



La dichiarazione di una funzione prevede l'indicazione del tipo di variabili che compongono i parametri, allo scopo di far sapere al compilatore in che modo inserire gli argomenti nella pila, al momento della chiamata. Si osservi l'esempio seguente in cui si dichiara una funzione con due parametri molto semplici: un intero normale e un intero di dimensione «doppia».

```
void f (int x, long long int y)
{
    ...
    ...
}
```

Partendo dal presupposto che la pila dei dati sia gestita a blocchi di «parole» del microprocessore, si può ipotizzare ragionevolmente in che modo siano impilati gli argomenti della chiamata. Si suppone di chiamare la funzione nel modo seguente e che la parola sia da 32 bit:

```
...
f (0x13579BDF, 0x123456789ABCDEF);
...
```

Alla chiamata della funzione, i parametri dovrebbero apparire nella pila come nella figura successiva, trascurando il problema dell'inversione eventuale dei byte:



Come si vede, gli argomenti vengono impilati in ordine inverso, in modo tale che il primo argomento appaia all'inizio della pila.

Ci sono molti dettagli da definire sul come vadano impilati gli argomenti di una chiamata; in particolare è da chiarire in che modo vadano trattati i dati la cui dimensione è inferiore alla parola del microprocessore, così come per quelli che si articolano in strutture. Questi dettagli vanno chiariti quando si vogliono scrivere funzioni da usare assieme a codice scritto in linguaggio assembleatore, oppure anche per altri linguaggi, se per quelli si utilizzano compilatori non conformi a quello usato per il C.

### 66.6.3 Elenco indefinito di parametri

«


Il linguaggio C ammette che le funzioni siano dichiarate con almeno un parametro esplicito e un elenco indefinito di parametri successivi. In altre parole, si ammette che ci sia un parametro certo e un elenco, eventuale, di altri parametri sconosciuti. Questo avviene, per esempio, con funzioni standard quali *printf()*:

```
int printf (const char *formato, ...);
```

Quando si chiama una funzione del genere, gli argomenti successivi al primo, se riguardano valori numerici, vengono «promossi» in modo tale da avere una dimensione minima di riferimento. Per la precisione, i valori interi di rango inferiore a quello di un intero comune, sono convertiti al livello di intero ‘**int**’ (con segno o senza, in base alle caratteristiche di partenza); i valori in virgola mobile, se sono espressi secondo un formato di rango inferiore a ‘**double**’, vengono trasformati semplicemente in ‘**double**’. Gli interi e i valori in virgola mobile di rango superiore, rimangono invariati.

È da osservare che, se si tenta di passare come argomento un valore che occupa uno spazio inferiore alla dimensione della parola del microprocessore, pur dichiarando tutti i parametri è molto probabile che il compilatore debba utilizzare ugualmente una parola intera, riempiendo in qualche modo lo spazio restante con dati nulli; pertanto, in presenza di parametri di dimensione non stabilita, è più che appropriata la promozione predefinita degli argomenti a valori multipli della parola.

Viene mostrato un esempio di programma contenente una funzione con un numero indefinito di parametri, nella quale, gli argomenti della chiamata vengono comunque estratti dalla pila dei dati, conoscendo le dimensioni usate nella chiamata. L’esempio funziona con un compilatore GNU C e serve solo per comprendere il meccanismo, ma per il momento non rappresenta il modo corretto di agire a

 questo proposito.

Listato 66.245. Per provare il codice attraverso un servizio *pastebin*: <http://ideone.com/JzZOP>.

```
#include <stdio.h>

void f (int w,...)
{
    //
    // Traduce l'indirizzo di «w» nel puntatore «p».
    //
    char *p = (char *) &w;
    //
    // Sposta il puntatore all'inizio del secondo parametro.
    //
    p = p + sizeof w;
    //
    // Mostra il valore del primo e del secondo parametro.
    //
    printf ("w = %i; ", w);
    printf ("x = %Lf; ", *((long double *)p));
    //
    // Sposta il puntatore all'inizio del terzo parametro.
    //
    p = p + sizeof (long double);
    //
    // Mostra il terzo parametro.
    //
    printf ("y = %lli; ", *((long long int *)p));
    //
    // Sposta il puntatore all'inizio del quarto parametro.
    //
    p = p + sizeof (long long int);
    //
}
```

```
// Mostra il quarto parametro.
//
printf ("z = %i\n", *((int *)p));
//
return;
}

int main (int argc, char *argv[])
{
    f (10, (long double) 12.34, (long long int) 13, 14);
    return 0;
}
```

Come si vede, per raggiungere gli argomenti successivi al primo, conoscendo le loro caratteristiche, si scandisce in pratica la memoria occupata dalla pila dei dati, prendendo come riferimento l'indirizzo del primo parametro, il quale costituisce il riferimento certo. Si misura la dimensione del primo parametro e si aggiusta il puntatore in modo da posizionarsi dopo la fine di questo, sapendo che da lì in poi si trovano gli argomenti successivi. Il puntatore è di tipo '**char \***', in modo da poterlo gestire a unità di «caratteri», conformemente al valore prodotto dall'operatore '**sizeof**'. Se tutto funziona come previsto, il programma mostra correttamente il messaggio seguente:

```
w = 10; x = 12.340000; y = 13; z = 14
```

Il modo corretto di estrapolare i valori dei parametri non dichiarati richiede l'uso di alcune macroistruzioni della libreria standard, contenute nel file di intestazione '**stdarg.h**'. Si osservi come va trasformato l'esempio già apparso per rispettare la formalità standard:



Listato 66.247. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/22g6f9rU>, <http://ideone.com/Xuy6s>.

```
#include <stdio.h>
#include <stdarg.h>

void f (int w,...)
{
    //
    // Dichiara le variabili che servono a contenere
    // gli argomenti privi di parametri formali.
    //
    long double x;
    long long int y;
    int z;
    //
    // Dichiara il puntatore ai parametri.
    //
    va_list ap;
    //
    // Posiziona il puntatore dopo il primo parametro,
    // ovvero dopo l'ultimo parametro dichiarato
    // esplicitamente.
    //
    va_start (ap, w);
    //
    // Estrapola il secondo argomento della chiamata (portando
    // avanti il puntatore di conseguenza.
    //
    x = va_arg (ap, long double);
    //
    // Mostra il valore del primo e del secondo argomento
    // ottenuto dalla chiamata della funzione.

```

```
//
printf ("w = %i; ", w);
printf ("x = %Lf; ", x);
//
// Estrapola il terzo argomento.
//
y = va_arg (ap, long long int);
//
// Mostra il terzo argomento.
//
printf ("y = %lli; ", y);
//
// Estrapola il quarto argomento.
//
z = va_arg (ap, int);
//
// Mostra il quarto e ultimo argomento.
//
printf ("z = %i\n", z);
//
// Conclude la scansione degli argomenti.
//
va_end (ap);
//
return;
}

int main (int argc, char *argv[])
{
    f (10, (long double) 12.34, (long long int) 13, 14);
    return 0;
}
```

Come si vede, è necessario incorporare il file di intestazione

'`stdarg.h`' della libreria standard. All'inizio della funzione si dichiara una variabile di tipo '`va_list`' per scandire l'elenco di parametri: si tratta evidentemente di un puntatore (molto probabilmente al tipo '`char`'). Subito dopo si inizializza la variabile da usare per la scansione con la macroistruzione '`va_start`' che ha l'apparenza di una funzione. A '`va_start`' viene passata la variabile da usare come puntatore per gli argomenti e l'ultimo parametro dichiarato espressamente nella funzione, allo scopo di aggiornare il puntatore e di portarlo all'inizio del primo argomento privo di un parametro esplicito. Successivamente si utilizza la macroistruzione '`va_arg`', anche questa con l'apparenza di una funzione, per estrapolare l'argomento a cui punta la variabile di tipo '`va_list`', usata per lo scopo, aggiornando conseguentemente la variabile-puntatore, in modo da essere pronta per l'argomento successivo. Al termine si usa '`va_end`', la quale può essere indifferentemente una macroistruzione o una funzione vera e propria, allo scopo di concludere l'uso del puntatore dichiarato per la scansione dei parametri.

Le macroistruzioni '`va_start`' e '`va_arg`' non potrebbero essere realizzate in forma di funzioni. Infatti, '`va_start`' utilizza apparentemente come argomento l'ultimo parametro della funzione, ma per calcolare la posizione del parametro successivo servirebbe invece l'indirizzo di tale variabile. In modo analogo, la macroistruzione '`va_arg`' richiede l'indicazione del tipo di dati da estrarre, mentre una funzione vera potrebbe accettare solo la dimensione restituita dall'operatore '`sizeof`'; inoltre restituisce un valore dello stesso tipo, mentre una funzione vera può restituire un solo tipo prestabilito.



Nell'esempio non si vede cosa accade quando si trasmette un argomento costituito da un carattere ('**char**'). In tal caso bisogna tenere in considerazione l'effetto della promozione a intero; pertanto, la macroistruzione *va\_arg* va usata indicando un tipo '**int**' (e non un tipo '**char**'). Lo stesso dicasi per i valori in virgola mobile, che vanno estratti prevedendo un formato '**double**', anche se nell'argomento originale dovesse trattarsi di '**float**' (e ammesso che l'argomento non sia espresso in un formato ancora più grande).

#### 66.6.4 Annotazioni su «printf()» e altre funzioni simili

Da quanto descritto a proposito della promozione dei valori numerici, interi o in virgola mobile, si comprende che le rappresentazioni di valori numerici vanno fatte preferibilmente a partire da interi di tipo '**int**' o da valori in virgola mobile di tipo '**double**'. Si osservino gli esempi seguenti:

- ```
printf ("%hd\n", 123);
```

 in linea di principio, lo specificatore di conversione '**%hd**' attende un valore di tipo '**short int**', ma il valore 123 che gli viene fornito è implicitamente di tipo '**int**';
- ```
printf ("%hd\n", (short int) 123);
```

 esattamente come nell'esempio precedente e a nulla serve il tentativo di indicare un cast nell'argomento della chiamata alla funzione;
- ```
printf ("%c\n", 'A');
```

 lo specificatore di conversione '**%c**' si attende un valore di tipo '**char**' (con o senza segno), ma il carattere '**A**' che gli viene fornito è implicitamente di tipo '**int**'.

Nel caso della funzione *scanf()*, questi problemi non ci sono, perché gli argomenti variabili sono costituiti tutti da puntatori ad aree di memoria che devono essere in grado di contenere le informazioni da inserire.

### 66.6.5 Costante predefinita «\_\_func\_\_»

«

Lo standard del linguaggio prescrive che, se all'interno di una funzione viene usato il nome '**\_\_func\_\_**', questo si deve tradurre nel nome della funzione che lo contiene. In pratica, il compilatore che incontra questo nome, dichiara automaticamente, all'interno della funzione, la costante seguente:

```
static const char __func__[] = "nome_funzione";
```

L'esempio seguente mostra in che modo se ne potrebbe fare uso.

Listato 66.251. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/1I0KIqiq>, <http://ideone.com/A9est>.

```
#include <stdio.h>
void f (void)
{
    printf ("Sono nella funzione \"%s\".\n", __func__);
}
int main (int argc, char *argv[])
{
    f ();
    return 0;
}
```

Una volta compilato il programma, eseguendolo si ottiene:

Sono nella funzione "f".

## 66.7 Struttura, unione, campo, enumerazione, costante composta

Fino a questo punto sono stati incontrati solo i tipi di dati primitivi, oltre agli array di questi (incluse le stringhe). Nel linguaggio C, come in altri, è possibile definire dei tipi di dati aggiuntivi, derivati dai tipi primitivi.

Qui si usa la convenzione di nominare le strutture, le unioni e le enumerazioni con una lettera iniziale maiuscola. Per quanto riguarda invece i tipi di dati derivati, ottenuti con l'istruzione `'typedef'`, si segue l'uso comune di aggiungere l'estensione `'_t'`.

### 66.7.1 Enumerazioni

È possibile dichiarare una variabile di tipo enumerativo, costituita tecnicamente da un intero, la quale può rappresentare solo un insieme prestabilito di valori, indicati simbolicamente attraverso delle definizioni. I valori simbolici che possono essere rappresentati sono tradotti in un numero intero, ma il programmatore non dovrebbe avere la necessità di avere a che fare direttamente con tali valori numerici corrispondenti. In altri termini, il tipo enumerativo è una forma di rappresentazione di un intero attraverso costanti mnemoniche.

```
enum nome { costante [, costante] ... }
```

La sintassi indicata mostra il modo in cui si definisce un tipo del genere: all'interno di parentesi graffe si elencano i nomi delle costanti

che possono essere assegnate a una variabile di questo tipo. Tuttavia, alle costanti si può associare un valore intero in modo esplicito; pertanto, la costante può essere espressa così:

```
nome_simbolico [=n]
```

Si osservi l'esempio seguente che comunque non rappresenta un programma completo:

```
...
enum Colore { nero, marrone, rosso, arancio, giallo, verde,
             blu, viola, grigio, bianco, argento=100,
             oro };
...
enum Colore c;    // Dichiara la variabile «c».
...
c = marrone + 1;  // Assegna a «c» il valore successivo a
                 // «marrone»; in pratica assegna il valore
                 // «rosso».
...
if (c <= rosso); // Se il colore va dal
  {              // nero al rosso,
  printf ("Non mi piace: %i\n", c); // visualizza un
  }              // messaggio e mostra
                 // anche il numero
                 // corrispondente.
...
```

All'inizio viene dichiarato il tipo enumerativo 'Colore', come insieme di colori principali, definiti simbolicamente per nome. Va osservato che nel caso dell'argento, viene associato espressamente il valore 100.

In mancanza di associazioni esplicite tra il valore simbolico e valore numerico, il compilatore associa al primo dei simboli il valore zero e dà a quelli successivi un numero ottenuto incrementando di una unità quello precedente. Nel caso dell'esempio, nero corrisponde a zero, marrone a uno, rosso a due e così di seguito fino al bianco. Il colore argento è definito espressamente (quindi dal nove del bianco si salta al 100 dell'argento) e il colore dell'oro viene determinato implicitamente come pari a *argento*+1, ovvero uguale a 101.

Seguendo l'esempio si vede la dichiarazione della variabile *c* di tipo `'enum Colore'`. In pratica, viene dichiarata una variabile di tipo intero, in grado di contenere i valori dell'enumerazione `'Colore'`.

Successivamente si assegna alla variabile *c* la somma tra la costante *marrone* (pari a uno) e il numero uno. In pratica si assegna il valore due, ma in base al contesto si intende di avere assegnato *rosso*.

Alla fine dell'esempio si vede un confronto tra la variabile *c* e un colore di quelli definiti simbolicamente. Di fatto si sta confrontando il valore della variabile con il numero due, ma in pratica sembra di valutare la cosa solo sul piano della sequenza ideale che è stata attribuita a quei colori.

La dichiarazione di una variabile enumerativa coincide quindi con la dichiarazione di un insieme di costanti simboliche, le quali non possono essere ridefinite. Pertanto, non è possibile dichiarare due variabili diverse che condividono costanti simboliche con lo stesso nome, a meno di essere in un campo di azione differente:



```
enum Colori { nero, marrone, rosso, arancio, giallo, verde,  
             blu, viola, grigio, bianco };  
  
enum Bianco_e_nero { nero, bianco };    // Non si può.
```

Le costanti simboliche definite attraverso le enumerazioni, possono essere usate anche al di fuori delle variabili dichiarate espressamente per questo scopo, purché possano ragionevolmente contenerne il valore. È anche evidente che al posto delle enumerazioni definite in questo modo sia possibile gestire direttamente le costanti. L'esempio seguente riporta i passi equivalenti di quanto già visto all'inizio della sezione:

```
...  
const int nero      = 0;  
const int marrone  = 1;  
const int rosso    = 2;  
const int arancio  = 3;  
const int giallo   = 4;  
const int verde    = 5;  
const int blu      = 6;  
const int viola    = 7;  
const int grigio   = 8;  
const int bianco   = 9;  
const int argento  = 100;  
const int oro      = 101;  
...  
int c;              // Dichiarata la variabile «c».  
...  
c = marrone + 1;    // Assegna a «c» il valore successivo a  
                   // «marrone»; in pratica assegna il valore  
                   // «rosso».  
...  
if (c <= rosso);   // Se il colore va dal
```

```
{ // nero al rosso,  
    printf ("Non mi piace: %i\n", c); // visualizza un  
} // messaggio e mostra  
// anche il numero  
// corrispondente.  
...
```

## 66.7.2 Strutture

Gli array sono sequenze di elementi uguali, tutti adiacenti nel modello di rappresentazione della memoria, ideale o reale che sia. In modo simile si possono definire strutture di dati più complesse in cui gli elementi adiacenti siano di tipo differente. Gli elementi che compongono una struttura sono i suoi *membri*. In pratica, una struttura è una sorta di mappa di accesso a un'area di memoria, attraverso i suoi membri.

La variabile contenente una struttura si comporta in modo analogo alle variabili di tipo primitivo, per cui, la variabile che è stata creata a partire da una struttura, rappresenta tutta la zona di memoria occupata dalla struttura stessa e non solo il riferimento al suo inizio. Questa distinzione è importante, per non fare confusione con il comportamento relativo agli array che sono sostanzialmente solo dei puntatori.

La dichiarazione di una struttura si articola in due fasi: la dichiarazione del tipo e la dichiarazione delle variabili che utilizzano quella struttura.

```
struct Datario { int giorno; int mese; int anno; };
```

L'esempio mostra la dichiarazione della struttura **'Datario'** (ovvero del tipo **'struct Datario'**) composta da tre interi dedicati

a contenere rispettivamente: il giorno, il mese e l'anno. In questo caso, trattandosi di tre elementi dello stesso tipo, sarebbe stato possibile utilizzare un array, ma come è possibile vedere in seguito, una struttura può essere conveniente anche in queste situazioni.

È importante osservare che le parentesi graffe sono parte dell'istruzione di dichiarazione della struttura e non rappresentano un blocco di istruzioni. Per questo motivo appare il punto e virgola finale, cosa che potrebbe sembrare strana, specialmente quando la struttura si articola su più righe come nell'esempio seguente:

```
struct Datario {
    int giorno;
    int mese;
    int anno;
}; // Il punto e virgole finale è necessario.
```

La dichiarazione delle variabili che utilizzano la struttura può avvenire contestualmente con la dichiarazione della struttura, oppure in un momento successivo. L'esempio seguente mostra la dichiarazione del tipo '**struct Datario**', seguito da un elenco di variabili che utilizzano quel tipo: *inizio* e *fine*.

```
struct Datario {
    int giorno;
    int mese;
    int anno;
} inizio, fine;
```

Tuttavia, il modo più elegante per dichiarare delle variabili a partire da una struttura è quello seguente:

```
struct Datario inizio, fine;
```

Quando una variabile è stata definita come organizzata secondo una



certa struttura, si accede ai suoi componenti attraverso l'indicazione del nome della variabile stessa, seguita dall'operatore punto ('.') e dal nome dell'elemento particolare.

```
inizio.giorno = 1;
inizio.mese = 1;
inizio.anno = 2012;
...
fine.giorno = inizio.giorno;
fine.mese = inizio.mese +1;
fine.anno = inizio.anno;
```

Una struttura può essere dichiarata in modo anonimo, definendo immediatamente tutte le variabili che fanno uso di quella struttura. La differenza sta nel fatto che la struttura non viene nominata nel momento della dichiarazione e, dopo la definizione dei suoi elementi, devono essere elencate tutte le variabili in questione. Evidentemente, non c'è la possibilità di riutilizzare questa struttura per altre variabili definite in un altro punto, ma soprattutto, come viene mostrato in seguito, diventa impossibile indicare il tipo di struttura come parametro formale di una funzione.



```
struct {
    int giorno;
    int mese;
    int anno;
} inizio, fine;
```

### 66.7.3 Assegnamento, inizializzazione, campo di azione e puntatori delle strutture

Nella sezione precedente si è visto come accedere ai vari componenti della struttura, attraverso una notazione che utilizza l'operatore



punto. Volendo è possibile assegnare a una variabile di questo tipo l'intero contenuto di un'altra che appartiene alla stessa struttura:

```

inizio.giorno = 1;
inizio.mese = 1;
inizio.anno = 2012
...
fine = inizio;
fine.mese++;

```

L'esempio mostra l'assegnamento alla variabile *fine* di tutta la variabile *inizio*. Questo è ammissibile solo perché si tratta di variabili dello stesso tipo, cioè di strutture di tipo 'Datario' (come deriva dagli esempi precedenti). Se invece si trattasse di variabili costruite a partire da strutture differenti, anche se realizzate nello stesso modo, con gli stessi membri, ciò non sarebbe ammissibile.

```

...
struct Datario {int giorno; int mese; int anno;};
struct Giorno  {int giorno; int mese; int anno;};
...
struct Datario ingresso = {31, 12, 2007};
struct Giorno  uscita;
uscita = ingresso;          // Errore: i dati sono incompatibili
...

```

Nel momento della dichiarazione di una struttura, è possibile anche inicializzarla utilizzando una forma simile a quella disponibile per gli array:

```

struct Datario inizio = { 1, 1, 2012 };

```

Oppure, per essere precisi e non dipendere dall'ordine dei campi nella struttura:

```
struct Datario inizio = { .giorno=1, .mese=1, .anno=2012 };
```

Dal momento che le strutture sono tipi di dati nuovi, per poterne fare uso occorre che la dichiarazione relativa sia accessibile a tutte le parti del programma che hanno bisogno di accedervi. Probabilmente, il luogo più adatto è al di fuori delle funzioni, eventualmente anche in un file di intestazione realizzato appositamente.

Ciò dovrebbe bastare a comprendere che le variabili che contengono una struttura vengono passate regolarmente attraverso le funzioni, purché la dichiarazione del tipo corrispondente sia precedente ed esterno alla descrizione delle funzioni stesse.

```
...
struct Datario { int giorno; int mese; int anno; };
...
void elabora (struct Datario oggi)
{
    ...
}
```

L'esempio seguente che rappresenta un programma completo, serve a dimostrare che, nella chiamata di una funzione, la struttura viene passata per valore (e non per riferimento come avviene con gli array).

Listato 66.267. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/YZqBySyx> , <http://ideone.com/ffO5U> .

```
#include <stdio.h>

struct Datario {int giorno; int mese; int anno;};

void f (struct Datario d)
```

```
{
    unsigned int indirizzo = (int) &d;
    d.giorno = 28;
    d.mese = 2;
    d.anno = 2007;
    printf ("data %i-%i-%i inserita all'indirizzo %u\n",
           d.giorno, d.mese, d.anno, indirizzo);
}

int main (void)
{
    struct Datario data = {31, 12, 2007};
    unsigned int ind = (int) &data;
    f (data);
    printf ("data %i-%i-%i inserita all'indirizzo %u\n",
           data.giorno, data.mese, data.anno, ind);
    return 0;
}
```

Se si esegue il programma si ottiene un messaggio simile a quello seguente, dove si vede che gli indirizzi delle variabili contenenti la struttura, prima della chiamata della funzione e all'interno della stessa, sono differenti:

```
data 28-2-2007 inserita all'indirizzo 3212916960
data 31-12-2007 inserita all'indirizzo 3212916992
```

D'altro canto, se la variabile fosse la stessa, le modifiche fatte all'interno della funzione sarebbero visibili anche dopo la chiamata.

Così come nel caso dei tipi primitivi, anche con le strutture si possono creare dei puntatori. La loro dichiarazione avviene in modo intuitivo, come nell'esempio seguente:

```
struct Datario *p_data_fattura;
...
p_data_fattura = &inizio;
...
```

Quando si utilizza un puntatore a una struttura, diventa un po' più difficile fare riferimento ai vari componenti della struttura stessa, perché l'operatore punto ('.') che serve a unire il nome della struttura a quello dell'elemento, ha priorità rispetto all'asterisco che si utilizza per dereferenziare il puntatore:

```
*p_data_fattura.giorno = 15; // Non è valido!
```

L'esempio appena mostrato, non è ciò che sembra, perché l'asterisco posto davanti viene valutato dopo l'elemento '`p_data_fattura.giorno`', il quale non esiste. Per risolvere il problema si possono usare le parentesi, come nell'esempio seguente:

```
(*p_data_fattura).giorno = 15; // Corretto.
```

In alternativa si può usare l'operatore '`->`', fatto espressamente per i puntatori a una struttura:

```
p_data_fattura->giorno = 15; // Corretto.
```

L'esempio seguente è una variante di quello già presentato in precedenza per dimostrare il passaggio per valore delle variabili che contengono una struttura. Ma in questo caso, il passaggio dei dati avviene esplicitamente per riferimento.

Listato 66.273. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/vW8ktxAg>, <http://ideone.com/dYBHZ>.

```
#include <stdio.h>
```

```
struct Datario {int giorno; int mese; int anno;};

void f (struct Datario *d)
{
    unsigned int indirizzo = (int) d;
    d->giorno = 28;
    d->mese = 2;
    d->anno = 2007;
    printf ("data %i-%i-%i inserita all'indirizzo %u\n",
           d->giorno, d->mese, d->anno, indirizzo);
}

int main (void)
{
    struct Datario data = {31, 12, 2007};
    unsigned int ind = (int) &data;
    f (&data);
    printf ("data %i-%i-%i inserita all'indirizzo %u\n",
           data.giorno, data.mese, data.anno, ind);
    return 0;
}
```

In tal caso, gli indirizzi della struttura appaiono uguali e le modifiche applicate all'interno della funzione si riflettono nella variabile originale:

```
data 28-2-2007 inserita all'indirizzo 3214580384
data 28-2-2007 inserita all'indirizzo 3214580384
```

## 66.7.4 Scostamento all'interno delle strutture



Il file `stddef.h` della libreria standard definisce una macroistruzione che, attraverso la parvenza di una funzione, consente di misurare lo scostamento di un membro della struttura, rispetto all'inizio della stessa:

```
offsetof (tipo, membro)
```

Si osservi l'esempio seguente.

Listato 66.275. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/wW7KtJB2>, <http://ideone.com/QJrQv>.

```
#include <stdio.h>
#include <stddef.h>

struct Elenco {
    char  uno;
    short due;
    int   tre;
};

int main (int argc, char *argv[])
{
    size_t offset = offsetof (struct Elenco, due);
    printf ("Il membro \"due\" si trova %i byte dopo "
           "l'inizio della struttura.\n", offset);
    return 0;
}
```

Come si può vedere, la macroistruzione *offsetof* produce un risultato

di tipo `'size_t'`. Supponendo che il compilatore allinei i membri della struttura secondo multipli di due byte, il messaggio emesso dal programma potrebbe essere così:

Il membro "due" si trova 2 byte dopo l'inizio della struttura. Pertanto, in questo caso, dopo il membro `'uno'` c'è un byte inutilizzato prima del membro `'due'`.

È il caso di ribadire che `'offsetof'` è una macroistruzione, ottenuta tramite le funzionalità del precompilatore. Diversamente, è probabile che sia impossibile realizzare una funzione che si comporti nello stesso modo apparente.

### 66.7.5 Unioni

«

L'unione permette di definire un tipo di dati accessibile in modi diversi, gestendolo come se si trattasse contemporaneamente di tipi differenti. La dichiarazione è simile a quella della struttura; quello che bisogna tenere a mente è che si fa riferimento alla stessa area di memoria; pertanto, lo spazio occupato è pari a quello del membro più grande.

```
union Livello {
    char c;
    int i;
};
```

Si immagini, per esempio, di voler utilizzare indifferentemente una serie di lettere alfabetiche, oppure una serie di numeri, per definire un livello di qualcosa («A» equivalente a uno, «B» equivalente a due, ecc.). Le variabili generate a partire da questa unione, possono



essere gestite nei modi stabiliti, come se fossero una struttura, ma condividendo la stessa area di memoria.

```
union Livello carburante;
```

L'esempio mostra in che modo si possa dichiarare una variabile di tipo '**union Livello**', riferita all'omonima unione. Il bello delle unioni sta però nella possibilità di combinarle con le strutture.

```
struct Livello {
    char tipo;
    union {
        char c;           // Usato se tipo == 'c'.
        int  i;           // Usato se tipo == 'n'.
    };
};
```

L'esempio non ha un grande significato pratico, ma serve a chiarire le possibilità. La variabile *tipo* serve ad annotare il tipo di informazione contenuta nell'unione, se di tipo carattere o numerico. L'unione viene dichiarata in modo anonimo come appartenente alla struttura.

L'esempio successivo, che è completo, permette di verificare l'ordine con cui vengono memorizzati i byte in memoria. L'unione dichiarata parte dal presupposto che un numero '**short int**' utilizzi l'equivalente di due caratteri:

Listato 66.280. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/K6to4aa3>, <http://ideone.com/YJk2f>.

```
#include <stdio.h>

union Little_big {
    short int i;           // 16 bit
```

```
    char c[2];           // 8 bit, 8 bit
};

int main (void)
{
    union Little_big lb;
    lb.i = 0x1234;
    printf ("%x %x%x\n", lb.i, lb.c[0], lb.c[1]);
    return 0;
}
```

Eseguendo il programma in un elaboratore con architettura *little endian* si ottiene il risultato seguente:

```
1234 3412
```

## 66.7.6 Campi



All'interno di una struttura è possibile definire l'accesso a ogni singolo bit di un tipo di dati determinato, oppure a gruppetti di bit. In pratica viene dato un nome a ogni bit o gruppetto.

```
struct Luci {
    unsigned char
        b0      :1,
        b1      :1,
        b2      :1,
        b3      :1,
        b4      :1,
        b5      :1,
        b6      :1,
        b7      :1;
};
```

L'esempio mostra l'abbinamento di otto nomi ai bit di un tipo 'char'. Il primo, *b0*, rappresenta il bit più a destra, ovvero quello meno significativo. Se il tipo 'char' occupasse una dimensione maggiore di 8 bit, la parte eccedente verrebbe semplicemente sprecata.

```
struct Luci salotto;  
...  
salotto.b2 = 1;
```

L'esempio mostra la dichiarazione della variabile *salotto* come appartenente alla struttura mostrata sopra, quindi l'assegnamento del terzo bit a uno, probabilmente per «accendere» la lampada associata.

Volendo indicare un gruppo di bit maggiore, basta aumentare il numero indicato a fianco dei nomi dei campi, come nell'esempio seguente:

```
struct Prova {  
    unsigned char  
        b0      :1,  
        b1      :1,  
        b2      :1,  
        stato   :4;  
};
```

Nell'esempio appena mostrato, si usano i primi tre bit in maniera singola (per qualche scopo) e altri quattro per contenere un'informazione «più grande». Ciò che resta (probabilmente solo un bit) viene semplicemente ignorato.

## 66.7.7 Istruzione «typedef»

&lt;&lt;

L'istruzione **'typedef'** permette di definire un nuovo di tipo di dati, in modo che la sua dichiarazione sia più agevole. Lo scopo di tutto ciò sta nell'informare il compilatore; **'typedef'** non ha altri effetti. La sintassi del suo utilizzo è molto semplice:

```
typedef tipo nuovo_tipo ;
```

Si osservi l'esempio seguente:

```
typedef int numero_t;  
numero_t x, y, z;
```

In questo modo viene definito il nuovo tipo **'numero\_t'**, corrispondente in pratica a un tipo intero, con il quale si dichiarano tre variabili: **x**, **y** e **z**. Le tre variabili sono di tipo **'numero\_t'**. L'esempio seguente riguarda le enumerazioni:

```
typedef enum Colore { nero, marrone, rosso, arancio,  
                    giallo, verde, blu, viola, grigio,  
                    bianco } colore_t;  
colore_t c, d;
```

In questo caso si definisce il tipo **'colore\_t'**, corrispondente a un'enumerazione con i nomi dei colori principali. Le variabili **c** e **d** vengono dichiarate con questa modalità. Dal momento che si usa **'typedef'**, si potrebbe definire l'enumerazione in modo anonimo:


```
typedef enum { nero, marrone, rosso, arancio, giallo,  
             verde, blu, viola, grigio, bianco } colore_t;  
colore_t c, d;
```

L'esempio successivo riguarda le strutture:

```

struct Datario {
    int giorno;
    int mese;
    int anno;
};
typedef struct Datario data_t;
data_t inizio, fine;

```

Attraverso **'typedef'** è stato definito il tipo **'data\_t'**, facilitando così la dichiarazione delle variabili *inizio* e *fine*. Ma in questo caso, si presta di più una struttura anonima: 


```

typedef struct {
    int giorno;
    int mese;
    int anno;
} data_t;
data_t inizio, fine;

```

Tradizionalmente, i nomi dei tipi di dati creati con l'istruzione **'typedef'** hanno estensione **'\_t'**.

## 66.7.8 Costanti letterali composte

È possibile rappresentare un array o una struttura attraverso una costante letterale, nota come *costante letterale composta*. Formalmente si definisce la costante letterale composta secondo il modello seguente, dove le parentesi graffe fanno parte della definizione: 

```

(tipo) { valore [, valore] }

```

Per comprenderne l'utilizzo servono degli esempi e il caso più semplice riguarda la definizione degli array:

```
int *p = (int []) {3, 5, 76};
```

In questo modo si dichiara un array di interi, contenente rispettivamente i valori 3, 5 e 76, il cui indirizzo iniziale viene assegnato al puntatore *p*. La variante seguente fa sì che il contenuto dell'array non possa essere modificato, ma per questo deve rendere altrettanto invariabile il contenuto raggiunto attraverso il puntatore:

```
const int *p = (const int []) {3, 5, 76};
```

Un array in forma letterale può essere trasmesso a una funzione. Quello che segue è un programma completo per dimostrare tale possibilità.

Listato 66.292. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/EkefhL40>, <http://ideone.com/dx4fQ>.

```
#include <stdio.h>

void f (int i[])
{
    printf ("i: %i %i %i\n", i[0], i[1], i[2]);
}

int main (int argc, char *argv[])
{
    f ((int []) {1, 3, 7});
    return 0;
}
```

In pratica, la funzione *f()* viene chiamata passando come argomento un array di tre interi, il quale logicamente viene trasmesso solo attraverso il puntatore al primo dei suoi elementi.

In modo analogo si possono rappresentare le strutture, ma in tal caso occorre disporre di un modello di riferimento, come si può vedere nell'esempio seguente che costituisce un altro programma completo.

Listato 66.293. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/1aTWnv01> , <http://ideone.com/PLEi0> .

```
#include <stdio.h>

struct Elenco {
    char  uno;
    short due;
    int   tre;
};

int main (int argc, char *argv[])
{
    struct Elenco e;
    e = (struct Elenco) { 33, 55, 77 };
    printf ("struttura: %i %i %i\n", e.uno, e.due, e.tre);
    return 0;
}
```

Ma naturalmente, i valori della struttura possono essere abbinati esplicitamente ai componenti a cui appartengono:

```
...
    e = (struct Elenco) { .uno=33, .tre=77, .due=55 };
...
```

Come per il caso degli array, anche le strutture rappresentate in forma letterale possono essere usate tra gli argomenti di una funzione. L'esempio seguente fa la stessa cosa di quello appena mostrato, con

la differenza che si avvale di una funzione per ottenere lo scopo.

Listato 66.295. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/4DUDe1bG>, <http://ideone.com/33Gx33G>.

```
#include <stdio.h>

struct Elenco {
    char  uno;
    short due;
    int   tre;
};

void f (struct Elenco e)
{
    printf ("struttura: %i %i %i\n", e.uno, e.due, e.tre);
}

int main (int argc, char *argv[])
{
    f ((struct Elenco) { 33, 55, 77 });
    return 0;
}
```

Anche in questo caso, naturalmente, si possono rendere espliciti i componenti della struttura a cui si attribuiscono i valori:

```
...
    f ((struct Elenco) { .uno=33, .tre=77, .due=55 });
...
```

A differenza dell'array, la struttura che si trova tra gli argomenti di una funzione viene passata integralmente; volendo trasmettere solo il suo indirizzo, si può usare l'operatore '&', come nell'esempio



seguinte.

Listato 66.297. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/FJnL564M>, <http://ideone.com/mcodf>.

```
#include <stdio.h>

struct Elenco {
    char  uno;
    short due;
    int   tre;
};

void f (struct Elenco *e)
{
    printf ("struttura: %i %i %i\n", e->uno, e->due, e->tre);
}

int main (int argc, char *argv[])
{
    f (&(struct Elenco) {.uno=33, .tre=77, .due=55});
    return 0;
}
```

## 66.8 Tipi di dati speciali, di uso comune

Il linguaggio C prevede un insieme di tipi di dati tradizionali, a cui ci si riferisce con maggiore frequenza, e vari altri tipi, alcuni dei quali è bene conoscere. <<

## 66.8.1 Tipo «\_Bool»

«

Lo standard C prevede un tipo particolare per la rappresentazione di valori logici, ovvero solo per i valori zero e uno. Nella tradizione del linguaggio, questo tipo manca e di norma si è rimediato rimpiazzandolo semplicemente con un valore intero, dal tipo ‘**char**’ in su. Dal momento che è frequente l’uso di un tipo personalizzato (o di una macro-variabile del precompilatore) denominato *bool*, lo standard ha inserito il tipo logico con il nome ‘**\_Bool**’, allo scopo di evitare conflitti con il codice esistente.

Il tipo ‘**\_Bool**’ può contenere solo i valori zero e uno; pertanto, la conversione di un numero di tipo diverso in un tipo ‘**\_Bool**’ avviene traducendo qualunque valore diverso da zero con il numero uno (*Vero*), mentre lo zero mantiene il suo valore invariato (*Falso*).

Lo standard non stabilisce come deve essere rappresentato in memoria il tipo ‘**\_Bool**’, anche se si tratta molto probabilmente di un byte intero che viene a essere sacrificato per lo scopo. Data la particolarità di questo tipo, non è detto che si possa utilizzare un puntatore per raggiungere l’area di memoria corrispondente.

Comprendendo il motivo per il quale questo tipo ha ricevuto un nome così particolare, diventa evidente che se lo si vuole utilizzare convenga creare una macro-variabile o un tipo derivato. D’altra parte, lo stesso file ‘`stdbool.h`’ prescrive la definizione della macro-variabile ‘**bool**’.

In conclusione, se si desidera utilizzare un tipo di dati booleano, conviene fare riferimento alla macro-variabile *bool*, la quale potrebbe anche essere ridefinita localmente nel proprio programma, se quello che si vuole non è conforme alle previsioni dello standard o delle

librerie del proprio compilatore.

## 66.8.2 Tipo «void»

Il tipo **'void'** rappresenta un'eccezione tra i tipi di dati usati nel linguaggio, in quanto rappresenta formalmente una variabile di rango nullo, e come tale incapace di contenere qualunque valore. La situazione più frequente di utilizzo del tipo **'void'** riguarda le funzioni, quando non devono restituire alcun valore: in tal caso si dichiara che sono di tipo **'void'**.

```
void procedura (int x)
{
    ...
    return;
}
```

L'esempio mostra una funzione che, non dovendo restituire alcun valore, viene dichiarata di tipo **'void'**. Come si vede, l'istruzione **'return'** va usata, in questo caso, senza l'indicazione di un valore.

Quando una funzione non richiede parametri, si deve indicare esplicitamente questo fatto con la parola chiave **'void'**, come dire che esiste sì un parametro, ma di rango nullo e come tale privo di qualunque informazione:

```
int funzione (void)
{
    ...
}
```

In questo esempio, la funzione restituisce un valore intero, ma non fa uso di alcun parametro.

Il cast di tipo `'void'` può servire per annullare il risultato di un'espressione, quando ciò che interessa della stessa sono solo i suoi «effetti collaterali». In altri termini, quando un'espressione esegue qualche tipo di operazione, ma complessivamente si vuole scartare il risultato che viene generato, si può usare un cast di tipo `'void'`. Per esempio, quando si vuole usare una funzione, la quale restituirebbe un valore, del quale non si vuole fare alcun uso, si può indicare nella chiamata un cast al tipo `'void'`, anche se di norma ciò non è necessario:

```
...  
    (void) mia_funzione (...);  
...
```

È possibile definire un puntatore generico al tipo `'void'`, sapendo che questo è convertibile in tutti gli altri tipi di puntatore, con un cast appropriato e che è sempre possibile fare anche l'inverso:

```
...  
void *p;  
...  
p = (void *) &a;  
...
```

Il puntatore nullo può essere definito, sia come un valore intero pari a zero, sia come tale valore tradotto in un puntatore generico, ovvero `'void *'`:

```
...  
int NULL = 0;  
...
```

```
...  
void *NULL = (void *) 0;  
...
```

Si osservi che un puntatore generico (**'void \*'**) non può essere incrementato o decrementato, perché fa riferimento a un'unità di memoria di dimensione nulla. Pertanto, per usare un puntatore del genere, quando si vuole scandire la memoria, prima va convertito in un puntatore di rango appropriato.

### 66.8.3 Tipo «size\_t»

Secondo lo standard il tipo **'size\_t'** è definito nel file `'stddef.h'`, ma in pratica, dal momento che viene usato dall'operatore **'sizeof'**, potrebbe essere incorporato direttamente nel compilatore, tra i tipi fondamentali. A ogni modo si tratta normalmente di un tipo equivalente a un **'unsigned long int'**, destinato però a contenere la dimensione di qualcosa, intesa come intervallo tra due indirizzi (tra due puntatori), ma espressa come valore assoluto.



Listato 66.304. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/65fb65zpFR> , <http://ideone.com/9UNUA> .

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
    double a = 1.1;
    double b = 2.2;
    char * A = (char *) &a;
    char * B = (char *) &b;
    size_t s = abs (A - B);
    printf ("distanza: %i\n", s);
    return 0;
}
```

L'esempio mostra la dichiarazione di due variabili e di due puntatori alle variabili. Tuttavia, i puntatori sono di tipo '**char \***', in modo che la sottrazione tra i due dia la distanza in byte. Volendo, per non fare riferimento a un tipo particolare di puntatore, si potrebbe usare il tipo '**void**', ottenendo lo stesso risultato.

Listato 66.305. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/GFoNtYU5> , <http://ideone.com/bp9Qi> .

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
    double a = 1.1;
    double b = 2.2;
    void * A = (void *) &a;
    void * B = (void *) &b;
    size_t s = abs (A - B);
    printf ("distanza: %i\n", s);
    return 0;
}
```

Va osservato che il risultato mostrato dall'esecuzione dell'esempio compilato, dipende dal compilatore. In pratica, è il compilatore che decide come collocare in memoria le variabili; se si presume che siano adiacenti, si dovrebbe ottenere una distanza di otto byte.

#### 66.8.4 Tipo «ptrdiff\_t»

Per rappresentare la differenza tra due indirizzi, tenendo conto del segno, si usa il tipo '**ptrdiff\_t**', definito anch'esso nel file '`stddef.h`'. Molto probabilmente si tratta di un tipo equivalente a un '**long int**'. Viene ripreso l'esempio già mostrato, senza calcolare il valore assoluto della differenza tra indirizzi.



Listato 66.306. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/WaunziA8> , <http://ideone.com/pzJC8> .

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
int main (int argc, char *argv[])
{
    double a = 1.1;
    double b = 2.2;
    void * A = (void *) &a;
    void * B = (void *) &b;
    ptrdiff_t s = (A - B);
    printf ("differenza: %i\n", s);
    return 0;
}
```

### 66.8.5 Tipo «va\_list»

«

Il tipo **‘va\_list’** è definito dallo standard nel file di intestazione **‘stdarg.h’**, allo scopo di agevolare la scansione degli argomenti variabili, passati alle funzioni. Lo standard è vago sul significato che deve avere il tipo **‘va\_list’**, ma in pratica dovrebbe trattarsi di un puntatore al tipo **‘char’**.<sup>23</sup> Tuttavia il suo utilizzo rimane relegato alla scansione degli argomenti variabili, come descritto nella sezione [66.6.3](#). Viene comunque riportata qui la copia di un esempio che ne mostra l’uso.



Listato 66.307. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/5kUJnQxn>, <http://ideone.com/2W05Y>.

```
#include <stdio.h>
#include <stdarg.h>

void
f (int w, ...)
{
    long double x;        // Dichiarare le variabili che servono
    long long int y;      // a contenere gli argomenti per i
    int z;                // quali mancano i parametri formali.

    va_list ap;          // Dichiarare il puntatore agli
                        // argomenti.

    va_start (ap, w);    // Posiziona il puntatore dopo la
                        // fine di «w».

    x = va_arg (ap, long double); // Estrae l'argomento
                                    // successivo portando
                                    // avanti il puntatore
                                    // di conseguenza.

    printf ("w = %i; ", w);        // Mostra il valore del
                                    // primo parametro.
    printf ("x = %Lf; ", x);       // Mostra il valore
                                    // dell'argomento successivo.

    y = va_arg (ap, long long int); // Estrapola e mostra
    printf ("y = %lli; ", y);      // il terzo argomento.

    z = va_arg (ap, int);          // Estrapola e mostra
    printf ("z = %i\n", z);       // il quarto argomento.
```

```
    va_end (ap);                // Conclude la scansione.

    return;
}

int main (int argc, char *argv[])
{
    f (10, (long double)12.34, (long long int)13, 14);
    return 0;
}
```

### 66.8.6 Tipo «wchar\_t»

«

Per rappresentare un carattere esteso, ovvero un carattere dell'insieme universale, non è sufficiente il tipo `'char'` e per questo esiste invece il tipo `'wchar_t'` (*wide character type*), definito nel file `'stddef.h'`.

Il tipo `'wchar_t'` è un intero, usato generalmente senza segno, di rango sufficiente a rappresentare tutti i caratteri che si intende di poter ammettere. È da osservare che per rappresentare l'insieme completo dei caratteri già definiti sono necessari anche più di 32 bit.

Il tipo `'wchar_t'` si usa sostanzialmente come il tipo `'char'`, anche per ciò che riguarda gli array e le stringhe (che per essere tali devono essere terminate con il carattere nullo), ma si tratta sempre di una gestione interna, perché la rappresentazione richiede invece una trasformazione nella forma prevista dalla configurazione locale (sezione [66.9](#)).

### 66.8.7 Tipo «wint\_t»

Molte delle funzioni standard che in qualche modo hanno a che fare con un carattere singolo (perché ne ricevono il valore come argomento o perché restituiscono il valore di un carattere), lo fanno trattando il carattere come un tipo `'int'`, ovvero, trattando il carattere senza segno e promuovendolo al rango di un intero normale. Questo sistema permette di distinguere tra tutti i caratteri dell'insieme ridotto e un valore ulteriore, rappresentato dalla macro-variabile *EOF*, usata per rappresentare un errore in base al contesto.

Nella gestione dei caratteri estesi ci sono funzioni analoghe che svolgono lo stesso tipo di adattamento, ma in tal caso il valore del carattere viene gestito in qualità di `'wint_t'`, il quale può rappresentare tutti i caratteri che sono ammessi dal tipo `'wchar_t'`, con l'aggiunta del valore corrispondente a `'WEOF'` (diverso da tutti gli altri).

Il tipo `'wint_t'` e la macro-variabile *WEOF* sono definiti nel file `'wchar.h'`. Il tipo `'wint_t'` è, evidentemente, un intero di rango tale da consentire la rappresentazione di tutti i valori necessari.

### 66.8.8 Tipo «time\_t»

Diverse funzioni dichiarate nel file `'time.h'` fanno riferimento al tipo `'time_t'` che rappresenta la quantità di unità di tempo trascorsa a partire da un'epoca di riferimento.

Frequentemente si tratta di un valore numerico intero che rappresenta la quantità di secondi trascorsi dall'epoca di riferimento (nei sistemi Unix è di norma l'ora zero del 1 gennaio 1970); inoltre, in un elaboratore che gestisca correttamente i fusi orari, è normale che questo valore sia riferito al tempo universale coordinato.

## 66.8.9 Tipo «struct tm»

«

La libreria standard, nel file ‘time.h’, prescrive che sia definito il tipo ‘**struct tm**’, con il quale è possibile rappresentare tutte le informazioni relative a un certo tempo, secondo le convenzioni umane:

```
struct tm {
    int tm_sec;           // Secondi:           da 0 a 60.
    int tm_min;          // Minuti:           da 0 a 59.
    int tm_hour;         // Ora:              da 0 a 23.
    int tm_mday;         // Giorno del mese:  da 1 a 31.
    int tm_mon;          // Mese dell'anno:   da 0 a 11.
    int tm_year;         // Anno dal 1900.
    int tm_wday;         // Giorno della settimana: da 0 a 6
                        // con lo zero corrispondente alla
                        // domenica.
    int tm_yday;         // Giorno dell'anno: da 0 a 365.
    int tm_isdst;        // Ora estiva. Contiene un valore
                        // positivo se è in vigore l'ora estiva;
                        // zero se l'ora è quella «normale»
                        // ovvero quella invernale; un valore
                        // negativo se l'informazione non è
                        // disponibile.
};
```

## 66.8.10 Tipo «FILE»

«

Il tipo ‘**FILE**’ rappresenta una variabile strutturata con tutte le informazioni necessarie a individuare un flusso di file aperto. Di norma vengono usati puntatori, ovvero variabili di tipo ‘**FILE \***’, per tutte le operazioni di accesso relative a flussi di file aperti, tanto che nel gergo comune si confondono le cose e tali puntatori sono chiamati generalmente *stream*.

### 66.8.11 Tipo «fpos\_t»

Alcune funzioni individuano la posizione di accesso ai file attraverso un insieme di dati. In quei casi, per rappresentare tale insieme di dati si usano variabili strutturate di tipo ‘**fpos\_t**’.

## 66.9 Configurazione locale

La libreria standard del linguaggio C prevede la gestione della configurazione locale, attraverso l’indicazione di una stringa da associare a una *categoria*, dove la categoria rappresenta il contesto particolare della configurazione locale a cui si vuole fare riferimento.

La stringa con cui si indica il tipo di configurazione desiderato, contiene le informazioni sulla lingua, la nazionalità e soprattutto la codifica da usare per la rappresentazione delle *sequenze multibyte*. La codifica scelta condiziona l’insieme di caratteri che possono essere gestiti, sia attraverso le sequenze multibyte, sia attraverso i caratteri estesi.

### 66.9.1 Configurazione locale nei sistemi Unix e simili

In un sistema Unix o simile, la configurazione locale viene definita impostando alcune variabili di ambiente. Si tratta precisamente di variabili il cui nome inizia per ‘**LC\_...**’, dove in particolare la variabile ‘**LC\_ALL**’, se usata, prevale su tutte, mentre la variabile ‘**LANG**’ (se ‘**LC\_ALL**’ non viene usata) serve per la configurazione predefinita di tutte le altre variabili ‘**LC\_...**’ che non fossero state dichiarate espressamente. A queste variabili di ambiente si associa una stringa secondo il formato seguente:

*lingua\_nazionalità.codifica*

Per esempio, la configurazione ‘**de\_CH.UTF-8**’ rappresenta la configurazione di lingua tedesca per la Svizzera, con una codifica UTF-8.

Ogni variabile di ambiente ‘**LC\_...**’, esclusa ‘**LC\_ALL**’, rappresenta una categoria, ovvero un contesto particolare a cui applicare la configurazione locale. Per esempio, pur volendo gestire i numeri con una rappresentazione europea (con la virgola per i decimali), si potrebbe voler gestire le valute in dollari americani. Pertanto ci potrebbe essere un uso contrastante delle variabili ‘**LC\_NUMERIC**’ e ‘**LC\_MONETARY**’.

## 66.9.2 Configurazione locale nel linguaggio C

«

Il linguaggio C non gestisce la configurazione locale attraverso le variabili di ambiente, perché non è detto che il sistema in cui si trova a operare il programma le preveda. Tuttavia definisce le categorie della configurazione locale attraverso macro-variabili (dichiarate nel file ‘*locale.h*’) con gli stessi nomi e significati usati per le variabili di ambiente dei sistemi Unix e simili (vale anche il fatto che la macro-variabile **LC\_ALL** si riferisca simultaneamente a tutte le categorie previste). Le macro-variabili in questione riguardano solo le categorie **LC\_...**, mentre la variabile di ambiente **LANG** non ha alcun corrispondente nel linguaggio e non rappresenta precisamente una categoria, ma solo un valore predefinito.

La configurazione locale di partenza per un programma scritto in linguaggio C è proprio la configurazione ‘**C**’, la quale coincide so-

stanzialmente con la modalità di funzionamento tradizionale del linguaggio, con una codifica ASCII o equivalente. Per impostare la configurazione locale si usa la funzione *setlocale()* secondo il modello seguente:

```
char *setlocale (int categoria, const char *configurazione);
```

Il primo parametro è un numero intero che si indica normalmente attraverso una macro-variabile *LC\_...*; il secondo è una stringa, contenente la definizione della configurazione, per esempio `'it_IT.UTF-8'`. Se la funzione è nelle condizioni di accettare la configurazione richiesta, restituisce un puntatore alla stringa che definisce la configurazione stessa; altrimenti dà solo il puntatore nullo.

Come accennato, all'avvio ogni programma si trova a funzionare come se fosse stata usata la configurazione `'C'`, ovvero come se fosse stata usata la funzione *setlocale()* così:

```
setlocale (LC_ALL, "C");
```

Per richiedere una configurazione più attuale e più utile, conviene specificare qualcosa che preveda la codifica UTF-8, con la quale è possibile rappresentare qualunque carattere della codifica universale:

```
setlocale (LC_ALL, "fr_CH.UTF-8");
```

Tuttavia, se il sistema operativo ha una gestione della configurazione locale, così come avviene nei sistemi Unix e simili, è meglio far sì che il programma erediti tale configurazione. Per ottenere questo, si usa la funzione *setlocale()* lasciando una stringa nulla (nel senso di

vuota) al posto della configurazione richiesta:

```
setlocale (LC_ALL, "");
```

Per interrogare la configurazione locale attiva per una certa categoria (o per tutte se si fa riferimento a ‘**LC\_ALL**’), è sufficiente fornire il puntatore nullo al posto della stringa. L’esempio seguente è completo e si vede anche l’incorporazione del file ‘`locale.h`’, contenente il prototipo della funzione *setlocale()* e la dichiarazione delle macro-variabili **LC\_...**

Listato 66.312. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/iomnt08Q>, <http://ideone.com/1Vk3V>.

```
#include <stdio.h>
#include <locale.h>
int main (int argc, char *argv[])
{
    setlocale (LC_ALL, "");
    char *loc;
    loc = setlocale (LC_ALL, NULL);
    printf ("LC_ALL: \"%s\"\n", loc);
    return 0;
}
```

Il programma potrebbe emettere il risultato seguente:

```
LC_ALL: "it_IT.UTF-8"
```

### 66.9.3 Caratteri multibyte e caratteri estesi

«

All’origine del linguaggio C esisteva una corrispondenza biunivoca tra carattere e byte. Attualmente, questa corrispondenza riguarda solo i caratteri dell’insieme minimo, il quale di norma coincide con quello della codifica ASCII. Per rappresentare caratteri che vanno al



di fuori dell'insieme minimo, si usano due metodi nel linguaggio: le sequenze multibyte, in cui un carattere è rappresentato attraverso una sequenza di più byte o comunque attraverso l'inserzione di codici che cambiano di volta in volta il sottoinsieme di riferimento, e i caratteri estesi che richiedono una unità di memorizzazione con un rango maggiore del byte. L'esempio seguente mostra l'uso di una stringa multibyte:

```
printf ("€àèìòασδφ\n");
```

È il contesto che fa capire la natura della stringa. In pratica, il file sorgente che contiene i caratteri deve essere scritto utilizzando una qualche codifica che preveda l'uso di più byte per rappresentare un carattere. La stessa codifica è quella che il programma deve usare durante il funzionamento per interpretare correttamente la stringa multibyte fornita.

In questo caso particolare, la funzione *printf()* non ha nemmeno bisogno di rendersi conto della codifica; semplicemente, se il programma funziona secondo la configurazione corretta, la visualizzazione del messaggio avviene come previsto.

Esistono diversi modi di gestire delle sequenze multibyte per rappresentare caratteri particolari, ma alcune sono più difficili da amministrare, perché richiedono il passaggio a sottoinsiemi di caratteri differenti attraverso l'uso di codici speciali, a cui si fa riferimento con il termine *shift*. In pratica, in tali condizioni, quando deve essere interpretata una stringa contenente sequenze multibyte, le funzioni devono tenere traccia dello stato di questa interpretazione, per sapere a quale sottoinsieme particolare di caratteri si sta facendo riferimento. Pertanto, l'interruzione e la ripresa di tale interpretazione devono

essere motivo di preoccupazione per il programmatore. Fortunatamente la tendenza è quella di usare la codifica UTF-8 per la rappresentazione dell'insieme universale dei caratteri, per tutte le lingue e tutte le nazionalità. Tale codifica ha il vantaggio di non richiedere la conservazione di uno stato (*shift status*), in quanto l'interpretazione di ogni carattere è indipendente dai precedenti: quello che è importante è evitare di spezzare l'interpretazione di un carattere a metà, ma anche se fosse, i caratteri successivi verrebbero individuati correttamente.

Dall'esempio mostrato si intende che una stringa multibyte si rappresenta letteralmente nello stesso modo di una stringa normale, con la differenza che la sua lunghezza in «caratteri», nel senso di unità **'char'**, è maggiore dei caratteri che rappresenta. quindi, eventualmente, nel dimensionare un array di caratteri, occorre tenere conto di questo particolare.

Per rappresentare un carattere che va al di fuori dell'insieme minimo del linguaggio C, si può usare un carattere esteso, ovvero un valore intero di rango maggiore rispetto al tipo **'char'**. Si tratta precisamente del tipo **'wchar\_t'** (*wide char*) che in condizioni normali va dai 16 ai 32 bit;

Evidentemente, il rango del tipo **'wchar\_t'** condiziona la quantità di caratteri che possono essere rappresentati. Per una rappresentazione abbastanza completa dell'insieme universale serve almeno un tipo **'wchar\_t'** da 32 bit.

Si può rappresentare una costante letterale di tipo **'wchar\_t'** mettendo anteriormente il prefisso **'L'**. Per esempio, **'L'€'** viene con-

vertito dal compilatore in un carattere esteso che rappresenta numericamente il simbolo dell'euro. In modo analogo è possibile costruire array di elementi `wchar_t`, per contenere stringhe estese (stringhe di caratteri `wchar_t` concluse da un valore nullo di terminazione, come per le stringhe normali). Anche per rappresentare le stringhe estese in modo letterale si può usare il prefisso `L`. Per esempio, `L"àèìòù"` viene tradotto dal compilatore in una stringa estesa.

Listato 66.315. Per provare il codice attraverso un servizio *pastebin*: <http://ideone.com/eWHAz>.

```
#include <stdio.h>
#include <locale.h>
#include <wchar.h>
int main (int argc, char *argv[])
{
    setlocale (LC_ALL, "en_US.UTF-8");
    wchar_t wc = L'ß';
    wchar_t wcs[] = L"€àèìòασδφ";
    printf ("%lc, %ls\n", wc, wcs);
    return 0;
}
```

L'esempio mostra l'uso delle costanti letterali riferite a caratteri e stringhe estese. In particolare, va osservato l'uso della funzione *printf()*, in cui si indicano lo specificatore di conversione `%lc` per tradurre un carattere esteso e `%ls` per una stringa estesa. Ecco il risultato che si attende di visualizzare da quel programma:

ß, €àèìòασδφ

A questo punto è bene sia chiaro un concetto logico ma non sempre evidente: per gestire caratteri al di fuori dell'insieme minimo, è necessario definire la configurazione locale con una codifica che sia

tale da permetterlo. Pertanto, se non si usa la funzione *setlocale()* (così come invece avviene nell'esempio), si sta lavorando con la configurazione predefinita 'C', per la quale non ci sono sequenze multibyte e diventa inutile l'uso del tipo 'wchar\_t'. Pertanto, se nell'esempio mancasse l'uso appropriato della funzione *setlocale()*, non si otterrebbe la visualizzazione del testo come previsto.

#### 66.9.4 Concatenamento eterogeneo

«

Il concatenamento di stringhe espresse in forma di costanti letterali, avviene, per le stringhe estese, esattamente come per le stringhe tradizionali, con l'eccezione che il concatenamento eterogeneo è ammissibile e implica sempre l'interpretazione di stringhe estese:

```
...
    wcp = "ciao amore" L"€àèìòασδφ";
...
```

In questo caso, la variabile *wcp* riceve il puntatore a una stringa estesa contenente precisamente la sequenza «ciao amore€àèìòασδφ», conclusa in modo appropriato.

Questo meccanismo consente, tra le altre cose, di concatenare delle macro-variabili che si espandono in stringhe letterali normali, in ogni circostanza, senza doverle duplicare per distinguerle in base al contesto.

#### 66.9.5 Conversione tra caratteri multibyte e caratteri estesi

«

Un gruppo di funzioni dichiarate come prototipo nel file 'stdlib.h' è importante per gestire la conversione tra caratteri multibyte e caratteri estesi. Le funzioni più importanti sono precisamente *mbstowcs()*

(*Multibyte string to wide character string*) e *wcstombs()* (*Wide character string to multibyte string*), con lo scopo di convertire stringhe da multibyte a caratteri estesi e viceversa.

```
size_t mbstowcs (wchar_t *restrict wcs,
                 const char *restrict s,
                 size_t n);
```

```
size_t wcstombs (char *restrict s,
                 const wchar_t *restrict wcs,
                 size_t n);
```

La funzione **'mbstowcs'** si usa per convertire una stringa contenente sequenze multibyte in una stringa estesa, ovvero un array di elementi **'wchar\_t'**. L'ultimo parametro rappresenta la quantità massima di caratteri estesi che devono essere inseriti nella stringa estesa di destinazione, contando anche il carattere nullo di terminazione. Il valore restituito è la quantità di caratteri che sono stati inseriti, escludendo il carattere nullo di terminazione, se c'è.

Listato 66.318. Per provare il codice attraverso un servizio *pastebin*: <http://ideone.com/Nx3eA>.

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    setlocale (LC_ALL, "en_US.UTF-8");
    wchar_t wca[] = L"€€€€€€€€€€€€";
    wchar_t wcb[] = L"€€€€€€€€€€€€";
```

```
size_t q;

q = mbstowcs (wca, "ääâ", 3);
printf ("mbstowcs: %i: \"%ls\"\n", q, wca);

q = mbstowcs (wcb, "ääâ", 6);
printf ("mbstowcs: %i: \"%ls\"\n", q, wcb);

return 0;
}
```

L'esempio mostra la dichiarazione di due stringhe estese contenenti 10 caratteri estesi (oltre al carattere di terminazione della stringa). La funzione *mbstowcs()* viene usata la prima volta per tradurre la stringa multibyte 'L"ääâ"' nei caratteri estesi corrispondenti all'inizio della prima delle due stringhe estese. Però, viene posto il limite al trasferimento di soli tre caratteri. Così facendo, il carattere di terminazione della stringa multibyte non viene convertito. Nel secondo caso, invece, si richiede il trasferimento di sei caratteri estesi, ma questo si ferma quando viene incontrato il carattere nullo di terminazione.

Entrambe le chiamate alla funzione *mbstowcs()* restituiscono il valore tre, perché sono solo tre i caratteri trasferiti, che siano diversi da quello di terminazione, ma nel secondo caso si può apprezzare la differenza nella stringa estesa risultante:

```
mbstowcs: 3: "ääâ€€€€€€€€"
mbstowcs: 3: "ääâ"
```

La funzione *wcstombs()* funziona in modo opposto, per convertire una stringa estesa in una stringa multibyte. In questo caso, l'ultimo

parametro rappresenta la quantità di byte che si vogliono ottenere con il trasferimento, incluso quello che rappresenta la terminazione della stringa. Logicamente, come nel caso dell'altra funzione, si ottiene la quantità di byte ottenuti dal trasferimento, ma senza contare il carattere nullo di terminazione.

Listato 66.320. Per provare il codice attraverso un servizio *pastebin*: <http://ideone.com/inlVK>.

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    setlocale (LC_ALL, "en_US.UTF-8");
    char mba[] = "*****";
    char mbb[] = "*****";
    size_t n;

    n = wcstombs (mba, L"ääâ", 6);
    printf ("wcstombs: %i: \"%s\"\n", n, mba);

    n = wcstombs (mbb, L"ääâ", 9);
    printf ("wcstombs: %i: \"%s\"\n", n, mbb);

    return 0;
}
```

Questo nuovo esempio è analogo al precedente, ma invertendo il ruolo delle stringhe: questa volta la stringa estesa viene convertita in una stringa multibyte. Nel caso particolare della codifica UTF-8, ognuna delle lettere che si vedono nella stringa estesa si traduce in una sequenza di due byte; pertanto, la conversione richiede che

siano convertiti almeno sette byte, per includere anche il carattere nullo di terminazione. Si può vedere che nel primo caso il carattere nullo non viene convertito, pertanto la stringa di destinazione continua ad apparire della lunghezza originale, pur con la prima parte sovrascritta. Naturalmente, rimangono solo quattro asterischi perché la sequenza multibyte necessaria a rappresentare quelle tre lettere è complessivamente di sei byte.

```
wcstombs: 6: "ääâ****"  
wcstombs: 6: "ääâ"
```

La conversione, in un verso o nell'altro, può fallire. Se queste funzioni incontrano dei problemi, restituiscono l'equivalente di  $-1$  tradotto secondo il tipo `'size_t'` (in pratica, utilizzando una rappresentazione dei valori negativi in complemento a due, si ottiene il valore positivo massimo che la variabile possa rappresentare, essendo `'size_t'` senza segno).

Ci sono altri dettagli sull'uso di queste funzioni, ma si possono approfondire leggendo la sezione [69.9.11](#) e le pagine di manuale *mbstowcs(3)* *wcstombs(3)*.

## 66.10 Organizzazione dei file sorgenti

«

Quando si scrive un programma che non sia estremamente banale, diventa importante organizzare i file dei sorgenti in un modo gestibile. Se l'esperienza di programmazione da cui si proviene, quando ci si rivolge al C, è quella dei linguaggi interpretati, si può essere tentati di scrivere tutto il proprio programma in un file solo, ma questo approccio può essere controproducente. D'altra parte, per dividere il lavoro in più file, occorre che tale suddivisione abbia un senso pratico, conforme alla filosofia del linguaggio.



## 66.10.1 File di intestazione

La direttiva `#include` del precompilatore consente di incorporare un altro file, scritto secondo le regole del linguaggio, come se il suo contenuto facesse parte del file incorporante. Tradizionalmente questi file che vengono incorporati sono «file di intestazione», a cui si dà un'estensione diversa, `.h`, proprio per distinguerne lo scopo.

Un file di intestazione, perché sia utile e non serva a creare maggiore confusione, può contenere la dichiarazione di macro-variabili, di macroistruzioni, di tipi derivati, di prototipi di funzione e di variabili pubbliche. Non ha senso inserire il codice completo delle funzioni all'interno di un file di intestazione, perché queste verrebbero replicate inutilmente nei file-oggetto, ogni volta che viene incorporato il file stesso.

Se si rispetta questo principio, un file di intestazione può essere incorporato in diversi file, garantendo un uso uniforme di quanto dichiarato al suo interno, senza duplicazioni inutili nel risultato della compilazione, anche se ciò che contiene tale file viene usato solo parzialmente o non viene usato affatto.

Un file di intestazione deve contenere ciò che serve alla soluzione di un certo tipo di problematica, ben delimitata. In particolare, dovrebbe contenere tutti i prototipi delle funzioni che servono, o possono servire, per quel tale problema.

## 66.10.2 Funzioni pubbliche

Le funzioni che devono poter essere usate in varie parti del programma è bene siano pubbliche (come avviene in modo predefinito) e che siano descritte come prototipo in un file di intestazione appropriata-

to. Per quanto possibile, le funzioni potrebbero essere scritte in file indipendenti, ovvero: un file distinto per ogni funzione.

Dal momento che le funzioni potrebbero avere bisogno di usare macro-variabili o macroistruzioni definite nel file di intestazione che ne dichiara i prototipi, nei file di queste funzioni dovrebbe apparire l'inclusione del file di intestazione rispettivo.

### 66.10.3 Funzioni e variabili private

«

Le funzioni dichiarate con la parola chiave **'static'** sono visibili solo all'interno del file-oggetto in cui vanno a finire. Queste funzioni statiche sono utili in quanto vengono chiamate da una sola o da poche funzioni; in tal caso, questo gruppo di funzioni è costretto a convivere nello stesso file.

Lo stesso problema riguarda le variabili che devono essere utilizzate da più funzioni, ma che non devono essere visibili alle altre, perché anche in questo caso si rende necessario il mettere tale insieme nello stesso file.

### 66.10.4 Esempio di «stdlib.h»

«

Per comprendere il senso di quanto appena descritto in modo così sintetico, è utile osservare l'organizzazione della libreria C standard, anche se poi nella realtà i contenuti dei file che la compongono non sono sempre facili da interpretare. A ogni modo, qui viene proposto il caso di quella parte della libreria C che fa capo al file di intestazione `'stdlib.h'`.

Per cominciare, già dal nome del file scelto come esempio, va osservato che un file di intestazione realizzato in modo conforme alla filosofia del linguaggio rappresenta una «libreria» di qualcosa, anche

se, per le funzioni, contiene solo i prototipi. Ecco, in breve, come potrebbe essere fatto il file 'stdlib.h', omettendo alcune porzioni ridondanti per i fini della spiegazione:

```
#ifndef _STDLIB_H
#define _STDLIB_H      1
#define NULL 0
typedef unsigned long int size_t;
typedef unsigned int wchar_t;
#include <limits.h>
typedef struct {int quot; int rem;} div_t;
...
#define RAND_MAX      INT_MAX
...
int  atoi      (const char *nptr);
...
int  rand      (void);
void srand     (unsigned int seed);
void *malloc   (size_t size);
void *realloc  (void *ptr, size_t size);
void free      (void *ptr);
#define calloc(nmemb, size) (malloc ((nmemb) * (size)))
...
#endif
```

Si può osservare che l'interpretazione del contenuto del file è subordinata al fatto che la macro-variabile *\_STDLIB\_H* non sia già stata dichiarata, mentre altrimenti viene dichiarata. In pratica, con questo meccanismo, se per qualunque ragione un file si trova a incorporare più volte il file di intestazione, il compilatore considera quel contenuto solo la prima volta.

Nell'esempio si vedono dichiarazioni di macro-variabili, di macroi-

struzioni (*calloc()* è, in questo caso, una macroistruzione), di tipi di dati derivati. Secondo il buon senso, tutte queste cose devono servire alle funzioni di cui sono presenti i prototipi, ma soprattutto per ciò che riguarda i prototipi. Per esempio, la macro-variabile *NULL* viene dichiarata nel file di intestazione perché è il valore che potrebbe essere restituito da funzioni come *malloc()* e deve essere uniformato; il tipo derivato `'size_t'` viene dichiarato perché viene usato dalla funzione *malloc()* e da altre; il file `'limits.h'` viene incorporato perché definisce il valore della macro-variabile *INT\_MAX* che in questo caso viene usato per definire *RAND\_MAX*, la quale deve essere uniformata per l'uso con la funzione *rand()*.

La funzione *atoi()* è utile per dimostrare in che modo mettere ogni funzione nel proprio file indipendente. Per esempio, quello che segue potrebbe essere il file `'atoi.c'`:

```
#include <stdlib.h>:
#include <ctype.h>:
int
atoi (const char *nptr)
{
    int i;
    int sign = +1;
    int n;

    for (i = 0; isspace (nptr[i]); i++)
        ; // Si limita a saltare gli spazi iniziali.
}

if (nptr[i] == '+')
{
    sign = +1;
```

```
        i++;
    }
    else if (nptr[i] == '-')
    {
        sign = -1;
        i++;
    }

    for (n = 0; isdigit (nptr[i]); i++)
    {
        n = (n * 10) + (nptr[i] - '0'); // Accumula il valore.
    }

    return sign * n;
}
```

Come si vede, questa versione di *atoi()* si avvale delle funzioni *isspace()* e *isdigit()*, dichiarate nel file `'ctype.h'` che viene aggiunto di conseguenza all'elenco delle inclusioni. Questa inclusione non è stata fatta nel file di intestazione `'stdlib.h'`, perché l'uso delle funzioni *isspace()* e *isdigit()* è dovuto soltanto a una scelta realizzativa di *atoi()* e non perché la libreria `'stdlib.h'` dipenda necessariamente da `'ctype.h'`.

Per realizzare le funzioni *rand()* e *srand()* deve essere condivisa una variabile, la quale può essere nascosta prudentemente al resto del programma. Pertanto serve un file unico che incorpori entrambe le funzioni:

```

#include <stdlib.h>
static unsigned int _srand = 1; // Il rango di «_srand»
                                // deve essere maggiore o
                                // uguale a quello di
                                // «RAND_MAX» e di
                                // «unsigned int».

int
rand (void)
{
    _srand = _srand * 1234567 + 12345;
    return _srand % ((unsigned int) RAND_MAX + 1);
}

void
srand (unsigned int seed)
{
    _srand = seed;
}

```

### 66.10.5 Parametri delle macroistruzioni

«

Quando si dichiara una macroistruzione, si usano delle macro-variabili interne che rappresentano i parametri per la «chiamata» di questa specie di funzione. Dal momento che il codice che costituisce la macroistruzione può avvalersi di altre macro-variabili già dichiarate e dato che di norma queste hanno nomi che utilizzano lettere maiuscole, è bene che quelle interne siano scritte con sole lettere minuscole. In pratica, conviene fare come nella macroistruzione già apparsa nella sezione precedente:

```
#define calloc(nmemb, size) (malloc ((nmemb) * (size)))
```

Al contrario, facendo come nell'esempio successivo, il rischio che sia già stata dichiarata la macro-variabile *SIZE* oppure *NMEMB* è

più alto:



```
#define calloc(NMEMB, SIZE) (malloc ((NMEMB) * (SIZE)))
```

## 66.10.6 Compilazione



I vari file con estensione ‘.c’ possono essere compilati separatamente, per ottenere altrettanti file-oggetto da collegare successivamente (i file ‘.h’ devono essere incorporati da file ‘.c’, pertanto non vanno compilati da soli). Per esempio, per un certo gruppo di file collocato in una certa directory, si potrebbe usare un file-make simile a quello seguente:

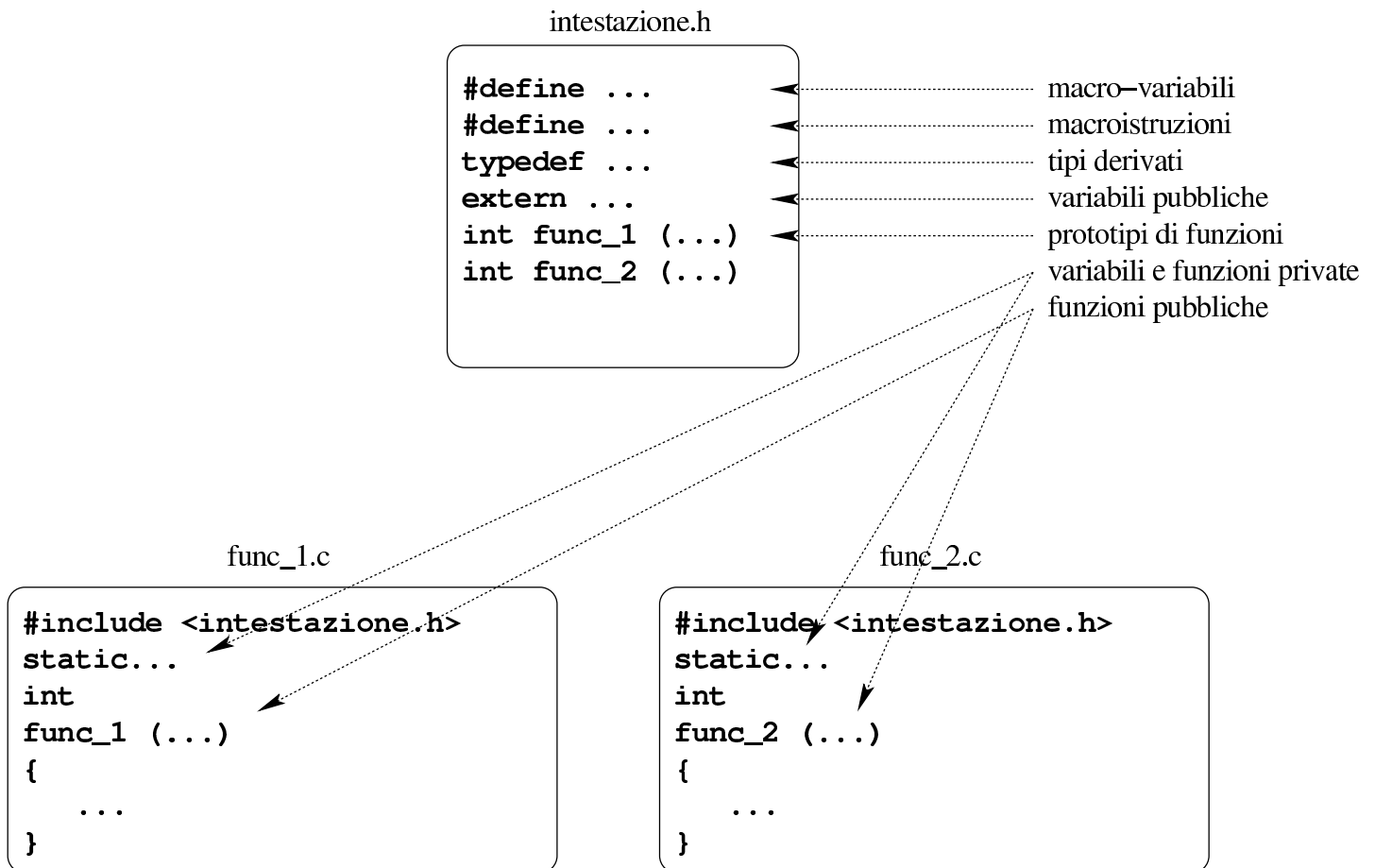
```
sorgenti = uno due tre
#
all: $(sorgenti)
#
clean:
    @rm *.o 2> /dev/null
#
$(sorgenti):
    @echo $@.c
    @gcc -Wall -Werror -o $@.o -c $@.c -I../include
```

In pratica, si presume che nella directory in cui si trova il file-make, ci siano i file ‘uno.c’, ‘due.c’ e ‘tre.c’, per i quali si vogliono ottenere altrettanti file-oggetto, con l’estensione appropriata. Si presume anche che i file di intestazione a cui i sorgenti fanno riferimento si trovino nella directory ‘../include/’.

Compilando in questo modo i file che contengono il minimo indispensabile (possibilmente una sola funzione per ciascuno), quando si verificano errori è più semplice concentrare l’attenzione per correggerli.

Quando si dispone dei file-oggetto si può passare al collegamento (*link*), ma anche in questa fase possono emergere dei problemi di tipo diverso: di solito si tratta di una funzione che viene chiamata, della quale esiste solo il prototipo e quindi non si trova in alcun file-oggetto. Naturalmente, il collegamento deve avvenire una volta sola, con tutti i file-oggetto che compongono il programma.

Figura 66.328. Indicazioni generali per la stesura di un insieme di file sorgenti ordinato.



## 66.11 K&R

«

Il linguaggio C, nella sua versione originale, nota come «K&R» (Kernigham e Ritchie), aveva delle caratteristiche che, fortunatamente, sono state perse. Generalmente non è necessario conoscere



le particolarità del vecchio linguaggio C, ma può capitare di leggere del vecchio codice, oppure può succedere di dover usare un vecchio compilatore.

Qui si annotano le caratteristiche principali della sintassi K&R, rispetto al linguaggio C, nella sua forma attuale.

### 66.11.1 Prototipi e chiamate delle funzioni

La differenza più importante tra la sintassi K&R e il linguaggio C attuale, sta nel modo di dichiarare i prototipi delle funzioni. Il prototipo di una funzione K&R non contiene la definizione dei tipi dei parametri (e tantomeno permette di attribuire loro dei nomi). Per esempio, il prototipo

```
int funzione (char a, short b, int c, long d, float e, double f);
```

si riduce, secondo la sintassi K&R, semplicemente nella dichiarazione seguente:

```
int funzione ();
```

Nella sintassi K&R, la mancanza di un prototipo vero e proprio, fa sì che nella chiamata di una funzione occorra essere molto precisi con i tipi degli argomenti; in altri termini, tutto quello che non ha il rango di **int**, va controllato attentamente. Per esempio, supponendo che il rango di **long int** sia effettivamente maggiore di quello di **int**, la chiamata seguente provoca certamente dei problemi:

```
x = funzione ('a', 123, 456, 789, 12.3, 45.6);
```

Si comprende che l'argomento attuale 789 sia effettivamente di tipo **int**, mentre la funzione si attende un rango maggiore, con risultati non prevedibili. Per quanto riguarda invece i tipi **char** e **short**

`int`, va osservato che la sintassi K&R prevede la promozione automatica a `int`, inoltre, per il tipo `float` è prevista la promozione a `double`.

Come si può intuire, **anche la quantità prevista degli argomenti di una chiamata non è determinabile per il compilatore**, con le conseguenze che si possono immaginare.

### 66.11.1.1 La macroistruzione «`_PROTOTYPE`» di Minix

«

Il codice del sistema operativo Minix è nato in un momento in cui si potevano incontrare sia compilatori C che riconoscevano e richiedevano l'uso di prototipi di funzione con l'indicazione dei parametri, sia di compilatori che potevano accettare solo la sintassi K&R. Per ovviare a questo problema, il codice del sistema Minix adotta l'uso di una macroistruzione, denominata *`_PROTOTYPE`*, dichiarata così:

```
#if _ANSI
...
#define _PROTOTYPE(function, params)    function params
...
#else
...
#define _PROTOTYPE(function, params)    function()
...
#endif
```

Successivamente, quando viene il momento di dichiarare un prototipo, questo viene scritto come nell'esempio seguente:

```
_PROTOTYPE( int printf, (const char *_format, ...) );
_PROTOTYPE( int scanf, (const char *_format, ...) );
```

## 66.11.2 Dichiarazione delle funzioni



Il modello sintattico che descrive la dichiarazione delle funzioni secondo il C di K&R, potrebbe essere espresso come nello schema seguente:

```
tipo nome_funzione (par_1 [, par_2] ...)  
tipo par_1 ;  
[ tipo par_2 ; ]  
...  
{  
    ...  
}
```

L'esempio seguente mostra la dichiarazione di una certa funzione, secondo la sintassi attuale del linguaggio C:

```
int  
funzione (int i, int j)  
{  
    int k;  
    k = i + j;  
    return (k);  
}
```

Così sarebbe invece secondo la sintassi K&R:

```
int
funzione (i, j)
int i;
int j;
{
    int k;
    k = i + j;
    return (k);
}
```

Tra l'altro, ciò può far incorrere in un errore, che il compilatore non segnala:

```
int
funzione (i, j)
int i;
int j;
{
    int i;
    int k;
    k = i + j;
    return (k);
}
```

In questo caso, la variabile *i* viene dichiarata anche nel corpo della funzione, oscurando il contenuto del parametro *i*.

### 66.11.3 Operatori composti di assegnamento



Nel linguaggio C comune si possono utilizzare degli operatori di assegnamento composti, come nell'esempio seguente in cui si vuole incrementare di due unità la variabile *i*:

```
i += 2;
```

Nella sintassi K&R, scrivere ‘+=’ oppure ‘+ =’ non fa differenza, mentre nello standard attuale del linguaggio ciò non è più ammissibile.

Nelle primissime versioni della sintassi K&R, gli operatori composti erano invertiti, pertanto, avrebbe potuto essere scritto:

```
i =+ 2; /* da non fare mai! */
```

Si può osservare che nella sintassi K&R non è possibile usare il segno ‘+’ al di fuori della somma, perché non avrebbe alcuna utilità (dal momento che  $+x$  è uguale a  $x$ ); pertanto, il fatto che si possa anche scrivere ‘`i = + 2;`’, non dovrebbe creare difficoltà. Tuttavia, scrivendo l’istruzione seguente, c’è da domandarsi cosa si intenda veramente:

```
i =- 2; /* da non fare mai! */
```

La variabile *i* viene ridotta di due unità, oppure le viene assegnato semplicemente il valore  $-2$ ?

#### 66.11.4 Tipi numerici

Nella sintassi K&R, le costanti numeriche in ottale possono contenere anche le cifre 8 e 9, senza che il compilatore si allarmi di ciò. Inoltre, le costanti numeriche rappresentano sempre un numero di tipo intero normale (`int`), a meno che gli si aggiunga la lettera `L`, per indicare che si tratta di un tipo `long int`.

I tipi numerici disponibili sono minori rispetto allo standard attuale del linguaggio C, mancando il tipo `long long int` e il tipo `long double`. Inoltre, nella sintassi K&R non è previsto l’uso dello specificatore `unsigned`.

### 66.11.5 Tipo «void \*»

«

Per la sintassi K&R, il tipo `'void *'` è equivalente al tipo `'char *'`. Pertanto, l'incremento di un tale puntatore porta ai byte successivi, mentre così non può avvenire secondo la sintassi attuale.

### 66.11.6 Direttive del preprocessore

«

Nella sintassi K&R, le direttive del preprocessore devono avere il cancelletto (`'#'`) esattamente nella prima colonna; inoltre non sono ammissibili direttive nulle, in cui il cancelletto sta da solo.

Le direttive `'#elif'`, `'#error'` e `'#pragma'` non sono disponibili. Gli operandi `'defined'`, `'#'` e `'##'` non sono disponibili.

### 66.11.7 Altre osservazioni su K&R

«

- Sulla funzione `main()` non si specifica cosa debba o possa restituire.
- Non sono disponibili le sequenze triplici, o *trigraph*, e di conseguenza non viene riconosciuta la sequenza `'\?'` nelle costanti carattere o nelle stringhe.
- La porzione significativa dei nomi degli identificatori (nomi di funzioni, di variabili, ecc.) è di soli otto caratteri.
- Le sequenze `'\a'`, `'\?'` e `'\x'`, nelle costanti carattere e nelle stringhe, non sono riconosciute; al contrario, sono ammissibili le sequenze `'\8'` e `'\9'`, che invece non dovrebbero, trattandosi di riferimenti a valori in ottale.
- Le costanti stringa, potrebbero risultare modificabili, mentre la sintassi attuale del linguaggio non lo deve consentire.

- L'operatore '&', usato per ottenere il puntatore a una variabile, non può essere usato con i nomi degli array.
- L'inizializzazione di una variabile, mentre viene dichiarata, può essere fatta omettendo il segno '='. Per esempio, al posto di scrivere `int x = 3 + 4;`, si può scrivere `int x 3 + 4;`.
- Nella struttura di controllo `switch`, l'espressione che viene valutata per la scelta dell'azione da compiere deve essere di tipo `int`.
- Le enumerazioni non sono disponibili.

## 66.11.8 Unproto

Se per qualche ragione si deve usare un compilatore C che è rimasto a standard precedenti al 1987, viene in aiuto il programma Unproto,<sup>24</sup> di Wietse Venema, che va inserito tra il preprocessore C e il compilatore vero e proprio.

Unproto è in grado di trasformare il risultato prodotto dal preprocessore in un codice C compatibile con la sintassi K&C, sia per la questione legata ai prototipi e la dichiarazione delle funzioni, sia per altri problemi meno appariscenti.

Unproto è anche incluso nella distribuzione Dev86, ovvero gli strumenti di sviluppo per 8088/8086, dove il compilatore BCC di Bruce Evans se ne avvale automaticamente.

## 66.12 Riferimenti

&lt;&lt;

- Brian W. Kernigham, Dennis M. Ritchie, *The C programming language*, Prentice-Hall 1978, prima edizione ISBN 0-13-110163-3; seconda edizione 0-13-110362-8, 0-13-110370-9; edizione italiana, Pearson, ISBN 88-7192-200-X, <http://cm.bell-labs.com/cm/cs/cbook/> ).
- Eric Giguere, *The ANSI Standard: A Summary for the C Programmer*, 1987, <http://www.ericgiguere.com/articles/ansi-c-summary.html>
- Open Standards, *C - Approved standards*, <http://www.open-std.org/jtc1/sc22/wg14/www/standards>
- ISO/IEC 9899:TC2, <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf>
- Richard Stallman e altri, *GNU coding standards*, <http://www.gnu.org/prep/standards/>
- Autori vari, *GCC manual*, <http://gcc.gnu.org/onlinedocs/gcc/> , <http://gcc.gnu.org/onlinedocs/gcc.pdf>
- Douglas Walls, *How to use the restrict qualifier in C*, [http://developers.sun.com/solaris/articles/cc\\_restrict.html](http://developers.sun.com/solaris/articles/cc_restrict.html)
- *SUPER-UX C Programmer's Guide, DIFFERENCES BETWEEN SUPER-UX ANSI C AND K&R C*, [http://static.cray-cyber.org/Documentation/NEC\\_SX\\_R10\\_1/G1AF02E/CHAP1.HTML#1.3](http://static.cray-cyber.org/Documentation/NEC_SX_R10_1/G1AF02E/CHAP1.HTML#1.3)
- Wietse Zweitze Venema, *Wietse's tools and papers, Unproto*, <ftp://ftp.porcupine.org/pub/security/index.html> , <ftp://ftp.porcupine.org/pub/unix/unproto5.shar.Z>



- Robert de Bath, *Dev86: a cross development C compiler, assembler and linker environment for the production of 8086 executables*, <http://homepage.ntlworld.com/robert.debath/dev86/>

<sup>1</sup> È bene osservare che un'istruzione composta, ovvero un raggruppamento di istruzioni tra parentesi graffe, non è concluso dal punto e virgola finale.

<sup>2</sup> In particolare, i nomi che iniziano con due trattini bassi ('\_\_'), oppure con un trattino basso seguito da una lettera maiuscola ('\_X') sono riservati.

<sup>3</sup> Tuttavia, le estensioni POSIX prevedono la possibilità di avere tre parametri: `int main (int argc, char *argv[], char *envp[])`.

<sup>4</sup> Il linguaggio C, puro e semplice, non comprende alcuna funzione, benché esistano comunque molte funzioni più o meno standardizzate, come nel caso di *printf()*.

<sup>5</sup> Quando il linguaggio C viene usato secondo lo standard POSIX, ovvero ciò che definisce le caratteristiche di un sistema operativo Unix, il byte deve essere precisamente di 8 bit, senza altre possibilità.

<sup>6</sup> Sono esistiti anche elaboratori in grado di indirizzare il singolo bit in memoria, come il Burroughs B1900, ma rimane il fatto che il linguaggio C si interessi di raggiungere un byte intero alla volta.

<sup>7</sup> Il qualificatore `signed` si può usare solo con il tipo `char`, dal momento che il tipo `char` puro e semplice può essere con o senza segno, in base alla realizzazione particolare del linguaggio che dipende dall'architettura dell'elaboratore e dalle convenzioni del

sistema operativo.

<sup>8</sup> La distinzione tra valori con segno o senza segno, riguarda solo i numeri interi, perché quelli in virgola mobile sono sempre espressi con segno.

<sup>9</sup> Come si può osservare, la dimensione è restituita dall'operatore `'sizeof'`, il quale, nell'esempio, risulta essere preceduto dalla notazione `'(int)'`. Si tratta di un cast, perché il valore restituito dall'operatore è di tipo speciale, precisamente si tratta del tipo `'size_t'`. Il cast è solo precauzionale perché generalmente tutto funziona in modo regolare senza questa indicazione.

<sup>10</sup> Per la precisione, il linguaggio C stabilisce che il «byte» corrisponda all'unità di memorizzazione minima che, però, sia anche in grado di rappresentare tutti i caratteri di un insieme minimo. Pertanto, ciò che restituisce l'operatore `sizeof()` è, in realtà, una quantità di byte, solo che non è detto si tratti di byte da 8 bit.

<sup>11</sup> Gli operandi di `'? :`' sono tre.

<sup>12</sup> Lo standard prevede il tipo di dati `'_Bool'` che va inteso come un valore numerico compreso tra zero e uno. Ciò significa che il tipo `'_Bool'` si presta particolarmente a rappresentare valori logici (binari), ma ciò sempre secondo la logica per la quale lo zero corrisponde a *Falso*, mentre qualunque altro valore corrisponde a *Vero*.

<sup>13</sup> Per la precisione, i parametri di una funzione corrispondono alla dichiarazione di variabili di tipo automatico.

<sup>14</sup> Questa descrizione è molto semplificata rispetto al problema del campo di azione delle variabili in C; in particolare, quelle che qui vengono chiamate «variabili globali», non hanno necessariamente un campo di azione esteso a tutto il programma, ma in condizioni

normali sono limitate al file in cui sono dichiarate. La questione viene approfondita in modo più adatto a questo linguaggio nella sezione [66.3](#).

<sup>15</sup> In pratica, *EXIT\_SUCCESS* equivale a zero, mentre *EXIT\_FAILURE* equivale a uno.

<sup>16</sup> Lo standard non impone che si tratti di file veri e propri; tuttavia, in un sistema Unix o in qualunque altro sistema operativo analogo, questi sarebbero file da cercare secondo criteri stabiliti, come viene descritto.

<sup>17</sup> In fase di collegamento (*link*) può darsi che il programma che svolge questo compito richieda che i file-oggetto siano indicati secondo una certa sequenza logica, ma questo problema, se esiste, è al di fuori della competenza del linguaggio C.

<sup>18</sup> Si ricorda che, in questo contesto, per «file» si intende il risultato dell'elaborazione da parte del precompilatore, il quale a sua volta potrebbe avere fuso assieme diversi file.

<sup>19</sup> Una variabile potrebbe rappresentare un registro del microprocessore e in tal caso non si potrebbe costruire un puntatore alla stessa. Pertanto, l'argomento sui puntatori parte dal presupposto che le variabili a cui eventualmente si vuole fare riferimento tramite un puntatore siano allocate in memoria.

<sup>20</sup> Per dereferenziare un puntatore si usa generalmente l'asterisco davanti al nome, pertanto il valore a cui punta la variabile *p* è accessibile attraverso l'espressione *\*p*. Tuttavia esiste un altro modo che viene chiarito a proposito dell'aritmetica dei puntatori, per cui lo stesso valore si raggiunge con l'espressione '*p[0]*'.

<sup>21</sup> In contesti particolari è ammissibile che *argc* sia pari a zero, a

indicare che non viene fornita alcuna informazione; oppure, se gli argomenti vengono forniti ma il nome del programma è assente, `argv[0][0]` deve essere pari a `<NUL>`, ovvero al carattere nullo.

<sup>22</sup> L'indirizzo gestito da un puntatore a una funzione, riguarda potenzialmente uno «spazio di indirizzamento» differente rispetto a quello usato per le variabili. Per esempio, il puntatore `p1`, riferito a una certa funzione, potrebbe avere lo stesso contenuto numerico del puntatore `p2` riferito a una variabile, ma nella memoria reale, i due puntatori raggiungerebbero posizioni differenti. Ciò serve per comprendere che la gestione dei puntatori alle funzioni non può essere confusa con quella dei dati, perché riguarda domini di indirizzamento diversi.

<sup>23</sup> È improbabile che sia utilizzato un tipo `'void *'`, perché non sarebbe possibile scandire la memoria, salvo convertirlo ogni volta in un formato `'char *'`.

<sup>24</sup> **Unproto** software libero

# Gestione dei flussi di file in C



67.1	Concetti generali .....	822
67.1.1	Dal file al flusso di file .....	822
67.1.2	File di testo e file binari .....	826
67.1.3	Fine del file .....	828
67.1.4	Memoria tampone .....	829
67.1.5	Flussi standard .....	830
67.1.6	Orientamento .....	831
67.2	Utilizzo comune dei file .....	832
67.2.1	Apertura e chiusura .....	832
67.2.2	Lettura e scrittura .....	835
67.2.3	Indicatore interno al file .....	840
67.2.4	File di testo .....	843
67.2.5	I/O standard .....	845
67.2.6	Ridirezione .....	846
67.2.7	Controllo degli errori .....	847
67.3	Conversione di input e output .....	850
67.3.1	Composizione dell'output .....	851
67.3.2	Rappresentazione degli specificatori di composizione per l'emissione dei dati	853
67.3.3	Funzioni per la composizione dell'output .....	860
67.3.4	Concatenamento di stringhe .....	861
67.3.5	Interpretazione dell'input .....	862

67.3.6	Rappresentazione degli specificatori di conversione	
866		
67.3.7	Funzioni per l'interpretazione dell'input	869
67.4	Riferimenti	871
EOF	828	
errno	847	
fclose()	832	
fgets()	843	
FILE	822	
fopen()	832	
fputs()	843	
fread()	835	
fseek()	840	
ftell()	840	
fwrite()	835	
printf()	860	
puts()	843	
reopen()	846	
scanf()	869	
stderr	845	
stdio	845	
stdio.h	822	
stdout	845	
vprintf()	860	
vscanf()	869	
WEOF	828	
%+...	853	
%...c	853 866	
%...d	853 866	
%...e	853 866	
%...f	853 866	
%...g	866	
%...hd	853 866	
%...hhd	866	
%...hhi	866	
%...hho	866	
%...hhu	866	
%...hxx	866	
%...hi	853 866	
%...ho	853	
%...hu	853 866	
%...hx	853 866	
%...i	853 866	
%...lc	853 866	
%...ld	853 866	
%...Le	853 866	
%...Lf	853 866	
%...Lg	866	
%...li	866	
%...lld	853 866	
%...lli	866	
%...llo	853 866	
%...llu	853	
%...llx	853 866	
%...lo	853 866	
%...ls	853 866	
%...lu	853	
%...lx	853 866	
%...o	853 866	
%...s	853 866	
%...u	853 866	
%...x	853 866	
%0...	853	
%-...	853	

## 67.1 Concetti generali

«

Il linguaggio C ha un proprio modo per gestire i file che, per poter essere compreso, richiede l'introduzione di alcuni concetti, presentati in questo capitolo. Va osservato che lo standard del linguaggio C prevede i flussi di file, i quali però, in un sistema che si rifà al modello di Unix sono gestiti attraverso i descrittori di file. Per scrivere codice C che sia compatibile nel modo migliore con qualunque sistema operativo, occorre avvalersi soltanto dei flussi, a cui qui ci si riferisce.

### 67.1.1 Dal file al flusso di file

Dal punto di vista del programma scritto in linguaggio C, il file viene utilizzato in qualità di *flusso logico di dati* (*stream*), ovvero flusso di file. Per la precisione, un file viene aperto attribuendogli un puntatore che rappresenta il flusso di file relativo; quando poi il flusso viene chiuso, l'associazione con il file si conclude.

La gestione del flusso di file avviene in modo trasparente, con l'ausilio di funzioni standard, ma ciò implica la presenza di una sorta di tabellina contenente una serie di informazioni legate all'accesso al file. Questa tabellina è formata in modo differente, a seconda del contesto in cui ci si trova a compilare il programma, ma in generale dovrebbe contenere almeno alcune informazioni basilari: il riferimento a un array di caratteri usato in qualità di memoria tampone, assieme ai vari puntatori necessari alla sua gestione; il tipo di accesso al file; i riferimenti per accedere al file secondo le caratteristiche del sistema operativo.

Quella tabellina che raccoglie tutte le informazioni su un certo flusso di file è definita da una variabile strutturata, dalla quale deriva un tipo di dati dichiarato nel file di intestazione `'stdio.h'`. Il tipo di dati in questione è denominato **'FILE'**.

L'apertura di un file, attraverso le funzioni standard, coincide con l'ottenimento di un puntatore al tipo **'FILE'**; pertanto, questo puntatore rappresenta il flusso di file e tutti i riferimenti a tale flusso si fanno con quel puntatore.

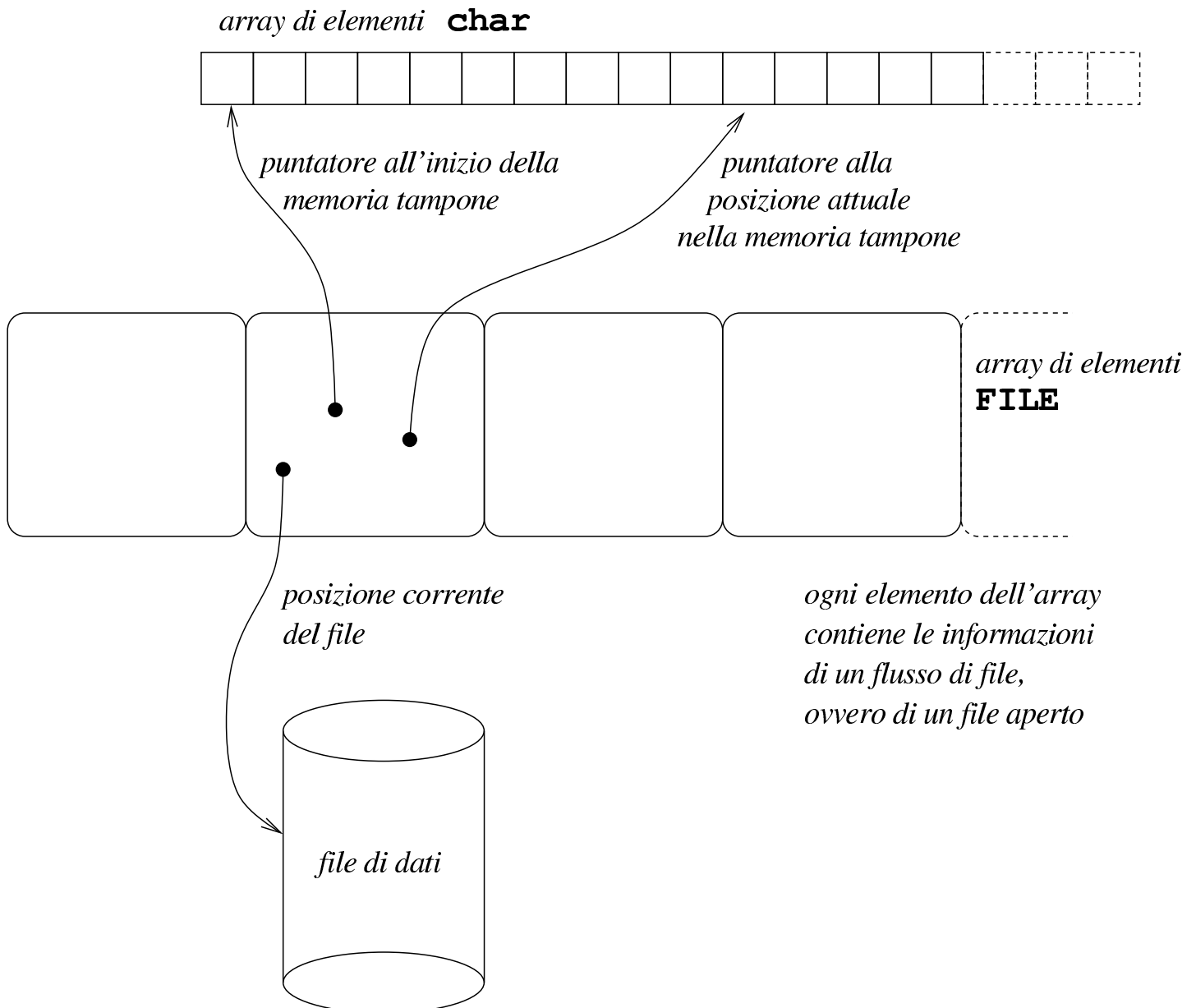
La modalità di accesso al file distingue tra lettura, scrittura e scrittura in aggiunta, utilizzando una simbologia particolare per esprimerla. Lo specchietto successivo sintetizza le operazioni consentite in base

alla modalità utilizzata:<sup>1</sup>

<b>r</b>	<b>w</b>	<b>a</b>	<b>r+</b>	<b>w+</b>	<b>a+</b>	Annotazioni
X			X			Per aprire un file in lettura, con queste modalità, è necessario che esista già.
	X			X		Quando si apre un file in scrittura, con queste modalità, se il file non esiste viene creato al volo, se invece esiste già, il suo contenuto precedente viene eliminato.
X			X	X	X	Con queste modalità è possibile leggere il contenuto del file.
	X	X	X	X	X	Con queste modalità è possibile modificare il contenuto del file.
		X			X	Con queste modalità è possibile scrivere nel file soltanto aggiungendo dati in coda.



Figura 67.2. Rappresentazione intuitiva dell'associazione tra una variabile strutturata di tipo **'FILE'** e il file a cui fa riferimento. Qui viene ipotizzato un array di elementi di tipo **'FILE'**, ma non è detto che l'organizzazione della libreria standard che si utilizza sia conforme a questa organizzazione.



## 67.1.2 File di testo e file binari

«

Il linguaggio C nasce per il sistema Unix, dove il file di testo ha una conformazione particolare che non è condivisa universalmente. Il file di testo in un sistema Unix o derivato è composto da una sequenza di caratteri (tradotti in byte),<sup>2</sup> dove la separazione tra le righe è segnalata dal codice *new-line*, corrispondente a  $\langle LF \rangle$ , ovvero la sequenza `'\n'`.

Nei sistemi Dos e MS-Windows si ha una rappresentazione simile, dove però il codice di interruzione di riga è rappresentato dalla sequenza  $\langle CR \rangle \langle LF \rangle$ . In altri sistemi si usano codice di interruzione di riga differenti e sono ammissibili forme molto diverse per rappresentare un file di testo.

Per questa ragione, il linguaggio C distingue l'accesso ai file attraverso due tipologie fondamentali: file di testo e file binari. In questo modo, quando si prevede un accesso in modalità testuale, la lettura e la scrittura del file avvengono attraverso una mediazione, tale da consentire al programmatore di trattare il file come se avesse la stessa rappresentazione di un sistema Unix. Naturalmente, in un sistema Unix e in qualunque altro sistema equivalente e conforme alla tradizione, non c'è distinzione tra l'accesso testuale ai file e quello binario.

Da quanto esposto vanno considerate due cose: quando si interviene su un file di testo, il codice corrispondente alla sequenza `'\n'` va inteso genericamente come codice di interruzione di riga; inoltre, il modo in cui si tiene traccia della posizione corrente all'interno di un file di testo non è predeterminabile, soprattutto perché non si può sapere quanti byte separano la fine di una riga dall'inizio della

successiva.

Il testo seguente è citato dalla documentazione standard *ISO/IEC 9899:TC2* e può servire per comprendere meglio il significato attribuito ai concetti di file di testo e di file binario:

*A text stream is an ordered sequence of characters composed into lines, each line consisting of zero or more characters plus a terminating new-line character. Whether the last line requires a terminating new-line character is implementation-defined. Characters may have to be added, altered, or deleted on input and output to conform to differing conventions for representing text in the host environment. Thus, there need not be a one- to-one correspondence between the characters in a stream and those in the external representation. Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if: the data consist only of printing characters and the control characters horizontal tab and new-line; no new-line character is immediately preceded by space characters; and the last character is a new-line character. Whether space characters that are written out immediately before a new-line character appear when read in is implementation-defined.*

*A binary stream is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream shall compare equal to the data that were earlier written out to that stream, under the same implementation. Such a stream may, however, have an implementation-defined number of null characters appended to the end of the stream.*

### 67.1.3 Fine del file

«

Nei documenti che trattano del linguaggio C si fa spesso riferimento alla macro-variabile *EOF* (dichiarata nel file `'stdio.h'`), in qualità di valore che si ottiene quando si tenta di leggere oltre la fine del file. La macro-variabile *EOF* corrisponde a un valore negativo che solitamente è  $-1$ , trattato come intero normale. Generalmente si può ottenere un valore di questo genere quando la lettura avviene carattere per carattere (inteso nel senso del tipo `'char'`, corrispondente al byte), perché in questi casi il carattere letto viene convertito in un valore senza segno, esteso alla dimensione di un intero normale. In questo modo, nessun carattere potrebbe confondersi con un valore negativo di un intero di tipo `'int'`.

Quando però la lettura di un file avviene attraverso funzioni che leggono un carattere esteso alla volta (l'equivalente di un carattere `'wchar_t'`), queste restituiscono un valore di tipo differente (`'wint_t'`) con cui si può rappresentare sia un carattere esteso, sia il valore rappresentato dalla macro-variabile *WEOF* che non individua alcun carattere esteso e rappresenta il raggiungimento della fine del file. A differenza di *EOF*, il valore di *WEOF* potrebbe essere positivo o negativo indifferentemente, perché conta solo che si tratti di un valore che non corrisponde ad alcun carattere esteso.

Di norma, il raggiungimento della fine di un file viene annotato all'interno della variabile strutturata che controlla il flusso (a cui ci si riferisce con un puntatore di tipo `'FILE *'`) e può essere interrogata con una funzione apposita. Naturalmente, l'uso di una funzione che porti alla modifica della posizione corrente, va ad azzerare tale indicazione.

## 67.1.4 Memoria tampone

I flussi di file possono disporre di una memoria tampone (*buffer*) che di norma è costituita da un array di caratteri ed è gestita da puntatori annotati all'interno delle variabili strutturate di tipo '**FILE**' associate ai flussi stessi.

Il programmatore ha la possibilità di controllare l'uso della memoria tampone, definendone la dimensione o arrivando a escluderla del tutto. In particolare, se si utilizza la memoria tampone, si può distinguere tra una gestione completa e una gestione a righe di testo.

L'uso della memoria tampone implica che le operazioni di scrittura possono avvenire con un certo ritardo. In generale, alla chiusura di un flusso di file si ottiene anche lo scarico della memoria tampone per ciò che riguarda le operazioni di scrittura ancora sospese; eventualmente è disponibile anche una funzione per richiedere espressamente l'esecuzione della scrittura in qualunque altro momento.

Va osservato che gli accessi ai file si prevedono in modo esclusivo; pertanto la gestione della memoria tampone è interna al programma. Per un accesso condiviso ai file la memoria tampone non può essere usata e comunque occorrono delle accortezze che le funzioni standard non possono offrire.

### 67.1.5 Flussi standard

«

Il linguaggio C prevede che ogni programma disponga, in modo predefinito, di tre flussi di file già costituiti: standard input, standard output e standard error. Il primo è predisposto per la lettura e di norma è collegato alla tastiera; il secondo e il terzo consentono solo la scrittura e sono collegati normalmente allo schermo.

Il fatto di disporre di tre flussi già in essere implica che ci siano tre puntatori di tipo `'FILE *'` già predisposti e associati correttamente alle strutture rispettive, per il controllo dei flussi di competenza. Va osservato che mentre i flussi standard non possono essere costituiti esplicitamente, potrebbero invece essere chiusi, oppure potrebbero essere riassegnati associandoli a file (o dispositivi) differenti.

L'associazione iniziale dei flussi standard a file o dispositivi dipende da ciò che succede in fase di avvio del programma (una shell potrebbe ridirigere i flussi a file diversi da quelli consueti). In condizioni normali, lo standard error è privo di memoria tampone, perché ciò che viene segnalato attraverso questo canale deve essere recepito il più presto possibile; per quanto riguarda invece gli altri due flussi, se questi non sono associati a dispositivi interattivi, di norma sono provvisti di memoria\_tampone.

Rimane da chiarire in che modo il file corrispondente al flusso sia aperto: l'associazione a una modalità di accesso binaria o testuale dovrebbe dipendere dal contesto e precisamente da ciò che determina il sistema operativo. È comunque possibile cambiare espressamente tale modalità, nel caso ciò fosse auspicabile.

## 67.1.6 Orientamento

I dati scritti e letti da un file vengono gestiti sempre attraverso sequenze di byte. Quando si devono rappresentare «caratteri estesi», tali da non poter essere espressi in un solo byte, si usano delle sequenze multibyte, secondo una codifica che normalmente dipende dalla configurazione locale.

La codifica multibyte utilizzata può essere priva di stato, in quanto ogni carattere esteso ha la propria sequenza indipendente, oppure può richiedere, di volta in volta, la selezione di un sottoinsieme di caratteri differente (attraverso quello che viene chiamato *shift state*). In ogni caso, sia la scrittura, sia la lettura, richiede di tenere traccia dello stato di completamento e, se necessario, della modalità di interpretazione in corso (*shift state*). Queste informazioni possono essere raccolte in un'area di memoria organizzata secondo il tipo `'mbstate_t'` (*Multibyte state*) che di solito è strutturata in più componenti.

Nella variabile strutturata di tipo `'FILE'` che rappresenta un flusso aperto, usata per gestire l'accesso al file relativo, deve essere presente un componente di tipo `'mbstate_t'` per poter seguire lo stato di interpretazione di una sequenza multibyte.

Onde evitare confusione, un flusso di file (aperto in modo binario o testuale, indifferentemente), deve essere *orientato*, nel senso che occorre stabilire se vada gestito a caratteri normali o estesi. In mancanza di una dichiarazione esplicita, l'orientamento viene definito in base all'uso del flusso attraverso funzioni specializzate per il trattamento di stringhe normali o di stringhe estese. Per esempio, si ottiene un orientamento orientato al byte (*byte-oriented*) se si utilizza la

funzione *fprintf()* (*file print formatted*), mentre si ottiene un orientamento esteso (*wide-oriented*) se si usa la funzione *fwprintf()* (*file wide print formatted*).

Una volta impostato l'orientamento, anche solo attraverso l'uso iniziale di una funzione invece di un'altra, questo può essere cambiato solo in modo esplicito, eventualmente riaprendo il flusso. Ma se questo cambiamento esplicito non viene eseguito, non è possibile utilizzare il flusso attraverso funzioni che non siano conformi all'orientamento esistente.

Si osservi che anche i tre flussi standard, all'inizio dell'esecuzione del programma, sono ancora privi di orientamento.

## 67.2 Utilizzo comune dei file

«

Nel linguaggio C, i file aperti sono flussi di file e l'apertura coincide con la predisposizione automatica di una variabile strutturata di tipo **FILE**, a cui, di conseguenza, si fa riferimento attraverso un puntatore (di tipo **FILE \***). Di solito, questo puntatore viene chiamato discorsivamente «puntatore al file», ovvero *file pointer*.

Quando si vuole accedere a un file, così come per poter usare le funzioni che consentono l'input e l'output elementare, è necessario includere il file `stdio.h`, dove, tra l'altro, è dichiarato il tipo **FILE**.

### 67.2.1 Apertura e chiusura

«

L'apertura dei file viene ottenuta normalmente con la funzione *fopen()* che restituisce il puntatore al file, oppure il puntatore nullo, **NULL**, in caso di fallimento dell'operazione. L'esempio seguen-



te mostra l'apertura del file 'mio\_file' contenuto nella directory corrente, con una modalità di accesso in sola lettura.

```
#include <stdio.h>
...
int main (void)
{
    FILE *fp_mio_file;
    ...
    fp_mio_file = fopen ("mio_file", "r");
    ...
}
```

Come si vede dall'esempio, è normale assegnare il puntatore ottenuto a una variabile adatta, che da quel momento identifica il file, finché questo resta aperto.

La chiusura del file avviene in modo analogo, attraverso la funzione *fclose()*, che restituisce zero se l'operazione è stata conclusa con successo, oppure il valore rappresentato da *EOF*. L'esempio seguente ne mostra l'utilizzo.

```
...
    fclose (fp_mio_file);
...
```

La chiusura del file conclude l'attività con questo, dopo avere scritto tutti i dati eventualmente ancora rimasti in sospeso (se il file è stato aperto in scrittura).

Normalmente, un file aperto viene definito come flusso di file, o *stream*; così, nello stesso modo viene identificata la variabile puntatore che vi si riferisce. In effetti, lo stesso file potrebbe anche essere aperto più volte con puntatori differenti, quindi è corretto distinguere tra file fisici su disco e file aperti, o flussi.

Seguono gli schemi sintattici di *fopen()* e *fclose()*, in forma di prototipo di funzione:

```
FILE *fopen (char *file, char *modalità);
```

```
int fclose (FILE *flusso_di_file);
```

La funzione *fopen()* richiede come secondo argomento una stringa contenente l'informazione della modalità di accesso. Questa può essere composta utilizzando i simboli seguenti, dove la lettera 'b' richiede espressamente un accesso binario, mentre la mancanza di tale lettera indica un accesso con le convenzioni dei file di testo:

Stringa	Descrizione
r rb	apre il file in sola lettura, posizionandosi all'inizio del file;
r+ rb+   r+b	apre il file in lettura e scrittura, posizionandosi all'inizio del file;

Stringa	Descrizione
w wb	apre il file in sola scrittura, creandolo se necessario, o troncadone a zero il suo contenuto se questo esiste già;
w+ wb+   w+b	apre il file in scrittura e lettura, creandolo se necessario, o troncadone a zero il suo contenuto se questo esiste già;
a ab	apre il file in scrittura in aggiunta ( <i>append</i> ), creandolo se necessario, o aggiungendovi dati a partire dalla fine e, di conseguenza, posizionandosi alla fine dello stesso;
a+ ab+   a+b	apre il file in scrittura in aggiunta e in lettura, creandolo se necessario, o aggiungendovi dati a partire dalla fine e, di conseguenza, posizionandosi alla fine dello stesso.

La funzione *fclose()* restituisce zero in caso di successo, oppure il valore corrispondente alla macro-variabile *EOF* (annotando anche un valore appropriato nella variabile *errno*).

## 67.2.2 Lettura e scrittura

L'accesso al contenuto dei file avviene generalmente a livello di byte e le operazioni di lettura e scrittura dipendono da un indicatore riferito a una posizione, espressa in byte, del contenuto del file stesso. Naturalmente, tale indicatore fa parte delle informazioni che si conservano nella variabile strutturata di tipo **'FILE'**, a cui si fa riferimento per identificare il flusso di file.

A seconda di come viene aperto il file, questo indicatore viene posizionato nel modo più logico, come descritto a proposito della fun-

zione *fopen()*. Questo indicatore viene spostato automaticamente a seconda delle operazioni di lettura e scrittura che si compiono, tuttavia, quando si passa da una modalità di accesso all'altra, è necessario spostare l'indicatore attraverso le istruzioni opportune, in modo da non creare ambiguità.

Per la lettura generica di un file in modo binario (nel senso di una lettura tale e quale del file) si può usare la funzione *fread()* che legge una quantità di byte trattandoli come un array. Per la precisione, si tratta di definire la dimensione di ogni elemento, espressa in byte, quindi la quantità di tali elementi. Il risultato della lettura viene inserito in un array, i cui elementi hanno la stessa dimensione. Si osservi l'esempio seguente:

```
...
    char ca[100];
    FILE *fp;
    int i;
    ...
    i = fread (ca, 1, 100, fp);
    ...
```

In questo modo si intende leggere 100 elementi della dimensione di un solo byte, collocandoli nell'array *ca*, organizzato nello stesso modo. Naturalmente, non è detto che la lettura abbia successo, o quantomeno non è detto che si riesca a leggere la quantità di elementi richiesta. Il valore restituito dalla funzione rappresenta la quantità di elementi letti effettivamente. Se si verifica un qualsiasi tipo di errore che impedisce la lettura, la funzione si limita a restituire zero.

Quando il file viene aperto in lettura, l'indicatore interno viene posizionato all'inizio del file; quindi, ogni operazione di lettura sposta

in avanti il puntatore, in modo che la lettura successiva avvenga a partire dalla posizione immediatamente seguente:

```
...  
    char ca[100];  
    FILE *fp;  
    int i;  
    ...  
    fp = fopen ("mio_file", "rb");  
    ...  
    while (1)          /* Ciclo senza fine */  
    {  
        i = fread (ca, 1, 100, fp);  
        if (i == 0)  
        {  
            break;      /* Termina il ciclo */  
        }  
        ...  
    }  
    ...
```

In questo modo, come mostra l'esempio, viene letto tutto il file a colpi di 100 byte alla volta, tranne l'ultima in cui si ottiene solo quello che resta da leggere.

Analogamente, la scrittura può essere eseguita con la funzione *fwrite()* che scrive una quantità di byte trattandoli come un array, nello stesso modo già visto con la funzione *fread()*. La scrittura procede a partire dalla posizione corrente riferita al file.

```
...
char ca[100];
FILE *fp;
int i;
...
i = fwrite (ca, 1, 100, fp);
...
```

L'esempio, come nel caso di *fread()*, mostra la scrittura di 100 elementi di un solo byte, prelevati da un array. Il valore restituito dalla funzione è la quantità di elementi che sono stati scritti con successo. Se si verifica un qualsiasi tipo di errore che impedisce la scrittura, la funzione si limita a restituire zero.

Anche in scrittura è importante l'indicatore della posizione interna del file. Di solito, quando si crea un file o lo si estende, l'indicatore si trova sempre alla fine. L'esempio seguente mostra lo scheletro di un programma che crea un file, copiando il contenuto di un altro (non viene utilizzato alcun tipo di controllo degli errori).

```
#include <stdio.h>
...
int main (void)
{
    char ca[1024];
    FILE *fp_in;
    FILE *fp_out;
    int i;
    ...
    fp_in = fopen ("file_in", "r");
    ...
    fp_out = fopen ("file_out", "w");
    ...
    while (1) // Ciclo senza fine.
```

```
{
    i = fread (ca, 1, 1024, fp_in);
    if (i == 0)
        {
            break;                // Termina il ciclo.
        }
    ...
    fwrite (ca, 1, i, fp_out);
    ...
}
...
fclose (fp_in);
fclose (fp_out);
...
return 0;
}
```

Seguono i modelli sintattici di *fread()* e *fwrite()*, espressi in forma di prototipi di funzione:

```
size_t fread (void *restrict ptr,
              size_t dimensione,
              size_t quantità,
              FILE *restrict stream);
```

```
size_t fwrite (const void *restrict ptr,
               size_t dimensione,
               size_t quantità,
               FILE *stream);
```

Il tipo di dati '**size\_t**' serve a garantire la compatibilità con qualun-

que tipo intero, mentre il tipo `'void'` per l'array permette l'utilizzo di qualunque tipo per i suoi elementi, anche se negli esempi è sempre stato visto il trattamento di sole sequenze di byte.

### 67.2.3 Indicatore interno al file

«

Lo spostamento diretto dell'indicatore interno della posizione di un file aperto è un'operazione necessaria quando il file è stato aperto simultaneamente in lettura e in scrittura, e da un tipo di operazione si vuole passare all'altro. Per questo si utilizza la funzione *fseek()* ed eventualmente anche *ftell()* per conoscere la posizione attuale. La posizione e gli spostamenti sono espressi in byte.

La funzione *fseek()* esegue lo spostamento a partire dall'inizio del file, oppure dalla posizione attuale, oppure dalla posizione finale. Per questo utilizza un parametro che può avere tre valori identificati rispettivamente da tre macro-variabili: *SEEK\_SET*, *SEEK\_CUR* e *SEEK\_END*. l'esempio seguente mostra lo spostamento del puntatore, riferito al flusso di file *fp*, in avanti di 10 byte, a partire dalla posizione attuale.

```
...  
i = fseek (fp, 10, SEEK_CUR);  
...
```

La funzione *fseek()* restituisce zero se lo spostamento avviene con successo, altrimenti si ottiene un valore negativo.

L'esempio seguente mostra lo scheletro di un programma, senza controlli sugli errori, che, dopo aver aperto un file in lettura e scrittura, lo legge a blocchi di dimensioni uguali, modifica questi blocchi e li riscrive nel file.



```
#include <stdio.h>

static const int dim = 100; // Dimensione del record logico.

int main (void)
{
    char ca[dim];
    FILE *fp;
    int qta;
    int posizione_1;
    int posizione_2;

    fp = fopen ("mio_file", "r+b"); // Lettura e scrittura.

    while (1) // Ciclo senza fine.
    {
        //
        // Salva la posizione del puntatore interno al file
        // prima di eseguire la lettura.
        //
        posizione_1 = ftell (fp);
        qta = fread (ca, 1, dim, fp);

        if (qta == 0)
        {
            break; // Termina il ciclo.
        }
        //
        // Salva la posizione del puntatore interno al file
        // dopo la lettura.
        //
        posizione_2 = ftell (fp);
        //
        // Sposta il puntatore alla posizione precedente
    }
}
```

```
// alla lettura.
//
fseek (fp, posizione_1, SEEK_SET);
//
// Esegue qualche modifica nei dati, per esempio
// mette un punto esclamativo all'inizio.
//
ca[0] = '!';
//
// Riscrive il record modificato.
//
fwrite (ca, 1, qta, fp);
//
// Riporta il puntatore interno al file alla
// posizione corretta per eseguire la lettura
// successiva.
//
fseek (fp, posizione_2, SEEK_SET);
}

fclose (fp);
return 0;
}
```

Segue il modello sintattico per l'uso della funzione *fseek()*, espresso attraverso il suo prototipo:

```
int fseek (FILE *stream, long int spostamento,
           int punto_di_partenza);
```

Il valore dello spostamento, fornito come secondo parametro, rappresenta una quantità di byte che può essere anche negativa, indicando in tal caso un arretramento dal punto di partenza. Il valore restitui-

to da *fseek()* è zero se l'operazione viene completata con successo, altrimenti viene restituito un valore diverso.

Segue il modello sintattico per l'uso della funzione *ftell()*, espresso attraverso il suo prototipo:

```
long int ftell (FILE *stream)
```

La funzione *ftell()* permette di conoscere la posizione dell'indicatore interno al file a cui fa riferimento il flusso di file fornito come parametro. Se si tratta di un file per il quale si esegue un accesso binario, la posizione ottenuta è assoluta, ovvero riferita all'inizio del file.

Il valore restituito in caso di successo è positivo, a indicare appunto la posizione dell'indicatore. Se si verifica un errore viene restituito un valore negativo:  $-1$ .

#### 67.2.4 File di testo

I file di testo possono essere gestiti in modo più semplice attraverso due funzioni: *fgets()* e *fputs()*. Queste permettono rispettivamente di leggere e scrivere un file una riga alla volta, intendendo come riga una porzione di testo che termina con il codice di interruzione di riga, secondo l'astrazione usata dal linguaggio.

La funzione *fgets()* permette di leggere una riga di testo di una data dimensione massima. Si osservi l'esempio seguente:

```
...  
fgets (ca, 100, fp);  
...
```

In questo caso, viene letta una riga di testo di una dimensione massima di 99 caratteri, dal file rappresentato dal puntatore *fp*. Questa riga viene posta all'interno dell'array *ca*, con l'aggiunta di un carattere '\0' finale. Questo fatto spiega il motivo per il quale il secondo parametro corrisponde a 100, mentre la dimensione massima della riga letta è di 99 caratteri. In pratica, l'array di destinazione è sempre una stringa, terminata correttamente.

Nello stesso modo funziona *fputs()*, che però richiede solo la stringa e il puntatore del file da scrivere. Dal momento che una stringa contiene già l'informazione della sua lunghezza perché possiede un carattere di conclusione, non è prevista l'indicazione della quantità di elementi da scrivere.

```
...  
fputs (ca, fp);  
...
```

Seguono i modelli sintattici delle funzioni *fputs()* e *fgets()*, in forma di prototipi di funzione:

```
char *fgets (char *stringa, int dimensione_max, FILE *stream);
```

```
int fputs (const char *stringa, FILE *stream)
```

Se l'operazione di lettura riesce, *fgets()* restituisce un puntatore corrispondente alla stessa stringa (cioè l'array di caratteri di destinazione), altrimenti restituisce il puntatore nullo, 'NULL', per esempio quando è già stata raggiunta la fine del file.

La funzione *fputs()* permette di scrivere una stringa in un file di

testo. La stringa viene scritta senza il codice di terminazione finale, ‘\0’, ma anche senza aggiungere il codice di interruzione di riga. Il valore restituito è un valore positivo in caso di successo, altrimenti **EOF**.

In alternativa a *fgets()* e a *fputs()* si possono considerare anche le funzioni *gets()* e *puts()*, le quali però utilizzano rispettivamente lo standard input e lo standard output. Ma la funzione *gets()* legge tutto quello che trova fino alla fine della riga o, in mancanza di questo, fino alla fine del file, mentre *puts()* **aggiungere automaticamente il codice di interruzione di riga** alla fine della stringa che viene scritta nel file.

```
char *gets (char *stringa) ;
```

```
int puts (const char *stringa)
```

### 67.2.5 I/O standard

Ci sono tre flussi di file che risultano aperti in modo predefinito, all'avvio del programma:

- standard input, corrispondente normalmente alla tastiera;
- standard output, corrispondente normalmente allo schermo del terminale;
- standard error, anch'esso corrispondente normalmente allo schermo del terminale.

Spesso si utilizzano questi flussi di file attraverso funzioni apposite (come nel caso di *gets()* e *puts()*) che vi fanno riferimento in modo

implicito, ma si potrebbe accedere anche attraverso funzioni generalizzate, utilizzando come puntatori i nomi: `'stdio'`, `'stdout'` e `'stderr'`.

## 67.2.6 Ridirezione

«

È possibile associare un flusso di file già in essere, a un file differente, attraverso la funzione *freopen()*, oppure è possibile modificarne la modalità di accesso. Evidentemente questo tipo di operazione richiede la chiusura del flusso di file, prima di associarvi un file differente o di cambiare la modalità, cosa che comunque tenta di eseguire automaticamente la stessa funzione *freopen()*:

```
FILE *freopen (const char *restrict nome_file_nuovo ,  
              const char *restrict modalità_di_accesso ,  
              FILE *restrict flusso di file) ;
```

La funzione, se riesce a eseguire il proprio compito, restituisce il puntatore allo stesso flusso di file indicato come terzo argomento, ovvero quello a cui viene applicata la ridirezione o la modifica dei permessi (o entrambe le cose). Per limitare l'effetto alla sola modifica della modalità di accesso, è sufficiente indicare il puntatore nullo al posto del nome del file. Viene mostrato un esempio che ridirige lo standard output:

```
#include <stdio.h>
int main (void)
{
    printf ("ciao 1\n");
    freopen ("mio", "w", stdout);
    printf ("ciao 2\n");
    freopen ("/dev/tty", "w", stdout);
    printf ("ciao 3\n");
    return 0;
}
```

In questo caso, dal momento che la funzione *printf()* scrive automaticamente attraverso lo standard output, quando il flusso di file **'stdout'** viene ridiretto nel file **'mio'**, il testo **'ciao 2'** viene scritto in tale file. Ipotizzando di operare in un sistema Unix o in un sistema equivalente, il file di dispositivo **'/dev/tty'** dovrebbe corrispondere allo schermo del terminale utilizzato in quel momento (anche se fosse un terminale grafico); pertanto, il messaggio **'ciao 3'** dovrebbe apparire nuovamente sullo schermo.

Logicamente, quando si riapre un file e si cambia la modalità, da binaria a testo o viceversa, può essere appropriato un riposizionamento, con l'aiuto di *fseek()*.

### 67.2.7 Controllo degli errori

Molte funzioni, quando si verifica un errore, annotano quanto accaduto, in forma di numero intero, in una variabile globale nota con il nome *errno*. In generale, il nome *errno* è un'espressione che si traduce nell'accesso, a un'area di memoria, condiviso dal programma, ed eventualmente distinto in base al thread, ovvero il flusso di controllo. Il significato del valore attribuito alla variabile *errno* è descritto da macro-variabili definite nel file **'errno.h'**, nel qua-



le viene anche dichiarata la variabile *errno*, o l'espressione che la rappresenta.

La lettura della variabile *errno* porta alla conoscenza dell'ultimo errore che si è presentato e non è previsto il suo azzeramento automatico.

La variabile strutturata che si utilizza per fare riferimento a un flusso di file prevede anche l'annotazione di uno stato di errore. In pratica, le funzioni che accedono ai file, oltre che aggiornare la variabile globale *errno*, gestiscono l'indicazione di questo stato, azzerandolo quando non è più significativo. Per verificare la presenza di uno stato di errore ancora valido, a proposito di un flusso di file, si usa la funzione *ferror()* che restituisce un valore diverso da zero se questo stato esiste effettivamente:

```
int ferror (FILE *flusso_di_file);
```

Per interpretare l'errore annotato nella variabile *errno* e visualizzare direttamente un messaggio attraverso lo standard error, si può usare la funzione *perror()*:

```
void perror (const char *s);
```

La funzione *perror()* mostra un messaggio in modo autonomo, aggiungendo davanti la stringa che può essere fornita come primo argomento (diversamente si può indicare il puntatore nullo o una stringa nulla, in quanto contenente solo il carattere di terminazione).

L'esempio seguente mostra un programma completo e molto semplice, in cui si crea un errore, tentando di scrivere un messaggio



attraverso lo standard input, cosa che produce un errore. Se effettivamente si rileva un errore associato a quel flusso di file, attraverso la funzione *ferror()*, allora si passa alla sua interpretazione con la funzione *perror()*.

Listato 67.15. Per provare il codice attraverso un servizio *pastebin*: <http://ideone.com/JvbUI>.

```
#include <stdio.h>
#include <errno.h>
int main (void)
{
    fprintf (stdin, "Ciao amore!\n");
    if (ferror (stdin))
        {
            perror ("Attenzione");
        }
    return 0;
}
```

Come si vede, è necessario includere anche il file ‘*errno.h*’, senza il quale la variabile *errno* non risulterebbe accessibile. Avviando questo programma in un sistema GNU/Linux si potrebbe ottenere il messaggio seguente:

```
Attenzione: Bad file descriptor
```

In alternativa alla funzione *perror()* si può usare anche *strerror()* (dal file ‘*string.h*’), con la quale si ottiene la stringa contenente il messaggio di errore:

```
char *strerror (int n_errore);
```

Si può modificare leggermente l’esempio già apparso, in modo da

usare la funzione *strerror()* per produrre lo stesso risultato.

Listato 67.17. Per provare il codice attraverso un servizio *pastebin*: <http://ideone.com/qDCYr>.

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
int main (void)
{
    char *cp;
    fprintf (stdin, "Ciao amore!\n");
    if (ferror (stdin))
    {
        cp = strerror (errno);
        fprintf (stderr, "Attenzione: %s\n", cp);
    }
    return 0;
}
```

## 67.3 Conversione di input e output

«

Il linguaggio C rappresenta in memoria i valori numerici in modo binario secondo una modalità diversa rispetto a quella usata per le stringhe che servono invece per l'interazione umana. In altri termini, un conto è il valore 100, un altro è la sequenza dei caratteri numerici con cui questo valore viene rappresentato sullo schermo o su carta.

Il linguaggio C non svolge automaticamente conversioni da valori numerici binari a stringhe di cifre numeriche e viceversa; per questo è necessario invece avvalersi di funzioni di conversione. Per la precisione esistono due gruppi di funzioni, *...printf()* e *...scanf()*, con cui è possibile comporre (nel senso tipografico) le informazioni in uscita, oppure interpretarle in senso inverso le informazioni in ingresso.

### 67.3.1 Composizione dell'output

Le funzioni del gruppo *...printf()* consentono di comporre una stringa (da memorizzare o da visualizzare), partendo da un'altra stringa contenente il formato di composizione e utilizzando un elenco variabile di argomenti:

```
...printf ( ... stringa_di_composizione [, argomento] ... )
```

Il modello sintattico dà solo una visione di massima: a seconda della funzione ci possono essere dei parametri che non vengono chiariti nello schema, quindi appare sempre la stringa di composizione, la quale può essere seguita da altri argomenti le cui caratteristiche non sono precisate nel prototipo della funzione.<sup>3</sup>

La stringa di composizione è una stringa normale, in cui si inseriscono delle sequenze precedute dal simbolo '%', note come *specificatori di conversione*. Conviene partire da un esempio, proprio con la funzione *printf()*, la quale emette la stringa generata dalla composizione attraverso lo standard output (attraverso il flusso di file associato allo standard output):

```
...  
printf ("Il capitale di %i al tasso %f%% dà l'interesse %i",  
        1000, 0.5, 5);  
...
```

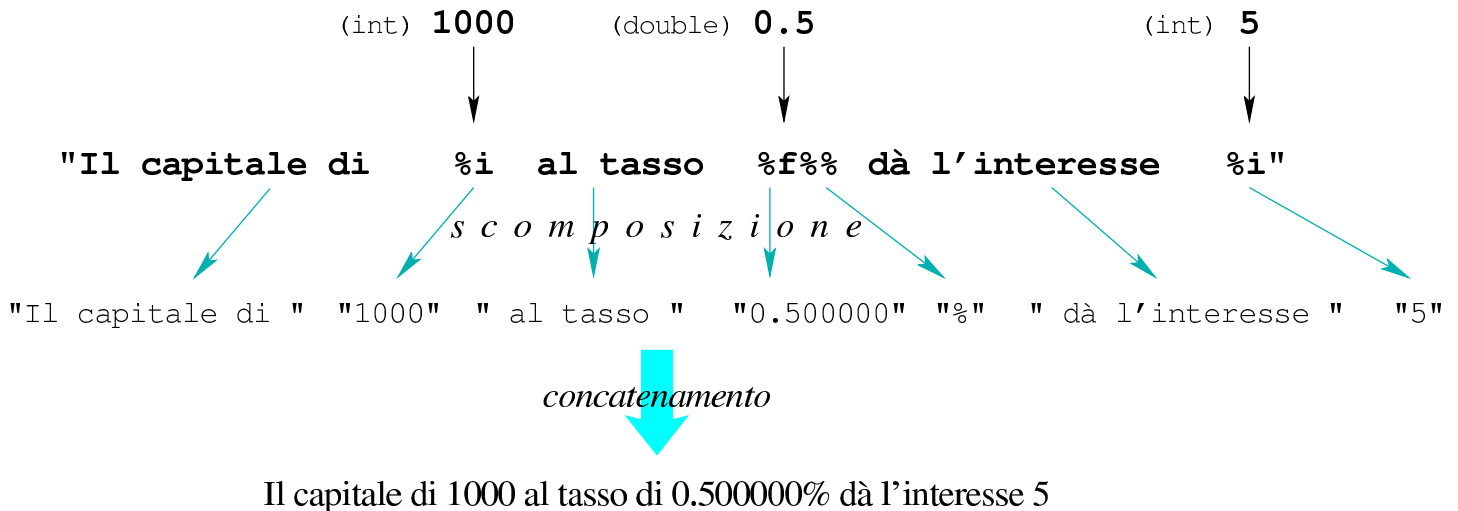
Da questa istruzione si ottiene la visualizzazione della frase seguente:

```
Il capitale di 1000 al tasso 0.500000% dà l'interesse 5
```

In pratica, al posto del primo specificatore '%i' è stato inserito il va-

lore 1000 dopo averlo convertito in modo da essere rappresentato da quattro caratteri ('1', '0', '0', '0'), al posto del secondo specificatore '%f' è stato inserito il valore 0.5 dopo un'opportuna conversione in caratteri, al posto del terzo specificatore '%%' è stato inserito un carattere di percentuale, infine, al posto del quarto specificatore '%i' è stato inserito il valore 5.

Figura 67.20. Schematizzazione della trasformazione di una stringa di composizione in una stringa finale.



Lo specificatore di conversione ha due compiti: indicare che tipo di informazione viene prelevato dagli argomenti (ammesso che si prelevi effettivamente un valore) e come questa deve essere rappresentata. Nel caso dell'esempio, il primo specificatore '%i' indica che il valore da prelevare dagli argomenti è di tipo '**int**'; il secondo specificatore '%f' indica un tipo '**double**'; il terzo non preleva alcun valore; il quarto indica ancora un altro '**int**'.

Una stringa di composizione che non contenga degli specificatori rimane evidentemente intatta e non richiede alcun dato aggiuntivo. La funzione *printf()* (che è stata usata nell'esempio) viene usata spesso come mezzo generico per emettere un messaggio attraverso lo stan-

dard output, anche quando non c'è alcun bisogno di comporre dei dati. Questo è lecito, ma non va dimenticato il contesto, pertanto, scrivere l'istruzione seguente sarebbe sbagliato:



```
...
printf ("Il capitale di 1000 al tasso 0.5% dà l'interesse 5");
...
```

Il modo giusto è quello seguente:



```
...
printf ("Il capitale di 1000 al tasso 0.5%% dà l'interesse 5");
...
```

### 67.3.2 Rappresentazione degli specificatori di composizione per l'emissione dei dati

Di norma, la scelta dello specificatore determina il tipo di dati dell'argomento e il tipo di trasformazione che deve ricevere. La tabella 67.23 elenca alcuni degli specificatori di conversione utilizzabili, nella loro forma più semplice. È bene ricordare che per rappresentare il simbolo di percentuale si usa uno specificatore fittizio composto dalla sequenza di due segni percentuali: '%%'.



Tabella 67.23. Alcuni specificatori di conversione.

Simbolo	Corrispondenza
%...c	Un carattere singolo.
%...s	Una stringa.
%...d	Un intero con segno da rappresentare in base dieci.

Simbolo	Corrispondenza
%...u	Un intero senza segno da rappresentare in base dieci.
%...o	Un intero senza segno da rappresentare in ottale.
%...x	Un intero senza segno da rappresentare in esadecimale.
%...e	Un numero a virgola mobile normale ( <b>'double'</b> ), da rappresentare in notazione esponenziale.
%...f	Un numero a virgola mobile normale ( <b>'double'</b> ), da rappresentare in notazione decimale fissa.

Leggendo la tabella si può osservare che la composizione dei dati in uscita può riguardare anche dati che sono già in forma di stringa (lo specificatore **'%...s'**), pertanto si usa questo metodo anche per il concatenamento delle stringhe.

Gli specificatori di conversione possono contenere indicazioni ulteriori tra il simbolo di percentuale e la lettera che definisce il tipo di trasformazione. Si tratta di inserire un simbolo composto da un carattere singolo, seguito eventualmente da altre informazioni aggiuntive, secondo la sintassi seguente:

```
% [simbolo] [n_ampiezza] [.n_precision] [hh|h|l|ll|j|z|t|L] tipo
```

Alcuni di questi simboli sono rappresentati dalla tabella 67.24. In presenza di valori numerici, si può indicare il numero di cifre decimali intere (ampiezza), aggiungendo eventualmente il numero di decimali (precisione), se si tratta di rappresentare un numero a virgola mobile. Quando è necessario modificare il tipo di dati provenienti dagli argomenti, ciò può essere precisato con una sigla, come

descritto nella tabella 67.25.

Tabella 67.24. Alcuni simboli per la conversione di valori numerici.

Simbolo	Corrispondenza
%+...	Il segno «+», usato all'inizio di uno specificatore di conversione, fa sì che i numeri positivi siano rappresentati con il segno in modo esplicito, mentre altrimenti il segno viene mostrato solo per quelli negativi.
%0 <i>ampiezza</i> ... %+0 <i>ampiezza</i> ...	Lo zero, usato all'inizio di uno specificatore di conversione, fa sì che si inseriscano degli zeri per allineare a destra un valore numerico, nell'ambito dell'ampiezza specificata. Lo zero può combinarsi con il segno «+».
% <i>ampiezza</i> ... %+ <i>ampiezza</i> ...	In mancanza di uno zero iniziale, in presenza dell'indicazione dell'ampiezza, il valore viene allineato a destra usando degli spazi.
%-... %-+...	Il segno meno richiede un allineamento a sinistra rispetto al campo, usando degli spazi a destra. Si può combinare con il segno «+», ma non con lo zero.

Tabella 67.25. Alcuni modificatori dell'estensione che ha in memoria il valore da estrarre e comporre.

Simbolo	Corrispondenza
%...ld %...lu %...lo %...lx	La lettera 'l', prima di quella che specifica il tipo di composizione, quando si tratta della conversione di un numero intero, specifica che il dato va trattato in qualità di <b>'long int'</b> , con o senza segno.

Simbolo	Corrispondenza
%...lc %...ls	La lettera 'l', prima di quella che specifica il tipo di composizione, quando si tratta della conversione di un carattere o di una stringa, specifica che il dato va trattato in qualità di carattere esteso ( <i>wide char</i> ) o di stringa estesa ( <i>wide string</i> ).
%...lld %...llu %...llo %...llx	La coppia di lettere 'll', prima di quella che specifica il tipo di composizione, quando si tratta della conversione di un numero intero, specifica che il dato va trattato in qualità di ' <b>long long int</b> ', con o senza segno.
%...Le %...Lf	La lettera 'L', prima di quella che specifica il tipo di composizione, quando si tratta della conversione di un valore in virgola mobile, specifica che il dato va trattato in qualità di ' <b>long double</b> '.

Nella stringa di composizione possono apparire anche sequenze di escape come già mostrato nella tabella 66.17. Si veda anche la pagina di manuale *printf(3)*.

Tabella 67.26. Esempi di utilizzo degli specificatori di conversione di *printf()*. Le costanti numeriche utilizzate negli esempi sono interpretate secondo le convenzioni del linguaggio, pertanto: 123 e -123 sono costanti di tipo '**int**'; mentre 123.456 e -123.456 sono costanti di tipo '**double**'.

Codice	Risultato emesso attraverso la funzione
<code>printf ("%i", 123);</code>	[123]
<code>printf ("%i", -123);</code>	[-123]



Codice	Risultato emesso attraverso la funzione
<pre>printf ("%2d", 123);</pre>	<p>[123]</p> <p>L'indicatore '<b>%2d</b>' specifica che si devono usare almeno due cifre, ma se le cifre della parte intera sono in numero maggiore, queste vanno indicate tutte ugualmente.</p>
<pre>printf ("%6d", 123);</pre>	[ 123]
<pre>printf ("%6d", -123);</pre>	[ -123]
<pre>printf ("%+6d", 123);</pre>	[ +123]
<pre>printf ("%06d", 123);</pre>	[000123]
<pre>printf ("%06d", -123);</pre>	[-00123]
<pre>printf ("%+06d", 123);</pre>	[+00123]
<pre>printf ("% -6d", 123);</pre>	[123 ]
<pre>printf ("%u", 123);</pre>	[123]
<pre>printf ("%u", -123);</pre>	<p>[4294967173]</p> <p>Evidentemente si ottiene la rappresentazione del valore binario, tale e quale, secondo la notazione usata per i valori negativi.</p>
<pre>printf ("%6x", 123);</pre>	[ 7b]
<pre>printf ("%06x", 123);</pre>	[00007b]
<pre>printf ("%x", 123);</pre>	[7b]

Codice	Risultato emesso attraverso la funzione
<code>printf ("%x", -123);</code>	[ffffff85] Evidentemente si ottiene la rappresentazione del valore binario, tale e quale, secondo la notazione usata per i valori negativi.
<code>printf ("%6x", 123);</code>	[ 7b]
<code>printf ("%06x", 123);</code>	[00007b]
<code>printf ("%o", 123);</code>	[173]
<code>printf ("%o", -123);</code>	[37777777605] Evidentemente si ottiene la rappresentazione del valore binario, tale e quale, secondo la notazione usata per i valori negativi.
<code>printf ("%6o", 123);</code>	[ 173]
<code>printf ("%06o", 123);</code>	[000173]
<code>printf ("%f", 123.456);</code>	[123.456000]
<code>printf ("%f", -123.456);</code>	[-123.456000]
<code>printf ("%12f", 123.456);</code>	[ 123.456000]
<code>printf ("% .4f", 123.456);</code>	[123.4560]
<code>printf ("%12.4f", 123.456);</code>	[ 123.4560]
<code>printf ("%12.4f", -123.456);</code>	[ -123.4560]
<code>printf ("% +12.4f", 123.456);</code>	[ +123.4560]
<code>printf ("%012.4f", 123.456);</code>	[0000123.4560]
<code>printf ("%012.4f", -123.456);</code>	[-000123.4560]
<code>printf ("% +012.4f", 123.456);</code>	[+000123.4560]

Codice	Risultato emesso attraverso la funzione
<code>printf ("% -12.4f", 123.456);</code>	[123.4560     ]
<code>printf ("%e", 123.456);</code>	[1.234560e+02]
<code>printf ("%e", -123.456);</code>	[-1.234560e+02]
<code>printf ("%15e", 123.456);</code>	[    1.234560e+02]
<code>printf ("% .4e", 123.456);</code>	[1.2346e+02]
<code>printf ("%15.4e", 123.456);</code>	[       1.2346e+02]
<code>printf ("%15.4e", -123.456);</code>	[       -1.2346e+02]
<code>printf ("% +15.4e", 123.456);</code>	[       +1.2346e+02]
<code>printf ("%015.4e", 123.456);</code>	[000001.2346e+02]
<code>printf ("%015.4e", -123.456);</code>	[-00001.2346e+02]
<code>printf ("% +015.4e", 123.456);</code>	[+00001.2346e+02]
<code>printf ("% -15.4e", 123.456);</code>	[1.2346e+02     ]
<code>printf ("%s", "ciao amore");</code>	[ciao amore]
<code>printf ("%7s", "ciao amore");</code>	[ciao amore] <b>La stringa è più lunga di sette caratteri, ma viene visualizzata completamente.</b>
<code>printf ("% .7s", "ciao amore");</code>	[ciao am] <b>La stringa viene troncata se è più lunga del valore della precisione.</b>
<code>printf ("% .14s", "ciao amore");</code>	[ciao amore]
<code>printf ("%14s", "ciao amore");</code>	[       ciao amore]
<code>printf ("%14.7s", "ciao amore");</code>	[       ciao am]
<code>printf ("% -14s", "ciao amore");</code>	[ciao amore     ]

Codice	Risultato emesso attraverso la funzione
<code>printf ("%[-14.7s]", "ciao amore");</code>	[ciao am            ]

### 67.3.3 Funzioni per la composizione dell'output

&lt;&lt;

Tutte le funzioni standard il cui nome finisce per **printf** interpretano una stringa di composizione secondo le modalità descritte nel capitolo, ovvero in modo analogo a *printf()* che, in particolare, emette il risultato della composizione attraverso lo standard output. In particolare, la funzione *fprintf()* scrive il risultato attraverso il flusso di file che costituisce il parametro *stream* (il primo argomento) e la funzione *sprintf()* copia il risultato, come stringa, a partire dal puntatore *s* (sempre il primo argomento).

```
int printf (const char *restrict composizione, ...);
```

```
int fprintf (FILE *restrict stream,  
            const char *restrict composizione, ...);
```

```
int sprintf (char *restrict s,  
            const char *restrict composizione, ...);
```

```
int snprintf (char *restrict s,  
             size_t n,  
             const char *restrict composizione, ...);
```

Le funzioni di cui è appena stato mostrato il modello sintattico, leggono gli argomenti successivi alla stringa di composizione in base a quanto indicato con gli specificatori di composizione. Altre funzioni equivalenti, con il nome che inizia con la lettera «v», hanno bisogno di un puntatore di tipo `va_list`:

```
int vprintf (const char *restrict composizione ,  
            va_list arg);
```

```
int vfprintf (FILE *restrict stream ,  
             const char *restrict composizione ,  
             va_list arg);
```

```
int vsprintf (char *restrict s ,  
            const char *restrict composizione ,  
            va_list arg);
```

```
int vsnprintf (char *restrict s ,  
             size_t n ,  
             const char *restrict composizione ,  
             va_list arg);
```

### 67.3.4 Concatenamento di stringhe

Il linguaggio C, di per sé, non agevola l'uso delle stringhe; al massimo si può contare sul fatto che una sequenza di stringhe letterali venga considerata una stringa sola, concatenata. Per il concatenamento



delle stringhe sono disponibili le funzioni *strcat()* e *strncat()*, ma l'uso delle funzioni previste per la composizione dell'output è molto più comodo, considerata la facilità con cui si inseriscono anche dati diversi dalle stringhe.

Listato 67.27. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/Nzbovidr>, <http://ideone.com/Jgn8z>.

```
#include <stdio.h>
int main (void)
{
    char s[] = "Ciao amore";
    printf ("%s... Ti %s.\n", s, "voglio tanto bene");
    return 0;
}
```

L'esempio appena mostrato dovrebbe dimostrare questa maggiore facilità. Il messaggio che viene visualizzato è: «Ciao amore... Ti voglio tanto bene.»

### 67.3.5 Interpretazione dell'input

«

Quando un programma interagisce con l'essere umano, scambia dati in forma grafica, nel senso che un numero appare e viene inserito come sequenza di caratteri grafici. Così come per la rappresentazione umana dei dati si usano comunemente le funzioni *..printf()*, per l'immissione dei dati si usano le funzioni *..scanf()* che hanno il ruolo opposto:

```
...scanf ( ... stringa_di_conversione [ , argomento ] ... )
```

Il modello sintattico dà solo una visione di massima: a seconda della

funzione ci possono essere dei parametri che non vengono chiariti nello schema, quindi appare sempre la stringa di conversione, la quale può essere seguita da altri argomenti costituiti da puntatori, le cui caratteristiche particolari non sono precisate nel prototipo della funzione.<sup>4</sup> Viene proposto un esempio con la funzione *scanf()* che riceve i dati in ingresso (da interpretare) dallo standard input:

```
...
printf ("Inserisci l'importo: ");
scanf ("%i", &i_importo);
...
```

Il pezzo di codice mostrato emette la frase seguente e resta in attesa dell'inserimento di un valore numerico intero, seguito da [*Invio*]:

```
Inserisci l'importo: _
```

Questo valore viene inserito nella variabile *i\_importo*. Si deve osservare il fatto che i parametri successivi alla stringa di conversione sono dei puntatori, per cui, avendo voluto inserire il dato nella variabile *i\_importo*, questa è stata indicata preceduta dall'operatore '*&*' in modo da fornire alla funzione l'indirizzo corrispondente (si veda la sezione [66.5](#) sulla gestione dei puntatori).

Con una stessa funzione di questo tipo è possibile inserire dati per diverse variabili, come si può osservare dall'esempio seguente, ma in questo caso, per ogni dato viene richiesta la pressione di [*Invio*] o l'inserimento di spazi tra un dato e l'altro.

```
...
printf ("Inserisci il capitale e il tasso: ");
scanf ("%i%f", &i_capitale, &i_tasso);
...
```

La stringa di conversione è il parametro più delicato di queste funzioni. Come visto negli esempi, una stringa del genere contiene principalmente degli specificatori di conversione che, come già accennato, si comportano in modo molto simile agli specificatori di composizione delle funzioni `...printf()`. Quello che segue è lo schema sintattico generale per la definizione di uno specificatore di conversione:

```
% [*] [n_ampiezza] [hh|h|l|ll|j|z|t|L] tipo
```

Come si può vedere, all'inizio è previsto un solo tipo di simbolo, costituito da un asterisco, il cui scopo è quello di annullare l'assegnamento del valore a una variabile. In pratica, con l'asterisco il dato corrispondente allo specificatore viene interpretato, ma poi non viene salvato in alcuna variabile.

Successivamente può apparire un numero che rappresenta l'ampiezza del dato da interpretare, in byte, il cui scopo è quello di limitare la lettura fino a un certo carattere (inteso come `'char'`, pertanto le sequenze multibyte contano per più di una unità singola). In questo caso non esiste la possibilità di indicare una precisione.

Dopo può apparire una sigla, composta da una o più lettere, il cui scopo è quello di modificare la dimensione predefinita della variabile di destinazione. In altri termini, senza questo modificatore si intende che la variabile ricevente debba essere di una certa grandezza, ma con l'aggiunta del «modificatore di lunghezza» si precisa invece qualcosa di diverso. In pratica, il modificatore di lunghezza usato da queste funzioni è equivalente a quello delle funzioni di composizione dell'output.

Al termine dello specificatore di conversione appare una lettera che



dichiara come deve essere interpretato il dato in ingresso e, in mancanza del modificatore di lunghezza, indica anche la dimensione predefinita della variabile ricevente.

Secondo la documentazione standard, il contenuto delle stringhe di conversione si suddivide in «direttive» che, in linea di massima, dovrebbero comporsi secondo il modello seguente:

```
[spazi] carattere_multibyte | %...
```

Pertanto, una direttiva può contenere degli spazi, un carattere (inteso in senso tipografico e quindi può occupare più di un byte) oppure uno specificatore di conversione. Visto da un altro punto di vista, la stringa di conversione è composta principalmente da specificatori di conversione che però possono essere alternati da spazi o altri caratteri: gli spazi indicano che in quella posizione possono esserci spazi che vengono ignorati; altri caratteri devono invece corrispondere esattamente nell'input e vengono poi ignorati. Tuttavia ci sono altre situazioni in cui gli spazi sono ugualmente esclusi in modo predefinito, come nell'esempio già visto, dove la stringa di conversione è composta solo da specificatori di conversione. Nell'esempio seguente, invece, si dimostra l'uso di caratteri estranei agli specificatori di conversione:

```
...  
printf ("Inserisci la data: ");  
scanf ("%i/%i/%i", &giorno, &mese, &anno);  
...
```

In questo caso la digitazione della data richiede anche l'inserzione delle barre oblique, senza le quali il riconoscimento fallisce.

Purtroppo, la sintassi per la scrittura delle stringhe di conversione non è molto soddisfacente ed è difficile avere un'idea chiara del loro utilizzo. Pertanto, è consigliabile di utilizzare sempre solo modelli molto semplici.

### 67.3.6 Rappresentazione degli specificatori di conversione

«

Di norma, la scelta dello specificatore di conversione determina il tipo di dati dell'argomento (ovvero il tipo di variabile a cui l'argomento punta) e il modo in cui deve essere interpretato. La tabella successiva elenca alcuni degli specificatori di conversione utilizzabili, nella loro forma più semplice. È bene ricordare che anche in questo caso si può usare uno specificatore costituito dall'unione di due caratteri percentuali ('%%'), il quale identifica semplicemente un carattere di percentuale singolo proveniente dai dati in ingresso, ma da ignorare.

Tabella 67.32. Tipi di conversione principali.

Simbolo	Corrispondenza
%...c	Corrisponde a un carattere, oppure, se viene specificata una lunghezza, a una sequenza di caratteri; pertanto, in condizioni normali il puntatore deve essere di tipo ' <b>char *</b> ' e, a partire dalla posizione indicata dallo stesso, ci deve essere abbastanza spazio per contenere tutti i caratteri previsti per l'immissione.

Simbolo	Corrispondenza
%...s	Corrisponde a una sequenza di caratteri diversi da quelli che producono uno spazio e a questa sequenza viene aggiunto automaticamente il carattere <code>&lt;NUL&gt;</code> , ovvero <code>'\0'</code> ; pertanto, in condizioni normali il puntatore deve essere di tipo <code>'char *'</code> e, a partire dalla posizione indicata dallo stesso, ci deve essere abbastanza spazio per contenere la stringa da immettere, incluso il carattere nullo conclusivo.
%...d	Corrisponde a un numero intero (con o senza segno), espresso in base dieci; pertanto, in condizioni normali il puntatore deve essere di tipo <code>'int *'</code> , con o senza segno.
%...i	Corrisponde a un numero intero (con o senza segno), espresso in base otto, in base dieci o in base sedici (purché la base di numerazione sia riconoscibile dal prefisso usato); pertanto, in condizioni normali il puntatore deve essere di tipo <code>'int *'</code> , con o senza segno.
%...u	Corrisponde a un numero intero senza segno, espresso in base dieci; pertanto, in condizioni normali il puntatore deve essere di tipo <code>'unsigned int *'</code> .
%...x	Corrisponde a un numero intero senza segno, espresso in base sedici; pertanto, in condizioni normali il puntatore deve essere di tipo <code>'unsigned int *'</code> .
%...o	Corrisponde a un numero intero senza segno, espresso in base otto; pertanto, in condizioni normali il puntatore deve essere di tipo <code>'unsigned int *'</code> .

Simbolo	Corrispondenza
%...e	Corrisponde a un numero a virgola mobile rappresentato in notazione decimale fissa o in notazione esponenziale; pertanto, in condizioni normali il puntatore deve essere di tipo <b>'double *'</b> .
%...f	
%...g	

Tabella 67.33. Alcuni modificatori dell'ampiezza del valore da immettere.

Simbolo	Corrispondenza
%...hd	La lettera <b>'h'</b> , prima di quella che specifica il tipo di conversione, quando si tratta dell'inserimento di un numero intero, specifica che il dato va trattato in qualità di <b>'short int *'</b> , con o senza segno, in base al contesto.
%...hi	
%...hu	
%...ho	
%...hx	
%...ld	La lettera <b>'l'</b> , prima di quella che specifica il tipo di conversione, quando si tratta dell'inserimento di un numero intero, specifica che il dato va trattato in qualità di <b>'long int *'</b> , con o senza segno, in base al contesto.
%...lu	
%...lo	
%...lx	

Simbolo	Corrispondenza
%...lc %...ls	La lettera ‘ <b>l</b> ’, prima di quella che specifica il tipo di conversione, quando si tratta dell’inserimento di un carattere o di una stringa, specifica che il dato va trattato in qualità di carattere esteso ( <i>wide char</i> ) o di stringa estesa ( <i>wide string</i> ). Di conseguenza, si intende che l’argomento sia di tipo ‘ <b>wchar_t *</b> ’.
%...Le %...Lf %...Lg	La lettera ‘ <b>L</b> ’, prima di quella che specifica il tipo di conversione, quando si tratta dell’inserimento di un valore in virgola mobile, specifica che il dato va trattato in qualità di ‘ <b>long double *</b> ’.

### 67.3.7 Funzioni per l’interpretazione dell’input

Tutte le funzioni standard il cui nome finisce per ‘**scanf**’ interpretano dei dati in ingresso attraverso una stringa di conversione, secondo le modalità descritte nel capitolo, ovvero in modo analogo a *scanf()* che, in particolare, legge i dati da interpretare dallo standard input. In particolare, la funzione *fscanf()* legge l’input attraverso il flusso di file che costituisce il parametro *stream* (il primo argomento) e la funzione *sscanf()* legge l’input da una stringa (che costituisce sempre il primo argomento).

```
int fscanf (FILE *restrict stream,
           const char *restrict conversione, ...);
```

```
int sscanf (const char *restrict s,  
           const char *restrict conversione, ...);
```

```
int scanf (const char *restrict conversione, ...);
```

Le funzioni di cui è appena stato mostrato il modello sintattico, utilizzano gli argomenti successivi alla stringa di conversione in base a quanto indicato con gli specificatori di conversione. Altre funzioni equivalenti, con il nome che inizia con la lettera «v», hanno bisogno di un puntatore di tipo `va_list`:

```
int vfscanf (FILE *restrict stream,  
            const char *restrict conversione,  
            va_list arg);
```

```
int vsscanf (const char *restrict s,  
            const char *restrict conversione,  
            va_list arg);
```

```
int vscanf (const char *restrict conversione,  
            va_list arg);
```

## 67.4 Riferimenti



- *Rationale for American National Standard for Information Systems - Programming Language - C: Input/Output*, <http://www.lysator.liu.se/c/rat/d9.html>
- *ISO/IEC 9899:TC2*, <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf>

<sup>1</sup> Nella tabella, in questa fase, non si distingue ancora tra accessi a file di testo rispetto a quelli relativi a file binari, pertanto non appare mai la sigla ‘**b**’.

<sup>2</sup> Può trattarsi anche di sequenze multibyte, ovvero di rappresentazioni dei caratteri che usano più byte per carattere.

<sup>3</sup> Questa è una semplificazione, perché ci sono altre funzioni dello stesso gruppo, che iniziano con la lettera ‘**v**’, le quali alla fine hanno un puntatore di tipo ‘**va\_list**’.

<sup>4</sup> Questa è una semplificazione, perché ci sono altre funzioni dello stesso gruppo, che iniziano con la lettera ‘**v**’, le quali alla fine hanno un puntatore di tipo ‘**va\_list**’.





# Esempi di programmazione in C



Problemi elementari .....	873
Somma tra due numeri positivi .....	874
Moltiplicazione di due numeri positivi attraverso la somma	876
Divisione intera tra due numeri positivi .....	877
Elevamento a potenza .....	879
Radice quadrata .....	881
Fattoriale .....	883
Massimo comune divisore .....	884
Numero primo .....	886
Successione di Fibonacci .....	887
Scansione di array .....	890
Ricerca sequenziale .....	890
Ricerca binaria .....	893
Algoritmi tradizionali .....	896
Bubblesort .....	896
Torre di Hanoi .....	899
Quicksort .....	900
Permutazioni .....	904

In questo capitolo vengono mostrati alcuni algoritmi elementari o comuni portati in C. Per la spiegazione di questi, se non sono già conosciuti, conviene leggere quanto riportato a partire dalla sezione [62.2](#).

# Problemi elementari

«

Somma tra due numeri positivi

«

Una copia di questo file dovrebbe essere disponibile presso [allegati/somma.c](#) .

Listato u3.1. Per provare il codice attraverso un servizio *pastebin*:  
<http://codepad.org/0fuGkfkz> , <http://ideone.com/SQB0I> .

```
#include <stdio.h>

int
somma (int x, int y)
{
    int z = x;
    int i;

    for (i = 1; i <= y; i++)
        {
            z++;
        };

    return z;
}

int
main (int argc, char *argv[])
{
    int x;
    int y;
    int z;

    // Converta le stringhe ottenute dalla riga di comando in
    // numeri interi e li assegna alle variabili x e y.
```

```
    sscanf (argv[1], "%i", &x);
    sscanf (argv[2], "%i", &y);

    z = somma (x, y);

    printf ("%i + %i = %i\n", x, y, z);

    return 0;
}
```

In alternativa si può tradurre il ciclo **‘for’** in un ciclo **‘while’**.

Listato u3.2. Per provare il codice attraverso un servizio *pastebin*:  
<http://codepad.org/TKwppy25> , <http://ideone.com/K33pD33> .

```
int
somma (int x, int y)
{
    int z = x;
    int i = 1;

    while (i <= y)
        {
            z++;
            i++;
        };

    return z;
}
```

## Moltiplicazione di due numeri positivi attraverso la somma

«

Una copia di questo file dovrebbe essere disponibile presso [allegati/moltiplica.c](#) .

Listato u3.3. Per provare il codice attraverso un servizio *pastebin*:  
<http://codepad.org/xjV7MZte> , <http://ideone.com/b5Wxx> .

```
#include <stdio.h>

int
moltiplica (int x, int y)
{
    int z = 0;
    int i;

    for (i = 1; i <= y; i++)
        {
            z = z + x;
        }

    return z;
}

int
main (int argc, char *argv[])
{
    int x;
    int y;
    int z;

    // Converta le stringhe ottenute dalla riga di comando
    // in numeri interi e li assegna alle variabili x e y.

    sscanf (argv[1], "%i", &x);
    sscanf (argv[2], "%i", &y);
```

```
z = moltiplica (x, y);

printf ("%i * %i = %i\n", x, y, z);

return 0;
}
```

In alternativa si può tradurre il ciclo **for** in un ciclo **while**.

Listato u3.4. Per provare il codice attraverso un servizio *pastebin*:  
<http://codepad.org/9GQLZmXk> , <http://ideone.com/CoQf0> .

```
int moltiplica (int x, int y)
{
    int z = 0;
    int i = 1;

    while (i <= y)
    {
        z = z + x;
        i++;
    }

    return z;
}
```

Divisione intera tra due numeri positivi

Una copia di questo file dovrebbe essere disponibile presso [allegati/dividi.c](#) .

Listato u3.5. Per provare il codice attraverso un servizio *pastebin*:  
<http://codepad.org/h15j1MVo> , <http://ideone.com/EmJ3X> .

```
#include <stdio.h>

int
dividi (int x, int y)
{
    int z = 0;
    int i = x;

    while (i >= y)
        {
            i = i - y;
            z++;
        }

    return z;
}

int
main (int argc, char *argv[])
{
    int x;
    int y;
    int z;

    // Converte le stringhe ottenute dalla riga di comando
    // in numeri interi e li assegna alle variabili x e y.

    sscanf (argv[1], "%i", &x);
    sscanf (argv[2], "%i", &y);

    z = dividi (x, y);
```

```
printf ("Divisione intera - %i:%i = %i\n", x, y, z);

return 0;

}
```

## Elevamento a potenza

Una copia di questo file dovrebbe essere disponibile presso [allegati/exp.c](#) .

Listato u3.6. Per provare il codice attraverso un servizio *pastebin*:  
<http://codepad.org/mxkVE5yi> , <http://ideone.com/guhJ6> .

```
#include <stdio.h>

int
exp (int x, int y)
{
    int z = 1;
    int i;

    for (i = 1; i <= y; i++)
        {
            z = z * x;
        }

    return z;
}

int
main (int argc, char *argv[])
{
    int x;
    int y;
    int z;
```

```
// Converte le stringhe ottenute dalla riga di comando
// in numeri interi e li assegna alle variabili x e y.
```

```
sscanf (argv[1], "%i", &x);
sscanf (argv[2], "%i", &y);

z = exp (x, y);

printf ("%i ** %i = %i\n", x, y, z);

return 0;
}
```

In alternativa si può tradurre il ciclo **‘for’** in un ciclo **‘while’**.

Listato u3.7. Per provare il codice attraverso un servizio *pastebin*:  
<http://codepad.org/7uP7uRGg> , <http://ideone.com/4X81h> .

```
int
exp (int x, int y)
{
    int z = 1;
    int i = 1;

    while (i <= y)
        {
            z = z * x;
            i++;
        };

    return z;
}
```

È possibile usare anche un algoritmo ricorsivo.



Listato u3.8. Per provare il codice attraverso un servizio *pastebin*:  
<http://codepad.org/mwiZu0T7> , <http://ideone.com/x0ei9> .

```
int exp (int x, int y)
{
    if (x == 0)
        {
            return 0;
        }
    else if (y == 0)
        {
            return 1;
        }
    else
        {
            return (x * exp (x, y-1));
        }
}
```

## Radice quadrata

Una copia di questo file dovrebbe essere disponibile presso [allegati/radice.c](#) .

Listato u3.9. Per provare il codice attraverso un servizio *pastebin*:  
<http://codepad.org/8FgV711v> , <http://ideone.com/rO8KS> .

```
#include <stdio.h>

int
radice (int x)
{
    int z = 0;
    int t = 0;

    while (1)
```

```

    {
        t = z * z;

        if (t > x)
        {
            // È stato superato il valore massimo.
            z--;
            return z;
        }

        z++;
    }

    // Teoricamente, non dovrebbe mai arrivare qui.
}

int
main (int argc, char *argv[])
{
    int x;
    int z;

    sscanf (argv[1], "%i", &x);

    z = radice (x);

    printf ("radq(%i) = %i\n", x, z);

    return 0;
}

```

## Fattoriale

Una copia di questo file dovrebbe essere disponibile presso [allegati/fatt.c](#).

Listato u3.10. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/vBGnZfEf>, <http://ideone.com/jYVgR>.

```
#include <stdio.h>

int
fatt (int x)
{
    int i = (x - 1);

    while (i > 0)
    {
        x = x * i;
        i--;
    }

    return x;
}

int
main (int argc, char *argv[])
{
    int x;
    int z;

    sscanf (argv[1], "%i", &x);

    z = fatt (x);

    printf ("%i! = %i\n", x, z);
}
```

```
    return 0;
}
```

In alternativa, l'algoritmo si può tradurre in modo ricorsivo.

Listato u3.11. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/LIC6Vyxp> , <http://ideone.com/6SfUj> .

```
int
fatt (int x)
{
    if (x > 1)
    {
        return (x * fatt (x - 1));
    }
    else
    {
        return 1;
    }
}
```

Massimo comune divisore

«

Una copia di questo file dovrebbe essere disponibile presso [allegati/mcd.c](#) .

Listato u3.12. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/ETf2XcdR> , <http://ideone.com/8H1og> .

```
#include <stdio.h>

int
mcd (int x, int y)
```

```

{
    while (x != y)
        {
            if (x > y)
                {
                    x = x - y;
                }
            else
                {
                    y = y - x;
                }
        }

    return x;
}

int
main (int argc, char *argv[])
{
    int x;
    int y;
    int z;

    sscanf (argv[1], "%i", &x);
    sscanf (argv[2], "%i", &y);

    z = mcd (x, y);

    printf ("Il massimo comune divisore di %i e %i è %i\n",
           x, y, z);

    return 0;
}

```

## Numero primo



Una copia di questo file dovrebbe essere disponibile presso [allegati/primoc](#).

Listato u3.13. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/T6vjaM2y>, <http://ideone.com/X1oos>.

```
#include <stdio.h>

unsigned int
primo (int x)
{
    unsigned int primo = 1;
    int i = 2;
    int j;

    while ((i < x) && primo)
    {
        j = x / i;
        j = x - (j * i);

        if (j == 0)
        {
            primo = 0;
        }
        else
        {
            i++;
        }
    }

    return primo;
}
```

```

int
main (int argc, char *argv[])
{
    int x;

    sscanf (argv[1], "%i", &x);

    if (primo (x))
        {
            printf ("%i è un numero primo\n", x);
        }
    else
        {
            printf ("%i non è un numero primo\n", x);
        }

    return 0;
}

```

## Successione di Fibonacci

Gli esempi mostrano una funzione che restituisce l'elemento  $n$ -esimo nella sequenza di Fibonacci, mentre la chiamata di questa funzione viene fatta in modo da ottenere l'elenco dei primi  $n$  numeri di Fibonacci. La prima soluzione mostra una funzione ricorsiva. Una copia di questo file dovrebbe essere disponibile presso [allegati/fibonacci.c](#) .

Listato u3.14. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/6CHi9taB> , <http://ideone.com/0Z1dz> .

```
#include <stdio.h>

unsigned int
fibonacci (unsigned int n)
{
    if (n == 0)
        {
            return 0;
        }
    else if (n == 1)
        {
            return 1;
        }
    else
        {
            return (fibonacci (n-1) + fibonacci (n-2));
        }
}

int
main (int argc, char *argv[])
{
    unsigned int n;
    unsigned int i;

    sscanf (argv[1], "%u", &n);

    for (i = 0; i <= n; i++)
        {
            printf ("%u ", fibonacci (i));
        }
}
```



```
printf ("\n");

return 0;
}
```

L'esempio seguente mostra solo la funzione, in forma iterativa:

Listato u3.15. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/1sH2hhIf>, <http://ideone.com/8z9J8>.

```
unsigned int
fibonacci (unsigned int n)
{
    unsigned int f1 = 1;
    unsigned int f0 = 0;
    unsigned int fn = n;
    unsigned int i;

    for (i = 2; i <= n; i++)
        {
            fn = f1 + f0;
            f0 = f1;
            f1 = fn;
        }

    return fn;
}
```

# Scansione di array

« Ricerca sequenziale

« Una copia di questo file dovrebbe essere disponibile presso [allegati/ricercaseq.c](#) .

Listato u3.16. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/9p0P3GU9> , <http://ideone.com/J8hQb> .

```
#include <stdio.h>

int
ricercaseq (int lista[], int x, int a, int z)
{
    int i;

    // Scandisce l'array alla ricerca dell'elemento.

    for (i = a; i <= z; i++)
    {
        if (x == lista[i])
        {
            return i;
        }
    }

    // La corrispondenza non è stata trovata.

    return -1;
}

int
main (int argc, char *argv[])
```

```

{
    int lista[argc - 2];
    int x;
    int i;

    // Acquisisce il primo argomento come valore da cercare.

    sscanf (argv[1], "%i", &x);

    // Considera gli argomenti successivi come gli elementi
    // dell'array da scandire.

    for (i = 2; i < argc; i++)
        {
            sscanf (argv[i], "%i", &lista[i-2]);
        }

    // Esegue la ricerca.

    i = ricercaseq (lista, x, 0, argc - 2);

    // Emette il risultato.

    printf ("%i si trova nella posizione %i\n", x, i);

    return 0;
}

```

Al posto di dichiarare l'array *lista* con una quantità di elementi definita in fase di funzionamento, si può usare la funzione *malloc()*, avendo cura di incorporare il file 'stdlib.h':

Listato u3.17. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/5nP79Nf798>, <http://ideone.com/YdMdC>.

```
#include <stdio.h>
#include <stdlib.h>
...
int
main (int argc, char *argv[])
{
    int *lista = (int *) malloc ((argc - 2) * sizeof (int));
    ...
```

Esiste anche una soluzione ricorsiva che viene mostrata nella subroutine seguente:

Listato u3.18. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/SC2AheV2> , <http://ideone.com/sk64m> .

```
int
ricercaseq (int lista[], int x, int a, int z)
{
    if (a > z)
        {
            // La corrispondenza non è stata trovata.

            return -1;
        }
    else if (x == lista[a])
        {
            return a;
        }
    else
        {
            return ricercaseq (lista, x, a+1, z);
        }
}
```

## Ricerca binaria

Una copia di questo file dovrebbe essere disponibile presso [allegati/ricercabin.c](#) .

Listato u3.19. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/vU9RwE9m> , <http://ideone.com/jEyYk> .

```
#include <stdio.h>

int
ricercabin (int lista[], int x, int a, int z)
```

```

{
    int m;

    // Determina l'elemento centrale.

    m = (a + z) / 2;

    if (m < a)
    {
        // Non restano elementi da controllare: l'elemento
        // cercato non c'è.

        return -1;
    }
    else if (x < lista[m])
    {
        // Si ripete la ricerca nella parte inferiore.

        return ricercabin (lista, x, a, m-1);
    }
    else if (x > lista[m])
    {
        // Si ripete la ricerca nella parte superiore.

        return ricercabin (lista, x, m+1, z);
    }
    else
    {
        // La variabile m rappresenta l'indice dell'elemento
        // cercato.

        return m;
    }
}

```

```

int main (int argc, char *argv[])
{
    int lista[argc - 2];
    int x;
    int i;

    // Acquisisce il primo argomento come valore da cercare.

    sscanf (argv[1], "%i", &x);

    // Considera gli argomenti successivi come gli elementi
    // dell'array da scandire.

    for (i = 2; i < argc; i++)
    {
        sscanf (argv[i], "%i", &lista[i-2]);
    }

    // Esegue la ricerca.

    i = ricercabin (lista, x, 0, argc-2);

    // Emette il risultato.

    printf ("%i si trova nella posizione %i\n", x, i);

    return 0;
}

```

Per questo esempio vale la stessa considerazione fatta nella sezione precedente, a proposito dell'uso di *malloc()* al posto di un array con una quantità di elementi definita dinamicamente durante il

funzionamento del programma.

## Algoritmi tradizionali

«

Bubblesort

«

Viene mostrata prima una soluzione iterativa e successivamente la funzione **'bsort'** in versione ricorsiva.

Una copia di questo file dovrebbe essere disponibile presso [allegati/bsort.c](#).

Listato u3.20. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/10ot10rUOE>, <http://ideone.com/hJlie>.

```
#include <stdio.h>

void
bsort (int lista[], int a, int z)
{
    int scambio;
    int j;
    int k;

    // Inizia il ciclo di scansione dell'array.

    for (j = a; j < z; j++)
    {
        // Scansione interna dell'array per collocare nella
        // posizione j l'elemento giusto.

        for (k = j+1; k <= z; k++)
        {
            if (lista[k] < lista[j])
            {
```



```

        // Scambia i valori.

        scambio = lista[k];
        lista[k] = lista[j];
        lista[j] = scambio;
    }
}
}

int
main (int argc, char *argv[])
{
    int lista[argc-1];
    int i;

    // Considera gli argomenti come gli elementi
    // dell'array da ordinare.

    for (i = 1; i < argc; i++)
    {
        sscanf (argv[i], "%i", &lista[i-1]);
    }

    // Esegue il riordino.

    bsort (lista, 0, argc-2);

    // Emette il risultato.

    for (i = 0; i < (argc-1); i++)
    {
        printf ("%i ", lista[i]);
    }
}

```

```
printf ("\n");

return 0;
}
```

Segue la funzione ‘**bsort**’ in versione ricorsiva.

Listato u3.21. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/izHITJ7y>, <http://ideone.com/5mSDT>.

```
void
bsort (int lista[], int a, int z)
{
    int scambio;
    int k;

    if (a < z)
        {
            // Scansione interna dell'array per collocare nella
            // posizione a l'elemento giusto.

            for (k = a+1; k <= z; k++)
                {
                    if (lista[k] < lista[a])
                        {
                            // Scambia i valori.

                            scambio = lista[k];
                            lista[k] = lista[a];
                            lista[a] = scambio;
                        }
                }

            bsort (lista, a+1, z);
        }
}
```

```
    }  
}
```

Al posto di dichiarare l'array *lista* con una quantità di elementi definita in fase di funzionamento, si può usare la funzione *malloc()*, avendo cura di incorporare il file 'stdlib.h':

Listato u3.22. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/M76CQ76gHg> , <http://ideone.com/j7hB5> .

```
#include <stdio.h>  
#include <stdlib.h>  
...  
int  
main (int argc, char *argv[])  
{  
    int *lista = (int *) malloc ((argc - 1) * sizeof (int));  
    ...  
}
```

## Torre di Hanoi

Una copia di questo file dovrebbe essere disponibile presso [allegati/hanoi.c](#) .

Listato u3.23. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/FvvVQmru> , <http://ideone.com/kr6DC> .

```
#include <stdio.h>  
  
void hanoi (int n, int p1, int p2)  
{  
    if (n > 0)
```

```

    {
        hanoi (n-1, p1, 6-p1-p2);
        printf ("Muovi l'anello %i dal piolo %i "
                "al piolo %i\n",
                n, p1, p2);
        hanoi (n-1, 6-p1-p2, p2);
    }
}

int main (int argc, char *argv[])
{
    int n;
    int p1;
    int p2;

    sscanf (argv[1], "%i", &n);
    sscanf (argv[2], "%i", &p1);
    sscanf (argv[3], "%i", &p2);

    hanoi (n, p1, p2);

    return 0;
}

```

## Quicksort



Una copia di questo file dovrebbe essere disponibile presso [allegati/qsart.c](#) .

Listato u3.24. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/QbDq6aMz> , <http://ideone.com/NJwFO> .

```
#include <stdio.h>

int
part (int lista[], int a, int z)
{
    // Viene preparata una variabile per lo scambio di
    // valori.

    int scambio = 0;

    // Si assume che «a» sia inferiore a «z».

    int i = a + 1;
    int cf = z;

    // Inizia il ciclo di scansione dell'array.

    while (1)
    {
        while (1)
        {
            // Sposta «i» a destra.

            if ((lista[i] > lista[a]) || (i >= cf))
            {
                break;
            }
            else
            {
                i += 1;
            }
        }
    }
}
```

```

    }
while (1)
{
    // Sposta «cf» a sinistra.

    if (lista[cf] <= lista[a])
    {
        break;
    }
    else
    {
        cf -= 1;
    }
}
if (cf <= i)
{
    // È avvenuto l'incontro tra «i» e «cf».

    break;
}
else
{
    // Vengono scambiati i valori.

    scambio = lista[cf];
    lista[cf] = lista[i];
    lista[i] = scambio;

    i += 1;
    cf -= 1;
}
}

// A questo punto lista[a..z] è stata ripartita e «cf» è

```

```

    // la collocazione di «lista[a]».

    scambio = lista[cf];
    lista[cf] = lista[a];
    lista[a] = scambio;

    // A questo punto, lista[cf] è un elemento (un valore)
    // nella giusta posizione.

    return cf;
}

void
quicksort (int lista[], int a, int z)
{
    // Viene preparata la variabile «cf».

    int (cf) = 0;

    if (z > a)
        {
            cf = part (lista, a, z);
            quicksort (lista, a, cf-1);
            quicksort (lista, cf+1, z);
        }
}

int
main (int argc, char *argv[])
{
    int lista[argc - 1];
    int i;

    // Considera gli argomenti come gli elementi

```

```

// dell'array da ordinare.

for (i = 1; i < argc; i++)
    {
        sscanf (argv[i], "%i", &lista[i-1]);
    }

// Esegue il riordino.

quicksort (lista, 0, argc-2);

// Emette il risultato.

for (i = 0; i < (argc-1); i++)
    {
        printf ("%i ", lista[i]);
    }
printf ("\n");

return 0;
}

```

Per questo esempio vale la stessa considerazione già fatta a proposito dell'uso di *malloc()* al posto di un array con una quantità di elementi definita dinamicamente durante il funzionamento del programma.

Permutazioni

«

Una copia di questo file dovrebbe essere disponibile presso [allegati/permuta.c](#) .



Listato u3.25. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/ca66C9da> , <http://ideone.com/I5bJV> .

```
#include <stdio.h>

void visualizza (int lista[], int dimensione)
{
    int i;

    for (i = 0; i < dimensione; i++)
        {
            printf ("%i ", lista[i]);
        }
    printf ("\n");
}

void permuta (int lista[], int a, int z, int dimensione)
{
    int scambio;
    int k;

    // Se il segmento di array contiene almeno due elementi,
    // si procede.

    if ((z - a) >= 1)
        {
            // Inizia un ciclo di scambi tra l'ultimo elemento e
            // uno degli altri contenuti nel segmento di array.

            for (k = z; k >= a; k--)
                {
                    // Scambia i valori.

                    scambio = lista[k];
```

```

        lista[k] = lista[z];
        lista[z] = scambio;

        // Esegue una chiamata ricorsiva per permutare
        // un segmento più piccolo dell'array.

        permuta (lista, a, z - 1, dimensione);

        // Scambia i valori.

        scambio = lista[k];
        lista[k] = lista[z];
        lista[z] = scambio;
    }
}
else
{
    // Visualizza l'array.

    visualizza (lista, dimensione);
}
}

int
main (int argc, char *argv[])
{
    int lista[argc - 1];
    int i;

    // Considera gli argomenti come gli elementi
    // dell'array da permutare.

    for (i = 1; i < argc; i++)
    {

```

```
        sscanf (argv[i], "%i", &lista[i-1]);
    }

    // Esegue le permutazioni.

    permuta (lista, 0, argc - 2, argc - 1);

    return 0;
}
```

Per questo esempio vale la stessa considerazione già fatta a proposito dell'uso di *malloc()* al posto di un array con una quantità di elementi definita dinamicamente durante il funzionamento del programma.



## Introduzione alle estensioni POSIX



68.1	Dal C a POSIX .....	912
68.1.1	Byte di otto bit .....	913
68.1.2	Errori .....	914
68.1.3	Gestione dei file .....	915
68.1.4	Altre funzionalità particolari .....	915
68.2	Espressioni regolari POSIX .....	916
68.2.1	Compilazione, confronto e rilascio della memoria 918	
68.2.2	Compilazione dell'espressione regolare .....	919
68.2.3	Liberazione della memoria .....	923
68.2.4	Confronto .....	923
68.2.5	Estrapolazione di sottostringhe .....	927
68.2.6	Informazioni diagnostiche .....	932
68.2.7	Esempio completo .....	936
68.3	Avvio e conclusione dei processi .....	940
68.3.1	Biforcazione: «fork» .....	940
68.3.2	Sostituzione: «exec» .....	943
68.3.3	Attesa della conclusione di un processo figlio .....	945
68.3.4	Adozione dei processi .....	946
68.3.5	Gli zombie .....	948
68.3.6	Variabili di ambiente .....	950
68.3.7	Variabili di ambiente e avvio di un programma ..	951

68.4	Nozioni sui thread POSIX	953
68.4.1	Numero identificativo del thread	954
68.4.2	Creazione e conclusione di un thread aggiuntivo	955
68.4.3	Caratteristiche della funzione che costituisce un thread aggiuntivo	956
68.4.4	Avvio di thread separati e fusione successiva	957
68.4.5	Conflitto nell'accesso ai dati	962
68.4.6	Accesso alle risorse in modo mutualmente esclusivo	966
68.4.7	Accesso esclusivo, ma condizionato	972
68.4.8	Osservazioni finali	978
68.5	I file secondo i sistemi POSIX	979
68.5.1	Apertura e chiusura di un file	980
68.5.2	Lettura e scrittura	987
68.5.3	Spostamento dell'indicatore interno al file	992
68.5.4	Controllo degli errori	995
68.6	Il file system Unix e la sua gestione tipica	996
68.6.1	Il blocco	996
68.6.2	Il super blocco	997
68.6.3	Gli inode	998
68.6.4	La directory	1001
68.6.5	Tabelle del sistema operativo	1002
68.7	Il file system Minix 1	1008

68.7.1	Blocchi e zone .....	1009
68.7.2	Struttura generale .....	1010
68.7.3	Super blocco .....	1011
68.7.4	Mappa di inode .....	1013
68.7.5	Mappa di zone .....	1015
68.7.6	Inode .....	1016
68.7.7	Directory .....	1020
68.7.8	Il problema dell'inversione dei byte .....	1022
68.8	Creazione ed eliminazione di file di qualunque tipo ..	1022
68.8.1	La funzione «mknod()» .....	1023
68.8.2	La funzione «mkdir()» .....	1031
68.8.3	La funzione «mkfifo()» .....	1032
68.8.4	La funzione «unlink()» .....	1032
68.8.5	La funzione «rmdir()» .....	1033
68.8.6	La funzione «remove()» .....	1034
68.9	Condotti .....	1034
68.9.1	Realizzazione materiale del condotto .....	1035
68.9.2	Condotti «senza nome» e condotti «con nome» ..	1037
68.9.3	Protocollo di accesso ai condotti .....	1037
68.9.4	Funzione «pipe()» .....	1038
68.9.5	Esempio di condotto attraverso un file FIFO ....	1043
68.10	Lettura delle directory .....	1047
68.10.1	Tipi derivati .....	1048
68.10.2	Procedura per accedere a una directory ....	1048

68.10.3	Attributo «FD_CLOEXEC» .....	1049					
68.10.4	Esempio di utilizzo delle funzioni di accesso alle directory .....	1050					
68.11	Riferimenti .....	1051					
close()	980	closedir()	1048	creat()	980	DIR	1048
dirent.h	1047	environ	950	errno	995	execl()	943
execve()	951	fcntl.h	979	FD_CLOEXEC	1049	fork()	940
init	946	lseek()	992	major()	1028	makedev()	1028
minor()	1028	mkdir()	1031	mkfifo()	1032 1043	mknod()	1023
open()	980	opendir()	1048	O_APPEND	980	O_CREAT	980
O_EXCL	980	O_NOCTTY	980	O_NONBLOCK	980	O_RDONLY	980
O_RDWR	980	O_SYNC	980	O_TRUNC	980	O_WRONLY	980
pipe()	1038	pthread_t	954	read()	987	readdir()	1048
regcomp()	918 919	regerror()	918 932	regex.h	918	regexec()	918 923 927
regex_t	918 919	regfree()	918	regmatch_t	918 923 927	remove()	1034
rewinddir()	1048	rmdir()	1033	stat.h	979	struct dirent	1048
S_IRGRP	980	S_IROTH	980	S_IRUSR	980	S_IRWXG	980
S_IRWXO	980	S_IRWXU	980	S_ISGID	980	S_ISUID	980
S_ISVTX	980	S_IWGRP	980	S_IWOTH	980	S_IWUSR	980
S_IXGRP	980	S_IXOTH	980	S_IXUSR	980	unistd.h	979
unlink()	1032	wait()	945	write()	987		

## 68.1 Dal C a POSIX



Lo standard del linguaggio C è definito in modo tale da consentire l'uso in contesti architetturali molto diversi tra loro. Tuttavia, il linguaggio C è nato per i sistemi Unix e quando si vuole considera-



re una libreria di funzioni più ampia, rispetto allo standard del linguaggio, ci si riferisce normalmente allo standard POSIX. Pertanto, lo standard POSIX estende la libreria del linguaggio C, con funzionalità che dipendono da un'organizzazione del sistema operativo conforme, almeno fondamentalmente, a quella di Unix.

Logicamente, i sistemi operativi possono essere più o meno conformi con lo standard POSIX e l'utilizzo delle estensioni introdotte da questo standard va preceduto da una ricerca sulla loro compatibilità. In generale va osservato che dal C puro a POSIX, la filosofia di programmazione cambia leggermente e occorre considerare alcuni particolari.

### 68.1.1 Byte di otto bit

Lo standard C prescrive che un byte sia di almeno 8 bit, mentre secondo POSIX, il byte diventa necessariamente di 8 bit. Infatti, nel file di intestazione `<stdint.h>`, secondo lo standard C la definizione dei tipi derivati `[u]intn_t` è facoltativa, ma nello standard POSIX diviene obbligatoria. Così facendo, dato che diviene obbligatorio disporre dei tipi derivati `int8_t` e `uint8_t`, non avendo un altro modo per definirli, è necessario che il tipo `char` sia esattamente di 8 bit: «

```
typedef signed char      int8_t;
...
typedef unsigned char   uint8_t;
```

Per lo stesso motivo, tutti gli altri tipi interi devono essere multipli (al quadrato), del byte, ma oltre a questo, quando la CPU fosse anche in grado di gestire facilmente interi più grandi di 32 bit, il tipo `int` potrebbe essere al massimo di soli 32 bit. Per esempio, dispo-

nendo di una CPU a 128 bit, si è praticamente costretti a dichiarare questi tipi derivati nel modo seguente, riservando, evidentemente, il tipo `'long long int'` per gli interi a 128 bit che per l'architettura sarebbero invece normali:

```
typedef signed char          int8_t;
typedef short int           int16_t;
typedef int                 int32_t;
typedef long int            int64_t;
//typedef long long int     int128_t;    // non standard

typedef unsigned char       uint8_t;
typedef unsigned short int  uint16_t;
typedef unsigned int        uint32_t;
typedef unsigned long int   uint64_t;
//typedef unsigned long long int uint128_t; // non standard
```

## 68.1.2 Errori

«

Molte funzioni, aggiunte dallo standard POSIX, che restituiscono un valore intero, utilizzano il valore `-1` per dichiarare la presenza di un errore che ha prodotto un esito non valido. In tal caso, si possono avere funzioni che restituiscono un valore, che, se maggiore o uguale a zero, è valido, mentre se è negativo non lo è, oppure funzioni che semplicemente restituiscono zero o `-1`.

Questo fatto si scontra con il principio per cui lo zero equivale a *Falso* e un valore diverso da zero equivale a *Vero*, perché in questo caso occorre verificare precisamente il valore, oppure occorre invertirlo logicamente perché abbia il significato atteso.

Questa inversione logica dei risultati si riscontra anche nei programmi di servizio di un sistema operativo POSIX, per cui il linguaggio

della shell POSIX considera lo zero come un valore pari a *Vero* e qualunque altro valore pari a *Falso*.

Oltre a questo fatto, i tipi di errore che si possono annotare nella variabile *errno* sono molto più numerosi rispetto a quanto previsto dallo standard C, pertanto aumentano le macro-variabili previste nel file ‘`errno.h`’.

### 68.1.3 Gestione dei file

La libreria standard del linguaggio C prevede una gestione dei file attraverso dei «flussi», mentre la libreria POSIX introduce il concetto di descrittore del file (*file descriptor*), ovvero un semplice numero intero.<sup>1</sup>

In uno stesso programma si possono usare entrambe le modalità di gestione dei file, ma è evidente che va evitato l’uso simultaneo sullo stesso file.

A parte la differenza nel modo di identificare un file aperto in un programma, la libreria POSIX mette in condizione di accedere alla gestione dei permessi e delle altre caratteristiche dei file, secondo le convenzioni di un sistema Unix; inoltre offre una gestione ordinata dei blocchi di accesso a porzioni degli stessi (*lock*).

### 68.1.4 Altre funzionalità particolari

La libreria POSIX offre anche altre funzionalità che possono essere importanti. Per esempio definisce delle funzioni per il trattamento delle espressioni regolari e per la gestione dei thread multipli.

## 68.2 Espressioni regolari POSIX

«

Un'espressione regolare è un modello che descrive la corrispondenza con una porzione di una stringa. Le espressioni regolari sono costruite, in maniera analoga alle espressioni matematiche, combinando espressioni più brevi. Lo standard POSIX distingue due tipi di espressioni regolari: quelle elementari, o BRE, e quelle estese, o ERE (si vedano in particolare le sezioni dedicate alle espressioni regolari in generale: [23.1](#) e [23.2](#)). Lo standard POSIX prevede una libreria specifica per la gestione delle espressioni regolari, a cui si fa riferimento in questo capitolo.

Va osservato che nel caso del compilatore GCC, con le librerie GNU, per la compilazione sono sufficienti le librerie principali, le quali vengono incluse automaticamente.

Tabella 68.3. Schema sintetico delle espressioni regolari, secondo la definizione POSIX: operatori fondamentali.

Descrizione	BRE POSIX	ERE POSIX
Escape: protezione del carattere successivo o attribuzione a tale carattere di un significato speciale.	\	\
Ancora all'inizio della riga o della stringa.	^	^
Ancora alla fine della riga o della stringa.	\$	\$
Alternativa.		
Raggruppamento.	\ ( \)	( )
Elenco (individua un solo carattere).	[ ]	[ ]

Descrizione	BRE POSIX	ERE POSIX
Riferimento al raggruppamento $n$ esimo.	$\backslash n$	

Tabella 68.4. Schema sintetico delle espressioni regolari, secondo la definizione POSIX: operatori interni alle espressioni tra parentesi quadre.

Descrizione	BRE POSIX	ERE POSIX
Sequenze.	$xy\dots$	$xy\dots$
Intervalli.	$x-y$	$x-y$
Elementi di collazione.	[ . . ]	[ . . ]
Caratteri equivalenti.	[ = = ]	[ = = ]
Classi di caratteri.	[ : : ]	[ : : ]

Tabella 68.5. Schema sintetico delle espressioni regolari, secondo la definizione POSIX: operatori di ripetizione.

Descrizione	BRE POSIX	ERE POSIX
Zero o più ripetizioni del carattere $x$ .	$x^*$	$x^*$
Una o nessuna occorrenza del carattere $x$ .		$x?$
Una o più ripetizioni del carattere $x$ .		$x^+$
Esattamente $n$ volte il carattere $x$ .	$x\{n\}$	$x\{n\}$
Almeno $n$ volte $x$ .	$x\{n,\}$	$x\{n,\}$

Descrizione	BRE POSIX	ERE POSIX
Da <i>n</i> a <i>m</i> volte <i>x</i> .	$x \{ n , m \}$	$x \{ n , m \}$

## 68.2.1 Compilazione, confronto e rilascio della memoria

«

Per eseguire un confronto con un'espressione regolare, attraverso le funzioni definite dallo standard POSIX, è necessario prima tradurre l'espressione regolare in una variabile strutturata di tipo `'regex_t'`. Tale operazione di analisi e traduzione dell'espressione regolare viene definita dallo standard come «compilazione». La compilazione dell'espressione regolare avviene con la funzione `regcomp()`, con la quale, oltre che fornire la stringa contenente l'espressione regolare stessa, si devono specificare delle opzioni sul modo in cui interpretarla o gestirla. Per esempio, in questa fase va stabilito se l'espressione è di tipo BRE o ERE, se conta la differenza tra lettere maiuscole e minuscole, se si intendono estrapolare delle sottostringhe attraverso la comparazione, se il codice di interruzione di riga ha un qualche valore particolare o meno.

Quando si dispone di un'espressione regolare compilata, si può passare alla comparazione di questa con una stringa, attraverso la funzione `regexec()`, con la quale si possono dare delle opzioni aggiuntive, nel modo finale di effettuare il confronto. Per la comparazione può essere necessaria la definizione di una variabile strutturata, di tipo `'regmatch_t'`.

Quando la variabile strutturata contenente l'espressione regolare non serve più, va rilasciata espressamente la memoria a cui gli elementi della stessa fanno riferimento. In altri termini, non basta liberare

la memoria della variabile che rappresenta la struttura, perché rimarrebbero allocati altri dati raggiunti attraverso dei puntatori. Per liberare un'espressione regolare compilata si utilizza la funzione *regfree()*.

Le funzioni *regcomp()* e *regexexec()*, hanno la caratteristica di restituire un valore intero, pari a zero se l'operazione è stata conclusa con successo, oppure un valore diverso se si è verificato qualche tipo di problema. I valori restituiti, se diversi da zero, sono codificati ordinatamente da macro-variabili simboliche appropriate. Eventualmente, con la funzione *regerror()*, è possibile ottenere la traduzione dell'errore, associato al riferimento all'espressione regolare compilata, in una stringa più esplicita.

Per utilizzare le espressioni regolari POSIX è necessario includere inizialmente il file di intestazione `'regex.h'`.

## 68.2.2 Compilazione dell'espressione regolare

Un'espressione regolare, in forma di stringa, viene compilata attraverso la funzione *regcomp()*, inserendo i dati necessari in una struttura di tipo `'regex_t'`. «

```
int regcomp (regex_t *restrict re, const char *restrict regex,
             int cflags);
```

Il prototipo della funzione mostra che il primo parametro, *re*, deve essere un puntatore al tipo `'regex_t'`: si tratta della struttura che viene modificata attraverso la compilazione. Il secondo parametro, *regex*, è la stringa che descrive l'espressione regolare (la stringa deve essere terminata regolarmente con un carattere nullo, come di

consueto). L'ultimo parametro, **'cflags'**, è un numero intero i cui bit descrivono le opzioni da considerare per l'interpretazione corretta della stringa dell'espressione regolare; tali bit vengono composti assieme attraverso l'uso di macro-variabili simboliche che fanno parte della stessa libreria della funzione *regcomp()*.

Tabella 68.6. Il tipo **'regex\_t'** definisce una variabile strutturata che contiene almeno il membro **'re\_sub'**.

Tipo	Nome	Descrizione
size_t	re_sub	Quantità di sottoespressioni tra parentesi tonde.

Va osservato che l'espressione regolare viene fornita attraverso una stringa «normale», ovvero un array di **'char'**.

L'esempio seguente dovrebbe servire a comprendere l'uso della funzione *regcomp()*:

```
#include <regex.h>
...
    regex_t re;
    regcomp (&re, "01[a-z]*", REG_EXTENDED|REG_NOSUB);
...
```

Come si può osservare, viene dichiarata una variabile di tipo **'regex\_t'**, della quale viene fornito il puntatore nella chiamata di *regcomp()*; inoltre, l'ultimo argomento della funzione è composto utilizzando due macro-variabili simboliche, sommate assieme con l'operatore **'|'**, ovvero un OR bit per bit.



Tabella 68.8. Macro-variabili simboliche da usare come opzioni per la compilazione di un'espressione regolare.

Macro-variabile	Significato
REG_EXTENDED	L'espressione regolare fornita è di tipo ERE (estesa). Se non si usa questa opzione, si intende che l'espressione sia di tipo BRE.
REG_ICASE	L'espressione regolare fornita va valutata senza distinguere tra lettere maiuscole o minuscole.
REG_NOSUB	Per la compilazione dell'espressione regolare non si intende tenere conto della corrispondenza eventuale di sottostringhe; in altri termini, non si vogliono considerare le parentesi '\(' e '\)', oppure '(' e ')' (a seconda che si tratti di ERE o BRE). In tal caso, l'espressione regolare serve per il confronto, ma non per estrapolare porzioni del risultato ottenuto.
REG_NEWLINE	In condizioni normali, il codice <i>new-line</i> contenuto nella stringa con cui l'espressione regolare deve essere confrontata, viene trattato come gli altri caratteri. Con l'opzione ' <b>REG_NEWLINE</b> ', invece, l'operatore '^' individua l'inizio di un testo che segue un codice <i>new-line</i> , mentre l'operatore '\$' individua la fine di un testo che precede un codice <i>new-line</i> .

La funzione *regcomp()* restituisce zero se il procedimento di compilazione dell'espressione regolare termina regolarmente, senza problemi nell'interpretazione della stringa che la rappresenta; altrimenti

restituisce un valore diverso che rappresenta un errore. Per poter valutare l'errore, occorre fare un confronto con delle macro-variabili simboliche, come descritto nella tabella successiva.

Tabella 68.9. Macro-variabili simboliche che rappresentano il tipo di errore restituito dalla funzione *regcomp()*.

Macro-variabile	Significato
REG_BADBR	Il contenuto di ‘\{...\}’ (nel caso di BRE) o di ‘{...}’ (nel caso di ERE), risulta non valido: potrebbe non trattarsi di un numero, oppure potrebbe esserci un numero troppo grande, oppure potrebbero esserci più di due numeri, oppure il primo potrebbe essere più grande del secondo. Infatti, il contenuto di tale raggruppamento deve essere un numero singolo, oppure due numeri separati da una virgola, dove il primo deve essere inferiore al secondo.
REG_BADPAT	Espressione regolare non valida (errore di sintassi).
REG_BADRPT	Un operatore di ripetizione, del tipo ‘?’, ‘*’ o ‘+’, non è preceduto a un'espressione regolare, ovvero si trova in una posizione sbagliata.
REG_ECOLLATE	Elemento di collazione ( <i>collating element</i> ) non valido, nell'ambito della configurazione locale attuale.
REG_ECTYPE	Riferimento a una classe di caratteri non valida.
REG_EESCAPE	L'espressione regolare termina con ‘\’ e ciò non è ammissibile.
REG_ESUBREG	Una sequenza ‘\n’, dove <i>n</i> è un numero, è errata.

Macro-variabile	Significato
REG_EBRACE	Le parentesi graffe che descrivono la ripetizione di qualcosa non bilanciano. Può trattarsi di sequenze del tipo ‘\{...\}’ per le espressioni BRE o del tipo ‘{...}’ per le espressioni ERE.
REG_EBRACK	Parentesi quadre non bilanciate (parentesi aperta e non chiusa, o viceversa).
REG_EPAREN	Le parentesi tonde che descrivono delle sottoespressioni non bilanciano. Può trattarsi di sequenze del tipo ‘\(...\)’ per le espressioni BRE o del tipo ‘(...)’ per le espressioni ERE.
REG_ERANGE	Un’estremità di un intervallo di valori non è valido.
REG_ESPACE	Il procedimento di interpretazione dell’espressione regolare porta all’esaurimento della memoria disponibile.

### 68.2.3 Liberazione della memoria

Un’espressione regolare compilata occupa memoria, non solo nella variabile strutturata che la rappresenta, ma anche in altre aree a cui il contenuto di tale variabile può puntare. Quando l’espressione regolare non serve più, la memoria relativa va liberata esplicitamente, attraverso la funzione *regfree()*, la quale non restituisce alcunché e richiede di indicare solo il puntatore alla variabile strutturata che rappresenta l’espressione regolare stessa.

```
void regfree (regex_t *re);
```

## 68.2.4 Confronto

&lt;&lt;

La comparazione di un'espressione regolare compilata e di una stringa, si svolge con la funzione *regexexec()*. Questa funzione richiede diversi argomenti, perché ci deve essere la possibilità di estrapolare anche delle sottostringhe, corrispondenti a delle sottoespressioni racchiuse tra parentesi tonde.

```
int regexexec (const regex_t *restrict re,
               const char *restrict stringa,
               size_t n_match, regmatch_t p_match[restrict],
               int eflags);
```

Il prototipo della funzione *regexexec()* può apparire inizialmente complicato da interpretare. I primi due parametri sono sostanzialmente l'espressione regolare compilata e la stringa da confrontare. Il terzo parametro rappresenta la quantità di elementi dell'array che viene fornito come quarto parametro (di tipo '*regmatch\_t*'). L'ultimo parametro rappresenta delle opzioni da applicare in fase di comparazione.

Per comprendere l'utilizzo della funzione, inizialmente conviene lasciare da parte i parametri *n\_match* e *pmatch*. Così facendo è possibile verificare se l'espressione regolare trova una corrispondenza nella stringa fornita. L'esempio seguente (che dovrebbe essere disponibile presso [allegati/c/esempio-posix-regex-1.c](#)) mostra la dichiarazione di una funzione che svolge tutti i passaggi, dalla compilazione alla liberazione della memoria.

Listato 68.10. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/t4ZYxS5l>, <http://ideone.com/oN111>.

```
#include <stdio.h>
#include <regex.h>

int
regex_match (char *pattern, const char *string)
{
    int      status;
    regex_t  re;
    status = regcomp (&re, pattern, REG_EXTENDED|REG_NOSUB);
    if (status != 0)
        {
            return (0);
        }
    status = regexec (&re, string, (size_t) 0, NULL, 0);
    if (status != 0)
        {
            regfree (&re);
            return (0);
        }
    regfree (&re);
    return (1);
}

int
main (void)
{
    char *string      = "Ciao amore mio";
    char *re          = "iao";

    if (regex_match (re, string))
        {
```

```

        printf ("Il modello \"%s\" trova corrispondenza ",
                re);
        printf ("nella stringa \"%s\"\n", string);
    }
else
    {
        printf ("Il modello \"%s\" ", re);
        printf ("NON trova corrispondenza ");
        printf ("nella stringa \"%s\"\n", string);
    }
return 0;
}

```

In pratica, la funzione *regexexec()* viene usata semplicemente fornendo il puntatore alla variabile strutturata contenente l'espressione regolare compilata e la stringa da confrontare:

```
regexexec (&re, string, (size_t) 0, NULL, 0);
```

Le opzioni che possono essere indicate alla funzione *regexexec()*, come ultimo argomento, sono solo due e riguardano la facoltà di considerare l'inizio o la fine della stringa come l'inizio o la fine di una riga.

Tabella 68.12. Macro-variabili simboliche da usare come opzioni per la comparazione di una stringa con un'espressione regolare.

Macro-variabile	Significato
REG_NOTBOL	In condizioni normali, il carattere '^' trova corrispondenza con l'inizio di una stringa. Con questa opzione, si inibisce tale corrispondenza ( <i>not begin of line</i> ).

Macro-variabile	Significato
REG_NOTEOL	In condizioni normali, il carattere '\$' trova corrispondenza con la fine di una stringa. Con questa opzione, si inibisce tale corrispondenza ( <i>not end of line</i> ).

Il valore restituito dalla funzione *regexexec()* è zero se il confronto avviene con successo, diversamente si ha un valore diverso da zero, per indicare la mancanza di corrispondenza o l'utilizzo eccessivo di memoria. Va osservato che la macro-variabile **REG\_ESPACE** è la stessa già vista per la funzione *regcomp()* e che **REG\_NOTBOL** rappresenta, opportunamente, un valore differente da tutte le altre macro-variabili **REG\_...**

Tabella 68.13. Macro-variabili simboliche che rappresentano il tipo di errore (o di insuccesso) restituito dalla funzione *regexexec()*.

Macro-variabile	Significato
REG_ESPACE	Il procedimento di confronto porta all'esaurimento della memoria disponibile.
REG_NOMATCH	Non c'è corrispondenza tra espressione regolare e stringa.

### 68.2.5 Estrapolazione di sottostringhe

Quando un'espressione regolare contiene una porzione del proprio codice racchiuso tra '\(' e '\)', nel caso di espressioni BRE, oppure tra '(' e ')', nel caso di espressioni ERE, è possibile estrapolare la porzione di stringa che corrisponde a tale sottoespressione. Per fare

questo si usa un array di variabili strutturate di tipo `'regmatch_t'` che viene fornito come argomento della chiamata di `regexexec()`.

Della variabile strutturata di tipo `'regmatch_t'` si sa solo che contiene almeno due campi, denominati `'rm_so'` e `'rm_eo'` (*regular expression match: start offset e end offset*). I due campi in questione, sono, a loro volta, di tipo `'regoff_t'`, corrispondente a un valore intero con segno, di rango appropriato.

Tabella 68.14. Organizzazione di una variabile strutturata di tipo `'regmatch_t'`.

Tipo	Nome del membro	Descrizione
<code>regoff_t</code>	<code>rm_so</code>	Scostamento in byte, dall'inizio della stringa, corrispondente all'inizio della sottostringa individuata.
<code>regoff_t</code>	<code>rm_eo</code>	Scostamento in byte, dall'inizio della stringa, corrispondente al carattere <b>successivo</b> alla sottostringa individuata.

La funzione `regexexec()` popola il contenuto dell'array di elementi `'regmatch_t'`, utilizzando il primo elemento (indice 0) per individuare la sottostringa corrispondente all'espressione regolare nel suo complesso, mentre gli elementi successivi riguardano le sottoespressioni eventuali. Pertanto, le sottoespressioni si trovano a partire dall'indice 1 di tale array.



Perché la funzione *regexexec()* possa estrapolare delle sottostringhe a partire da sottoespressioni, è necessario che l'espressione regolare sia stata compilata senza l'opzione 'REG\_NOSUB'. Infatti, tale opzione viene usata per risparmiare risorse quando si sa che non ci si intende avvalere di tale possibilità.

L'esempio seguente mostra un piccolo programma, completo, in cui la funzione *regex\_match()* si occupa di verificare la corrispondenza con un'espressione regolare e, se c'è corrispondenza, compila anche un array di stringhe con le sottostringhe estratte. Naturalmente, tale array di stringhe deve essere già stato predisposto prima della chiamata della funzione e deve avere una dimensione adeguata a contenere sia la corrispondenza con l'espressione regolare nel suo complesso, sia la corrispondenza con le altre sottoespressioni eventuali. Il file dovrebbe essere disponibile presso [allegati/c/esempio-posix-regex-2.c](#).

Listato 68.15. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/ymVOGkON>, <http://ideone.com/U9Tka>.

```
#include <stdio.h>
#include <regex.h>

int
regex_match (char *pattern, const char *string,
             size_t sub_size, char **sub)
{
    regmatch_t match[sub_size];
    size_t      n;
    size_t      m;
    size_t      d;
```

```
int      status;
regex_t  re;
status = regcomp (&re, pattern, REG_EXTENDED);
if (status != 0)
    {
        return (status);
    }
status = regexec (&re, string, sub_size, match, 0);
if (status == 0)
    {
        for (n=0; n < sub_size; n++)
            {
                for (d = 0, m = match[n].rm_so;
                    m >= 0 && m < match[n].rm_eo;
                    m++, d++)
                    {
                        sub[n][d] = string[m];
                    }
                sub[n][d] = '\0';
            }
        regfree (&re);
        return (status);
    }

int
main (void)
{
    int  result;
    char *string      = "Ciao amore mio";
    char *re          = "Ciao (amo)re";
    char  sub0[200];
        sub0[0] = '\0';
    char  sub1[200];
```

```
        sub1[0] = '\\0';
char *sub[] = {sub0, sub1};

result = regex_match (re, string, 2, sub);

if (result == 0)
    {
        printf ("Il modello \"%s\" trova corrispondenza ",
                re);
        printf ("nella stringa \"%s\", precisamente ",
                string);
        printf ("nella porzione \"%s\", mentre la ",
                sub[0]);
        printf ("sottostringa estratta è \"%s\".\n",
                sub[1]);
    }
else
    {
        printf ("Il modello \"%s\" ", re);
        printf ("NON trova corrispondenza ");
        printf ("nella stringa \"%s\".\n", string);
    }
return 0;
}
```

**Compilando il programma ed eseguendolo con i dati che si vedono, si ottiene la visualizzazione del testo seguente:**

Il modello "Ciao (amo)re" trova corrispondenza nella stringa "Ciao amore mio", precisamente nella porzione "Ciao amore", mentre la sottostringa estratta è "amo".

## 68.2.6 Informazioni diagnostiche

&lt;&lt;

Le funzioni *regcomp()* e *regexexec()* restituiscono un valore intero che rappresenta l'esito dell'operazione svolta: se è zero l'operazione ha avuto successo, altrimenti c'è un qualche tipo di problema che può essere individuato confrontando tale valore con una serie di macro-variabili prestabilite. Tuttavia, si può ottenere un risultato tradotto in un testo più comprensibile attraverso la funzione *regerror()*, la quale richiede l'indicazione del numero dell'errore, del puntatore alla variabile strutturata che rappresenta l'espressione regolare a cui si riferisce il problema e le informazioni necessarie a compilare correttamente una stringa con il messaggio appropriato.

```
size_t regerror (int errore, const regex_t *restrict re,  
                char *restrict testo, size_t dimensione);
```

In pratica, il primo parametro è il numero dell'errore o comunque dell'esito dell'operazione svolta, come restituito dalle funzioni *regcomp()* e *regexexec()*; il secondo parametro è il puntatore alla variabile strutturata che rappresenta l'espressione regolare a cui si riferisce l'esito in questione; il terzo parametro è un array di tipo '**char**', in cui la funzione deve poter scrivere il testo della spiegazione; l'ultimo parametro è la dimensione massima di tale array (oltre la quale la funzione non deve scrivere).

Il valore restituito dalla funzione *regerror()* è la dimensione utilizzata effettivamente nell'array per scrivere il testo dell'esito (inclusa la terminazione con il byte a zero).

L'esempio seguente mostra un piccolo programma, completo, ottenuto dalla modifica di quello apparso nella sezione precedente, dove

in presenza di un esito non soddisfacente per le funzioni *regcomp()* e *regexexec()* viene visualizzato un messaggio esplicito del problema verificatosi. Il programma richiede volutamente un confronto non corretto per produrre un errore. Il file dovrebbe essere disponibile presso [allegati/c/esempio-posix-regex-3.c](#).

Listato 68.17. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/onqxRRyX>, <http://ideone.com/Q0w9k>.

```
#include <stdio.h>
#include <regex.h>

int
regex_match (char *pattern, const char *string,
             size_t sub_size, char **sub)
{
    const int msg_size = 200;
    char      msg[msg_size];
             msg[0] = '\0';

    //
    regmatch_t match[sub_size];
    size_t      n;
    size_t      m;
    size_t      d;
    int         status;
    regex_t     re;
    status = regcomp (&re, pattern, REG_EXTENDED);
    if (status != 0)
        {
            regerror (status, &re, msg, msg_size);
            fprintf (stderr, "%s\n", msg);
            return (status);
        }
    status = regexexec (&re, string, sub_size, match, 0);
```

```
if (status != 0)
{
    regerror (status, &re, msg, msg_size);
    fprintf (stderr, "%s\n", msg);
}
else
{
    for (n=0; n < sub_size; n++)
    {
        for (d = 0, m = match[n].rm_so;
             m >= 0 && m < match[n].rm_eo;
             m++, d++)
        {
            sub[n][d] = string[m];
        }
        sub[n][d] = '\0';
    }
}
regfree (&re);
return (status);
}

int
main (void)
{
    int    result;
    char *string      = "Ciao amore mio";
    char *re          = "*Ciao (amo)re";
    char  sub0[200];
            sub0[0] = '\0';
    char  sub1[200];
            sub1[0] = '\0';
    char *sub[] = {sub0, sub1};
}
```

```
result = regex_match (re, string, 2, sub);

if (result == 0)
{
    printf ("Il modello \"%s\" trova corrispondenza ",
           re);
    printf ("nella stringa \"%s\", precisamente ",
           string);
    printf ("nella porzione \"%s\", mentre la ",
           sub[0]);
    printf ("sottostringa estratta è \"%s\".\n",
           sub[1]);
}
else
{
    printf ("Il modello \"%s\" ", re);
    printf ("NON trova corrispondenza ");
    printf ("nella stringa \"%s\"\n", string);
}
return 0;
}
```

L'espressione regolare contiene un errore che consiste nell'uso dell'asterisco all'inizio della stessa. Provando a eseguire il programma si dovrebbe visualizzare il testo seguente:

#### **Invalid preceding regular expression**

```
Il modello "*Ciao (amo)re" NON trova corrispondenza nella stringa
"Ciao amore mio"
```

## 68.2.7 Esempio completo



Viene proposta una funzione per semplificare il confronto di una stringa con un'espressione regolare, più completa rispetto a quanto già mostrato negli esempi delle sezioni precedenti. A sua volta la funzione viene mostrata in un programma di prova completo.

```
int regex_match (char *restrict pattern ,
                 const char *restrict string ,
                 size_t sub_size ,
                 char *restrict sub [restrict] ,
                 int cflags , int eflags , int verbose) ;
```

Il primo parametro è la stringa che contiene l'espressione regolare; il secondo parametro è la stringa da confrontare con l'espressione; il terzo parametro è la dimensione massima dell'array che costituisce il quarto parametro; il quarto parametro è un array di puntatori a stringhe, di cui però non si conosce l'ampiezza massima; il quinto parametro è un intero che rappresenta le opzioni da usare con la funzione *regcomp()*; il sesto parametro è un intero che rappresenta le opzioni da usare con la funzione *regexexec()*; l'ultimo parametro, se diverso da zero, richiede la visualizzazione dei messaggi di errore attraverso lo standard error.

La funzione restituisce un valore pari a zero se tutto il procedimento si completa con successo; altrimenti restituisce l'esito prodotto dalla funzione *regcomp()* o da *regexexec()*.

Si osservi che all'inizio del programma è possibile definire la macrovariabile '**restrict**' come commento. Ciò è necessario se il compilatore non riconosce ancora tale parola chiave nella definizione dei



parametri che sono puntatori. Infatti, si tratta di una caratteristica utile solo nei compilatori ottimizzati, in grado di gestire l'elaborazione degli array in modo diverso da quello tradizionale. Il file dovrebbe essere disponibile presso [allegati/c/esempio-posix-regex-ok.c](#).

Listato 68.19. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/c06Z8k6Z>, <http://ideone.com/J008h>.

```
#include <stdio.h>
#include <regex.h>
//
// #define restrict /**/
//
//
int
regex_match (char *restrict pattern,
             const char *restrict string,
             size_t sub_size, char *restrict sub[restrict],
             int cflags, int eflags, int verbose)
{
    const int  msg_size = 200;
    char      msg[msg_size];
    //
    regmatch_t pmatch[sub_size];
    size_t     n;
    size_t     m;
    size_t     d;
    //
    int        status;
    regex_t    re;
    //
    status = regcomp (&re, pattern, cflags);
    //
    if (status != 0)
```

```
{
    if (verbose)
    {
        regerror (status, &re, msg, msg_size);
        fprintf (stderr, "%s\n", msg);
    }
    return (status);
}
//
status = regexec (&re, string, sub_size, pmatch,
                 eflags);
//
if (status != 0)
{
    if (verbose)
    {
        regerror (status, &re, msg, msg_size);
        fprintf (stderr, "%s\n", msg);
    }
}
else
{
    for (n=0; n < sub_size; n++)
    {
        for (d = 0, m = pmatch[n].rm_so;
             m >= 0 && m < pmatch[n].rm_eo;
             m++, d++)
        {
            sub[n][d] = string[m];
        }
        sub[n][d] = '\0';
    }
}
//
```

```
    regfree (&re);
    //
    return (status);
}
//
//
//
int
main (void)
{
    int    result;
    char *string      = "Ciao amore mio";
    char *re          = "Ciao (amo)re";
    char  sub0[200];
        sub0[0] = '\0';
    char  sub1[200];
        sub1[0] = '\0';
    char *sub[] = {sub0, sub1};
    //
    result = regex_match (re, string, 2, sub, REG_EXTENDED,
                          0, 1);
    //
    if (result == 0)
        {
            printf ("Il modello \"%s\" trova corrispondenza ",
                    re);
            printf ("nella stringa \"%s\", precisamente ",
                    string);
            printf ("nella porzione \"%s\", mentre la ",
                    sub[0]);
            printf ("sottostringa estratta è \"%s\".\n",
                    sub[1]);
        }
    else
```

```
    {
        printf ("Il modello \"%s\" ", re);
        printf ("NON trova corrispondenza ");
        printf ("nella stringa \"%s\"\n", string);
    }
    return 0;
}
```

## 68.3 Avvio e conclusione dei processi

«

In un sistema Unix, l'avvio di un processo si ottiene attraverso l'uso di due chiamate di sistema: una produce una copia del processo esistente, con la possibilità di distinguere poi tra chi è il genitore e chi è invece il figlio; l'altra carica un processo e lo mette in funzione al posto di quello da cui ha avuto origine. Inizialmente, la questione può sembrare complicata o almeno strana. Se però si ha la possibilità di approfondire il funzionamento basilare di un sistema Unix tradizionale, si scopre che è tutto perfettamente logico e lineare, ovvero, che si tratta della scelta progettuale più semplice che si potesse attuare.

Attorno a questi concetti ci sono poi altre questioni legate ai processi, che è bene introdurre assieme al resto, per avere una visione iniziale relativamente completa.

### 68.3.1 Biforcazione: «fork»

«

Attraverso la funzione *fork()*, definita nel file di intestazione `unistd.h`, si ottiene la duplicazione del processo elaborativo corrente, associandogli un numero PID differente, diventando questo figlio del processo da cui la chiamata ha avuto origine. Ciò che va

chiarito è che il processo ottenuto dalla duplicazione continua a funzionare dal punto in cui si trovava il processo originario, pertanto è dal valore restituito dalla funzione *fork()* che si riesce a capire se ci si trova a funzionare come genitore o figlio di quel contesto particolare.

Listato 68.20. Per provare il codice attraverso un servizio *pastebin*: <http://ideone.com/XpVBYY>.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
//-----
int
main (void)
{
    pid_t  pid;
    //
    printf ("Io sono il genitore e il mio numero PID "
           "è %i.\n",
           (int) getpid ());
    pid = fork ();
    if (pid == 0)
    {
        printf ("Io sono il figlio della biforcazione "
               "e il mio genitore ha il numero "
               "PID %i.\n", (int) getppid ());
        exit (0);
    }
    else
    {
        printf ("Ho avviato una biforcazione di me "
               "stesso, la quale ha ottenuto il "
               "numero PID %i.\n", pid);
    }
}
```

```
    return (0);  
}
```

Il listato mostra un esempio completo di programma che avvia una biforcazione di se stesso. La funzione *fork()* viene usata senza argomenti e restituisce un numero PID: se questo è diverso da zero, significa che si tratta dell'esecuzione del processo genitore; se invece è zero, è in corso l'esecuzione del processo duplicato. Nell'esempio si fa uso delle funzioni *getpid()* e di *getppid()*, per ottenere, rispettivamente il numero del processo in corso e quello del processo genitore.

Come si può osservare, nell'esempio il processo figlio ha vita breve, perché si limita a dichiarare la propria esistenza, quindi chiama la funzione *exit()* per concludere esplicitamente la propria attività.

Il risultato dell'esecuzione di questo programma potrebbe essere costituito dai messaggi seguenti:

```
Io sono il genitore e il mio numero PID è 5531.
```

```
Io sono il figlio della biforcazione e il mio genitore ha il ↵  
↪numero PID 5531.
```

```
Ho avviato una biforcazione di me stesso, la quale ha ↵  
↪ottenuto il numero PID 5532.
```

Il processo ottenuto dalla biforcazione è sostanzialmente uguale a quello del genitore (a parte la distinzione del numero PID e della gerarchia genitore-figlio); tuttavia, le differenze emergono in base al livello di complessità del programma in questione. Pertanto, è sempre bene accertarsi nel dettaglio di cosa erediti il processo figlio, dalla pagina di manuale *fork(2)*, quando si vuole usare questa funzione.

Le cose essenziali da sapere riguardano principalmente i file aperti

e i *thread*, ovvero i flussi elaborativi. La biforcazione produce la duplicazione dei file aperti nel nuovo processo, condividendo però l'indice che rappresenta la posizione corrente. In tal modo, le operazioni di lettura e scrittura sui file possono svolgersi in modo coordinato: uno legge fino a un certo punto, l'altro legge da lì fino a un'altra posizione, e lo stesso vale per la scrittura. La biforcazione produce un solo *thread* nel processo figlio, costituito precisamente da quello che ha chiamato la funzione *fork()*.

### 68.3.2 Sostituzione: «exec»

Un gruppo di funzioni, contraddistinte dal prefisso '**exec**', consente di rimpiazzare il processo corrente con un altro, caricando un programma. Rimpiazzare il processo corrente significa che questo si conclude e, da quel punto, dovrebbe iniziare a funzionare un altro programma dall'inizio.

In condizioni normali, un processo che voglia avviare un programma, esegue prima una biforcazione, quindi, nel codice che riguarda il processo figlio esegue una funzione *exec...()*, con cui quel figlio viene rimpiazzato con il nuovo programma.

Nell'esempio successivo viene mostrato l'uso della funzione *execl()*, con la quale si indica il percorso del programma da avviare, seguito dagli argomenti da dare a questo, tenendo conto che il primo deve corrispondere al nome del programma stesso e che l'ultimo deve essere un puntatore a carattere nullo:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
//-----
int
main (void)
{
    pid_t  pid;
    int    status;
    //
    pid = fork ();
    if (pid == 0)
        {
            status = execl ("../prog", "prog", (char *) NULL);
            perror (NULL);
            exit (0);
        }
    return (0);
}
```

Se il programma **'prog'** viene avviato correttamente dalla directory corrente, come indicato nel percorso di avvio, la funzione *execl()* «non ritorna», nel senso che il processo che la avvia scompare. Se invece si verifica un errore, la funzione restituisce il valore  $-1$  e l'esecuzione del processo originario prosegue. In questo caso, si usa la funzione *perror()* per visualizzare l'errore annotato nella variabile *errno*, quindi la funzione *exit()* conclude comunque il funzionamento del processo.



### 68.3.3 Attesa della conclusione di un processo figlio

Quando un processo esegue una biforcazione, dalla quale poi si può passare all'esecuzione di un altro programma o meno, ci può essere la necessità di attendere che il processo figlio termini il suo funzionamento. Per fare questo si usa normalmente la funzione *wait()*, oppure *waitpid()* con argomenti appropriati.

Listato 68.23. Per provare il codice attraverso un servizio *pastebin*: <http://ideone.com/vD2hs> .

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
//-----
int
main (void)
{
    pid_t  pid_child;
    pid_t  pid_dead;
    int    status;
    //
    pid_child = fork ();
    if (pid_child == 0)
    {
        sleep (1);
        printf ("ciao!\n");
        exit (7);
    }
    printf ("Ho avviato il processo %i.\n", pid_child);
    //
    pid_dead = wait (&status);
    //
    printf ("Il processo %i si è concluso restituendo "
```

```
        "il valore %x.\n",  
        pid_dead, WEXITSTATUS (status));  
    //  
    return (0);  
}
```

L'esempio mostra l'avvio di un processo figlio, in cui, dopo una pausa di un secondo si visualizza un messaggio e quindi quel processo termina restituendo il valore 7. Il processo genitore mostra subito un messaggio in cui dichiara il numero PID del figlio, quindi si mette in attesa della sua conclusione. Il valore restituito dal processo figlio confluisce nella variabile *status*, ma deve essere interpretato attraverso la macroistruzione **WEXITSTATUS()**. Il risultato prodotto a video dal programma di esempio mostrato è molto simile al testo seguente:

```
Ho avviato il processo 6138.
```

```
ciao!
```

```
Il processo 6138 si è concluso restituendo il valore 7.
```

### 68.3.4 Adozione dei processi

«

I sistemi Unix e di conseguenza lo standard POSIX, seguono una convenzione nella numerazione dei processi: il kernel è il processo zero ed è implicito; il processo numero uno è **'init'** (o altro in situazioni particolari) e ha il compito di essere quello che genera tutti gli altri.

Quando un processo termina di funzionare, i suoi processi figli vengono affidati a **'init'** (o comunque a quel processo che si trova ad avere il numero uno).

Listato 68.25. Per provare il codice attraverso un servizio *pastebin*: <http://ideone.com/Wq5je>.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
//-----
int
main (void)
{
    pid_t  pid;
    //
    pid = fork ();
    if (pid == 0)
    {
        printf ("Io sono il figlio della biforcazione e "
                "il mio genitore ha il numero "
                "PID %i.\n", (int) getppid ());
        sleep (2);
        printf ("Sono passati due secondi e il mio "
                "genitore ha il numero PID %i.\n",
                (int) getppid ());
        exit (0);
    }
    else
    {
        printf ("Io sono il processo %i e ho avviato una "
                "biforcazione di me stesso, la quale ha "
                "ottenuto il numero PID %i, ma adesso "
                "concludo il mio funzionamento.\n",
                (int) getpid (), pid);
    }
    return (0);
}
```

L'esempio appena mostrato dovrebbe chiarire questo fatto: quando il processo figlio ha superato l'attesa di due secondi, il suo genitore ha già smesso di funzionare, e in quel momento è già stato adottato dal processo numero uno. I messaggi prodotti dal programma sono come quelli seguenti:

```

Io sono il processo 6602 e ho avviato una biforcazione ↵
↳di me stesso, la quale ha ottenuto il numero ↵
↳PID 6603, ma adesso concludo il mio funzionamento.
Io sono il figlio della biforcazione e il mio genitore ↵
↳ha il numero PID 6602.
Sono passati due secondi e il mio genitore ha il ↵
↳numero PID 1.

```

### 68.3.5 Gli zombie

«

Quando un processo elaborativo conclude il suo funzionamento, per qualunque motivo e in qualunque modo sia, si presume che debba restituire un valore al proprio genitore. Come descritto in precedenza, la funzione *wait()* consente a un genitore di recepire la conclusione di un suo processo figlio, ottenendo anche il valore restituito. Tuttavia, non è detto che un genitore sia sempre lì pronto a recepire la conclusione di un proprio figlio, pertanto, i processi conclusi continuano a rimanere annotati nel sistema, fino a quando le loro informazioni devono rimanere disponibili. Un processo concluso, ma in attesa di essere eliminato, è noto come «zombie».

La conclusione di un processo produce automaticamente l'invio di un segnale '**SIGCHLD**' al genitore. Questo segnale, in particolare, se non viene intercettato, produce l'eliminazione dei processi figli defunti. Tuttavia potrebbe essere utilizzato da un genitore per intervenire contestualmente e recepire la conclusione di un processo figlio,

senza rimanere in attesa con la funzione *wait()* per questo, come nell'esempio seguente:

Listato 68.27. Per provare il codice attraverso un servizio *pastebin*: <http://ideone.com/JYqHo> .

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>
//-----
void signal_handler (int signal)
{
    int    status;
    pid_t  pid;
    if (signal == SIGCHLD)
        {
            pid = wait (&status);
            printf ("Il processo %i si è concluso restituendo "
                   "il valore %x.\n",
                   pid, WEXITSTATUS (status));
        }
}
//-----
int
main (void)
{
    pid_t  pid;
    //
    signal (SIGCHLD, signal_handler);
    //
    pid = fork ();
    if (pid == 0)
        {
            sleep (1);
```

```
        printf ("ciao!\n");
        exit (7);
    }
    printf ("Ho avviato il processo %i.\n", pid);
    //
    sleep (60);
    //
    return (0);
}
```

Come già chiarito, quando un processo muore, i suoi figli vengono adottati automaticamente da `'init'`, o comunque dal processo numero uno. In questa circostanza, però, `'init'` riceve anche il segnale `'SIGCHLD'`, perché i processi adottati potrebbero trovarsi già nello stato di «zombie», ovvero in attesa di essere considerati per poter morire definitivamente.

### 68.3.6 Variabili di ambiente

«

Una caratteristica dei programmi di un sistema Unix, e quindi POSIX, è la disponibilità di quelle che sono note come variabili di ambiente. Va osservato che questo concetto non è presente nel linguaggio C puro e semplice; inoltre, per la stessa ragione, il prototipo della funzione `main()` diventa più articolato rispetto a quello di un programma C comune:

```
int main (int argc, char *argv[], char *envp[]);
```

I parametri `argc` e `argv[]` sono gli stessi, già conosciuti nel linguaggio C, con l'accortezza di avere l'elemento `argv[argc]` pari al puntatore nullo (`'NULL'`). Il parametro `envp[]` è inteso come un array di

stringhe, il cui contenuto deve avere la forma *'nome=valore'* e l'ultimo elemento, anche in questo caso, deve essere un puntatore nullo (per poter riconoscere la sua conclusione).

In pratica, l'array *envp[]* diventa il veicolo per le variabili di ambiente da fornire al programma che si vuole avviare.

Ma la questione non si esaurisce così, perché per motivi storici l'array di stringhe che descrivono le variabili di ambiente è accessibile anche attraverso una variabile globale (esterna), denominata *environ*. In tal modo, anche se la funzione *main()* non fosse provvista del parametro *envp[]*, sarebbe comunque possibile accedere alle stringhe delle variabili di ambiente.

```
extern char **environ;
```

Per leggere e modificare ciò che rappresenta le variabili di ambiente, si usano poi delle funzioni apposite, i cui prototipi appaiono nel file di intestazione *'stdlib.h'*. Queste funzioni hanno in comune un nome terminante per «env», come *setenv()*, *unsetenv()* e *putenv()*.

### 68.3.7 Variabili di ambiente e avvio di un programma

Quando si avvia un nuovo programma, attraverso una delle funzioni *exec...()*, questo ottiene un insieme di variabili di ambiente, ereditandole dal processo originario (che viene rimpiazzato), oppure attraverso una dichiarazione esplicita. Ciò dipende da quale funzione *exec...()* viene usata effettivamente. La funzione da cui poi hanno origine le altre della famiglia *exec...()* è *execve()*:



```
int execve (const char *path, char *const argv[],  
           char *const envp[]);
```

Logicamente, il primo parametro (*path*) rappresenta il percorso del programma da avviare, mentre gli altri due corrispondono agli array di stringhe che di norma hanno lo stesso nome nel prototipo della funzione *main()*.

```
#include <unistd.h>  
#include <stdio.h>  
#include <stdlib.h>  
  
//-----  
int  
main (int argc, char *argv[], char *envp[])  
{  
    pid_t pid;  
    char *exec_argv[3];  
    char *exec_envp[3];  
    //  
    exec_envp[0] = "PATH=/bin:/usr/bin:/sbin:/usr/sbin";  
    exec_envp[1] = "CONSOLE=/dev/console";  
    exec_envp[2] = NULL;  
    //  
    exec_argv[0] = "./mio_prog";  
    exec_argv[1] = "-x";  
    exec_argv[2] = NULL;  
    //  
    pid = fork ();  
    if (pid == 0)  
    {  
        execve (exec_argv[0], exec_argv, exec_envp);  
        perror (NULL);  
        exit (1);  
    }  
}
```



```
    return (0);  
}
```

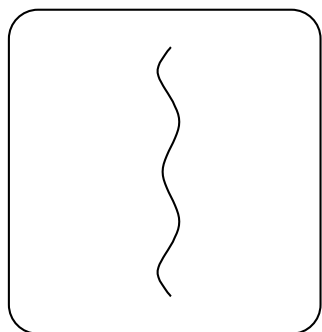
L'esempio mostra la costruzione degli array contenenti le variabili di ambiente e gli argomenti del programma da avviare.

## 68.4 Nozioni sui thread POSIX

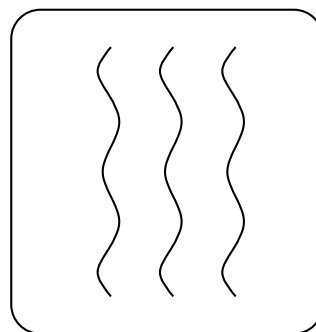
Un programma comune si traduce solitamente in un solo flusso di controllo (o flusso elaborativo), ovvero in un solo *thread*, nel senso che il procedimento esecutivo è unico, dall'avvio alla conclusione del processo. Un programma più sofisticato, potrebbe gestire gli stessi dati attraverso più flussi di controllo concorrenti e in tal caso si dice che questo utilizza più thread. Pertanto, non va confuso il concetto di processo elaborativo con il flusso di controllo o thread, perché i thread di un processo condividono la stessa memoria, mentre i processi elaborativi, tra di loro, hanno aree di memoria indipendenti.

Il termine inglese thread si traduce letteralmente come «filetto», pertanto viene rappresentato frequentemente in questo modo.

processo elaborativo  
a thread singolo



processo elaborativo  
con più thread simultanei



La simultaneità di esecuzione dei thread può essere simulata, attraverso la suddivisione del tempo di CPU, oppure può essere anche reale, quando l'elaboratore dispone di più CPU. Tuttavia anche quando si dispone di una sola CPU, l'organizzazione corretta di un programma in più thread può migliorarne le prestazioni.

Lo standard POSIX definisce alcune funzioni per la gestione dei thread, per le quali è necessario includere il file di intestazione `'pthread.h'` (dove la «p» sta per «POSIX»).

In un sistema GNU, o comunque quando si utilizza il compilatore GCC con la libreria dei sistemi GNU, per l'utilizzo delle funzioni che consentono di gestire i thread POSIX, è necessario includere esplicitamente la libreria `'pthread'`, con l'opzione `'-lpthread'`. In pratica, per compilare gli esempi di questo capitolo si usano comandi del tipo:

```
$ cc -Wall -lpthread -o file_eseguibile file_sorgente_c [Invio]
```

### 68.4.1 Numero identificativo del thread

«

Ogni thread ha un proprio numero identificativo, rappresentato attraverso un tipo di dati apposito, denominato `'pthread_t'`. Quando si crea un thread occorre fare riferimento a una variabile di tipo `'pthread_t'`, in modo tale che questa sia aggiornata con il numero corretto; successivamente, per ricondurre un thread al flusso principale del processo elaborativo, si utilizza nuovamente quel numero per poterlo individuare.

L'esempio seguente crea la variabile scalare ***mio\_thread***, per annotare il numero di un thread:

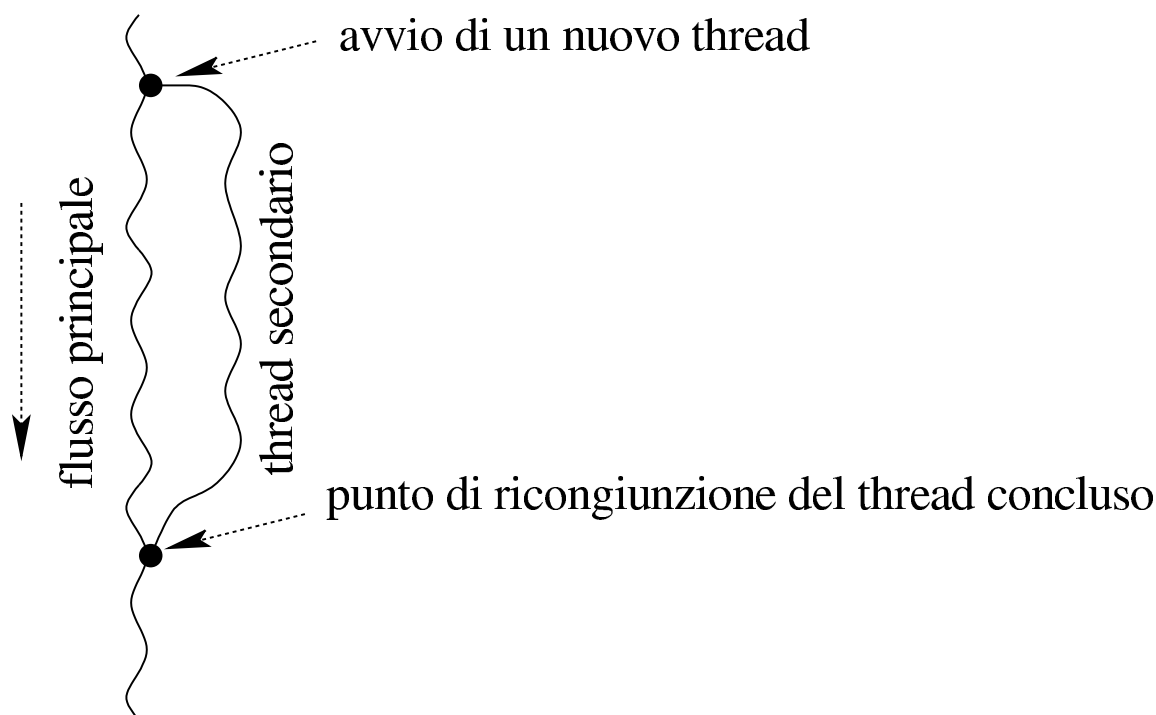
```
pthread_t mio_thread;
```

L'esempio successivo, invece, predispone l'array *miei\_thread[]* per annotare il numero di identificazione di un massimo di cinque thread:

```
pthread_t miei_thread[5];
```

## 68.4.2 Creazione e conclusione di un thread aggiuntivo

Un programma ha sempre almeno un thread, ovvero quello principale, la cui creazione è implicita. Tutti gli altri thread che si vogliono gestire vanno creati appositamente: si tratta di fare in modo che una certa funzione sia eseguita senza attenderne la sua conclusione. Ma a un certo punto del flusso principale del programma, è necessario formalizzare la conclusione dei thread aggiuntivi (e se non sono ancora terminati occorre attendere che lo siano effettivamente).



### 68.4.3 Caratteristiche della funzione che costituisce un thread aggiuntivo



La creazione di un nuovo thread coincide con l'avvio di una funzione senza attendere la sua conclusione. Tale funzione deve però avere una forma precisa: riceve esattamente un argomento, costituito da un puntatore indefinito (`'void *'`), e restituisce un valore, costituito da un puntatore indefinito.

```
void *funzione (void *arg);
```

In pratica, per passare degli argomenti a una funzione di questo tipo, si predispose una struttura con tutto ciò che serve e se ne passa il puntatore; d'altro canto, la funzione deve essere in grado di estrapolare i dati dalla struttura. Come si comprende, tale funzione ha anche difficoltà a restituire un valore, perché può solo produrre un puntatore a qualcosa che deve risultare già definito prima della sua chiamata.

Per comprendere la cosa viene proposto un programma estremamente banale, in cui la funzione *function()* si limita a mostrare ripetutamente un certo carattere, in base ai dati forniti attraverso il riferimento a una struttura.

Listato 68.33. Per provare il codice attraverso un servizio *pastebin*: <http://codepad.org/2ezjFCNy>, <http://ideone.com/xK3xC>.

```
#include <stdio.h>

struct Arguments {
    char x;
    int max;
```

```
};

void *
function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    for (i = 0; i < arg->max; i++)
        {
            printf ("%c", arg->x);
        }
    return NULL;
}

int
main (void)
{
    struct Arguments arg = {'x', 10};
    function (&arg);
    printf ("\n");
    return (0);
}
```

Come si vede, la funzione deve sapere come si articola la struttura, per poter accedere ai dati che questa contiene. Generalmente, come nel caso dell'esempio, in una funzione di questo tipo non si restituisce alcunché.

#### 68.4.4 Avvio di thread separati e fusione successiva

Per comprendere il meccanismo di avvio di un thread separato e della sua fusione successiva, viene proposto un esempio molto semplice, con cui si mostrano solo i passaggi indispensabili. Per la preci- <<

sione, oltre al flusso principale, vengono avviati tre thread ulteriori, attraverso la stessa funzione. Nell'esempio, la funzione usata per avviare i thread, riceve gli argomenti tramite una struttura articolata nello stesso modo già visto nella sezione precedente. Il file dovrebbe essere disponibile presso [allegati/c/esempio-posix-thread-1.c](#) :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

struct Arguments {
    char x;
    int max;
};

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10);

    for (i = 0; i < (arg->max * m); i++)
    {
        if ((i % m) == 0)
        {
            printf ("%c", arg->x);
            fflush (stdout);
        }
    }
    return NULL;
}

int
```

```
main (void)
{
    pthread_t pthread_1;
    pthread_t pthread_2;
    pthread_t pthread_3;

    struct Arguments arg_1 = {'a', 12};
    struct Arguments arg_2 = {'b', 6};
    struct Arguments arg_3 = {'c', 3};

    int status_1;
    int status_2;
    int status_3;

    srand (1234567);

    status_3 = pthread_create (&pthread_3, NULL,
                               pthread_function, &arg_3);
    status_2 = pthread_create (&pthread_2, NULL,
                               pthread_function, &arg_2);
    status_1 = pthread_create (&pthread_1, NULL,
                               pthread_function, &arg_1);

    if ((status_1 + status_2 + status_3) != 0)
    {
        fprintf (stderr, "Errore nella creazione "
                "dei \"thread\"!\n");
        abort ();
    }

    status_1 = pthread_join (pthread_1, NULL);
    status_2 = pthread_join (pthread_2, NULL);
    status_3 = pthread_join (pthread_3, NULL);
}
```

```
    if ((status_1 + status_2 + status_3) != 0)
    {
        fprintf (stderr, "Errore nella fusione "
                "dei \"thread\"!\n");
        abort ();
    }

    return (0);
}
```

All'inizio della funzione *main()* si può vedere la dichiarazione di tre variabili di tipo `'pthread_t'`, ognuna delle quali viene usata per annotare un numero di identificazione di un thread. Con la chiamata alla funzione *pthread\_create()* vengono avviati i thread, indicando il riferimento alla variabile da usare per annotare il thread rispettivo, il riferimento alla funzione da avviare e il riferimento alla struttura contenente gli argomenti per tale funzione. Il prototipo seguente è semplificato, per facilitarne la lettura:

```
int pthread_create (pthread_t *tid, pthread_attr_t *attr,
                  void *(*funzione) (void *),
                  void *argomenti);
```

Generalmente, si usa la funzione *pthread\_create()* senza specificare attributi particolari per il thread che si vuole creare, pertanto si utilizza semplicemente come secondo argomento il valore `'NULL'`.

La funzione *pthread\_create()* restituisce un valore intero, dove lo zero manifesta il successo dell'operazione, mentre un valore differente indica un problema, decodificabile attraverso il confronto con delle macro-variabili prestabilite. La funzione che avvia il thread for-



nisce il numero dello stesso modificando il contenuto della variabile a cui si riferisce il puntatore fornito come primo argomento.

Una volta accertato che i thread sono stati creati con successo (diversamente il programma termina di funzionare, attraverso la chiamata della funzione *abort()*), non essendoci altro da fare in questo esempio, viene richiesto di attendere la loro conclusione, attraverso la chiamata della funzione *pthread\_join()*. Tale funzione richiede di indicare il numero del thread di cui si vuole attendere la conclusione, oltre a un puntatore, utile per raggiungere il valore che potrebbe essere restituito dalla funzione che costituiva il thread.

```
int pthread_join (pthread_t tid, void **valore);
```

In pratica, la funzione *pthread\_join()* sospende l'esecuzione del flusso principale, fino a quando il thread individuato dal numero fornito come primo argomento si conclude (si osservi che in questo caso il numero del thread viene fornito come valore e non più come puntatore). Se il thread non deve produrre alcun risultato utile, il secondo argomento di *pthread\_join()* può essere il valore nullo ('**NULL**'), altrimenti si deve indicare un puntatore generico, a una variabile che contiene a sua volta un puntatore: tale variabile deve essere quella usata dalla funzione che costituiva il thread per porvi al suo interno il puntatore al risultato dell'elaborazione.

La funzione *pthread\_join()* restituisce un valore intero, dove lo zero indica il successo dell'operazione, mentre un valore diverso rappresenta un problema, individuabile attraverso il confronto con delle macro-variabili prestabilite.

Il programma di esempio, dopo la fusione dei thread e dopo il con-

trollo dell'esito di tale fusione, si conclude semplicemente. Va osservato che la fusione dei thread è necessaria anche in questo caso, perché il programma non può concludersi (attraverso la fine del flusso principale) prima che tutti i thread accessori siano stati fusi.

La funzione utilizzata per i thread dell'esempio, ovvero *pthread\_function()*, trova un numero casuale abbastanza grande e lo usa per eseguire un ciclo per un numero molto elevato di volte. Al primo ciclo, e poi anche ogni volta che l'indice del ciclo risulta divisibile per il numero casuale trovato, mostra una lettera sullo schermo. A questo proposito, va osservato l'uso della funzione *fflush()* per garantire che la lettera emessa attraverso lo standard output venga visualizzata subito, senza rimanere in attesa nella memoria tampone.

I thread dell'esempio vengono avviati con insiemi di dati differenti, in modo che: il thread *pthread\_3* emetta la lettera «c» per tre volte, il thread *pthread\_2* emetta la lettera «b» per sei volte e che il thread *pthread\_1* emetta la lettera «a» per dodici volte.

Il risultato visibile sullo schermo assomiglia a una sequenza come questa:

```
cbaaaaaabcaaaabacabbb
```

#### 68.4.5 Conflitto nell'accesso ai dati

«

Quando un thread opera su dati propri (a cui nessun altro thread, nemmeno quello principale, accede in scrittura), tutto fila liscio senza preoccupazioni. Ma la realtà richiede generalmente che i thread si scambino dei dati, pertanto, quando si aggiorna un'informazione, occorre un modo per escludere gli altri thread dall'interferire.

Nell'esempio successivo si crea volutamente una situazione di conflitto tra alcuni thread che modificano simultaneamente una variabile, denominata *global*, il cui scopo sarebbe quello di contare i caratteri mostrati sullo schermo. Il file dovrebbe essere disponibile presso [allegati/c/esempio-posix-thread-2.c](#) .

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

struct Arguments {
    char x;
    int  max;
};

int global;

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10);
    int g;

    for (i = 0; i < (arg->max * m); i++)
    {
        g = global;
        if ((i % m) == 0)
        {
            printf ("%c", arg->x);
            fflush (stdout);
            g++;
            global = g;
        }
    }
}
```

```
        }
    }
    return NULL;
}

int
main (void)
{
    pthread_t pthread_1;
    pthread_t pthread_2;
    pthread_t pthread_3;

    struct Arguments arg_1 = {'a', 12};
    struct Arguments arg_2 = {'b', 6};
    struct Arguments arg_3 = {'c', 3};

    int status_1;
    int status_2;
    int status_3;

    global = 0;

    srand (1234567);

    status_3 = pthread_create (&pthread_3, NULL,
                               pthread_function, &arg_3);
    status_2 = pthread_create (&pthread_2, NULL,
                               pthread_function, &arg_2);
    status_1 = pthread_create (&pthread_1, NULL,
                               pthread_function, &arg_1);

    if ((status_1 + status_2 + status_3) != 0)
    {
        fprintf (stderr, "Errore nella creazione "

```

```

                                "dei \"thread\\\"!\n");
    abort ();
}

status_1 = pthread_join (pthread_1, NULL);
status_2 = pthread_join (pthread_2, NULL);
status_3 = pthread_join (pthread_3, NULL);

if ((status_1 + status_2 + status_3) != 0)
{
    fprintf (stderr, "Errore nella fusione "
            "dei \"thread\\\"!\n");
    abort ();
}

printf ("\n");
printf ("La variabile globale ha raggiunto "
        "il valore %i.\n", global);

return (0);
}

```

Provando a eseguire il programma di esempio, si potrebbero osservare messaggi molto simili a quelli seguenti:

```
cbaaaaaacbaaaaabcabbb
```

```
La variabile globale ha raggiunto il valore 19.
```

In questo caso la variabile globale che viene modificata dalla funzione *pthread\_function()* ha raggiunto solo il valore 19, mentre il valore atteso sarebbe di 21 (essendo visualizzati 21 caratteri sullo schermo). Naturalmente può succedere che il valore ottenuto dalla variabile sia corretto, ma non ci si può contare, perché non è possibile

prevedere la sequenza effettiva delle operazioni.

Naturalmente, si può migliorare la funzione *pthread\_function()* per ridurre al minimo la possibilità di accavallamenti tra le attività dei vari thread, ma anche così non si può avere la garanzia di evitare i conflitti:<sup>2</sup>

```
void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10);
    int g;
    //
    for (i = 0; i < (arg->max * m); i++)
        {
            if ((i % m) == 0)
                {
                    printf ("%c", arg->x);
                    fflush (stdout);
                    g = global;
                    g++;
                    global = g;
                }
        }
    return NULL;
}
```

#### 68.4.6 Accesso alle risorse in modo mutualmente esclusivo



Per accedere simultaneamente, in modo ordinato, a dati condivisi, occorre definire dei *mutex*, ovvero delle variabili che hanno il ruolo di «lucchetto» per definire un accesso mutualmente esclusivo a

una certa area di dati. In altri termini, una volta definita una certa attività da svolgere in modo esclusivo, in un certo insieme di dati, gli si associa una variabile speciale con funzione di mutex (lucchetto mutualmente esclusivo) e prima di entrare nella zona critica che richiede un accesso esclusivo a quell'insieme di dati, si cerca di ottenere tale esclusività con una funzione che interroga e modifica la variabile mutex.

```
...
pthread_mutex_t mutex_1 = PTHREAD_MUTEX_INITIALIZER;

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 1000);
    int g;

    for (i = 0; i < (arg->max * m); i++)
        {
            pthread_mutex_lock (&mutex_1);
            g = global;
            if ((i % m) == 0)
                {
                    printf ("%c", arg->x);
                    fflush (stdout);
                    g++;
                    global = g;
                }
            pthread_mutex_unlock (&mutex_1);
        }
    return NULL;
}
```

...

L'estratto di esempio appena mostrato mette in evidenza le modifiche da apportare per gestire il meccanismo di accesso mutualmente esclusivo. In questo caso la porzione di codice da eseguire in modo mutualmente esclusivo va dalla lettura della variabile globale alla sua modifica successiva: nell'esempio le operazioni sono tenute distanti per dimostrare il funzionamento, dato che sarebbe meglio ridurre al minimo il tempo in cui un thread blocca un mutex.

La variabile globale `'mutex_1'` viene dichiarata di tipo `'pthread_mutex_t'` (presumibilmente si tratta di una struttura) e viene inizializzata attraverso una macro-variabile appropriata alle sue caratteristiche. Successivamente, prima di entrare nella zona critica, il thread deve richiedere l'accesso esclusivo attraverso la funzione `pthread_mutex_lock()`, specificando il riferimento alla variabile che costituisce il mutex del contesto. Quando il thread ottiene l'accesso esclusivo può riprendere la sua esecuzione e, quando non ha più bisogno di impegnare il mutex, lo libera, con la funzione `pthread_mutex_unlock()`.

Per completezza viene mostrato il programma di esempio completo. Il file dovrebbe essere disponibile presso [allegati/c/esempio-posix-thread-3.c](#).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

struct Arguments {
    char x;
```



```
    int  max;
};

int global;

pthread_mutex_t mutex_1 = PTHREAD_MUTEX_INITIALIZER;

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 1000);
    int g;
    //
    for (i = 0; i < (arg->max * m); i++)
        {
            pthread_mutex_lock (&mutex_1);
            g = global;
            if ((i % m) == 0)
                {
                    printf ("%c", arg->x);
                    fflush (stdout);
                    g++;
                    global = g;
                }
            pthread_mutex_unlock (&mutex_1);
        }
    return NULL;
}

int
main (void)
{
```

```
pthread_t pthread_1;
pthread_t pthread_2;
pthread_t pthread_3;

struct Arguments arg_1 = {'a', 12};
struct Arguments arg_2 = {'b', 6};
struct Arguments arg_3 = {'c', 3};

int status_1;
int status_2;
int status_3;

global = 0;

srand (1234567);

status_3 = pthread_create (&pthread_3, NULL,
                          pthread_function, &arg_3);
status_2 = pthread_create (&pthread_2, NULL,
                          pthread_function, &arg_2);
status_1 = pthread_create (&pthread_1, NULL,
                          pthread_function, &arg_1);

if ((status_1 + status_2 + status_3) != 0)
{
    fprintf (stderr, "Errore nella creazione "
            "dei \"thread\"!\n");
    abort ();
}

status_1 = pthread_join (pthread_1, NULL);
status_2 = pthread_join (pthread_2, NULL);
status_3 = pthread_join (pthread_3, NULL);
```

```

    if ((status_1 + status_2 + status_3) != 0)
    {
        fprintf (stderr, "Errore nella fusione "
                "dei \"thread\"\n");
        abort ();
    }

    printf ("\n");
    printf ("La variabile globale ha raggiunto "
           "il valore %i.\n", global);

    return (0);
}

```

Eseguendo il programma si può osservare che non si creano più accavallamenti nella scrittura della variabile globale e il risultato finale è sempre corretto:

```
cbaaaaaabcaaaaaabcbbb
```

La variabile globale ha raggiunto il valore 21.

Come già descritto nella sezione precedente, è sicuramente meglio ridurre al minimo la zona critica, in modo che anche con l'ausilio delle variabili mutex non sia penalizzata la simultaneità di esecuzione dei thread:

```

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10000);
    int g;

```

```
for (i = 0; i < (arg->max * m); i++)
{
    if ((i % m) == 0)
    {
        printf ("%c", arg->x);
        fflush (stdout);
        pthread_mutex_lock (&mutex_1);
        g = global;
        g++;
        global = g;
        pthread_mutex_unlock (&mutex_1);
    }
}
return NULL;
}
```

### 68.4.7 Accesso esclusivo, ma condizionato



Può darsi che l'accesso esclusivo a una zona critica debba avvenire solo al verificarsi di una certa condizione. In altri termini, può darsi che prima di intervenire effettivamente in un certo insieme di dati, un thread debba attendere che questi siano pronti. Per ottenere questo risultato, generalmente, a fianco della variabili mutex, si associano delle variabili che rappresentano il verificarsi di una certa condizione, da gestire anche queste attraverso funzioni apposite dei thread POSIX.

L'estratto seguente mostra le modifiche importanti agli esempi già apparsi, per produrre una situazione in cui i thread devono attendere il verificarsi di una condizione per procedere con il loro intervento nell'area critica:

```
struct Arguments {
    char x;
    int max;
    int delay;
};

int global;

pthread_mutex_t mutex_1 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_1 = PTHREAD_COND_INITIALIZER;

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10000);
    int g;

    for (i = 0; i < (arg->max * m); i++)
        {
            if ((i % m) == 0)
                {
                    pthread_mutex_lock (&mutex_1);
                    while (global < arg->delay)
                        {
                            pthread_cond_wait (&cond_1, &mutex_1);
                        }
                    printf ("%c", arg->x);
                    fflush (stdout);
                    g = global;
                    g++;
                    global = g;
                    pthread_cond_broadcast (&cond_1);
                }
        }
}
```

```
        pthread_mutex_unlock (&mutex_1);  
    }  
}  
return NULL;  
}
```

Questa volta, la struttura che costituisce gli argomenti della funzione *pthread\_function()* ha un'informazione in più, che rappresenta un ritardo da inserire prima di iniziare a mostrare i caratteri sullo schermo. In pratica, se la variabile globale ha raggiunto o superato quel tale valore, il thread può procedere con il proprio lavoro, altrimenti deve rimanere in attesa.

Per ottenere questo risultato, la variabile globale `'cond_1'` viene dichiarata con il tipo `'pthread_cond_t'`, allo scopo di poter rappresentare le condizioni dei thread, e viene inizializzata con una macrovariabile appropriata alle sue caratteristiche effettive. Il thread, prima cerca di ottenere un accesso esclusivo, quindi, se lo ottiene, inizia un ciclo in attesa del verificarsi della condizione, richiamando ripetutamente la funzione *pthread\_cond\_wait()*, con il riferimento alla variabile della condizione e a quella del mutex.

La chiamata della funzione *pthread\_cond\_wait()* fa sì che il thread che aveva ottenuto l'accesso esclusivo venga messo in pausa, a vantaggio di un altro che può così ottenere l'accesso esclusivo alla zona critica. La pausa in cui si trova il primo thread può terminare nel momento in cui viene usata la funzione *pthread\_cond\_broadcast()*, con il riferimento alla condizione che aveva prodotto la sospensione e poi anche la funzione *pthread\_mutex\_unlock()*.

Il thread che era stato messo in pausa dalla funzione

***pthread\_cond\_wait()***, riprende quando tale funzione ha riottenuto l'accesso esclusivo in base alla propria variabile mutex.

Logicamente, occorre fare attenzione a non creare una situazione in per cui tutti i thread si mettono in pausa per qualcosa che non si verifica.

Segue il programma di esempio, completo di tutte le sue parti. Il file dovrebbe essere disponibile presso [allegati/c/esempio-posix-thread-4.c](#).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

struct Arguments {
    char x;
    int  max;
    int  delay;
};

int global;

pthread_mutex_t mutex_1 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond_1  = PTHREAD_COND_INITIALIZER;

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10000);
    int g;

    for (i = 0; i < (arg->max * m); i++)
```

```
{
    if ((i % m) == 0)
    {
        pthread_mutex_lock (&mutex_1);
        while (global < arg->delay)
        {
            pthread_cond_wait (&cond_1, &mutex_1);
        }
        printf ("%c", arg->x);
        fflush (stdout);
        g = global;
        g++;
        global = g;
        pthread_cond_broadcast (&cond_1);
        pthread_mutex_unlock (&mutex_1);
    }
}
return NULL;
}

int
main (void)
{
    pthread_t pthread_1;
    pthread_t pthread_2;
    pthread_t pthread_3;

    struct Arguments arg_1 = {'a', 12, 0};
    struct Arguments arg_2 = {'b', 6, 5};
    struct Arguments arg_3 = {'c', 3, 10};

    int status_1;
    int status_2;
    int status_3;
```



```
global = 0;

srand (1234567);

status_3 = pthread_create (&pthread_3, NULL,
                           pthread_function, &arg_3);
status_2 = pthread_create (&pthread_2, NULL,
                           pthread_function, &arg_2);
status_1 = pthread_create (&pthread_1, NULL,
                           pthread_function, &arg_1);

if ((status_1 + status_2 + status_3) != 0)
{
    fprintf (stderr, "Errore nella creazione "
             "dei \"thread\"!\n");
    abort ();
}

status_1 = pthread_join (pthread_1, NULL);
status_2 = pthread_join (pthread_2, NULL);
status_3 = pthread_join (pthread_3, NULL);

if ((status_1 + status_2 + status_3) != 0)
{
    fprintf (stderr, "Errore nella fusione "
             "dei \"thread\"!\n");
    abort ();
}

printf ("\n");
printf ("La variabile globale ha raggiunto "
        "il valore %i.\n", global);
```

```
return (0);  
}
```

Eseguendo il programma si può ottenere un risultato simile a quello seguente:

```
aaaaabbbbbcbccaaaaaaa
```

La variabile globale ha raggiunto il valore 21.

Nel programma di esempio, il thread associato alla variabile `'pthread_1'` può visualizzare subito i suoi caratteri sullo schermo, mentre quello associato a `'pthread_2'` deve attendere che sia stato visualizzato il quinto, mentre quello associato a `'pthread_3'` deve attendere che sia stato visualizzato il decimo. Naturalmente, se tutti i thread avviati dovessero attendere qualche carattere prima di poter iniziare, questi si bloccherebbero a vicenda, irrimediabilmente; inoltre, lo stesso succederebbe se ci fosse anche un solo thread che deve attendere un valore per la variabile `'global'` che non può essere raggiunto senza il proprio apporto.

#### 68.4.8 Osservazioni finali

«

Lo standard POSIX prevede una discreta quantità di funzioni per la gestione dei thread; pertanto quanto descritto in questo capitolo è solo il minimo indispensabile per comprenderne il meccanismo. In modo particolare, va tenuto in considerazione che per l'inizializzazione delle variabili mutex e di quelle che rappresentano le condizioni, si possono usare funzioni apposite che non sono state descritte.

Il fatto che ci siano thread distinti rispetto a quello principale, ha delle implicazioni anche sull'invio dei segnali e sulla possibilità di una loro conclusione anticipata. Pertanto sono disponibili le funzioni

*pthread\_kill()* e *pthread\_exit()*, rivolte specificatamente ai thread (escluso sempre il flusso principale):

```
int pthread_kill (pthread_t tid, int segnale);
```

```
void pthread_exit (void *valore_da_restituire);
```

Infine può essere utile la funzione *pthread\_self()*, per ottenere il numero identificativo del thread attuale:

```
pthread_t pthread_self (void);
```

## 68.5 I file secondo i sistemi POSIX

Il linguaggio C, puro e semplice, prevede una gestione dei file basilare, attraverso il tipo derivato **FILE**, per cui un file aperto è un «flusso», identificato da un puntatore al tipo **FILE**. Lo standard dei sistemi Unix comporta un'infrastruttura più articolata per la gestione dei file, al di sotto di quanto già descrive il C, introducendo il concetto di *descrittore di file*, corrispondente a un numero intero normale positivo. Le funzioni e le macro-variabili principali per l'apertura e il controllo dei file, secondo la mediazione del concetto di descrittore, sono indicati nel file di intestazione `fcntl.h` (*file control*), ma per amministrare le caratteristiche dei file, servono le definizioni e le funzioni del file di intestazione `sys/stat.h`; inoltre, altre funzioni importanti al riguardo si trovano nel file `unistd.h`.

L'apertura di un file, dal punto di vista dei sistemi Unix (e quindi POSIX), implica non solo l'associazione al numero del descrittore, ma anche l'attribuzione di opzioni di funzionamento ed eventualmente un sistema di blocco di porzioni del file. La creazione di un file implica l'attribuzione di permessi, nel rispetto però della maschera dei permessi esistente.

Va osservato che anche i flussi di file standard del linguaggio C, trovano una corrispondenza nello standard POSIX in altrettanti descrittori già assegnati, costituiti precisamente dai primi tre:

Denominazione	flusso di file C	numero del descrittore POSIX
standard input	<code>stdin</code>	0
standard output	<code>stdout</code>	1
standard error	<code>stderr</code>	2

Lo standard POSIX prescrive che i numeri dei descrittori siano assegnati usando sempre il valore libero più piccolo; pertanto, il primo descrittore a essere utilizzato, dato che i primi tre sono impegnati per i flussi standard, è il numero tre e di seguito vanno i successivi.

### 68.5.1 Apertura e chiusura di un file

«

L'apertura ed eventuale creazione di un file, secondo le convenzioni POSIX, va eseguita utilizzando la funzione *open()*. Per motivi storici esiste anche la funzione *creat()* che però ha meno possibilità di *open()*, pertanto il suo utilizzo non è indispensabile.

```
int open (const char *file, int oflag [, mode_t mode ] );
```

```
int creat (const char *file, mode_t mode );
```

La funzione *open()* apre un file, indicato attraverso una stringa che descrive il suo percorso (relativo o assoluto che sia) secondo le convenzioni POSIX e restituisce il numero del suo descrittore; se però restituisce il valore  $-1$ , significa che l'operazione non ha avuto successo e di conseguenza è stato modificato il contenuto della variabile *errno* (la quale può essere esaminata per determinarne la causa).

Il valore costituito dal parametro *oflag* viene ottenuto combinando assieme, con l'operatore OR binario, una serie di macro-variabili definite nel file 'fcntl.h', tenendo conto che non tutte le combinazioni sono ammissibili simultaneamente. Se si utilizza l'opzione rappresentata dalla macro-variabile *O\_CREAT*, per richiedere la creazione del file, va usato anche il terzo parametro della funzione, con cui si specifica la modalità di creazione dello stesso.

Tabella 68.47. Macro-variabili da utilizzare per combinare il valore del parametro *oflag*. Di queste macro-variabili, in particolare, ne va scelta una sola e si è obbligati a usarla per specificare la modalità di accesso al file: in sola lettura, in sola scrittura o in entrambi i modi.

Macro-variabile	Significato
<i>O_RDONLY</i>	Si richiede l'apertura del file in sola lettura.

Macro-variabile	Significato
O_WRONLY	Si richiede l'apertura del file in sola scrittura.
O_RDWR	Si richiede l'apertura del file in lettura e scrittura.

Tabella 68.48. Alcune delle macro-variabili da utilizzare per combinare il valore del parametro *oflag*. Non tutte le combinazioni di queste opzioni sono ammissibili.

Macro-variabile	Significato
O_APPEND	Questa opzione fa in modo che la scrittura avvenga sempre in estensione del contenuto già esistente.
O_CREAT	Questa opzione richiede la creazione del file, ammesso che non esista già. Nel caso il file sia effettivamente da creare, vale la modalità di creazione specificata con il parametro <i>mode</i> .
O_TRUNC	Questa opzione si può associare solo a una richiesta di accesso in scrittura (in sola scrittura o in lettura e scrittura simultaneamente) relativa a un file normale, con lo scopo di azzerarne il contenuto se questo file esiste già.
O_EXCL	Questa opzione può essere usata solo in abbinamento a 'O_CREAT' e richiede la creazione del file o il fallimento dell'operazione se questo esiste già, anche nel caso si tratti solo di un collegamento simbolico che punta a un file inesistente.
O_NOCTTY	Questa opzione fa sì che se il file indicato corrisponde a un dispositivo di terminale, questo non possa diventare in alcun caso il terminale di controllo abbinato al processo.

Macro-variabile	Significato
O_NONBLOCK	Questa opzione richiede di non attendere per la conclusione delle operazioni, ammesso che ciò sia possibile in base al contesto.
O_SYNC	Questa opzione richiede che le operazioni di accesso al file avvengano in modo sincrono rispetto all'hardware. Pertanto, in questo modo, le operazioni di scrittura comportano l'attesa che queste siano realizzate effettivamente.

Il parametro *mode* riguarda esclusivamente la creazione del file (specificando l'indicatore '**O\_CREAT**'). In tal caso, si tratta del numero che esprime i permessi da dare al file. Si tratta degli stessi permessi che si indicano con programmi come '**chmod**', quando si usa la forma numerica, e si scrivono preferibilmente in base otto. Naturalmente, i permessi indicati vengono poi filtrati attraverso la maschera dei permessi, come se fosse eseguita questa operazione: '*mode & ~umask*'. Se lo si preferisce, al posto di indicare i permessi richiesti, direttamente in forma numerica, ci si può avvalere di macro-variabili dichiarate nel file di intestazione '`sys/stat.h`', come descritto nella tabella successiva.

Tabella 68.49. Macro-variabili per esprimere i permessi da attribuire a un file che si vuole creare.

Macro-variabile	Valore numerico equivalente	Significato
S_IRWXU	0700 <sub>8</sub>	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per l'utente proprietario.

Macro-variabile	Valore numerico equivalente	Significato
S_IRUSR	0400 <sub>8</sub>	Rappresenta il permesso di lettura per l'utente proprietario.
S_IWUSR	0200 <sub>8</sub>	Rappresenta il permesso di scrittura per l'utente proprietario.
S_IXUSR	0100 <sub>8</sub>	Rappresenta il permesso di esecuzione o attraversamento per l'utente proprietario.
S_IRWXG	0070 <sub>8</sub>	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per il gruppo proprietario.
S_IRGRP	0040 <sub>8</sub>	Rappresenta il permesso di lettura per il gruppo proprietario.
S_IWGRP	0020 <sub>8</sub>	Rappresenta il permesso di scrittura per il gruppo proprietario.
S_IXGRP	0010 <sub>8</sub>	Rappresenta il permesso di esecuzione o attraversamento per il gruppo proprietario.
S_IRWXO	0007 <sub>8</sub>	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per gli altri utenti.
S_IROTH	0004 <sub>8</sub>	Rappresenta il permesso di lettura per gli altri utenti.
S_IWOTH	0002 <sub>8</sub>	Rappresenta il permesso di scrittura per gli altri utenti.
S_IXOTH	0001 <sub>8</sub>	Rappresenta il permesso di esecuzione o attraversamento per gli altri utenti.
S_ISUID	4000 <sub>8</sub>	Rappresenta l'attivazione del bit S-UID.
S_ISGID	2000 <sub>8</sub>	Rappresenta l'attivazione del bit S-GID.



Macro-variabile	Valore numerico equivalente	Significato
S_ISVTX	1000 <sub>8</sub>	Rappresenta l'attivazione del bit Sticky.

Come già accennato, la funzione *creat()* non è più indispensabile e può essere sostituita da *open()*, usata nel modo seguente:

```
open (file, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

Segue un esempio molto semplice in cui si apre un file in scrittura, specificando che se non esiste già, questo va creato, con tutti i permessi che la maschera dei permessi esistente consenta di attribuire. In caso di errore, il contenuto della variabile *errno* viene considerato con l'aiuto della funzione *perror()*. Il file dell'esempio dovrebbe essere disponibile presso [allegati/c/esempio-posix-fcntl-open.c](#).

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int
main (void)
{
    const char *file = "/tmp/test";
    int fdn;

    fdn = open (file, O_WRONLY|O_CREAT, 0777);

    if (fdn >= 0)
    {
        printf ("Aperto il file \"%s\", ", file);
```

```
        printf ("associandolo al descrittore %i.\n", fdn);
        close (fdn);
    }
else
    {
        printf ("Non è possibile aprire il file \"%s\"!\n",
                file);
        perror (NULL);
    }

return (0);
}
```

La funzione *open()* richiede l'inclusione del file 'fcntl.h', nel quale sono dichiarate anche la macro-variabili *O\_WRONLY* e *O\_CREAT*; ma per la funzione *close()* è necessario includere anche il file 'unistd.h'.

Come si vede, i permessi da attribuire al file che venisse creato sono tutti quelli disponibili (7777<sub>8</sub>). Eventualmente, aggiungendo anche l'inclusione del file 'sys/stat.h', sarebbe possibile indicare tale richiesta attraverso macro-variabili convenzionali:

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

...
fdn = open (file, O_WRONLY|O_CREAT,
           S_IRUSR|S_IWUSR|S_IRGRP);
...
```

In questo esempio, però, i permessi richiesti sono minori, corrispondenti al numero  $0640_8$ .

Va osservato che la funzione *open()* può aprire ogni tipo di file, ma può creare solo dei file «normali». Per creare directory e altri tipi di file speciali si usano funzioni apposite. D'altro canto, per attribuire a un file dei permessi, è possibile usare la funzione *chmod()* e non è strettamente necessario occuparsene nel momento della creazione.

La funzione *close()*, già mostrata nell'esempio, ha un prototipo molto semplice: richiede l'indicazione del descrittore del file da chiudere e restituisce zero se tutto va bene, altrimenti produce il valore  $-1$  e aggiorna la variabile *errno*:

```
int close (int descrittore);
```

## 68.5.2 Lettura e scrittura

Come per le funzioni dello standard C, anche per quelle a livello POSIX si accede al contenuto del file attraverso un indicatore della posizione espresso in byte (con la differenza che non si pone il problema di distinguere tra file di testo e file binari). A ogni descrittore di file sono associate delle informazioni, amministrare in modo trasparente dal sistema operativo, e a queste si accede solo attraverso delle funzioni. Tra queste informazioni si trova anche l'indicatore che consente di determinare la posizione iniziale per la lettura o la scrittura.

A seconda di come viene aperto il file, l'indicatore della posizione che lo riguarda viene inizializzato nel modo più logico, come descritto a proposito della funzione *open()*. Questo indicatore vie-

ne spostato automaticamente a seconda delle operazioni di lettura e scrittura che si compiono, tuttavia, quando si passa da una modalità di accesso all'altra, è necessario spostare l'indicatore attraverso le istruzioni opportune, in modo da non creare ambiguità.

Per la lettura di un file aperto e qualificato con un descrittore, si può usare la funzione *read()* che legge una quantità di byte trattandoli come un array. Si osservi l'esempio seguente:

```
...
char buf[100];
int fdn;
ssize_t dim;
...
fdn = open (...);
...
dim = read (fdn, buf, 100);
...
```

In questo modo si intende leggere 100 byte, collocandoli nell'array *buf*, con la stessa capacità massima. Naturalmente, non è detto che la lettura abbia successo, o quantomeno non è detto che si riesca a leggere la quantità di elementi richiesta. Il valore restituito dalla funzione rappresenta la quantità di byte letti effettivamente. Se si verifica un qualsiasi tipo di errore che impedisce la lettura, la funzione si limita a restituire  $-1$ , mentre lo zero è un risultato valido e indica che la lettura è giunta alla fine del file.

Quando il file viene aperto in lettura, in condizioni normali l'indicatore interno viene posizionato all'inizio del file; quindi, ogni operazione di lettura sposta in avanti il puntatore, in modo che la prossima lettura avvenga a partire dalla posizione immediatamente successiva:

```
...
char buf[100];
int fdn;
ssize_t dim;
...
fdn = open (...);
...
while (1)                // Ciclo senza fine.
{
    dim = read (fdn, buf, 100);
    if (dim == 0)
        {
            break;        // Termina il ciclo.
        }
    ...
}
...
```

In questo modo, come mostra l'esempio, viene letto tutto il file a colpi di 100 byte alla volta, tranne l'ultima in cui si ottiene solo quello che resta da leggere.

Analogamente, la scrittura può essere eseguita con la funzione *write()* che scrive una quantità di byte trattandoli come un array, nello stesso modo già visto con la funzione *read()*. Anche in questo caso, la scrittura procede a partire dalla posizione corrente riferita al file.

```
...
    char buf[100];
    int fdn;
    ssize_t dim;
    ...
    fdn = open (...);
    ...
    dim = write (fdn, buf, 100);
    ...
```

L'esempio, come nel caso di *read()*, mostra la scrittura di 100 byte, prelevati da un array. Il valore restituito dalla funzione è la quantità di elementi che sono stati scritti con successo. Se si verifica un errore la funzione restituisce il valore  $-1$ , mentre lo zero è un valore valido.

Anche in scrittura è importante l'indicatore della posizione interna del file. Di solito, quando si crea un file o lo si estende, l'indicatore si trova sempre alla fine. L'esempio seguente mostra lo scheletro di un programma che crea un file, copiando il contenuto di un altro (non viene utilizzato alcun tipo di controllo degli errori).

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
...
int
main (void)
{
    char          buf[1024];
    int           fdn_in;
    int           fdn_out;
    ssize_t       dim;
    ...
    fdn_in = open (file, O_RDONLY);
```

```
...
fdn_out = open (file, O_WRONLY);
...
while (1)                                // Ciclo senza fine.
{
    dim = read (fdn, buf, 1024);
    if (dim == 0)
        {
            break;                        // Termina il ciclo.
        }
    ...
    write (fdn_out, buf, dim);
    ...
}
...
close (fdn_in);
close (fdn_out);
return 0;
}
```

Seguono i modelli sintattici di *read()* e *write()*, espressi in forma di prototipi di funzione:

```
ssize_t read (int fdn, void *buf, size_t n);
```

```
ssize_t write (int fdn, const void *buf, size_t n);
```

Il tipo di dati '**ssize\_t**' rappresenta l'equivalente di '**size\_t**', ma con segno, allo scopo di poter rappresentare il valore  $-1$ , che indica un esito errato; il tipo '**void**' per l'array in cui vanno scritti o da cui vanno letti i dati, permette l'utilizzo di qualunque tipo per i suoi

elementi, anche se le operazioni di lettura e scrittura operano solo al livello di byte.

### 68.5.3 Spostamento dell'indicatore interno al file

«

Lo spostamento diretto dell'indicatore interno della posizione di un file aperto è un'operazione necessaria quando il file è stato aperto simultaneamente in lettura e in scrittura, e da un tipo di operazione si vuole passare all'altro. Per questo si utilizza la funzione *lseek()*, con la quale è possibile leggere e modificare tale posizione attuale. La posizione e gli spostamenti sono espressi in byte; la variabile usata per rappresentare questi spostamenti è di tipo 'off\_t' (*offset*).

La funzione *lseek()* esegue lo spostamento a partire dall'inizio del file, oppure dalla posizione attuale, oppure dalla posizione finale. Per questo utilizza un parametro che può avere tre valori identificati rispettivamente da tre macro-variabili, definite all'interno del file 'stdio.h': **SEEK\_SET**, **SEEK\_CUR** e **SEEK\_END**. l'esempio seguente mostra lo spostamento del puntatore, riferito al descrittore di file *fdn*, in avanti di 10 byte, a partire dalla posizione attuale.

```
...  
i = lseek (fdn, 10, SEEK_CUR);  
...
```

La funzione *lseek()* restituisce la posizione raggiunta all'interno del file, partendo dall'inizio dello stesso, se lo spostamento avviene con successo, altrimenti produce il valore -1.

L'esempio seguente mostra lo scheletro di un programma, senza controlli sugli errori, che, dopo aver aperto un file in lettura e scrittura, lo legge a blocchi di dimensioni uguali, modifica questi blocchi





```
//  
// Salva la posizione del puntatore interno al file  
// dopo la lettura.  
//  
pos_2 = lseek (fdn, 0, SEEK_CUR);  
//  
// Sposta il puntatore alla posizione precedente alla  
// lettura.  
//  
lseek (fdn, pos_1, SEEK_SET);  
//  
// Esegue qualche modifica nei dati, per esempio  
// mette un punto esclamativo all'inizio.  
//  
buf[0] = '!';  
//  
// Riscrive il record modificato.  
//  
write (fdn, buf, dim);  
//  
// Riporta il puntatore interno al file alla posizione  
// corretta per eseguire la lettura successiva  
//  
lseek (fdn, pos_2, SEEK_SET);  
}  
  
close (fdn);  
return 0;  
}
```

Segue il modello sintattico per l'uso della funzione *lseek()*, espresso attraverso il suo prototipo:

```
off_t lseek (int fdn, off_t spostamento,  
            int punto_di_partenza);
```

Il valore dello spostamento, costituito dal secondo parametro, rappresenta una quantità di byte che può essere anche negativa, indicando in tal caso un arretramento dal punto di partenza specificato dal terzo parametro. Il valore restituito da *lseek()* è la nuova posizione all'interno del file, espressa a partire dall'inizio dello stesso (pertanto deve trattarsi di un valore maggiore o uguale a zero); se invece si presenta un errore, si ottiene  $-1$  (precisamente si definisce come `(off_t) -1`).

#### 68.5.4 Controllo degli errori

Le funzioni descritte, quando si verifica un errore, annotano il numero dell'errore nella variabile globale *errno* (il nome *errno* dovrebbe essere precisamente un'espressione che si traduce nell'accesso a un'area di memoria condiviso dal programma, distinto in base al thread). Il significato del valore attribuito alla variabile *errno* è descritto da macro-variabili definite nel file `errno.h`, il quale fa già parte dello standard C, ma viene esteso da POSIX. «

La lettura della variabile *errno* porta alla conoscenza dell'ultimo errore che si è presentato e non è previsto il suo azzeramento automatico; pertanto, occorre accertarsi del verificarsi di un problema, prima di interrogare la variabile, oppure la si deve azzerare prima di chiamare una funzione di cui si vuole verificare l'esito.

Per interpretare l'errore annotato nella variabile *errno* e visualizzare direttamente un messaggio attraverso lo standard error, si può usare

la funzione *perror()*, già descritta nei capitoli sul C:

```
void perror (const char *s);
```

## 68.6 Il file system Unix e la sua gestione tipica

«

Per comprendere il senso dell'organizzazione della libreria C e di quella POSIX, per quanto riguarda la gestione dei file, è necessario conoscere l'impostazione originale della gestione di un file system in un sistema Unix. A tale riguardo ci sono due livelli: quello del file system, così come viene strutturato nell'unità di memorizzazione e la gestione dei file aperti, a livelli diversi, partendo dall'inode, fino al flusso di file del C, passando per il concetto di descrittore del file.

### 68.6.1 Il blocco

«

In un file system Unix tradizionale, lo spazio di un'unità di memorizzazione è suddiviso in blocchi di byte, di dimensione pari a un multiplo del settore fisico, ma si tratta comunque di un valore che si ottiene come potenza di 2. Considerato che le unità di memorizzazione comuni hanno settori fisici da 512 byte, il blocco di un tale file system può essere da 1024, 2048, 4096 byte,... La dimensione effettiva di tale blocco dipende però dalle caratteristiche specifiche di quel tipo di file system, tenendo conto che spesso è possibile scegliere la sua dimensione in fase di inizializzazione.

Nel caso del file system Minix, si distingue tra blocchi e zone. La zona è un concetto specifico dei sistemi Minix e, in realtà, la zona di Minix è l'entità del file system che più si avvicina al blocco dei sistemi Unix tradizionali.

## 68.6.2 Il super blocco

Le unità di memorizzazione possono essere organizzate in due modi: con partizioni o senza. Una partizione è a sua volta come un'unità singola, non divisa in partizioni.

L'inizio di un'unità di memorizzazione viene riservato generalmente per il codice di avvio del sistema operativo, tenendo conto che questo vale sia per le unità suddivise in partizioni, sia per quelle non suddivise, sia per le partizioni stesse. Pertanto, nessun file system sovrascrive il primo settore di un'unità, anche se può considerarlo parte della propria gestione.

Dopo lo spazio che viene lasciato per il codice di avvio del sistema operativo (di uno o più settori), si colloca generalmente quello che è noto come *super blocco*, il quale si può considerare come una tabella riassuntiva delle caratteristiche generali del file system e della sua situazione.

Le informazioni contenute nel super blocco devono consentire di sapere: qual è la dimensione del blocco (ammesso che questa non sia fissa); qual è la dimensione dell'unità in blocchi; dove sono le tabelle che rappresentano gli inode; quanti sono gli inode, quali sono quelli liberi e quali invece sono impegnati; quali sono i blocchi che possono essere utilizzati per i dati (file, directory e altre tabelle per i riferimenti indiretti) e quali invece sono impegnati.

Per conoscere quali sono gli inode e le zone libere o impegnate, si possono usare sistemi diversi. In generale è probabile che si usino mappe di bit (come nel caso di Minix), oppure delle liste. Ma in generale, ciò che consente di sapere e di annotare gli inode e le zone impegnate o libere, fa parte concettualmente del super blocco, anche

se materialmente può trattarsi di strutture di dati separate.

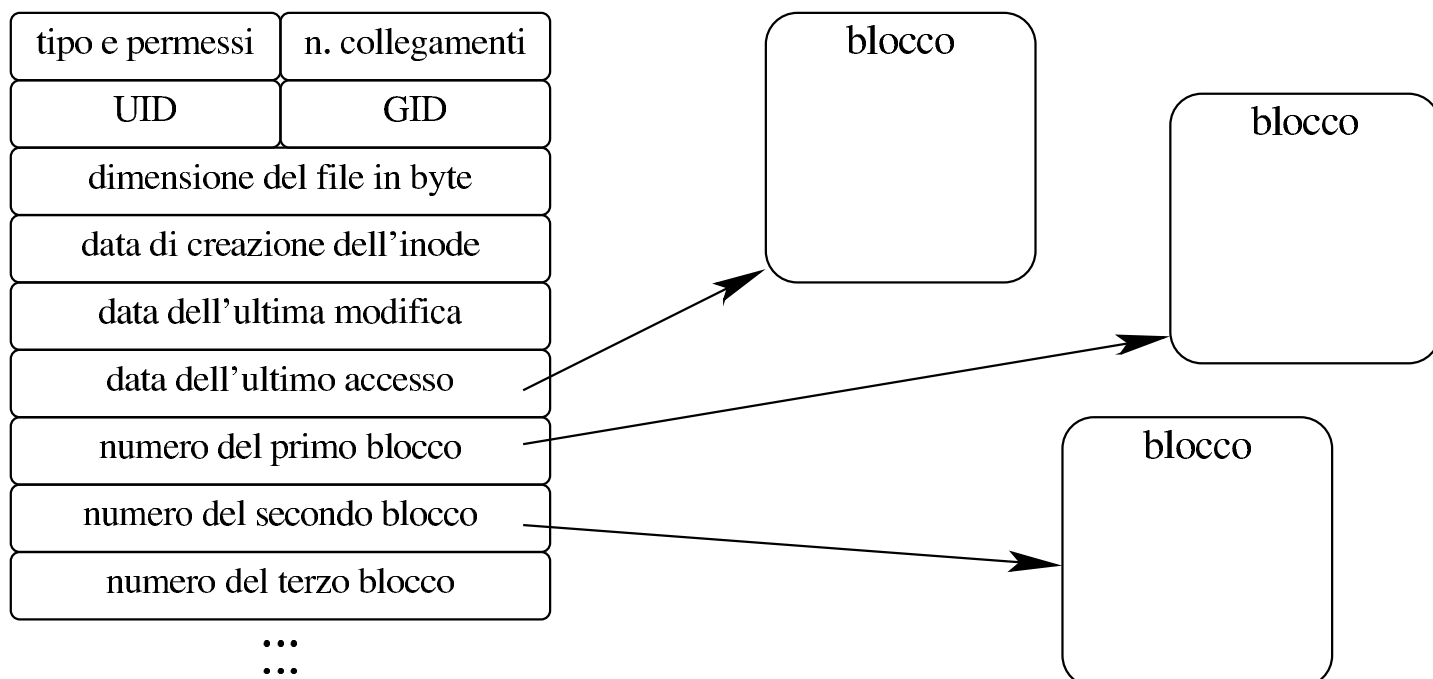
### 68.6.3 Gli inode

«

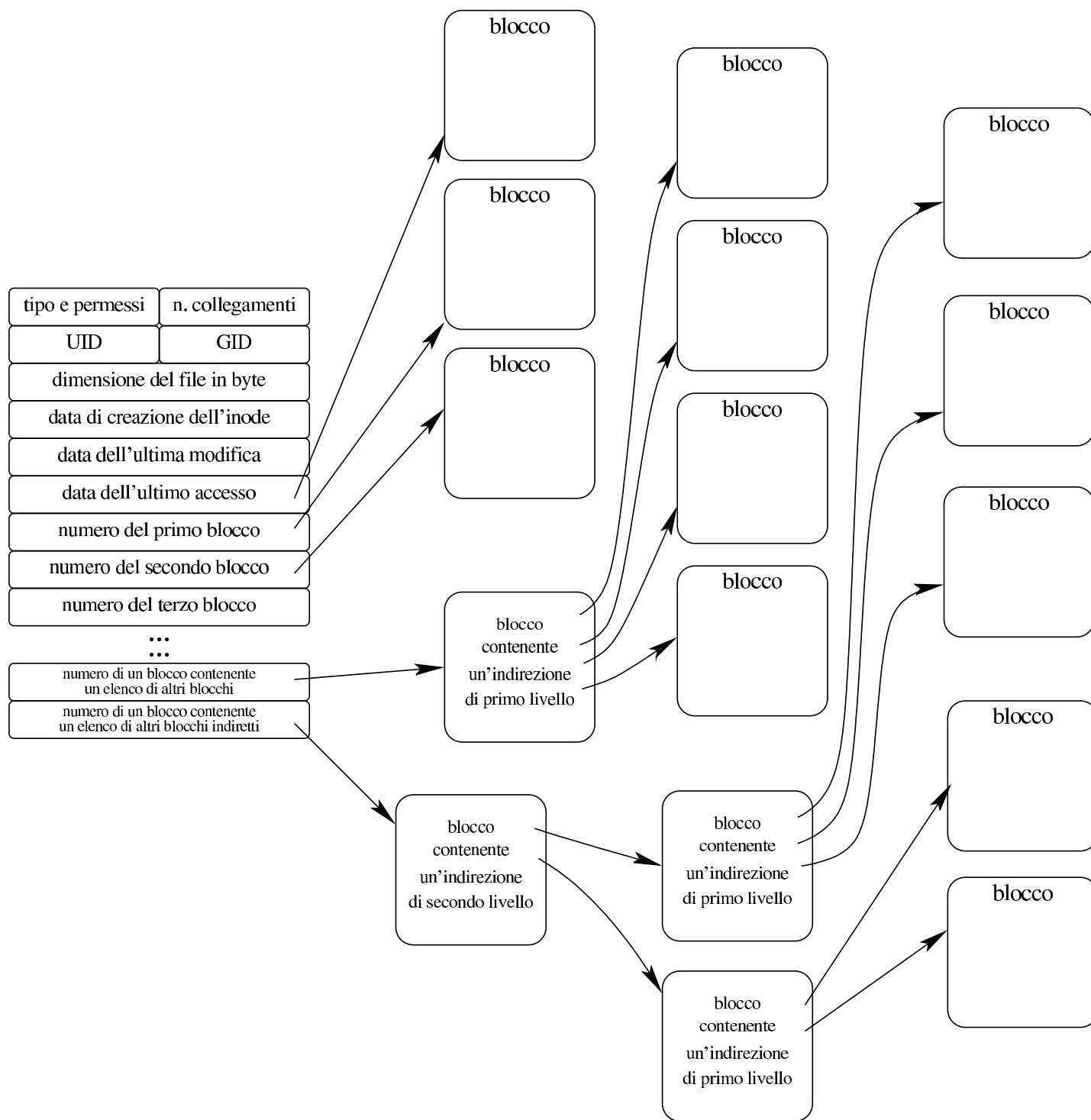
Dopo il super blocco, includendo in questo anche ciò che consente di sapere quali inode e quali blocchi sono impegnati o liberi, si collocano generalmente delle tabelle che rappresentano, ognuna, un inode. Di solito, un inode occupa un sottomultiplo dello spazio di un blocco, in modo da facilitare i calcoli per individuarne la collocazione.

Un inode contiene almeno queste informazioni: il tipo di file e i permessi di accesso, in un valore a 16 bit; il numero UID del proprietario del file; il numero GID del gruppo proprietario; la dimensione del file (o della directory) espressa in byte; le date di accesso, modifica del file e creazione dell'inode; la quantità di riferimenti provenienti dalle directory; una serie di numeri di blocchi occupati dal file o dalla directory.

Il contenuto del file rappresentato dall'inode si articola in blocchi, i quali non sono necessariamente contigui e nemmeno ordinati: è l'insieme dei riferimenti contenuti nell'inode che determina la posizione e l'ordine in cui questi vanno considerati. Inoltre, dato che non è possibile allocare nel file system uno spazio più piccolo di un blocco, è indispensabile l'informazione sulla dimensione del file per sapere quando questo termina nel suo ultimo blocco utilizzato.



Dal momento che deve essere possibile rappresentare file di grandi dimensioni, i primi riferimenti ai blocchi utilizzati sono diretti, come si vede nel disegno appena mostrato, mentre si prevedono generalmente dei riferimenti indiretti, a blocchi che contengono a loro volta i riferimenti di altri blocchi. In tal caso si parla di «indirezione» di primo, secondo ed eventualmente anche di terzo livello.



In questa struttura di collegamenti ai blocchi, va osservato che un file potrebbe non avere allocato tutti blocchi che risulterebbero dalla dimensione riportata. Per esempio, utilizzando blocchi da 1024 byte, un file che risulta essere grande 10240 byte, non è detto che occupi effettivamente 10 blocchi come sembrerebbe. Infatti, la scrittura nel



file potrebbe essere avvenuta specificando una piccola parte verso la fine e in altre posizioni. In pratica, almeno in linea teorica, questo tipo di organizzazione a inode, consente di scrivere il file dove si vuole, sapendo che lo spazio intermedio, se non viene allocato, risulta contenere dei dati con bit a zero.

Nella tabella che rappresenta l'inode, i blocchi non allocati risultano indicati con il numero zero, pertanto, il vero blocco zero non può essere accessibile attraverso gli inode (ma d'altra parte è normale che il blocco zero sia impegnato dal codice di avvio, dal super blocco ed eventualmente dalle tabelle degli inode stessi).

È importante osservare che in un file system Unix non esiste mai l'inode con il numero zero, perché questo valore viene utilizzato per fare riferimento a un errore o comunque a situazioni speciali. Di solito, l'inode numero uno corrisponde alla directory principale, dell'unità presa in considerazione (ovvero di una sua partizione).

#### 68.6.4 La directory

La directory è un file come gli altri, riconoscibile perché, nell'inode, il campo che definisce il tipo e i permessi, riporta l'indicazione relativa. Nei file system tradizionali, il file che rappresenta la directory è formato normalmente da *record* di lunghezza uniforme, in cui si distingue un campo contenente il numero di un inode e un altro contenente il nome di un file.



n. inode	nome
	•
	••
	<b>cat</b>
	<b>cp</b>
	<b>dd</b>

In pratica, si associa il tale inode a un certo nome. Va ricordato che l'inode zero non esiste e che, di norma, l'inode uno è quello della directory radice dell'unità di memorizzazione (o della partizione relativa).

Nel file di intestazione `'limits.h'`, la macro-variabile ***NAME\_MAX*** rappresenta la quantità minima di caratteri che possono essere usati per i nomi dei file (nelle directory). Questa quantità non include il carattere nullo di terminazione delle stringhe; pertanto, se corrispondesse al valore 14, vorrebbe dire che i nomi possono avere effettivamente 14 caratteri. Ciò avviene a differenza della macro-variabile ***PATH\_MAX***, per i percorsi, la quale deve invece includere anche il carattere nullo di terminazione.

La directory si costruisce come descritto, ma rimane il fatto che le prime due voci debbano essere «.» e «..»: la prima corrispondente al riferimento dell'inode della directory stessa; la seconda corrispondente al riferimento dell'inode della directory genitrice (ovvero quella precedente in senso gerarchico), con la variante che la directory radice può solo puntare a se stessa, in ogni caso.

## 68.6.5 Tabelle del sistema operativo

Il file system dei sistemi Unix, oltre che avere una certa forma nella fisicità dell'unità di memorizzazione, ha anche una rappresentazione astratta tradizionale nel sistema operativo, nel modo in cui si prendono in considerazione i file aperti e ciò da cui questi dipendono.

Nella semplificazione dei sistemi tradizionali, si utilizzano delle tabelle per: i super blocchi delle unità innestate; gli inode in corso di utilizzazione; i file aperti; i descrittori dei file aperti; i flussi di file abbinati ai descrittori.

### 68.6.5.1 Tabella dei super blocchi

Per accedere a un file, è necessario poter raggiungere il file system di un certo dispositivo, il quale deve essere stato innestato, ovvero reso disponibile nel file system generale del sistema operativo. Per raccogliere la situazione delle unità innestate serve una tabella, la quale può essere vista come quella dei dispositivi o dei super blocchi.

Le voci della tabella dei super blocchi riproducono i super blocchi delle unità innestate, incluse le mappe o le tabelle di utilizzo degli inode e dei blocchi, oltre ad altre informazioni accessorie. Quando si crea o si elimina un file, quando lo si estende e lo si riduce, la voce relativa di tale tabella dei super blocchi va aggiornata anche nell'unità di memorizzazione, per quanto riguarda la situazione di utilizzo degli inode e dei blocchi di dati.

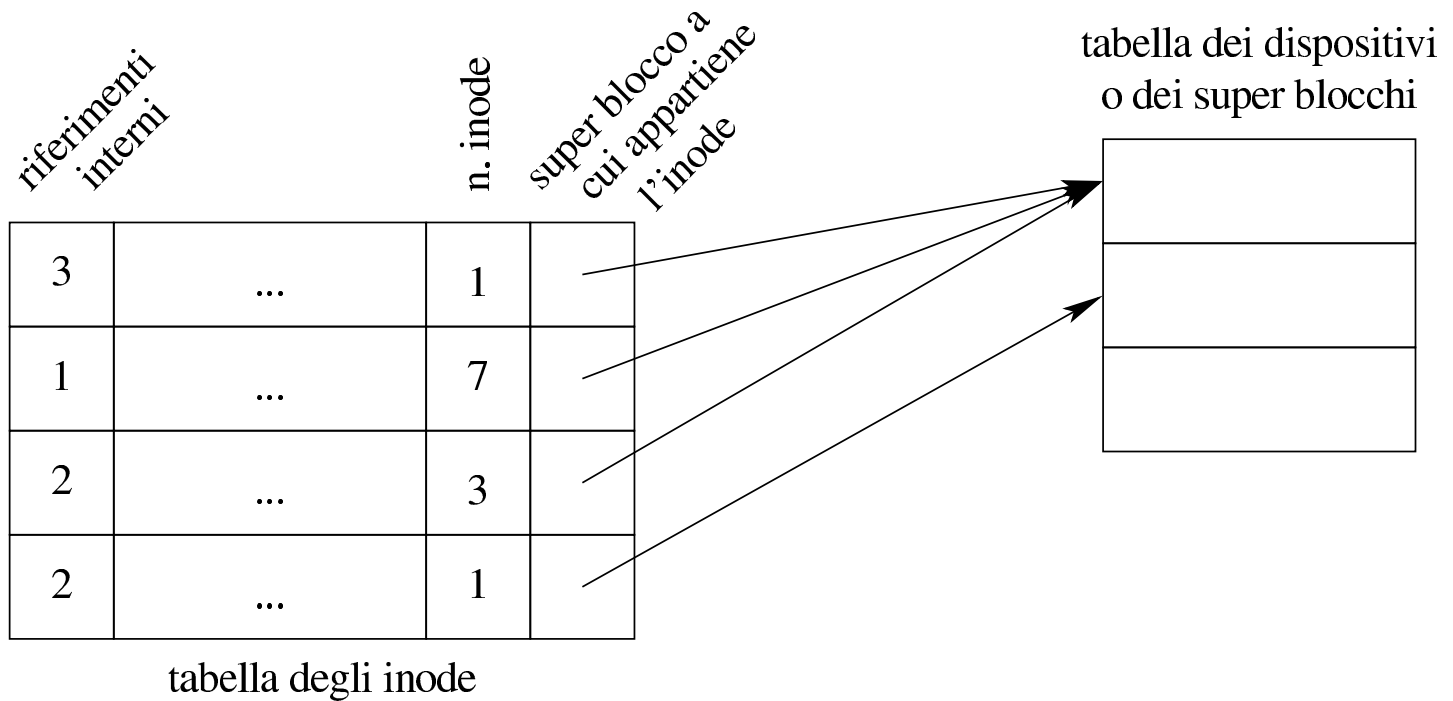
Va osservato che il sistema operativo deve innestare almeno una unità, contenente il file system principale, pertanto, almeno un super blocco deve essere sempre presente nella tabella.

## 68.6.5.2 Tabella degli inode



Quando si accede a un file per la prima volta, le informazioni relative al suo inode vengono caricate in una voce della tabella degli inode. Il contenuto minimo di questa voce è costituito di norma da tutti i dati dell'inode contenuti nel file system, incluso il numero di questo, aggiungendo il riferimento alla voce che rappresenta il super blocco da cui proviene e la quantità di riferimenti interni (i riferimenti interni non vanno confusi con i collegamenti nel file system, provenienti dalle directory).

Quando si apre più volte lo stesso file, o comunque ciò che fa capo allo stesso inode, nella tabella di inode si ha sempre solo una voce, dove il contatore dei riferimenti interni serve a sapere quante volte risulta aperto. Quando poi tale contatore arriva a zero, perché i file vengono chiusi mano a mano, la voce della tabella è libera e può essere riutilizzata per un altro inode, oppure può essere semplicemente ripresa così come si trova, se il file viene riaperto (incrementando nuovamente il contatore).



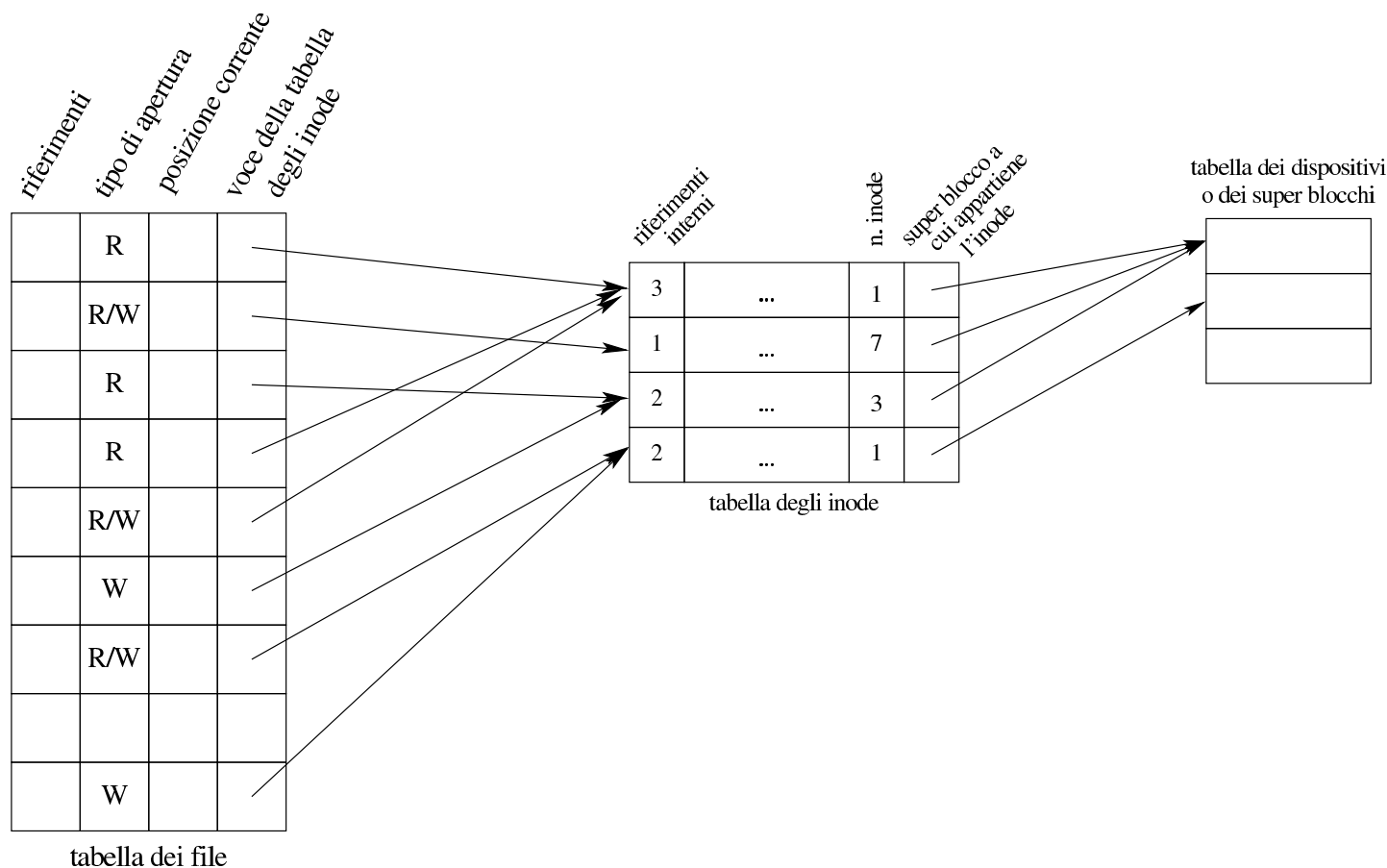
Va però osservato che, durante il funzionamento del sistema operativo, potrebbe farsi riferimento a inode astratti, privi del collegamento a un dispositivo o super blocco. Questo caso riguarda in particolare i condotti «privi di nome», ovvero quelli che non derivano dall'apertura di un file speciale di tipo FIFO.

### 68.6.5.3 Tabella dei file

L'apertura di un file, oltre che coinvolgere la tabella degli inode, implica l'aggiunta di una voce nella tabella dei file di sistema (ovvero dei file aperti complessivamente nel sistema operativo). Le voci di questa tabella devono avere un riferimento all'inode, la modalità di apertura (lettura, scrittura o entrambe), la posizione corrente nel file per le letture o le scritture successive, un contatore di riferimenti interni.

L'apertura di un file implica sempre l'aggiunta di una nuova voce nella tabella dei file, ma ci sono delle situazioni in cui uno stesso

processo o più processi differenti possono condividere la stessa voce della tabella dei file di sistema. Come nel caso degli inode, quando il contatore dei riferimenti raggiunge lo zero, significa che la voce corrispondente è chiusa (o libera) e può essere riutilizzata per il prossimo file da aprire.

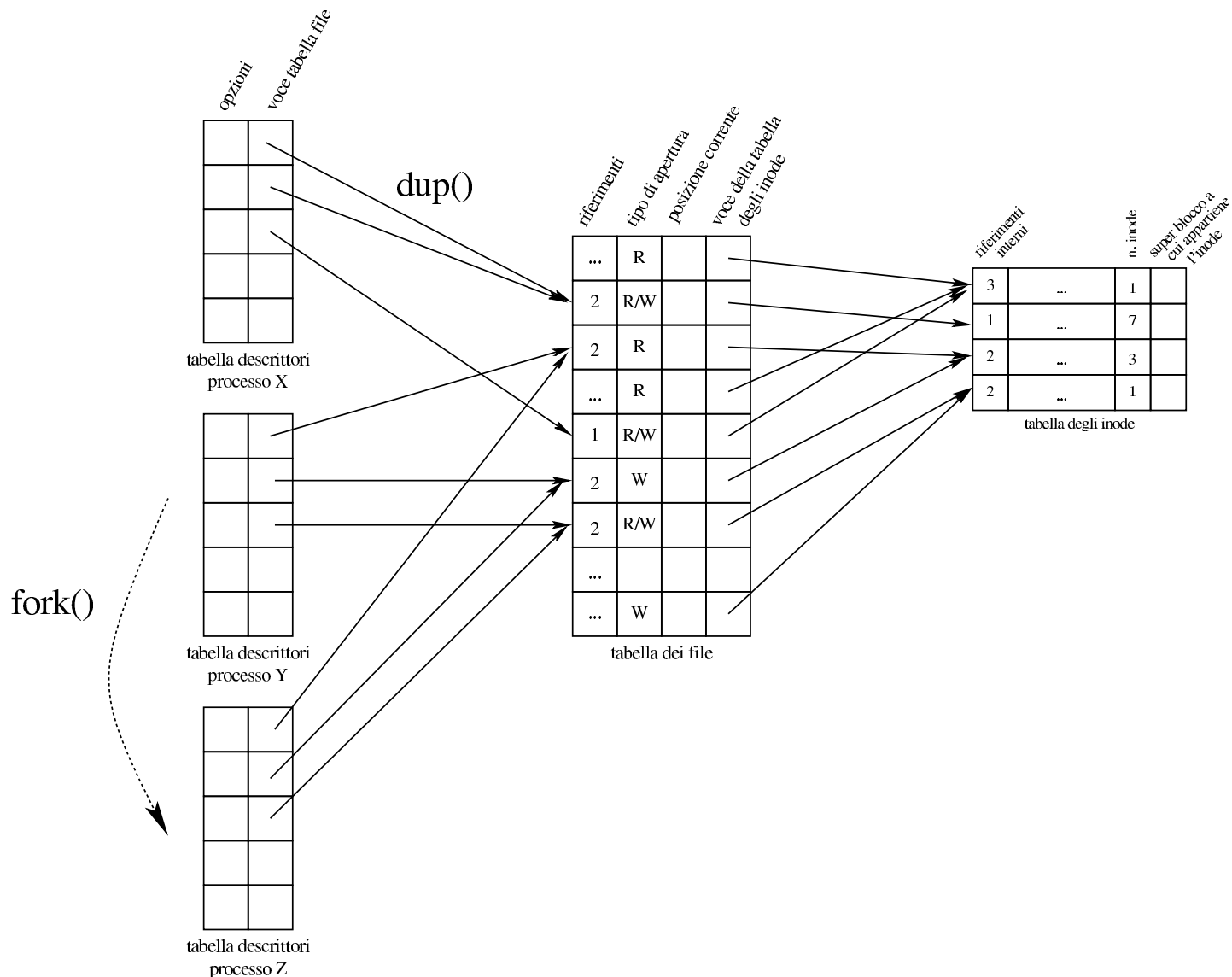


#### 68.6.5.4 Tabella dei descrittori dei file

«

Ogni processo elaborativo ha una propria tabella dei descrittori dei file, nella quale, le voci rappresentano i file aperti dal processo stesso. Le voci della tabella includono il riferimento alla tabella dei file di sistema e le opzioni date in fase di apertura, riguardanti aspetti più precisi rispetto alla semplice distinzione di un accesso in lettura o in scrittura.

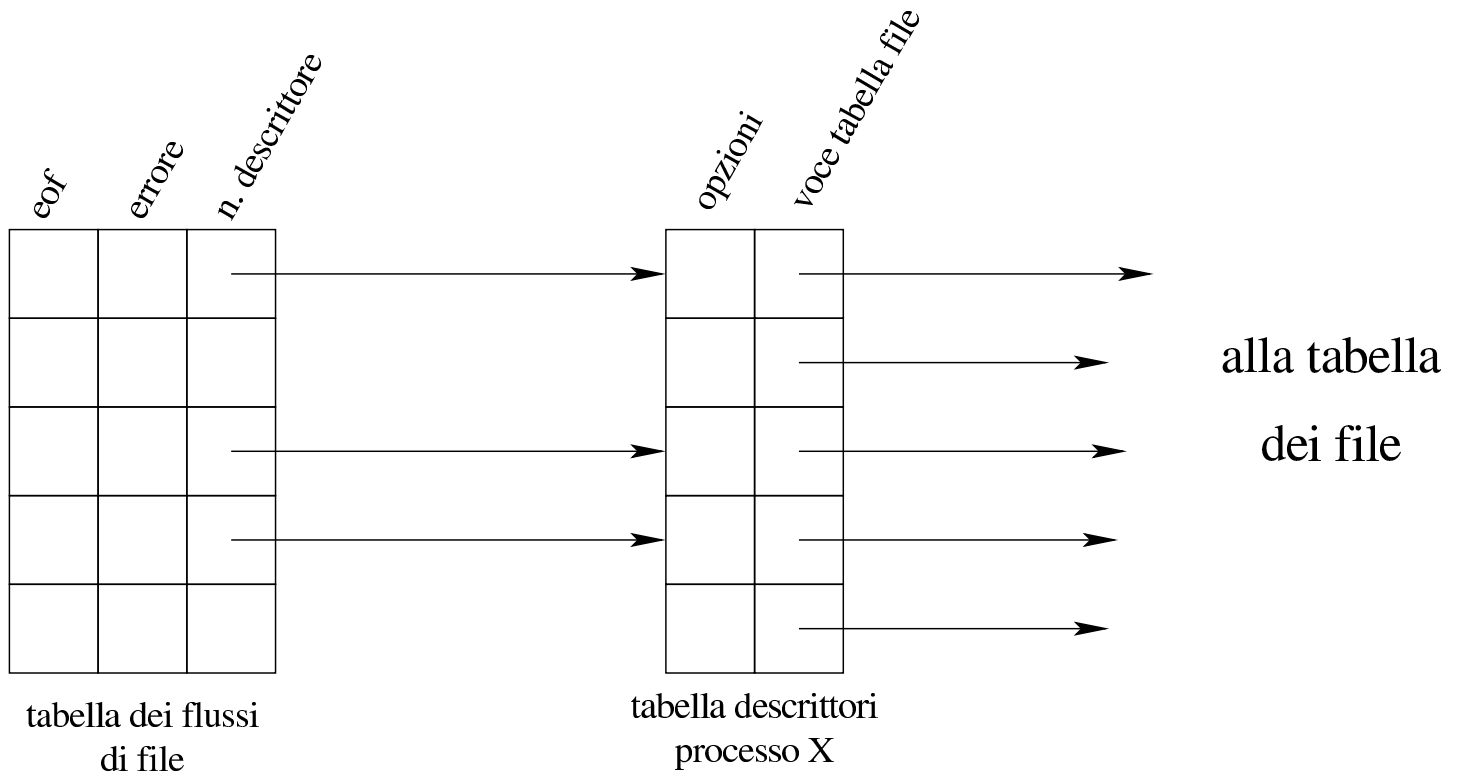
Quando un processo elaborativo si sdoppia, attraverso la chiamata di sistema che fa capo alla funzione *fork()*, i descrittori dei file vengono duplicati e i riferimenti corrispondenti nella tabella dei file di sistema si incrementano. Nello stesso modo, se un processo elaborativo utilizza la funzione della libreria standard *dup()*, ottiene la duplicazione di un descrittore, incrementando il contatore dei riferimenti nella tabella dei file.



## 68.6.5.5 Flussi di file

«

Dal punto di vista del processo elaborativo, la gestione dei file in forma di descrittori o di flussi, potrebbe sembrare indipendente, ma in pratica ciò non può essere. La gestione dei flussi di file implica la presenza di una tabella aggiuntiva (oltre a quella dei descrittori), contenente, per ogni voce, il riferimento al descrittore, un indicatore di errore e un altro indicatore di fine file.



Come suggerisce intuitivamente il disegno, in un sistema operativo POSIX, un flusso di file aperto ha un proprio descrittore di file corrispondente, anche se nell'ambito del programma può rimanere sconosciuto il numero del descrittore abbinato. Tuttavia, per converso, è possibile aprire un file attraverso un descrittore, senza che sia coinvolto necessariamente il flusso che gli corrisponderebbe. A tale proposito, lo standard prescrive la presenza di funzioni che consentono di ristabilire il collegamento esplicito tra flussi e descrittori.



## 68.7 Il file system Minix 1

Come esempio di come può essere strutturato effettivamente un file system, conforme alle richieste dello standard POSIX, viene proposta la spiegazione dettagliata del tipo usato dal sistema operativo Minix, nelle sue primissime edizioni.

Il kernel Linux consente di accedere a file system Minix 1, eventualmente con l'estensione dei nomi a 30 byte, mentre manca una gestione efficace del file system Minix 2 e manca del tutto la possibilità di accedere alla versione Minix 3.

### 68.7.1 Blocchi e zone

Il file system Minix 1 suddivide lo spazio disponibile in *blocchi* da 1024 byte; così, qualunque oggetto sia memorizzato occupa un multiplo di tale dimensione. Di solito, le unità di memorizzazione di massa sono organizzate in settori da 512 byte, pertanto tale organizzazione in blocchi si adatta perfettamente alle unità comuni.

Per l'indirizzamento dei dati, all'interno del file system, si utilizza il concetto di *zona*, corrispondente a un multiplo del blocco, ottenuto però come potenza di due. Pertanto possono esserci zone della stessa dimensione dei blocchi, oppure doppie, quadruple,... In pratica, deve essere possibile rappresentare con un numero intero, il logaritmo in base due del rapporto tra la dimensione della zona e la dimensione del blocco.

$$\log_2 \frac{\text{dim\_zona}}{\text{dim\_blocco}}$$

Per esempio, una zona da 8192 byte, porta a un rapporto tra zona e blocco di 8 e  $\log_2 8$  è pari a 3 (in quanto  $2^3 = 8$ ).

Il valore del logaritmo in base due, del rapporto tra zona e blocco, fa parte delle informazioni contenute nel file system Minix 1, perché serve a ottenere la dimensione della zona, attraverso lo scorrimento a sinistra del valore 1024. Per esempio così:

```
y = 1024 << 3;
```

In tal caso, la variabile  $y$  va a contenere il valore 8192. Ma in alternativa, basta calcolare i multipli di blocco, ottenendo così il valore 8:

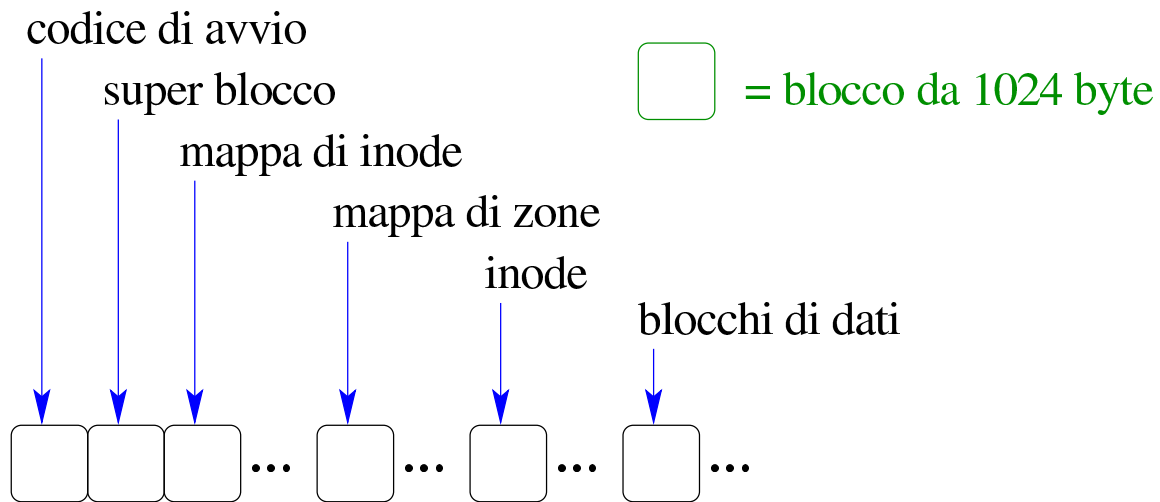
```
y = 1 << 3;
```

Nella tradizione Unix non esiste la «zona», la quale riguarda specificatamente il file system Minix. La zona di Minix rappresenta in pratica il concetto di «blocco» dei sistemi Unix.

## 68.7.2 Struttura generale

«

La struttura generale del file system Minix 1 è schematizzata dalla figura successiva. Il primo blocco (da 1024 byte) è riservato al codice di avvio, anche se di solito questo occupa soltanto un settore da 512 byte. Il secondo blocco contiene il «super blocco», ovvero l'instestazione del file system, con le informazioni generali sullo stesso. Il terzo blocco, ed eventuali altri blocchi successivi, sono utilizzati per una mappa degli inode, la quale ha lo scopo di annotare quali sono utilizzati e quali sono liberi. A partire dal blocco successivo inizia la mappa delle zone utilizzate (zone, intese come multipli dei blocchi, come spiegato nella sezione precedente). Dopo la mappa delle zone appaiono i blocchi contenenti gli inode (tanti quanti sono previsti nella mappa di inode). Successivamente appaiono i blocchi usati dalle zone di dati che utilizzano lo spazio rimanente.



Va osservato che se le zone hanno una dimensione maggiore dei blocchi, il primo blocco utile per la memorizzazione dei dati (dopo gli inode) deve iniziare all'inizio di una zona, contando le zone a partire dal primo blocco (quello riservato dal codice di avvio). Pertanto, potrebbero rimanere anche blocchi non utilizzabili, dopo quelli delle mappe e prima di quelli dei dati.

### 68.7.3 Super blocco

Il super blocco raccoglie le informazioni più importanti del file system e dalla sua integrità dipende l'accessibilità di tutto il resto del contenuto presente. Anche se gli viene riservato un blocco intero, in pratica, il super blocco di Minix 1 occupa molto meno spazio.



Il secondo campo del super blocco, come si vede dalla figura, rappresenta la dimensione dell'unità di memorizzazione (o della partizione considerata), espressa in zone. Per esempio, se si utilizzano zone uguali ai blocchi, un dischetto da 1440 Kibyte è composto esattamente da 1440 zone.

Il quinto campo indica la prima zona dati, contando a partire da zero. La prima zona dati è la prima zona che possa essere usata, dopo gli inode. Nella mappa delle zone utilizzate, il bit che rappresenta la zona dati numero uno, si riferisce a questa prima zona dati (nella mappa delle zone, la zona dati zero risulta sempre utilizzata ma in realtà non esiste).

Il sesto campo indica il logaritmo in base due, del rapporto tra di-

menzione della zona e del blocco. Per esempio, un valore pari a zero indica che la zona è uguale al blocco; uno indica che la zona è costituita da due blocchi; tre indica che la zona è composta da quattro blocchi e così di seguito.

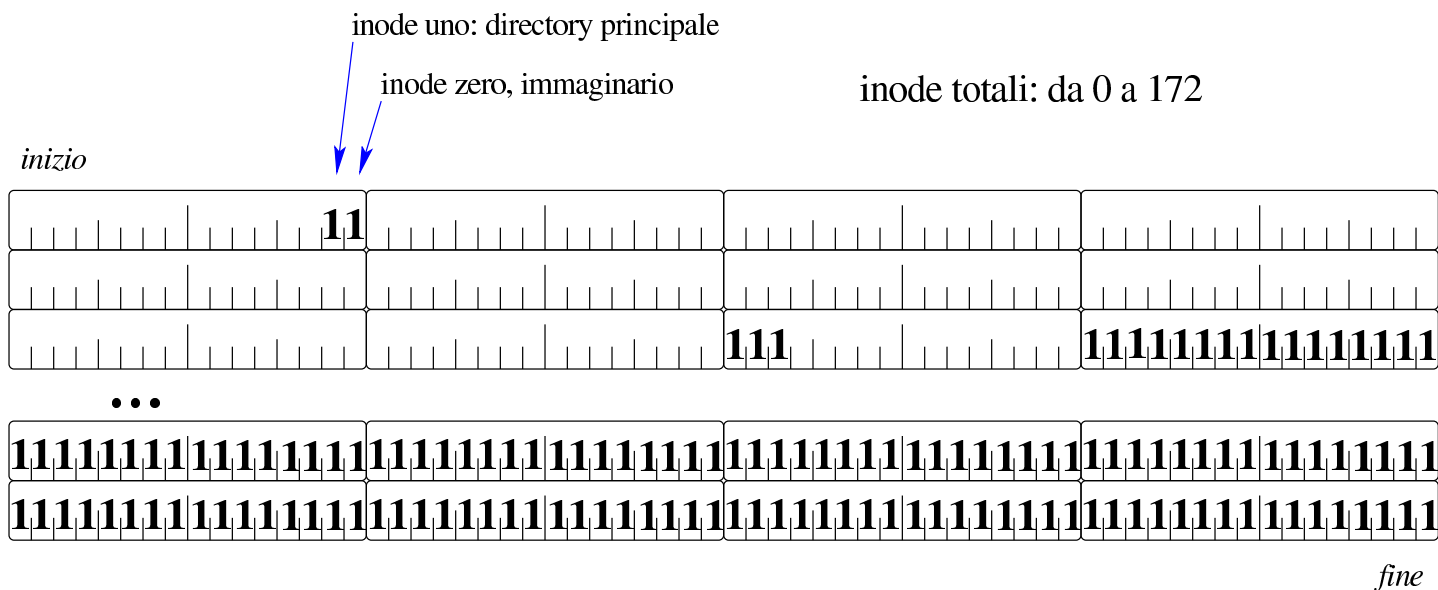
Il «numero magico» è il codice di riconoscimento, usato per verificare che si tratti effettivamente di un file system Minix 1. Tale numero deve essere  $137F_{16}$ .

#### 68.7.4 Mappa di inode

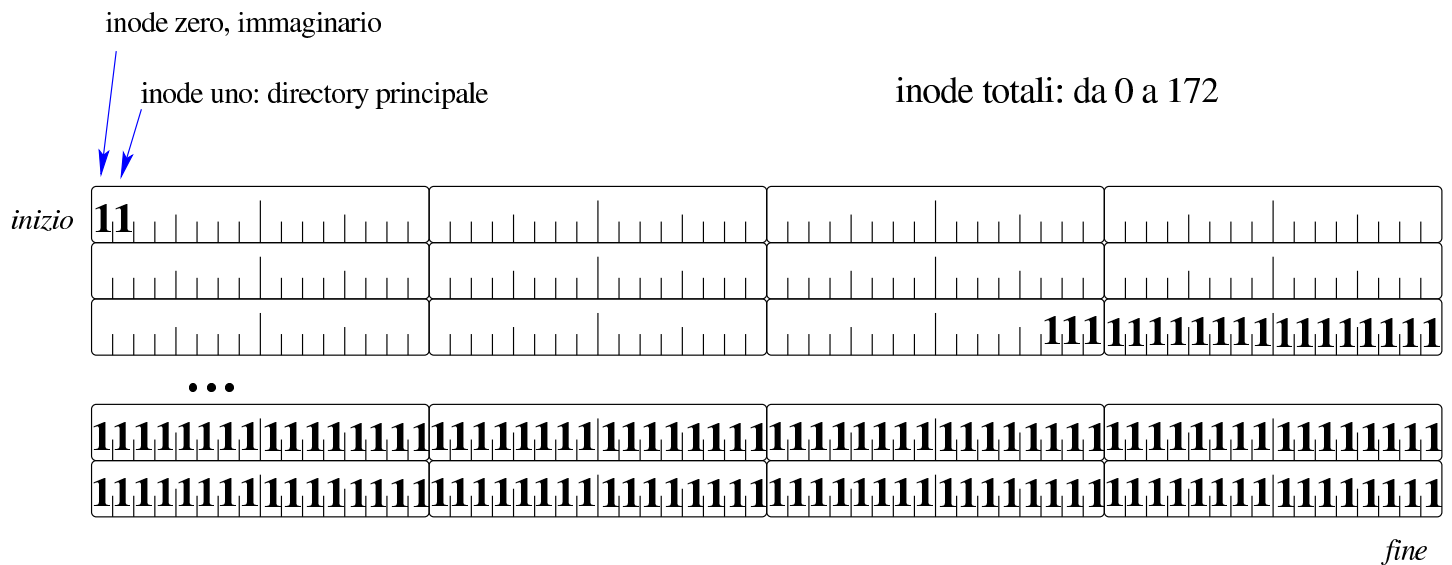
La mappa di inode è costituita da un insieme di bit, ognuno dei quali rappresenta lo stato di utilizzazione di un certo inode: 1 indica un inode utilizzato; 0 indica un inode libero. In questa mappa, il primo bit, riferito all'inode zero, è sempre attivo. Ma l'inode zero, in pratica, non viene rappresentato e il primo vero inode, ovvero quello riferito alla directory radice, ha sempre il numero uno. «

In base al fatto che l'inode zero, in pratica, non esiste, anche se risulta sempre utilizzato, va considerato che il valore presente nel primo campo del super blocco indica la quantità reale di inode, ovvero l'indice massimo (partendo da zero) che si può utilizzare nella loro scansione.

La mappa di bit va però scandita, suddividendola a blocchi da 16 bit. L'esempio seguente rappresenta una mappa per 172 inode, dove si vede il primo (zero) impegnato e il secondo che già è predisposto per la directory principale:



Nel disegno sono rappresentati solo i bit a uno, lasciando gli altri come spazi vuoti. Va osservato che l'ordine in cui si dispongono i bit non è quello che ci si aspetterebbe: il primo, quello dell'inode zero, appare a destra del suo insieme di 16 bit (si tratta quindi del bit meno significativo), mentre lo si attenderebbe a sinistra secondo il senso di lettura latino. Considerato che i bit inutilizzati vanno posti a uno, come se esistessero altrettanti inode impegnati, l'ultimo insieme di 16 bit va interpretato con attenzione. Per avere una visione più umana della mappa, occorrerebbe invertire la sequenza di bit in ogni gruppetto:



Al problema dell'inversione della sequenza di bit, si aggiunge il fatto che il file system è nato per un'architettura *little endian*, ovvero a byte invertiti, ma la questione viene trattata alla fine del capitolo.

Per sapere dove si trova un certo inode  $n$ , occorre considerare che questi si collocano dopo i blocchi della mappa di zone, che il primo vero inode è quello con indice uno, ovvero il secondo, in base alla numerazione della mappa. Come viene descritto successivamente, ogni inode occupa 32 byte, pertanto, in ogni blocco ci stanno esattamente 32 inode.

### 68.7.5 Mappa di zone

La mappa delle zone dei dati, funziona come quella di inode, dove i bit a uno indicano una zona utilizzata e il primo bit, riferito alla zona zero, è sempre a uno, ma in realtà la prima vera zona dati è quella a cui corrisponde l'indice uno. Come per la mappa di inode, anche in questo caso valgono le stesse considerazioni relative al fatto che la scansione deve essere fatta a gruppi di 16 bit e che il conteggio inizia dalla parte numericamente meno significativa di tali gruppi.

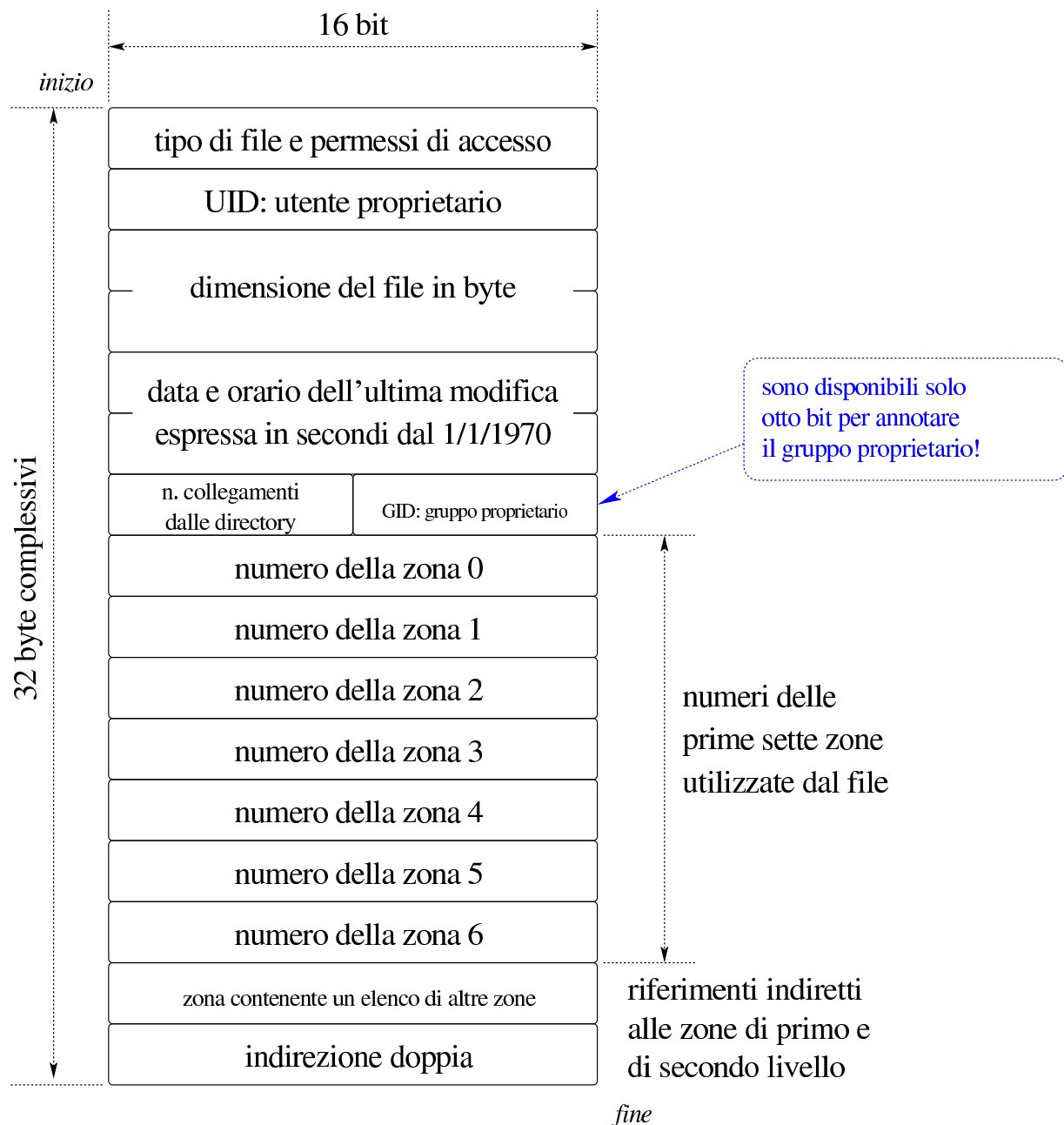
È bene precisare che la mappa si riferisce alle zone dei dati, pertanto riguarda quelle zone che iniziano dopo tutte le informazioni già descritte, compresa la stessa mappa e la tabella di inode successiva. Per fare un esempio, se nel super blocco è scritto che la prima zona dati è quella con il numero 19, significa che il bit con indice uno della mappa (il secondo) individua la zona 19 e le zone precedenti non possono essere utilizzate per i dati.

### 68.7.6 Inode

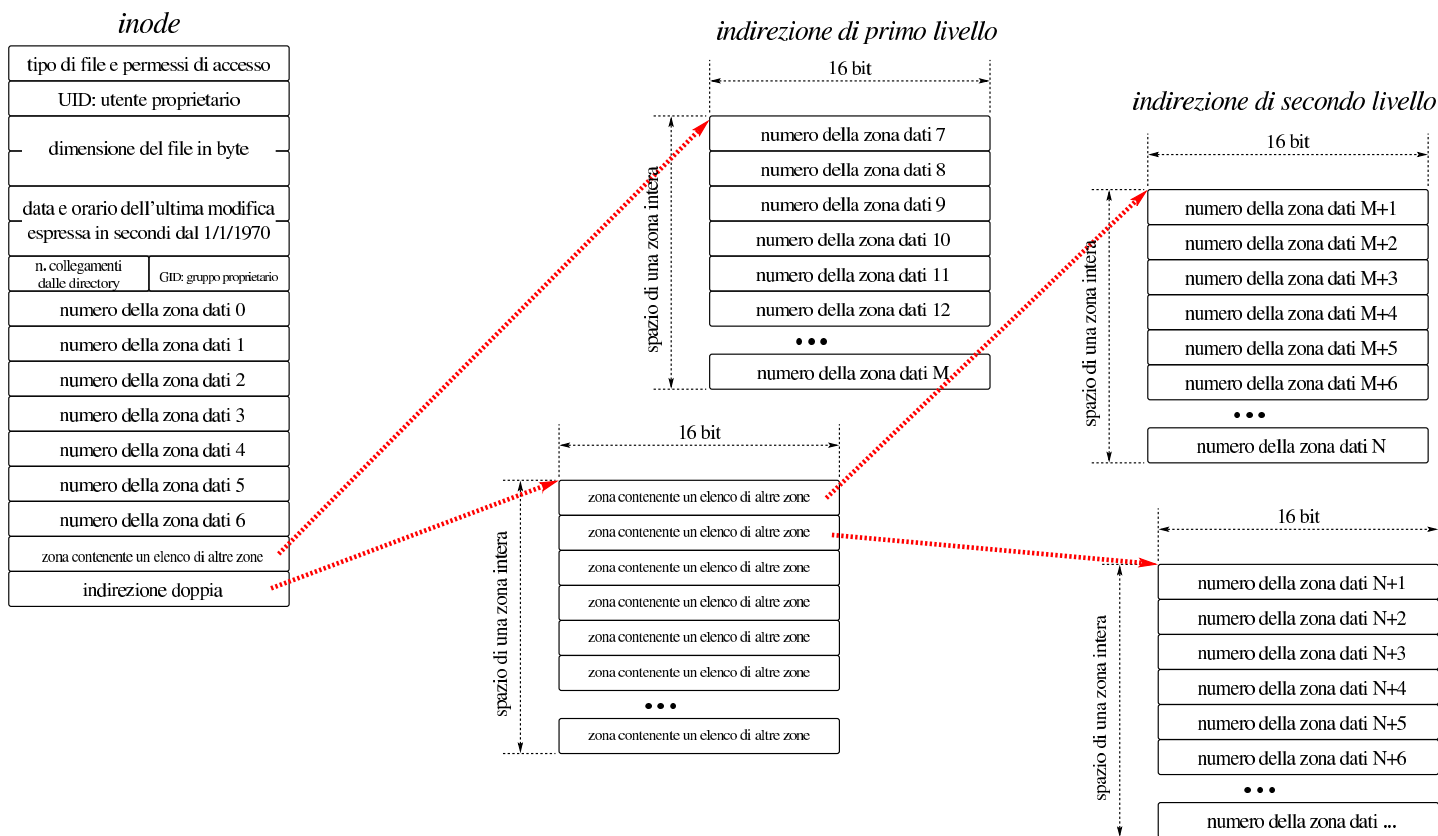
«

I blocchi successivi alla mappa delle zone dei dati, sono utilizzati per gli inode, di cui si conosce la quantità, perché questa è annotata nel super blocco. Nel file system Minix 1, ogni inode occupa esattamente 32 byte.





La figura successiva mostra il meccanismo usato per indirizzare file che occupano più di sette zone, attraverso elenchi aggiuntivi, ognuno dei quali occupa a sua volta una zona intera. I riferimenti indiretti alle zone possono essere quindi di primo livello, o di secondo livello, come suggerito dalla figura stessa.

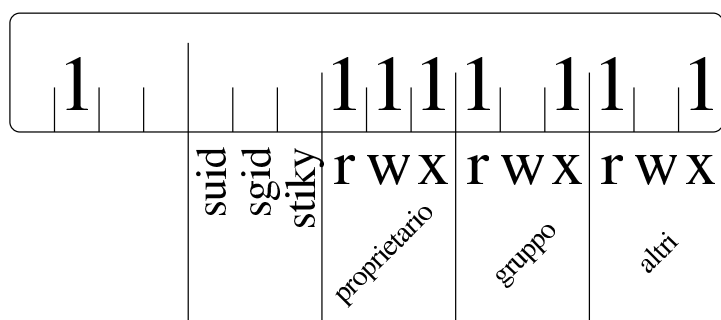


Ipotizzando di avere zone della stessa dimensione dei blocchi (1024 byte), dal momento che gli elenchi indiretti possono contenere a loro volta 512 numeri di zona, sarebbe possibile gestire file con una dimensione massima di  $7+512+512 \times 512$  Kibyte, ovvero 262663 Kibyte. Disponendo di zone della dimensione di quattro blocchi, si potrebbero gestire file da  $7+2048+2048 \times 2048$  Kibyte, ovvero 4196359 Kibyte. Con lo stesso criterio, con zone da otto blocchi, si potrebbero gestire file da poco più di 16 Gibyte; con zone da 16 blocchi si arriverebbe a poco più di 64 Gibyte. A questi limiti si aggiunge però il fatto che le zone sono individuate da numeri a 16 bit; pertanto, con zone da un solo blocco, si possono indirizzare al massimo 65536 Kibyte, ovvero 64 Mibyte; con zone da due blocchi si arriva a 128 Mibyte; con zone da 16 blocchi si arriva al massimo a 1 Gibyte. Pertanto, la doppia «indirizzazione» può essere usata solo parzialmente e non avrebbe senso un'indirizzazione tripla.

A parte la limitazione nella dimensione dei file, va annotato un fatto che può risultare più spiacevole: il numero del gruppo proprietario del file (GID) viene rappresentato con soli 8 bit. Ciò significa che si possono indicare gruppi fino al numero 255 e in pratica, quando vi si copia un file, il numero del gruppo viene troncato nella parte più significativa. Un altro limite importante riguarda il fatto che l'inode riporti solo la data di modifica del file, mancando così la data di accesso e la data di creazione dell'inode stesso.

Il primo campo da 16 bit di un inode, rappresenta il tipo e i permessi del file a cui si riferisce l'inode (si veda anche la sezione [70.2.1](#) a proposito della «modalità» POSIX). L'interpretazione di questo valore deve avvenire secondo gli standard dei sistemi POSIX, ovvero secondo lo schema seguente, dove si ipotizza una directory con permessi di accesso e di lettura per tutti gli utenti:

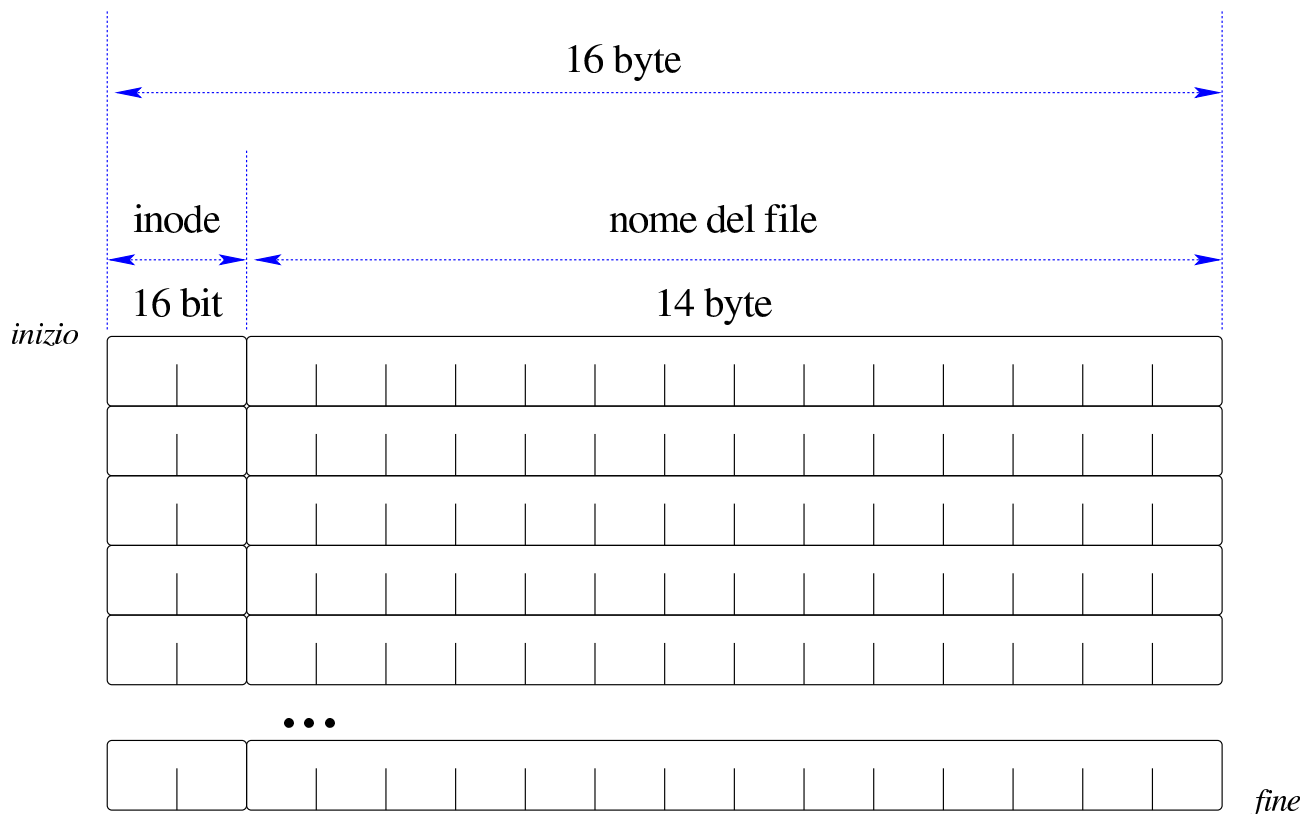
0 0 0 1	file FIFO
0 0 1 0	dispositivo a caratteri
0 1 0 0	directory
0 1 1 0	dispositivo a blocchi
1 0 0 0	file normale
1 0 1 0	collegamento simbolico
1 1 0 0	socket di dominio Unix



## 68.7.7 Directory

«

Le directory si collocano nelle zone dei dati, come gli altri file, e a loro si accede a partire da un inode (come per tutti gli altri file). La struttura di una directory è molto semplice, come si può vedere nella figura successiva:



In pratica, si tratta di un file suddiviso in *record* a dimensione fissa da 16 byte, dove i primi due byte rappresentano il numero inode del nome che occupa i restanti 14 byte. Il file system Minix 1 consente così di rappresentare nomi di file fino a un massimo di 14 caratteri, ma nei sistemi GNU/Linux si utilizza spesso un'estensione con directory aventi *record* da 32 byte, in modo da poter rappresentare nomi fino a 30 caratteri.

La directory è un file come gli altri, pertanto, per sapere quante sono le voci che la compongono, occorre conoscere la dimensione del file, come annotato nel suo inode. Per esempio, una directory con quattro voci (inclusi i nomi '.' e '..'), occupa 64 byte.

### 68.7.8 Il problema dell'inversione dei byte

«

Il sistema operativo Minix nasce negli anni 1980 per elaboratori a 16 bit con l'inversione dei byte (*little endian*). Per questa ragione, i byte che costituiscono l'organizzazione del file system Minix 1 sono invertiti. Ciò diventa un problema quando si legge il contenuto del file system in modo diretto, in esadecimale, perché tutte le voci che prevedono una rappresentazione a 16 o a 32 bit, vanno rovesciate in modo appropriato, per poterle interpretare correttamente. Per esempio, in un'altra sezione è stato descritto il modo in cui viene popolata la mappa degli inode e delle zone dei dati; ai problemi lì descritti si aggiungerebbe anche l'inversione dei byte.

Se si considera che un file system serve per scrivere dati anche su unità di memorizzazione rimovibili, utilizzabili presumibilmente su altri sistemi e altre architetture, sarebbe più appropriata una progettazione che preveda sempre la scrittura «ordinata» dei byte, come si fa per i dati trasmessi in rete. Nel caso particolare di Minix, con l'evolvere del sistema e con l'adattamento anche ad altre architetture, si è reso necessario considerare se il file system a cui si accede giunge ordinato secondo la propria architettura, oppure se per questa è inverso. In pratica, un numero magico  $137F_{16}$  indica che il file system va bene così; altrimenti, il numero  $7F13_{16}$  richiede che i valori a 16 e a 32 siano invertiti (byte per byte), per poter essere interpretati correttamente.

### 68.8 Creazione ed eliminazione di file di qualunque tipo

«

Un file «normale», definito in inglese come *regular file*, è il contenitore di una sequenza di byte, rappresentato in qualche modo nel

file system. Nei sistemi Unix, anche le directory sono dei file, benché si tratti evidentemente di un tipo speciale, per il quale si richiede un trattamento particolareggiato; inoltre, altri tipi di entità rientrano nella gestione complessiva del concetto di file per i sistemi Unix. Originariamente è stato usato il termine «nodo», da cui deriva il nome della funzione *mknod()*, con cui si poteva creare qualunque tipo di file.

### 68.8.1 La funzione «mknod()»

La funzione *mknod()*, dichiarata nel file di intestazione `'sys/stat.h'`, potenzialmente, è in grado di creare qualunque tipo di file, ma completamente vuoto, ammesso che si tratti di un tipo di file che ha un contenuto rappresentato nel file system. Teoricamente questa funzione potrebbe creare anche delle directory, ammesso che il sistema operativo lo consenta, ma si tratterebbe comunque di directory prive delle voci obbligatorie `'.'` e `'..'`, quindi si tratterebbe di directory incomplete ed errate per il file system.

```
int mknod (const char *path, mode_t mode, dev_t dev);
```

Il primo parametro della funzione è una stringa che rappresenta il percorso del file da creare nel file system; il secondo parametro, *mode*, individua il tipo di file ed eventualmente i permessi di accesso; l'ultimo parametro, *dev*, il numero del file di dispositivo, ammesso che si tratti della creazione di questo tipo di file. La tabella successiva elenca le macro-variabili da usare per comporre il valore del parametro *mode*, usando l'operatore OR binario, avendo la cura di specificare una sola macro-variabile per il tipo.

Tabella 68.76. Macro-variabili per esprimere, complessivamente il tipo e i permessi di un file.

Macro-variabile	Valore numerico equivalente	Significato
S_IFBLK	vedere 'sys/stat.h'	File di dispositivo a blocchi.
S_IFCHR	vedere 'sys/stat.h'	File di dispositivo a caratteri.
S_IFIFO	vedere 'sys/stat.h'	File FIFO.
S_IFREG	vedere 'sys/stat.h'	File normale ( <i>regular file</i> ).
S_IFDIR	vedere 'sys/stat.h'	Directory.
S_IFLNK	vedere 'sys/stat.h'	Collegamento simbolico.
S_IFSOCK	vedere 'sys/stat.h'	Socket di dominio Unix.
S_ISUID	4000 <sub>8</sub>	Rappresenta l'attivazione del bit S-UID.
S_ISGID	2000 <sub>8</sub>	Rappresenta l'attivazione del bit S-GID.
S_ISVTX	1000 <sub>8</sub>	Rappresenta l'attivazione del bit Sticky.
S_IRWXU	0700 <sub>8</sub>	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per l'utente proprietario.
S_IRUSR	0400 <sub>8</sub>	Rappresenta il permesso di lettura per l'utente proprietario.
S_IWUSR	0200 <sub>8</sub>	Rappresenta il permesso di scrittura per l'utente proprietario.



Macro-variabile	Valore numerico equivalente	Significato
S_IXUSR	0100 <sub>8</sub>	Rappresenta il permesso di esecuzione o attraversamento per l'utente proprietario.
S_IRWXG	0070 <sub>8</sub>	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per il gruppo proprietario.
S_IRGRP	0040 <sub>8</sub>	Rappresenta il permesso di lettura per il gruppo proprietario.
S_IWGRP	0020 <sub>8</sub>	Rappresenta il permesso di scrittura per il gruppo proprietario.
S_IXGRP	0010 <sub>8</sub>	Rappresenta il permesso di esecuzione o attraversamento per il gruppo proprietario.
S_IRWXO	0007 <sub>8</sub>	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per gli altri utenti.
S_IROTH	0004 <sub>8</sub>	Rappresenta il permesso di lettura per gli altri utenti.
S_IWOTH	0002 <sub>8</sub>	Rappresenta il permesso di scrittura per gli altri utenti.
S_IXOTH	0001 <sub>8</sub>	Rappresenta il permesso di esecuzione o attraversamento per gli altri utenti.

### 68.8.1.1 Creazione di un file «normale»

L'esempio seguente mostra l'uso della funzione *mknod()* per la creazione di un file comune: «

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
    int status;
    status = mknod ("/tmp/miofile",
                   (mode_t) (S_IFREG | S_IRUSR | S_IWUSR
                              | S_IXUSR),
                   (dev_t) 0);
    if (status != 0) perror (NULL);
    return (0);
}
```

### 68.8.1.2 Creazione di una directory

«

L'esempio seguente mostra un programma elementare che ha lo scopo di creare una directory vuota, priva anche delle voci obbligatorie '.' e '..'. In condizioni normali, il sistema operativo dovrebbe impedire tale azione, producendo un messaggio di errore.

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
    int status;
    status = mknod ("/tmp/miadir",
                   (mode_t) (S_IFDIR | S_IRUSR | S_IWUSR
                              | S_IXUSR),
                   (dev_t) 0);
    if (status != 0) perror (NULL);
    return (0);
}
```

### 68.8.1.3 Creazione di un file FIFO

Un file FIFO è un «condotto» (*pipe*) rappresentato da un file e si distingue dai condotti creati internamente, senza tale associazione simbolica. Pertanto si distingue anche tra «condotti con nome» (*pipe* con nome) o file FIFO e «condotti senza nome» (*pipe* senza nome) o solo *pipe*.

L'esempio seguente mostra un programma elementare che ha lo scopo di creare un file FIFO.

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
    int status;
    status = mknod ("/tmp/miofifo",
                   (mode_t) (S_IFIFO | S_IRUSR | S_IWUSR
                              | S_IXUSR),
                   (dev_t) 0);
    if (status != 0) perror (NULL);
    return (0);
}
```

#### 68.8.1.4 Creazione di file di dispositivo

«

La creazione di un file di dispositivo richiede l'indicazione del numero del dispositivo. Ciò comporta una complicazione, perché quel numero contiene simultaneamente le informazioni sul numero primario e sul numero secondario.

Originariamente, il numero del dispositivo era formato da 16 bit, di cui l'ottetto (il byte) più significativo rappresentava il numero primario, mentre quello meno significativo il numero secondario. Dal momento che questa organizzazione è sì quella tradizionale, ma non è richiesta dallo standard, diventa necessario disporre di funzioni o macroistruzioni che aiutino a comporre correttamente il numero di dispositivo complessivo, o a estrapolare le sue componenti.

Vari sistemi che si rifanno al modello di Unix introducono tre funzioni o macroistruzioni, utili per manipolare i numeri di dispositivo. Purtroppo queste funzioni non sono standard, benché abbastanza

diffuse:

```
dev_t makedev (int major, int minor);
```

```
int major (dev_t device);
```

```
int minor (dev_t device);
```

La funzione *makedev()* assembla il numero primario e il numero secondario ottenuti come argomenti, restituendo un numero di dispositivo complessivo; per converso, le funzioni *major()* e *minor()* estrapolano rispettivamente il numero primario e il numero secondario, a partire da un numero di dispositivo complessivo. Tali funzioni dovrebbero essere dichiarate nel file di intestazione ‘`sys/types.h`’.

I due esempi seguenti mostrano la creazione di due file di dispositivo, uno a caratteri e uno a blocchi. Ci si avvale della funzione *makedev()* per assemblare il numero di dispositivo complessivo, a partire dal numero primario e dal numero secondario.

```
#include <sys/stat.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
    int status;
    status = mknod ("/tmp/miodev1",
                   (mode_t) (S_IFCHR | S_IRUSR | S_IWUSR
                              | S_IXUSR),
                   makedev (1, 2));
    if (status != 0) perror (NULL);
    return (0);
}
```

```
#include <sys/stat.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
    int status;
    status = mknod ("/tmp/miodev2",
                   (mode_t) (S_IFBLK | S_IRUSR | S_IWUSR
                              | S_IXUSR),
                   makedev (3, 4));
    if (status != 0) perror (NULL);
    return (0);
}
```

Va osservato che la creazione di un file di dispositivo dovrebbe risultare concessa solo a un processo in funzione con i privilegi dell'utente con numero UID pari a zero ('**root**').

## 68.8.2 La funzione «`mkdir()`»

La funzione *`mkdir()`* costituisce il modo corretto per creare una directory vuota (ma provvista delle voci ‘.’ e ‘..’ obbligatorie).

```
int mkdir (const char *path, mode_t mode);
```

A differenza di *`mknod()`*, il parametro *`mode`* va usato esclusivamente per indicare i permessi di accesso richiesti, tenendo conto, naturalmente, che questi vengono filtrati ulteriormente in base alla maschera dei permessi (*`user mask`*). In altri termini, nel parametro *`mode`* non si può specificare il tipo di file, cosa che comunque sarebbe ignorata, dato che si tratta della creazione di una directory e di nulla altro.

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
    int status;
    status = mkdir ("/tmp/miadir",
                  (mode_t) (S_IRUSR | S_IWUSR | S_IXUSR));
    if (status != 0) perror (NULL);
    return (0);
}
```

L'esempio mostra la creazione della directory ‘/tmp/miadir/’ in un programma completo e molto semplice.

### 68.8.3 La funzione «mkfifo()»

«

La funzione *mkfifo()* consente di creare un file FIFO, specificando il percorso e i permessi, in modo analogo a quanto si farebbe con *mkdir()* per la creazione delle directory, con la differenza che in questo caso l'uso della funzione *mknod()* sarebbe comunque corretto.

```
int mkfifo (const char *path, mode_t mode);
```

Segue un esempio molto semplice, al pari di quelli già apparsi nel capitolo.

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
    int status;
    status = mkfifo ("/tmp/miofifo",
                    (mode_t) (S_IRUSR | S_IWUSR | S_IXUSR));
    if (status != 0) perror (NULL);
    return (0);
}
```

### 68.8.4 La funzione «unlink()»

«

La funzione *unlink()* consente di eliminare un file, possibilmente di qualunque tipo, «scollegandolo» dalla directory a cui si riferisce il percorso indicato per l'operazione. Dal momento che un file è rappresentato in un file system Unix da un inode, tale inode viene eliminato effettivamente se non ci più altri riferimenti allo stesso.



In linea di principio, con *unlink()* non dovrebbe essere possibile la cancellazione di una directory; inoltre, se nel frattempo il file in questione risulta utilizzato da un processo, l'operazione di cancellazione (scollegamento) dovrebbe completarsi soltanto nel momento in cui il file risulta chiuso a tutti gli effetti.

```
int unlink (const char *path);
```

L'esempio seguente mostra la cancellazione del file `‘/tmp/cancellami’`:

```
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
    int status;
    status = unlink ("/tmp/cancellami");
    if (status != 0) perror (NULL);
    return (0);
}
```

### 68.8.5 La funzione «*rmdir()*»

La funzione *rmdir()* consente di eliminare una directory, purché vuota (contenente soltanto le voci `‘.’` e `‘..’`), specificandone il percorso. «

```
int rmdir (const char *path);
```

L'esempio seguente mostra la cancellazione della directory `‘/tmp/cancellami/’`:

```
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
    int status;
    status = rmdir ("/tmp/cancellami");
    if (status != 0) perror (NULL);
    return (0);
}
```

### 68.8.6 La funzione «remove()»

«

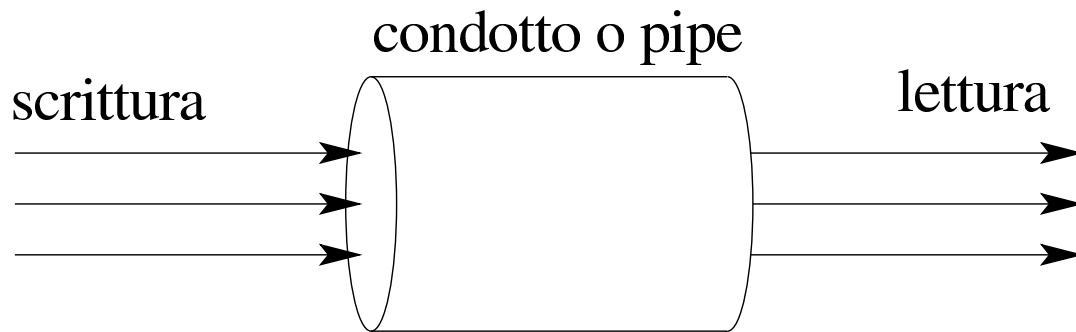
La funzione *remove()* cancella un file, utilizzando *unlink()* oppure *rmdir()*, in base al tipo di file specificato per la rimozione stessa.

```
int remove (const char *path);
```

### 68.9 Condotti

«

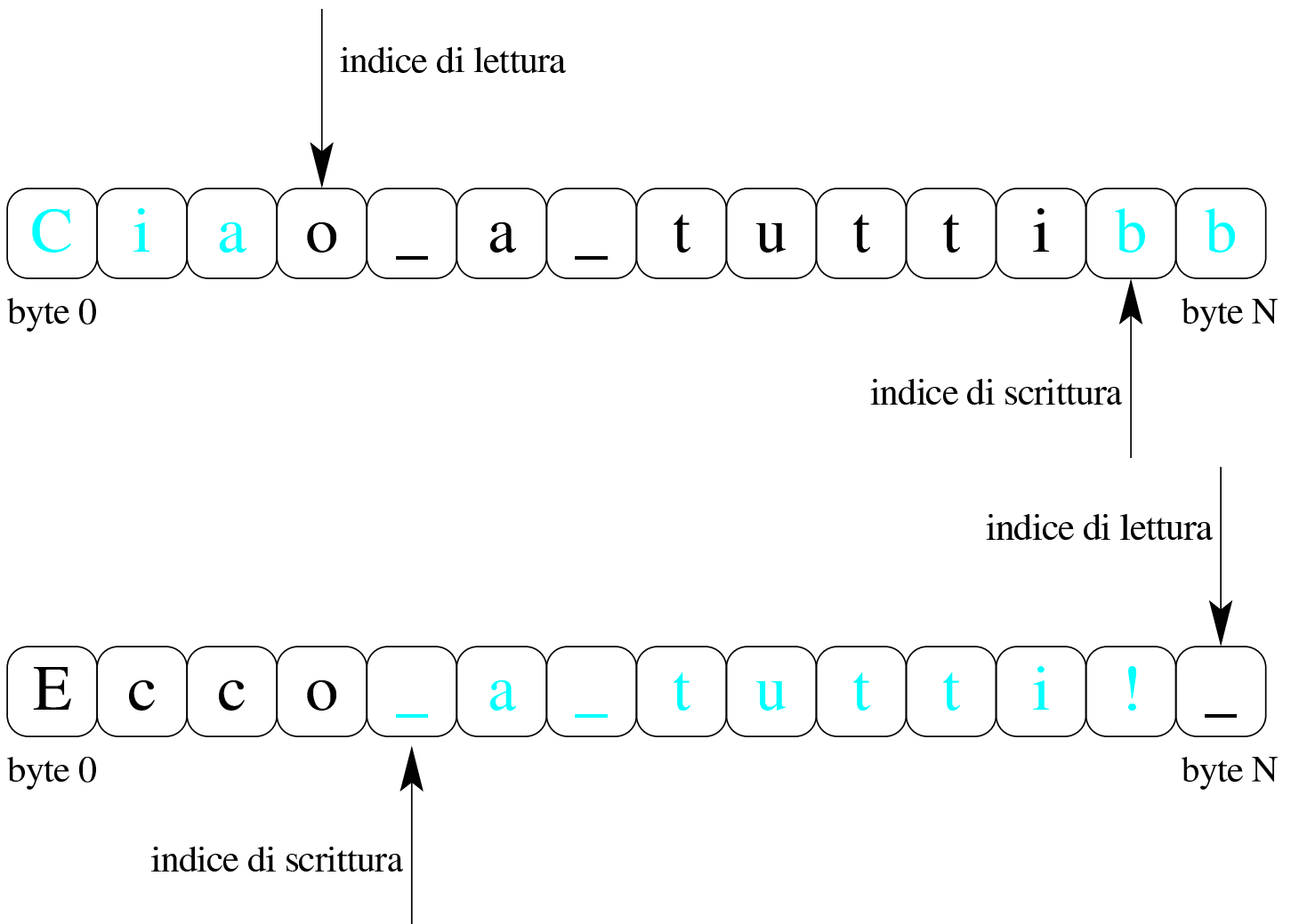
I *condotti*, o *pipe*, sono dei file virtuali, ad accesso FIFO (*First in, first out*). In altri termini sono delle *code*, in forma di file. Un condotto richiede che ci siano sia processi che vi scrivono, sia processi che vi leggono le informazioni. La lettura comporta il prelievo di dati e la liberazione di spazio disponibile per ulteriori operazioni di scrittura.



Se più di un processo apre uno stesso condotto in scrittura, non è possibile prevedere l'ordine in cui le operazioni di scrittura procedono; nello stesso modo, se più processi aprono uno stesso condotto in lettura, non è possibile prevedere con quale ordine vengano raccolti i dati dal condotto stesso. In altri termini, è compito dei processi di definire un protocollo tra di loro, se i dati devono confluire secondo un certo criterio.

### 68.9.1 Realizzazione materiale del condotto

In memoria centrale, il condotto si realizza come un array di byte, a cui si accede attraverso due indici: uno per la scrittura e l'altro per la lettura, tenendo conto che raggiunta la fine, si riprende dall'inizio. <<



Lo schema mostra un array scandito con due indici, dove i caratteri di colore nero rappresentano i byte scritti e ancora da leggere, mentre quelli in azzurro-ciano (ovvero quelli di colore più chiaro), rappresentano i byte già letti che possono essere sovrascritti. Lo schema mostra due momenti differenti, dove nel secondo caso l'indice di scrittura, una volta raggiunta la fine dell'array, riprende dall'inizio.

Dal momento che un condotto viene rappresentato in memoria come un inode, con tanto di elenco di riferimenti ai blocchi utilizzati nel file system, dato che tali annotazioni non servono perché nulla viene memorizzato in un file system, originariamente si utilizzava proprio quella porzione di memoria (quella dei blocchi diretti) per

l'array che consente di conservare temporaneamente i dati. Tuttavia, dal momento che lo standard di oggi richiede che lo spazio nella coda di un condotto sia abbastanza grande, è improbabile che si utilizzi ancora questo metodo.

### 68.9.2 Condotti «senza nome» e condotti «con nome»

Il condotto, come concetto, è un file virtuale già «aperto» e utilizzato da qualche processo elaborativo. In questi termini, un condotto potrebbe essere creato al volo, da un processo che successivamente ne avvia un altro con il quale deve comunicare. Un condotto realizzato al volo non ha alcun riferimento nel file system, pertanto gli si attribuisce la caratteristica di essere «senza nome». D'altro canto, un condotto può essere rappresentato nel file system da un file speciale, di tipo FIFO, da trattare come se fosse un file normale, benché non lo sia. Nel secondo caso si tratta di un condotto «con nome», perché c'è un nome nel file system, ovvero si tratta di un file FIFO.

### 68.9.3 Protocollo di accesso ai condotti

Una volta creato un condotto (che questo sia senza nome o che derivi dall'apertura di un file FIFO, ciò non fa differenza), solo dopo che questo risulta utilizzato sia in scrittura, sia in lettura, si può procedere con le operazioni di scrittura e lettura.

Quando un processo tenta di leggere da un condotto nel quale non sono disponibili dati nuovi, questo viene sospeso, in attesa di dati; nello stesso modo, un processo che tenta di scrivere in un condotto che non ha spazio disponibile (perché i dati già inseriti non sono ancora stati letti), viene sospeso in attesa di tale disponibilità.

Quando un processo tenta di leggere da un condotto che non viene più utilizzato in scrittura (perché non ci sono più descrittori di file associati al condotto in scrittura), si trova di fronte a un file concluso (nel senso che si avvera la condizione di fine del file); quando invece un processo tenta di scrivere in un condotto a cui non corrisponde più alcun descrittore in lettura, questo processo riceve il segnale SIGPIPE e, se il processo lo ignora, l'operazione di scrittura termina con un errore **'EPIPE'**.

Dal momento che la natura di un condotto è quella di essere ad accesso sequenziale, non è possibile posizionare l'indice di lettura o scrittura, con l'ausilio della funzione *lseek()*.

#### 68.9.4 Funzione «pipe()»

«

La funzione *pipe()* crea un condotto al volo, restituendo attraverso un array di due elementi i numeri dei descrittori per l'accesso in lettura e in scrittura. Tale condotto non ha un file FIFO corrispondente, ma risulta ugualmente aperto e disponibile al processo che lo crea.

```
int pipe (int fd[2]);
```

La funzione restituisce l'esito dell'operazione: zero in caso di successo, oppure  $-1$  in caso di problemi (aggiornando il valore della variabile *errno*). I descrittori per l'accesso vengono ottenuti dal contenuto dell'array *fd[]*, dove *fd[0]* è il descrittore per l'accesso in lettura, mentre *fd[1]* è quello per l'accesso in scrittura.

Con questa funzione, un processo crea un condotto e ottiene i due descrittori di accesso; tuttavia, lo scopo di un condotto è quello di mettere in comunicazione due o più processi. Pertanto, dopo un'o-

perazione di questo tipo, si passa quasi certamente a una biforcazione ed eventualmente all'esecuzione di un altro programma. Dopo la biforcazione è normale che il processo originario chiuda il descrittore che non gli serve (dal momento che utilizza probabilmente solo quello di scrittura o solo quello di lettura) e che così faccia anche il processo sdoppiato: il programma che eventualmente venisse caricato al posto del processo sdoppiato troverebbe già tutto pronto per iniziare a lavorare correttamente. Il listato successivo mostra un esempio, disponibile eventualmente presso [allegati/c/esempio-posix-pipe.c](#), ottenuto modificando un esempio analogo che appare nella pagina di manuale *pipe(2)* di un sistema GNU/Linux.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//-----
int
main (void)
{
    int                pipefd[2];
    pid_t              child;
    char               buffer;
    char               *message = "ciao a tutti voi amici "
                                "vicini e lontani\n";

    int                i;
    size_t             size;
    ssize_t            written;
//
```

```
//  
//  
if (pipe (pipefd) == -1)  
  {  
    perror ("pipe");  
    exit (EXIT_FAILURE);  
  }  
//  
//  
//  
child = fork ();  
if (child == -1)  
  {  
    perror ("fork");  
    exit (EXIT_FAILURE);  
  }  
//  
//  
//  
if (child == 0)  
  {  
    //  
    // Questo è il figlio e deve leggere dal condotto;  
    // pertanto, chiude il descrittore di scrittura nel  
    // condotto.  
    //  
    close (pipefd[1]);  
    //  
    // Legge un byte alla volta, finché c'è qualcosa da  
    // poter leggere.  
    //  
    while (read (pipefd[0], &buffer, 1) > 0)  
      {  
        //  
        //  
      }  
  }  
}
```





```
        if (written < 0)
        {
            //
            // Essendosi verificato un errore, chiude
            // il descrittore del condotto e si mette
            // in attesa della morte del proprio
            // processo figlio. Al termine conclude il
            // proprio funzionamento.
            //
            perror ("pipe");
            close (pipefd[1]);
            wait (NULL); // Wait for child.
            exit (EXIT_FAILURE);
        }
    }
}
//
// Dal momento che il codice precedente è racchiuso in
// un ciclo infinito, ciò che segue non può essere mai
// eseguito.
// Comunque, nel caso si volesse gestire il ciclo
// precedente, a questo punto verrebbe chiuso il
// condotto da parte del processo genitore, attendendo
// la morte del proprio processo figlio, prima di
// concludere regolarmente il proprio funzionamento.
//
close (pipefd[1]);
wait (NULL);
exit (EXIT_SUCCESS);
}
}
```

Il programma dell'esempio, sdoppiandosi in due processi, da un lato (quello del genitore) scrive nel condotto la stringa *message*

una quantità di volte indefinita, mentre dall'altro legge dal condotto il messaggio riproducendolo attraverso lo standard output sullo schermo, fino a quando uno dei due processi viene interrotto.

### 68.9.5 Esempio di condotto attraverso un file FIFO

Nella sezione precedente appare un esempio completo di programma che crea e utilizza un condotto. Nel listato successivo, disponibile eventualmente presso [allegati/c/esempio-posix-fifo.c](#), si vede un altro esempio, pressoché equivalente, in cui i due processi prodotti comunicano attraverso un condotto derivante da un file FIFO. Il file FIFO viene creato preventivamente dal processo genitore, prima di sdoppiarsi, poi i due processi aprono il file e riproducono lo stesso comportamento già descritto nella sezione precedente.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/wait.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>

//-----
int
main (void)
{
    int          fd;
    pid_t        child;
    char         buffer;
    char         *message = "ciao a tutti voi amici "
                           "vicini e lontani\n";

    int          i;
    size_t       size;
```

```
ssize_t          written;
int              status;
//
// Prima di creare il file FIFO, cancella quello che
// potrebbe esserci già.
//
unlink ("/tmp/fifo");
//
// Usa 'mknod()', ma potrebbe usare 'mkfifo()' per creare
// il file FIFO.
//
status = mknod ("/tmp/fifo",
                S_IFIFO | S_IRUSR | S_IWUSR, 0);
if (status != 0)
{
    perror ("mknod fifo");
    exit (EXIT_FAILURE);
}
//
// Il processo si sdoppia.
//
child = fork ();
if (child == -1)
{
    perror ("fork");
    exit (EXIT_FAILURE);
}
//
//
//
if (child == 0)
{
    //
    // Questo è il processo figlio che apre il file FIFO
```

```
// in lettura.
//
fd = open ("/tmp/fifo", O_RDONLY);
if (fd < 0)
{
    perror ("fifo read open");
    exit (EXIT_FAILURE);
}
//
// Legge un byte alla volta riproducendolo attraverso
// lo standard output.
//
while (read (fd, &buffer, 1) > 0)
{
    write (STDOUT_FILENO, &buffer, 1);
}
//
// Chiude il file FIFO e si conclude il funzionamento
// del processo figlio.
//
close (fd);
//
exit (EXIT_SUCCESS);
}
else
{
    //
    // Questo è il processo genitore che apre il file
    // FIFO in scrittura.
    //
    fd = open ("/tmp/fifo", O_WRONLY);
    if (fd < 0)
    {
        perror ("fifo write open");
    }
}
```

```
    exit (EXIT_FAILURE);
}
//
// Inizia un ciclo infinito.
//
while (1)
{
    //
    // Scrive il messaggio contenuto nella stringa
    // 'message' attraverso il condotto.
    //
    for (i = 0, written = 0, size = strlen (message);
        i < strlen (message);
        i += written, size -= written)
    {
        written = write (fd, &message[i], size);
        if (written < 0)
        {
            //
            // Essendosi verificato un errore, chiude
            // il file FIFO e si mette in attesa della
            // morte del proprio processo figlio.
            // Al termine conclude il proprio
            // funzionamento.
            //
            perror ("pipe");
            close (fd);
            wait (NULL);
            exit (EXIT_FAILURE);
        }
    }
}
//
// Dal momento che il codice precedente è racchiuso
```

```
// in un ciclo infinito, ciò che segue non può essere
// mai eseguito.
// Comunque, nel caso si volesse gestire il ciclo
// precedente, a questo punto verrebbe chiuso il file
// FIFO da parte del processo genitore, attendendo la
// morte del proprio processo figlio, prima di
// concludere regolarmente il proprio funzionamento.
//
close (fd);
wait (NULL);
exit (EXIT_SUCCESS);
}
}
```

## 68.10 Lettura delle directory

Le directory si creano con la funzione *mkdir()*, in modo da garantire che le voci obbligatorie ‘.’ e ‘..’ siano presenti. Successivamente, l’aggiornamento delle directory avviene in modo trasparente da parte del sistema operativo, in base alle operazioni di creazione ed eliminazione dei file. Rimane il problema della lettura delle directory, dalla quale ottenere nomi e riferimenti a inode, per poter conoscere il loro «contenuto».

Di norma è possibile leggere le directory come se fossero dei file puri e semplici, ma così facendo occorre interpretarne il contenuto secondo le regole di quel file system particolare. In generale ciò è sconsigliabile, pertanto vengono in aiuto alcune funzioni descritte nel file di intestazione ‘dirent.h’ (si veda anche la sezione [70.6](#) sul file di intestazione ‘dirent.h’).

## 68.10.1 Tipi derivati



La gestione delle directory, secondo il file di intestazione `'dirent.h'`, prevede due tipi derivati: `'DIR'` e `'struct dirent'`. Il tipo `'DIR'` serve a rappresentare una variabile strutturata con tutte le informazioni relative a un flusso di file riferito a una directory. In altri termini, è l'equivalente del tipo `'FILE'`, ma utile solo per l'accesso alle directory. Il tipo `'struct dirent'` serve a poter rappresentare i due componenti indispensabili di ogni voce di directory dei sistemi Unix: numero di inode e nome. Pertanto, il tipo `'struct dirent'` contiene almeno i membri `'d_ino'` e `'d_name'`, per contenere rispettivamente il numero di inode e il nome relativo:

```
struct dirent {
    ino_t    d_ino;
    char    d_name[];
    ...
};
```

## 68.10.2 Procedura per accedere a una directory



Per accedere a una directory, occorre prima aprirla, con la funzione `opendir()`, la quale restituisce un puntatore a una variabile di tipo `'DIR'`, dove tale puntatore rappresenta così la directory in forma di flusso.

```
DIR *opendir (const char *path);
```

La lettura di una directory avviene a blocchi di una voce per volta, utilizzando la funzione `readdir()`: ogni lettura produce la voce



successiva della *directory*, in forma di variabile strutturata di tipo `'struct dirent'`, della quale si ottiene il puntatore.

```
struct dirent *readdir (DIR *dp);
```

Evidentemente, da come è strutturato il prototipo della funzione, si intuisce che la variabile strutturata a cui punta ciò che restituisce la funzione stessa, deve trovarsi in una zona di memoria statica, la quale viene riutilizzata ogni volta che si chiama la funzione *readdir()*.

Dato che le letture si susseguono in modo sequenziale, quando si vuole che la prossima lettura ricominci dalla prima voce, si utilizza la funzione *rewinddir()*:

```
void rewinddir (DIR *dp);
```

Al termine, come per i file normali, il flusso aperto di *directory* va chiuso con la funzione *closedir()*, dove il valore restituito rappresenta il successo o meno dell'operazione:

```
int closedir (DIR *dp);
```

### 68.10.3 Attributo «FD\_CLOEXEC»

Un sistema Unix dispone di un metodo di accesso ai file basato sui descrittori, sopra il quale si inserisce la gestione dei flussi di file. Per quanto riguarda le *directory*, lo standard non specifica se i flussi relativi debbano avvalersi dei descrittori o meno. Tuttavia, se si usano i descrittori, si presenta una situazione particolare: se si esegue un

altro processo, in sostituzione di quello in corso (con una delle funzioni *exec...()*), i descrittori aperti vengono ereditati tali e quali dal nuovo processo. Nel caso delle *directory*, ciò va evitato.

Nei sistemi in cui il tipo **DIR** viene gestito tramite riferimenti a descrittori, l'apertura di una *directory* comporta l'attivazione dell'indicatore rappresentato dalla macro-variabile *FD\_CLOEXEC* (file di intestazione `'fcntl.h'`), con il quale si assicura che il descrittore venga chiuso nel caso di utilizzo di una funzione *exec...()*.

#### 68.10.4 Esempio di utilizzo delle funzioni di accesso alle *directory*

«

Il listato successivo mostra un esempio molto semplice di programma che legge la *directory* corrente, mostrando l'elenco dei nomi che contiene, senza indicare però altre informazioni. Eventualmente si può ottenere il file dell'esempio dall'indirizzo [allegati/c/esempio-posix-dirent.c](#)

```
#include <errno.h>
#include <stdio.h>
#include <dirent.h>
//-----
int
main (int argc, char *argv[], char *envp[])
{
    DIR          *dp;
    struct dirent *dir;
    //
    dp = opendir (".");
    if (dp == NULL)
        {
            perror (NULL);
            return (1);
        }
    //
    while ((dir = readdir (dp)) != NULL)
        {
            printf ("%s\n", dir->d_name);
        }
    //
    closedir (dp);
    //
    return (0);
}
```

Eventualmente si veda una realizzazione molto semplice del programma ‘**ls**’ nei sorgenti di os32 (listato [96.1.23](#)).

## 68.11 Riferimenti

- Maurice J. Bach, *The design of the UNIX operating system*, Prentice Hall, 1990, ISBN 0132017997



- The Open Group, *The UNIX System*, <http://www.unix.org/>
- Free Software Foundation, *The GNU C Library*, <http://www.gnu.org/software/libc/manual/>
- The Open Group, *The Single UNIX® Specification, Version 2, Regular Expressions*, [http://pubs.opengroup.org/onlinepubs/000095399/basedefs/xbd\\_chap09.html](http://pubs.opengroup.org/onlinepubs/000095399/basedefs/xbd_chap09.html)
- Daniel Robbins, *POSIX threads explained*, <http://www.ibm.com/developerworks/library/l-posix1/>, <http://www.ibm.com/developerworks/library/l-posix2/>, <http://www.ibm.com/developerworks/library/l-posix3/>
- pagina di manuale *pthread(7)* di un sistema GNU/Linux
- Mark Hayes, *POSIX threads tutorial*, <http://math.arizona.edu/~swig/documentation/pthreads/>
- The Open Group, *The Single UNIX® Specification, Version 2, pthread.h*, <http://pubs.opengroup.org/onlinepubs/000095399/basedefs/pthread.h.html>
- The Open Group, *The Single UNIX® Specification, Version 2, dirent.h*, <http://pubs.opengroup.org/onlinepubs/000095399/basedefs/dirent.h.html>
- Pagine di manuale: *mknod(2)*, *mkdir(2)*, *mkfifo(3)*, *unlink(2)*, *pipe(2)*, *mkfifo(3)*, *opendir(3)*, *readdir(3)*, *rewinddir(3)*, *closedir(3)*

<sup>1</sup> Per la libreria POSIX, la gestione dei flussi del linguaggio C è costruita avvalendosi del sistema dei descrittori, con l'aggiunta però di una memoria tampone.

<sup>2</sup> Nel programma di esempio si può fare sicuramente di meglio, incrementando direttamente la variabile globale, senza tanti travasi come invece viene fatto. Ma lo scopo di questi esempi è simulare una situazione più complessa, senza complicazioni che esulano dal problema specifico che si vuole descrivere.



## Libreria C, con qualche estensione POSIX



69.1	Funzionalità di libreria non dichiarate .....	1069
69.2	File «assert.h» .....	1069
69.2.1	Utilizzo .....	1070
69.3	File «limits.h» .....	1071
69.3.1	Confronto tra architetture .....	1075
69.3.2	Estensioni POSIX .....	1075
69.4	File «stdint.h» .....	1077
69.4.1	Tipi interi ad ampiezza esatta 1078	
69.4.2	Tipi interi di rango minimo 1079	
69.4.3	Tipi interi «veloci» .....	1082
69.4.4	Tipi interi per rappresentare dei puntatori ..	1084
69.4.5	Tipi interi di rango massimo .....	1085
69.4.6	Limiti per altri tipi interi .....	1086
69.5	File «errno.h» .....	1087
69.6	File «locale.h» .....	1092
69.6.1	Impostazione della configurazione locale .....	1093
69.6.2	Composizione dei valori numerici .....	1096
69.7	File «ctype.h» .....	1101
69.7.1	Funzioni «is...()» .....	1102

69.7.2	macroistruzioni «is...()» .....	1116
69.7.3	Funzioni di conversione .....	1117
69.7.4	macroistruzioni di conversione .....	1120
69.7.5	Esempio di utilizzo delle funzioni .....	1120
69.8	File «stdarg.h» .....	1123
69.8.1	Realizzazione .....	1124
69.8.2	Esempio di utilizzo delle macro .....	1125
69.8.3	Promozione .....	1127
69.9	File «stdlib.h» .....	1127
69.9.1	Tipi di dati speciali .....	1128
69.9.2	Macro-variabili .....	1128
69.9.3	Conversioni numeriche .....	1130
69.9.4	Funzioni per la generazione di numeri in modo pseudo-casuale .....	1136
69.9.5	Funzioni standard per la generazione di numeri pseudo-casuali .....	1139
69.9.6	Amministrazione della memoria .....	1140
69.9.7	Conclusione forzata del programma .....	1141
69.9.8	Funzioni di comunicazione con l'ambiente ....	1142
69.9.9	Funzioni di ricerca e riordino .....	1143
69.9.10	Funzioni per l'aritmetica con i numeri interi 1146	
69.9.11	Funzioni per la gestione di caratteri estesi e sequenze multibyte .....	1148
69.9.12	Funzione «mblen()» .....	1149



69.9.13	Funzioni «mbtowc()» e «wctomb()» .....	1151
69.9.14	Funzioni «mbstowcs()» e «wcstombs()» .....	1153
69.10	File «inttypes.h» .....	1156
69.10.1	Divisione intera con interi di rango massimo ..	1157
69.10.2	Macro-variabili in qualità di specificatori di conversione 1158	
69.10.3	Valore assoluto .....	1165
69.10.4	Conversione da stringa a numero intero ...	1166
69.11	File «iso646.h» .....	1167
69.12	File «stdbool.h» .....	1168
69.13	File «stddef.h» .....	1169
69.14	File «string.h» .....	1170
69.14.1	Copia .....	1170
69.14.2	Concatenamento .....	1180
69.14.3	Comparazione .....	1184
69.14.4	Ricerca .....	1191
69.14.5	Funzioni varie .....	1217
69.15	File «signal.h» .....	1223
69.15.1	Dichiarazione contorta .....	1223
69.15.2	Tipo speciale .....	1225
69.15.3	Denominazione dei segnali .....	1226
69.15.4	Segnali secondo POSIX 1227	

69.15.5	Gestori fittizi di segnali .....	1231
69.15.6	Funzioni .....	1233
69.15.7	Esempio .....	1235
69.16	File «time.h» .....	1239
69.16.1	Il tempo di CPU .....	1239
69.16.2	Rappresentazione interna del tempo .....	1241
69.16.3	Rappresentazione strutturata del tempo .....	1242
69.16.4	Funzioni per l'elaborazione di valori legati al tempo 1243	
69.16.5	Conversione in stringa .....	1247
69.17	File «stdio.h» .....	1254
69.17.1	Tipi .....	1255
69.17.2	Macro-variabili varie .....	1256
69.17.3	Ipotesi di gestione del tipo «FILE» .....	1259
69.17.4	Flussi standard .....	1260
69.17.5	Funzioni per la rimozione e la ridenominazione dei file .....	1261
69.17.6	Funzioni per la gestione dei file temporanei ..	1261
69.17.7	Funzioni per l'apertura e la chiusura dei flussi di file 1264	
69.17.8	Funzioni per la gestione della memoria tampone 1268	
69.17.9	Funzioni per la composizione dell'output 1270	
69.17.10	Funzioni per l'interpretazione dell'input .....	1280

69.17.11	Funzioni per la lettura e la scrittura di un carattere alla volta	1289					
69.17.12	Funzioni per l'input e l'output di file di testo	1292					
69.17.13	Funzioni per l'input e output diretto	1294					
69.17.14	Funzioni per il posizionamento	1295					
69.17.15	Accesso esclusivo ai flussi di file	1297					
69.17.16	Condotti	1299					
69.17.17	Informazioni sul terminale	1299					
69.17.18	Gestione degli errori	1300					
69.17.19	Realizzazione di «vsnprintf()» e altre collegate	1301					
69.18	Riferimenti	1302					
abort()	abs()	and	and_eq	1141	1146	1167	1167
asctime()	assert()	assert.h		1247	1069	1069	1069
atexit()	atof()	atoi()	atol()	1141	1130	1130	1130
atoll()	bitand	bool	bsearch()	1130	1167	1168	1143
BUFSIZ	calloc()	CHAR_BIT	CHAR_MAX	1256	1140	1071	1071
CHAR_MIN	clearerr()	clock()		1071	1300	1239	
CLOCKS_PER_SEC	clock_t	compl		1239	1239	1167	
ctermid()	ctime()	ctype.h		1299	1248	1101	
difftime()	div()	div_t	EDOM	1244	1146	1128	1087
EILSEQ	errno	errno.h		1087	1087	1087	1087
EXIT_FAILURE	EXIT_SUCCESS			1128	1128		
false	fclose()	fdopen()	feof()	1168	1264	1264	
ferror()	fflush()	fgetc()		1300	1268	1289	
fgetpos()	fgets()	FILE	FILENAME_MAX	1295	1292	1255	1256
flockfile()	fopen()	FOPEN_MAX		1297	1264	1256	

fpos\_t 1255 fprintf() 1278 fputc() 1289 fputs() 1292  
 fread() 1294 free() 1140 freopen() 1264 fscanf()  
 1287 fseek() 1295 fseeko() 1295 fsetpos() 1295  
 ftell() 1295 ftello() 1295 ftrylockfile() 1297  
 funlockfile() 1297 fwrite() 1294getc() 1289  
 getchar() 1289getchar\_unlocked() 1297  
 getc\_unlocked() 1297getenv() 1142gets() 1292  
 gmtime() 1246imaxabs() 1165imaxdiv() 1157  
 imaxdiv\_t 1157INT16\_C() 1079INT16\_MAX 1078  
 INT16\_MIN 1078int16\_t 1078INT32\_C() 1079  
 INT32\_MAX 1078INT32\_MIN 1078int32\_t 1078  
 INT64\_C() 1079INT64\_MAX 1078INT64\_MIN 1078  
 int64\_t 1078INT8\_C() 1079INT8\_MAX 1078INT8\_MIN  
 1078int8\_t 1078INTMAX\_C() 1085INTMAX\_MAX 1085  
 INTMAX\_MIN 1085intmax\_t 1085INTPTR\_MAX 1084  
 INTPTR\_MIN 1084intptr\_t 1084inttypes.h 1156  
 INT\_FAST16\_MAX 1082INT\_FAST16\_MIN 1082  
 int\_fast16\_t 1082INT\_FAST32\_MAX 1082  
 INT\_FAST32\_MIN 1082int\_fast32\_t 1082  
 INT\_FAST64\_MAX 1082INT\_FAST64\_MIN 1082  
 int\_fast64\_t 1082INT\_FAST8\_MAX 1082  
 INT\_FAST8\_MIN 1082int\_fast8\_t 1082  
 INT\_LEAST16\_MAX 1079INT\_LEAST16\_MIN 1079  
 int\_least16\_t 1079INT\_LEAST32\_MAX 1079  
 INT\_LEAST32\_MIN 1079int\_least32\_t 1079  
 INT\_LEAST64\_MAX 1079INT\_LEAST64\_MIN 1079  
 int\_least64\_t 1079INT\_LEAST8\_MAX 1079  
 INT\_LEAST8\_MIN 1079int\_least8\_t 1079INT\_MAX 1071  
 INT\_MIN 1071isalnum() 1102isalpha() 1103

isascii() 1114      isblank() 1104      iscntrl() 1105  
 isdigit() 1106      isgraph() 1107      islower() 1108  
 iso646.h 1167      isprint() 1109      ispunct() 1110  
 isspace() 1111      isupper() 1112      isxdigit() 1113  
 labs() 1146      LC\_TIME 1249      ldiv() 1146      ldiv\_t 1128  
 limits.h 1071      llabs() 1146      lldiv() 1146      lldiv\_t 1128  
 LLONG\_MAX 1071      LLONG\_MIN 1071      locale.h 1092  
 localtime() 1246      LONG\_BIT 1075      LONG\_MAX 1071  
 LONG\_MIN 1071      L\_ctermid 1256      L\_tmpnam 1256  
 malloc() 1140      mblen() 1149      mbstowcs() 1153  
 mbtowc() 1151      MB\_CUR\_MAX 1128      MB\_LEN\_MAX 1071  
 memccpy() 1172      memchr() 1192      memcmp() 1184  
 memcpy() 1171      memmove() 1174      memset() 1217  
 mktime() 1244      NDEBUG 1069      not 1167      not\_eq 1167      NULL  
 1169      offsetof 1169      or 1167      or\_eq 1167      pclose() 1299  
 perror() 1300      popen() 1299      PRId16 1158      PRId32 1158  
 PRId64 1158      PRId8 1158      PRIdFAST16 1158      PRIdFAST32  
 1158      PRIdFAST64 1158      PRIdFAST8 1158      PRIdLEAST16  
 1158      PRIdLEAST32 1158      PRIdLEAST64 1158      PRIdLEAST8  
 1158      PRIdMAX 1158      PRIdPTR 1158      PRIi16 1158      PRIi32  
 1158      PRIi64 1158      PRIi8 1158      PRIiFAST16 1158  
 PRIiFAST32 1158      PRIiFAST64 1158      PRIiFAST8 1158  
 PRIiLEAST16 1158      PRIiLEAST32 1158      PRIiLEAST64 1158  
 PRIiLEAST8 1158      PRIiMAX 1158      PRIiPTR 1158      printf()  
 1278      PRIo16 1158      PRIo32 1158      PRIo64 1158      PRIo8 1158  
 PRIoFAST16 1158      PRIoFAST32 1158      PRIoFAST64 1158  
 PRIoFAST8 1158      PRIoLEAST16 1158      PRIoLEAST32 1158  
 PRIoLEAST64 1158      PRIoLEAST8 1158      PRIoMAX 1158  
 PRIoPTR 1158      PRIu16 1158      PRIu32 1158      PRIu64 1158

PRIu8 [1158](#)      PRIuFAST16 [1158](#)      PRIuFAST32 [1158](#)  
 PRIuFAST64 [1158](#)    PRIuFAST8 [1158](#)    PRIuLEAST16 [1158](#)  
 PRIuLEAST32 [1158](#)    PRIuLEAST64 [1158](#)    PRIuLEAST8 [1158](#)  
 PRIuMAX [1158](#)    PRIuPTR [1158](#)    PRIx16 [1158](#)    PRIx16 [1158](#)  
 PRIx32 [1158](#)    PRIx32 [1158](#)    PRIx64 [1158](#)    PRIx64 [1158](#)  
 PRIx8 [1158](#)    PRIx8 [1158](#)    PRIxFAST16 [1158](#)    PRIxFAST16  
[1158](#)    PRIxFAST32 [1158](#)    PRIxFAST32 [1158](#)    PRIxFAST64  
[1158](#)    PRIxFAST64 [1158](#)    PRIxFAST8 [1158](#)    PRIxFAST8 [1158](#)  
 PRIxLEAST16 [1158](#)    PRIxLEAST16 [1158](#)    PRIxLEAST32 [1158](#)  
 PRIxLEAST32 [1158](#)    PRIxLEAST64 [1158](#)    PRIxLEAST64 [1158](#)  
 PRIxLEAST8 [1158](#)      PRIxLEAST8 [1158](#)      PRIxMAX [1158](#)  
 PRIxMAX [1158](#)    PRIxPTR [1158](#)    PRIxPTR [1158](#)    PTRDIFF\_MAX  
[1086](#)    PTRDIFF\_MIN [1086](#)    ptrdiff\_t [1086](#) [1169](#)    putchar()  
[1289](#)      putchar() [1289](#)      putchar\_unlocked() [1297](#)  
 putchar\_unlocked() [1297](#)    puts() [1292](#)    P\_tmpdir [1256](#)  
 qsort() [1143](#)    raise() [1233](#)    rand() [1136](#)    RAND\_MAX [1128](#)  
 realloc() [1140](#)      remove() [1261](#)      rename() [1261](#)  
 rewind() [1295](#)      scanf() [1287](#)      SCHAR\_MAX [1071](#)  
 SCHAR\_MIN [1071](#)    SCNd16 [1158](#)    SCNd32 [1158](#)    SCNd64 [1158](#)  
 SCNd8 [1158](#)      SCNdFAST16 [1158](#)      SCNdFAST32 [1158](#)  
 SCNdFAST64 [1158](#)    SCNdFAST8 [1158](#)    SCNdLEAST16 [1158](#)  
 SCNdLEAST32 [1158](#)    SCNdLEAST64 [1158](#)    SCNdLEAST8 [1158](#)  
 SCNdMAX [1158](#)    SCNdPTR [1158](#)    SCNi16 [1158](#)    SCNi32 [1158](#)  
 SCNi64 [1158](#)    SCNi8 [1158](#)    SCNiFAST16 [1158](#)    SCNiFAST32  
[1158](#)    SCNiFAST64 [1158](#)    SCNiFAST8 [1158](#)    SCNiLEAST16  
[1158](#)    SCNiLEAST32 [1158](#)    SCNiLEAST64 [1158](#)    SCNiLEAST8  
[1158](#)    SCNiMAX [1158](#)    SCNiPTR [1158](#)    SCNo16 [1158](#)    SCNo32  
[1158](#)      SCNo64 [1158](#)      SCNo8 [1158](#)      SCNoFAST16 [1158](#)  
 SCNoFAST32 [1158](#)    SCNoFAST64 [1158](#)    SCNoFAST8 [1158](#)

SCNoLEAST16 [1158](#) SCNoLEAST32 [1158](#) SCNoLEAST64 [1158](#)  
SCNoLEAST8 [1158](#) SCNoMAX [1158](#) SCNoPTR [1158](#) SCNu16  
[1158](#) SCNu32 [1158](#) SCNu64 [1158](#) SCNu8 [1158](#) SCNuFAST16  
[1158](#) SCNuFAST32 [1158](#) SCNuFAST64 [1158](#) SCNuFAST8 [1158](#)  
SCNuLEAST16 [1158](#) SCNuLEAST32 [1158](#) SCNuLEAST64 [1158](#)  
SCNuLEAST8 [1158](#) SCNuMAX [1158](#) SCNuPTR [1158](#) SCNx16  
[1158](#) SCNx32 [1158](#) SCNx64 [1158](#) SCNx8 [1158](#) SCNxFAST16  
[1158](#) SCNxFAST32 [1158](#) SCNxFAST64 [1158](#) SCNxFAST8 [1158](#)  
SCNxLEAST16 [1158](#) SCNxLEAST32 [1158](#) SCNxLEAST64 [1158](#)  
SCNxLEAST8 [1158](#) SCNxMAX [1158](#) SCNxPTR [1158](#) SEEK\_CUR  
[1256](#) SEEK\_END [1256](#) SEEK\_SET [1256](#) setbuf() [1268](#)  
setvbuf() [1268](#) SHRT\_MAX [1071](#) SHRT\_MIN [1071](#) SIGABRT  
[1226](#) [1227](#) SIGALRM [1227](#) SIGBUS [1227](#) SIGCHLD [1227](#)  
SIGCONT [1227](#) SIGFPE [1226](#) [1227](#) SIGHUP [1227](#) SIGILL [1226](#)  
[1227](#) SIGINT [1226](#) [1227](#) SIGKILL [1227](#) signal() [1233](#)  
signal.h [1223](#) SIGPIPE [1227](#) SIGPOLL [1227](#) SIGPROF [1227](#)  
SIGQUIT [1227](#) SIGSEGV [1226](#) [1227](#) SIGSTOP [1227](#) SIGSYS  
[1227](#) SIGTERM [1226](#) [1227](#) SIGTRAP [1227](#) SIGTTIN [1227](#)  
SIGTTOU [1227](#) SIGURG [1227](#) SIGUSR1 [1227](#) SIGUSR2 [1227](#)  
SIGVTALRM [1227](#) SIGXCPU [1227](#) SIGXFSZ [1227](#)  
SIG\_ATOMIC\_MAX [1086](#) SIG\_ATOMIC\_MIN [1086](#)  
sig\_atomic\_t [1086](#) [1225](#) SIG\_DFL [1231](#) SIG\_ERR [1231](#)  
SIG\_IGN [1231](#) SIZE\_MAX [1086](#) size\_t [1086](#) [1169](#)  
snprintf() [1278](#) sprintf() [1278](#) srand() [1136](#)  
scanf() [1287](#) SSIZE\_MAX [1075](#) stdarg.h [1123](#)  
stdbool.h [1168](#) stddef.h [1169](#) stderr [1260](#) stdint.h  
[1077](#) stdio [1260](#) stdio.h [1254](#) stdlib.h [1127](#) stdout  
[1260](#) strcat() [1181](#) strchr() [1193](#) strcmp() [1186](#)  
strcoll() [1188](#) strcpy() [1175](#) strcspn() [1198](#)

strdup() 1178    strerror() 1219    strerror\_r() 1220  
 strftime() 1249    string.h 1170    strlen() 1221  
 strncat() 1182    strncmp() 1188    strncpy() 1176  
 strpbrk() 1200    strrchr() 1195    strspn() 1197  
 strstr() 1202    strtod() 1130    strtof() 1130  
 strtouimax() 1166    strtok() 1204    strtok\_r() 1211  
 strtol() 1130    strtold() 1130    strtoll() 1130  
 strtouimax() 1166    strtoul() 1130    strtoull() 1130  
 struct tm 1242    strxfrm() 1190    system() 1142  
 tempnam() 1261    time() 1243    time.h 1239    time\_t 1241  
 1243    tmpfile() 1261    tmpnam() 1261    TMP\_MAX 1256  
 toascii() 1120    tolower() 1117    toupper() 1118    true  
 1168    UCHAR\_MAX 1071    UINT16\_C() 1079    UINT16\_MAX 1078  
 uint16\_t 1078    UINT32\_C() 1079    UINT32\_MAX 1078  
 uint32\_t 1078    UINT64\_C() 1079    UINT64\_MAX 1078  
 uint64\_t 1078    UINT8\_C() 1079    UINT8\_MAX 1078  
 uint8\_t 1078    UINTMAX\_C() 1085    UINTMAX\_MAX 1085  
 uintmax\_t 1085    UINTPTR\_MAX 1084    uintptr\_t 1084  
 UINT\_FAST16\_MAX 1082    uint\_fast16\_t 1082  
 UINT\_FAST32\_MAX 1082    uint\_fast32\_t 1082  
 UINT\_FAST64\_MAX 1082    uint\_fast64\_t 1082  
 UINT\_FAST8\_MAX 1082    uint\_fast8\_t 1082  
 UINT\_LEAST16\_MAX 1079    uint\_least16\_t 1079  
 UINT\_LEAST32\_MAX 1079    uint\_least32\_t 1079  
 UINT\_LEAST64\_MAX 1079    uint\_least64\_t 1079  
 UINT\_LEAST8\_MAX 1079    uint\_least8\_t 1079    UINT\_MAX  
 1071    ULLONG\_MAX 1071    ULONG\_MAX 1071    ungetc() 1289  
 USHRT\_MAX 1071    va\_arg() 1123    va\_copy() 1123  
 va\_end() 1123    va\_list 1123    va\_start() 1123



vfprintf() 1279      vfscanf() 1288      vprintf() 1279  
 vscanf() 1288      vsnprintf() 1279      vsprintf() 1279  
 vsscanf() 1288      WCHAR\_MAX 1086      WCHAR\_MIN 1086  
 wchar\_t 1086 1169      wcstoimax() 1166      wcstombs() 1153  
 wcstouimax() 1166      wctomb() 1151      WINT\_MAX 1086  
 WINT\_MIN 1086      wint\_t 1086      WORD\_BIT 1075      xor 1167  
 xor\_eq 1167      \_Exit() 1141      \_IOFBF 1256      \_IOLBF 1256  
 \_IONBF 1256      \_POSIX2\_... 1075      \_POSIX\_... 1075  
 \_XOPEN\_... 1075      \_\_bool\_true\_false\_are\_defined  
 1168      \_\_udivdi3() 1069      \_\_umoddi3() 1069      %+... 1270      %...c  
 1270      %...d 1270      %...e 1270      %...f 1270      %...g 1270      %...hd 1270  
 %...hhd 1270      %...hhi 1270      %...hhn 1270      %...hho 1270      %...hhu  
 1270      %...hhx 1270      %...hi 1270      %...hn 1270      %...ho 1270      %...hu  
 1270      %...hx 1270      %...i 1270      %...lc 1270      %...ld 1270      %...Le 1270  
 %...Lf 1270      %...Lg 1270      %...li 1270      %...lld 1270      %...lli 1270  
 %...lln 1270      %...llo 1270      %...llu 1270      %...llx 1270      %...ln  
 1270      %...lo 1270      %...ls 1270      %...lu 1270      %...lx 1270      %...n 1270  
 %...o 1270      %...s 1270      %...u 1270      %...x 1270      %0... 1270      %-... 1270

Complessivamente, la libreria C è ciò che consente l'uso di funzioni, macroistruzioni e macro-variabili definite dallo standard (ed eventualmente dalle estensioni presenti nel proprio contesto). Generalmente le funzioni vengono fornite già compilate all'interno di una libreria dinamica o statica (per esempio possono essere i file `/lib/libc.so` o `/usr/lib/libc.a`), ma dal punto di vista formale, la libreria standard è percepita attraverso i file di intestazione.

Per la precisione, lo standard stabilisce che si debba fare riferimento a delle «intestazioni» nel sorgente di un programma scritto in linguaggio C, ma il contesto particolare può essere tale per cui queste potrebbero non esistere fisicamente come ci si attenderebbe da un sistema operativo tradizionale. Anche per questo, nella documentazione standard ci si riferisce solo a intestazioni, senza precisare che debba trattarsi di file.

In pratica, i file di intestazione, o ciò che ne fa la funzione, sono sempre necessari e al loro interno si dichiarano le macro-variabili, le macroistruzioni e i prototipi delle funzioni, le quali normalmente sono già precompilate in un file separato. A ogni modo, di norma il compilatore è predisposto per utilizzare automaticamente i file precompilati necessari.

Nei capitoli successivi vengono descritti alcuni dei file di intestazione previsti dallo standard del linguaggio, mostrando come potrebbero essere realizzati e, in alcuni casi, anche fornendo una soluzione completa per le funzioni (gli esempi dovrebbero essere disponibili a partire da [allegati/c/](#)).

La libreria C viene estesa dallo standard POSIX con componenti aggiuntivi. In alcuni casi, nei capitoli successivi, si fa riferimento anche a estensioni POSIX, con le annotazioni appropriate al riguardo. Va però osservato che per scrivere un programma in linguaggio C, che abbia la massima portabilità fra sistemi operativi molto differenti tra loro, occorre evitare il più possibile le estensioni di qualunque genere.

Ciò che non si vede negli esempi dei capitoli successivi è la tecnica comune che si usa per evitare di includere ricorsivamente lo stesso file di intestazione più volte: si associa a ogni file una macro-variabile e se all'inizio della lettura questa non risulta dichiarata, il contenuto viene acquisito, altrimenti viene ignorato semplicemente, perché deve essere già stato incluso in precedenza. L'esempio seguente riguarda il file `'limits.h'`:

```
#ifndef _LIMITS_H
#define _LIMITS_H      1
    ...
    contenuto_del_file
    ...
#endif // __LIMITS_H
```

In pratica viene verificato se la macro-variabile `__LIMITS_H` è già stata definita; se lo è, il contenuto del file viene ignorato. Se invece la macro-variabile non è stata dichiarata, questa allora viene dichiarata e quindi si procede con il lavoro normale del file.

Tabella 69.2. File di intestazione standard.

Intestazione	Descrizione	Riferimenti
<code>assert.h</code>	Verifica diagnostica di un'espressione (asserzione).	sezione <a href="#">69.2</a>
<code>complex.h</code>	Aritmetica complessa.	--
<code>ctype.h</code>	Classificazione dei caratteri.	sezione <a href="#">69.7</a>
<code>errno.h</code>	Definizione degli errori.	sezione <a href="#">69.5</a>
<code>fenv.h</code>	Gestione di valori in virgola mobile.	--

Intestazione	Descrizione	Riferimenti
<code>float.h</code>	Limiti dei valori in virgola mobile.	--
<code>inttypes.h</code>	Estensione di <code>'stdint.h'</code> .	sezione <a href="#">69.10</a>
<code>iso646.h</code>	Macro-variabili da usare in sostituzione di vari operatori.	sezione <a href="#">69.11</a>
<code>limits.h</code>	Limiti per i numeri interi.	sezione <a href="#">69.3</a>
<code>locale.h</code>	Gestione della configurazione locale (nel senso di «localizzazione»).	sezione <a href="#">69.6</a>
<code>math.h</code>	Funzioni matematiche comuni.	--
<code>setjmp.h</code>	Funzionalità per il salto incondizionato.	--
<code>signal.h</code>	Gestione dei segnali.	sezione <a href="#">69.15</a>
<code>stdarg.h</code>	Gestione degli argomenti variabili.	sezione <a href="#">69.8</a>
<code>stdbool.h</code>	Tipo e valori booleani.	sezione <a href="#">69.12</a>
<code>stddef.h</code>	Definizioni comuni; in particolare i tipi <code>'size_t'</code> , <code>'wchar_t'</code> e il puntatore nullo <code>'NULL'</code> .	sezione <a href="#">69.13</a>
<code>stdint.h</code>	Definizioni di interi con un rango prestabilito, assieme ai valori minimi e massimi.	sezione <a href="#">69.4</a>
<code>stdio.h</code>	Gestione di input e output dei dati.	sezione <a href="#">69.17</a>
<code>stdlib.h</code>	Funzioni, macro e tipi di utilità generale.	sezione <a href="#">69.9</a>

Intestazione	Descrizione	Riferimenti
<code>string.h</code>	Gestione delle stringhe.	sezione <a href="#">69.14</a>
<code>tgmath.h</code>	macroistruzioni matematiche, indipendenti dal tipo.	--
<code>time.h</code>	Gestione di date e orari.	sezione <a href="#">69.16</a>
<code>wchar.h</code>	Gestione facilitata di caratteri estesi.	--
<code>wctype.h</code>	Classificazione dei caratteri estesi.	--

## 69.1 Funzionalità di libreria non dichiarate

Può succedere che il compilatore, per assolvere a funzionalità che figurano essere indipendenti da librerie, debba invece avvalersi di funzioni esterne che non sono previste dallo standard. In particolare, questo problema può verificarsi di fronte alla necessità di svolgere calcoli al di fuori della portata normale del microprocessore.

A titolo di esempio, il compilatore GNU C per la piattaforma x86-32 prevede un tipo intero `'long long int'` da 64 bit. Quando si vuole ottenere una divisione intera o il resto di una divisione con variabili di questo tipo, il compilatore GNU C richiama rispettivamente le funzioni `__udivdi3()` e `__umoddi3()`. In generale il problema non si avverte, ma se si vuole scrivere la propria libreria C, senza tali funzioni, in pratica non è possibile usare questo tipo intero molto grande.

Si vedano eventualmente i listati della sezione [95.2](#), relativi a os32, in cui si realizzano queste funzioni con il solo ausilio del linguaggio C.

## 69.2 File «assert.h»



Il file ‘assert.h’ della libreria standard definisce la macroistruzione **assert()**, da usare per generare informazioni diagnostiche, sulla base dell’esito della valutazione di un’espressione.

La macroistruzione **assert()** viene definita in due modi alternativi, in base alla presenza o meno della macro-variabile **NDEBUG**. Per la precisione, in presenza della macro-variabile **NDEBUG** la macroistruzione **assert()** deve risultare inerte.

```
#include <stdio.h>
#ifdef NDEBUG
#define assert(ignore) ((void)0)
#else
#define assert(ASSERTION) \
    ({if ((ASSERTION)==0) \
        fprintf (stderr, \
            "Assertion failed: " # ASSERTION \
            ", function %s, file %s, line %u.\n", \
            __func__, __FILE__, __LINE__);})
#endif
```

### 69.2.1 Utilizzo



La macroistruzione **assert()** va usata con la sintassi seguente, dove il parametro indica un’espressione di tipo non specificato, purché di tipo scalare:

```
void assert (espressione);
```

Se l’espressione si traduce in un valore *Falso*, ovvero pari a zero, la macroistruzione emette, attraverso lo standard error, un messaggio

contenente l'espressione stessa e altre indicazioni. Precisamente, oltre all'espressione deve apparire: il nome della funzione in cui ci si trova, il nome del file (sorgente) e il numero della riga.

Tuttavia, se la macro-variabile *NDEBUG* risulta definita, prima dell'inclusione del file 'assert.h', la macroistruzione *assert()* deve essere trasformata dal compilatore come un'istruzione inerte, ovvero l'equivalente di '(void) 0'.

Segue un l'esempio di un programma completo in cui si utilizza *assert()*:

```
#include <stdio.h>
int
main (int argc, char *argv[])
{
    assert (123==124);
    return 0;
}
```

L'espressione verificata da *assert()* non può essere vera, pertanto, se non è stata dichiarata la macro-variabile *NDEBUG*, questo programma dovrebbe produrre un messaggio come quello seguente:

```
Assertion failed: 123==124, function main, file assert.c,
line 5.
```

### 69.3 File «limits.h»

Il file 'limits.h' della libreria standard definisce delle macro-variabili che riepilogano i limiti dei valori rappresentabili con le variabili scalari intere. Lo standard prescrive dei limiti minimi per la conformità, ma le realizzazioni comuni consentono mediamente di rappresentare valori più grandi (in senso assoluto), a parità di tipo



di intero. Infatti, i limiti effettivi dipendono principalmente dalla dimensione della parola del microprocessore e dal modo in cui si rappresentano i valori negativi. Si può osservare che nelle architetture comuni, in cui i valori negativi si rappresentano con il complemento a due, il valore negativo più grande (in senso assoluto) di una variabile è pari a una unità in più rispetto al valore positivo massimo (per esempio il tipo **'signed char'** va solitamente da -128 a 127).

L'esempio proposto si riferisce a un'architettura a 32 bit con i valori negativi rappresentati attraverso il complemento a due.

```
#define CHAR_UNSIGNED    0

#define CHAR_BIT        8

#define SCHAR_MIN      (-0x80)
#define SCHAR_MAX      0x7F
#define UCHAR_MAX      0xFF

#ifdef CHAR_UNSIGNED
#   define CHAR_MIN    0
#   define CHAR_MAX    UCHAR_MAX
#else
#   define CHAR_MIN    SCHAR_MIN
#   define CHAR_MAX    SCHAR_MAX
#endif

#define MB_LEN_MAX      16

#define SHRT_MIN        (-0x8000)
#define SHRT_MAX        0x7FFF
#define USHRT_MAX       0xFFFF

#define INT_MIN          (-0x80000000)
#define INT_MAX          0x7FFFFFFF
```



```

#define UINT_MAX          0xFFFFFFFFU

#define LONG_MIN          (-0x80000000L)
#define LONG_MAX          0x7FFFFFFFL
#define ULONG_MAX         0xFFFFFFFFUL

#define LLONG_MIN         (-0x8000000000000000LL)
#define LLONG_MAX         0x7FFFFFFFFFFFFFFFLL
#define ULLONG_MAX        0xFFFFFFFFFFFFFFFFULL

```

Tabella 69.7. Macro-variabili standard per la rappresentazione dei limiti riferiti a variabili scalari intere.

Macro-variabile	Descrizione
CHAR_BIT	Indica la quantità di bit utilizzata per rappresentare il tipo <b>'char'</b> , con o senza segno. In altri termini è l'unità di memorizzazione più piccola con cui si può gestire l'insieme di caratteri minimo. Di norma si tratta di 8 bit.
SCHAR_MIN SCHAR_MAX	Indicano il valore minimo e il valore massimo rappresentabile in una variabile <b>'signed char'</b> .
UCHAR_MAX	Indica il valore massimo rappresentabile in una variabile <b>'unsigned char'</b> . Il valore minimo è zero.
CHAR_MIN CHAR_MAX	Indicano il valore minimo e il valore massimo rappresentabile in una variabile <b>'char'</b> . Questi valori dipendono dal fatto che il tipo <b>'char'</b> sia da intendere equivalente a un tipo <b>'unsigned char'</b> o <b>'signed char'</b> , da cui ereditano i limiti.
MB_LEN_MAX	Indica la quantità massima di byte che possono essere usati per rappresentare un carattere multibyte, qualunque sia la configurazione locale.

Macro-variabile	Descrizione
SHRT_MIN SHRT_MAX	Indicano il valore minimo e il valore massimo rappresentabile in una variabile <b>'short int'</b> .
USHRT_MAX	Indica il valore massimo rappresentabile in una variabile <b>'unsigned short int'</b> . Il valore minimo è zero.
INT_MIN INT_MAX	Indicano il valore minimo e il valore massimo rappresentabile in una variabile <b>'int'</b> .
UINT_MAX	Indica il valore massimo rappresentabile in una variabile <b>'unsigned int'</b> . Il valore minimo è zero.
LONG_MIN LONG_MAX	Indicano il valore minimo e il valore massimo rappresentabile in una variabile <b>'long int'</b> .
ULONG_MAX	Indica il valore massimo rappresentabile in una variabile <b>'unsigned long int'</b> . Il valore minimo è zero.
LLONG_MIN LLONG_MAX	Indicano il valore minimo e il valore massimo rappresentabile in una variabile <b>'long long int'</b> .
ULLONG_MAX	Indica il valore massimo rappresentabile in una variabile <b>'unsigned long long int'</b> . Il valore minimo è zero.

Eventualmente si veda la realizzazione di questo file nei sorgenti di os32 (listato [95.1.6](#)).

### 69.3.1 Confronto tra architetture

Per avere un'idea di come potrebbero svilupparsi i valori del file `limits.h` tra le varie architetture, viene mostrata una tabella in cui si possono paragonare quelli minimi stabiliti dallo standard con quelli usati nei sistemi GNU/Linux con architetture x86-32 e x86-64. Per semplicità si indicano solo i valori senza segno:

Macro-variabile	Standard	GNU/Linux x86-32	GNU/Linux x86-64
UCHAR_MAX	$2^8-1$	$2^8-1$	$2^8-1$
USHRT_MAX	$2^{16}-1$	$2^{16}-1$	$2^{16}-1$
UINT_MAX	$2^{16}-1$	$2^{32}-1$	$2^{32}-1$
ULONG_MAX	$2^{32}-1$	$2^{32}-1$	$2^{64}-1$
ULLONG_MAX	$2^{64}-1$	$2^{64}-1$	$2^{128}-1$

### 69.3.2 Estensioni POSIX

Per lo standard POSIX, il file `limits.h` serve anche per annotare limiti numerici relativi al funzionamento del sistema operativo, come per esempio la quantità massima di file aperti simultaneamente per ogni processo.

Un gruppo di macro-variabili definite nel file `limits.h`, caratterizzate per avere il prefisso `_POSIX_...`, `_POSIX2_...` e `_XOPEN_...`, definisce dei limiti minimi di compatibilità con lo standard. Per esempio, la macro-variabile `_POSIX_LINK_MAX` deve tradursi nel numero 8 e stabilisce che deve essere consentita la creazione di al-

meno otto collegamenti fisici per ogni file, in qualunque sistema POSIX.

Un secondo gruppo di macro-variabili definite nel file ‘limits.h’, dichiara i limiti massimi effettivi del sistema, riconducibili ai minimi già fissati nel primo gruppo già descritto. Per esempio, la macro-variabile **LINK\_MAX** indica il numero massimo effettivo di collegamenti fisici per file, tenendo conto che deve essere necessariamente maggiore o uguale al valore di **\_POSIX\_LINK\_MAX**.

Le macro-variabili del secondo gruppo sono facoltative, in quanto i limiti effettivi del sistema, per le varie voci, possono dipendere da fattori dinamici di funzionamento. In ogni caso, devono essere garantiti i valori minimi delle macro-variabili del primo gruppo.

Nell’ambito delle dichiarazioni che fanno già parte dello standard C, va osservato che lo standard POSIX richiede che il byte sia esattamente di 8 bit, pertanto la macro-variabile **CHAR\_BIT** deve tradursi necessariamente nel numero otto. Inoltre, si aggiungono anche qui alcune macro-variabili:

```
#define WORD_BIT    32
#define LONG_BIT    32
#define SSIZE_MAX   0x7FFFFFFFL
```

Tabella 69.10. Alcune macro-variabili aggiunte dallo standard POSIX.

Macro-variabile	Descrizione
WORD_BIT	Rappresenta la quantità di bit utilizzata per rappresentare il tipo ‘ <b>int</b> ’, con o senza segno. Il minimo valore accettabile è 32.

Macro-variabile	Descrizione
LONG_BIT	Rappresenta la quantità di bit utilizzata per rappresentare il tipo <code>'long int'</code> , con o senza segno. Il minimo valore accettabile è 32.
SSIZE_MAX	Rappresenta il valore positivo massimo che possa esprimere una variabile di tipo <code>'ssize_t'</code> (un tipo come <code>'size_t'</code> , ma con segno). Il minimo valore accettabile è quello della macro-variabile <code>_POSIX_SSIZE_MAX</code> , ovvero $7FFF_{16}$ .

## 69.4 File «stdint.h»

Il file `'stdint.h'` della libreria standard definisce principalmente dei tipi interi, alternativi a quelli tradizionali, riferiti in modo più diretto al rango. Assieme a questi tipi interi definisce anche delle macro-variabili che consentono di conoscere i limiti esatti di tali tipi, oltre ad altre macro-variabili con i limiti di tipi interi speciali, dichiarati in altri file (si veda eventualmente la realizzazione di questo file nei sorgenti di `os32`, listato [95.1.13](#)).

Lo standard prescrive che alcuni dei tipi definiti nel file `'stdint.h'` siano opzionali, purché sia rispettato un certo ordine (per esempio, se viene definito un tipo intero con segno, deve essere prevista anche una versione di quel tipo senza segno e viceversa). A tale proposito, le macro-variabili con le quali si possono verificare i limiti, servono anche per consentire al programmatore di verificare la disponibilità o meno del tipo relativo, attraverso istruzioni del precompilatore del tipo `'#ifdef'`.

L'esempio proposto si riferisce a un elaboratore x86-32 ed è abbastanza conforme alla configurazione che si può trovare in un sistema

## GNU/Linux.

## 69.4.1 Tipi interi ad ampiezza esatta

«

Lo standard prescrive un gruppo facoltativo di tipi interi il cui rango è definito precisamente dal nome. Si tratta dei tipi ‘`int $n$ _t`’ (con segno) e ‘`uint $n$ _t`’ (senza segno), dove  $n$  esprime la quantità di bit che compone l’intero. Si tratta necessariamente di tipi facoltativi, perché non è possibile stabilire in modo sicuro che in ogni architettura siano gestibili tipi interi di una data quantità di bit; per esempio, in una certa architettura «X» potrebbero essere gestiti tipi interi a 8, 16 e 32 bit, mentre in un’architettura «Y» i tipi disponibili effettivamente potrebbero essere a 8, 16, 24 e 32 bit.

Nei sistemi POSIX, questi tipi sono invece obbligatori, costringendo così ad avere byte esattamente di otto bit.

POSIX

```
typedef signed char      int8_t;
typedef short int       int16_t;
typedef int              int32_t;           // x86-32
typedef long long int   int64_t;           // x86-32

typedef unsigned char    uint8_t;
typedef unsigned short int uint16_t;
typedef unsigned int     uint32_t;        // x86-32
typedef unsigned long long int uint64_t;  // x86-32
```

Le macro-variabili usate per definire i limiti di questi valori interi hanno nomi del tipo ‘`INT $n$ _MIN`’, ‘`INT $n$ _MAX`’ e ‘`UINT $n$ _MAX`’, per indicare rispettivamente: il valore minimo dei tipi con segno; il valore massimo dei tipi con segno; il valore massimo dei tipi senza segno.

Lo standard prescrive precisamente questi valori minimi e massimi, intendendo implicitamente che i valori negativi si rappresentino con il complemento a due:

Macro-variabile	Valore standard
<code>INT<math>n</math>_MIN</code>	$-(2^{n-1})$
<code>INT<math>n</math>_MAX</code>	$2^{n-1}-1$
<code>UINT<math>n</math>_MAX</code>	$2^n-1$

<code>#define INT8_MIN</code>	<code>(-0x80)</code>
<code>#define INT16_MIN</code>	<code>(-0x8000)</code>
<code>#define INT32_MIN</code>	<code>(-0x80000000)</code>
<code>#define INT64_MIN</code>	<code>(-0x8000000000000000LL)</code>
<code>#define INT8_MAX</code>	<code>0x7F</code>
<code>#define INT16_MAX</code>	<code>0x7FFF</code>
<code>#define INT32_MAX</code>	<code>0x7FFFFFFF</code>
<code>#define INT64_MAX</code>	<code>0x7FFFFFFFFFFFFFFFLL</code>
<code>#define UINT8_MAX</code>	<code>0xFF</code>
<code>#define UINT16_MAX</code>	<code>0xFFFF</code>
<code>#define UINT32_MAX</code>	<code>0xFFFFFFFFU</code>
<code>#define UINT64_MAX</code>	<code>0xFFFFFFFFFFFFFFFFULL</code>

## 69.4.2 Tipi interi di rango minimo

Un gruppo richiesto espressamente dallo standard riguarda tipi interi il cui rango sia tale da garantire la rappresentazione di almeno  $n$  bit, utilizzando comunque la quantità minima possibile di bit. In questo caso i nomi sono `'int_least $n$ _t'` per i tipi con segno



e ‘`uint_leastn_t`’ per quelli senza segno. Lo standard prescrive che siano previsti necessariamente i tipi a 8, 16, 32 e 64 bit, mentre ammette che ne siano disponibili anche altri.

```
typedef signed char          int_least8_t;
typedef short int           int_least16_t;
typedef int                 int_least32_t; // x86-32
typedef long long int      int_least64_t; // x86-32

typedef unsigned char       uint_least8_t;
typedef unsigned short int  uint_least16_t;
typedef unsigned int        uint_least32_t; // x86-32
typedef unsigned long long int uint_least64_t; // x86-32
```

Le macro-variabili usate per definire i limiti di questi valori interi hanno nomi del tipo ‘`INT_LEASTn_MIN`’, ‘`INT_LEASTn_MAX`’ e ‘`UINT_LEASTn_MAX`’, per indicare rispettivamente: il valore minimo dei tipi con segno; il valore massimo dei tipi con segno; il valore massimo dei tipi senza segno. Lo standard attribuisce questi limiti in modo indipendente dalla rappresentazione dei valori negativi, ma tali limiti possono essere estesi in senso assoluto:

Macro-variabile	Valore standard che può essere superato in termini assoluti
<code>INT_LEASTn_MIN</code>	$-(2^{n-1}-1)$
<code>INT_LEASTn_MAX</code>	$2^{n-1}-1$
<code>UINT_LEASTn_MAX</code>	$2^n-1$

```
#define INT_LEAST8_MIN      (-0x80)
#define INT_LEAST16_MIN     (-0x8000)
#define INT_LEAST32_MIN     (-0x80000000)
#define INT_LEAST64_MIN     (-0x8000000000000000LL)
```



```
#define INT_LEAST8_MAX          0x7F
#define INT_LEAST16_MAX         0x7FFF
#define INT_LEAST32_MAX         0x7FFFFFFF
#define INT_LEAST64_MAX         0x7FFFFFFFFFFFFFFFLL

#define UINT_LEAST8_MAX         0xFF
#define UINT_LEAST16_MAX        0xFFFF
#define UINT_LEAST32_MAX        0xFFFFFFFFU
#define UINT_LEAST64_MAX        0xFFFFFFFFFFFFFFFFULL
```

Ai tipi interi di rango minimo sono associate anche delle macroistruzioni, il cui scopo è quello di consentire la rappresentazione corretta dei valori costanti:

```
INTn_C (valore)
```

```
UINTn_C (valore)
```

In pratica, per indicare il valore costante 1234567890, precisando che va inteso come un tipo ‘**uint\_least64\_t**’, si deve scrivere: ‘**UINT64\_C(1234567890)**’. Il valore costante in sé, può essere espresso in qualunque modo, purché sia ammissibile nel contesto comune.

```

#define INT8_C (VAL)      VAL
#define INT16_C (VAL)     VAL
#define INT32_C (VAL)    VAL           // x86-32
#define INT64_C (VAL)    VAL ## LL    // x86-32

#define UINT8_C (VAL)     VAL
#define UINT16_C (VAL)   VAL
#define UINT32_C (VAL)   VAL ## U     // x86-32
#define UINT64_C (VAL)   VAL ## ULL   // x86-32

```

È evidente che, nel caso dell'esempio mostrato, `'UINT64_C(1234567890)'` corrisponde a `'1234567890ULL'`.

### 69.4.3 Tipi interi «veloci»

«

Un altro gruppo di tipi richiesti dallo standard è quello il cui rango è tale da consentire la rappresentazione di almeno  $n$  bit, utilizzando la quantità minima di bit che garantisce tempi ottimali di elaborazione. In questo caso i nomi sono `'int_fastn_t'` per i tipi con segno e `'uint_fastn_t'` per quelli senza segno. Lo standard prescrive che siano previsti necessariamente i tipi a 8, 16, 32 e 64 bit, mentre ammette che ne siano disponibili anche altri.

```

typedef signed char      int_fast8_t;
typedef int              int_fast16_t; // x86-32
typedef int              int_fast32_t; // x86-32
typedef long long int    int_fast64_t; // x86-32

typedef unsigned char    uint_fast8_t;
typedef unsigned int     uint_fast16_t; // x86-32
typedef unsigned int     uint_fast32_t; // x86-32
typedef unsigned long long int uint_fast64_t; // x86-32

```

Come suggerisce l'esempio, è ragionevole pensare che, dove possibile, il rango usato effettivamente sia quello del tipo intero

normale.

Le macro-variabili usate per definire i limiti di questi valori interi hanno nomi del tipo ‘**INT\_FAST $n$ \_MIN**’, ‘**INT\_FAST $n$ \_MAX**’ e ‘**UINT\_FAST $n$ \_MAX**’, per indicare rispettivamente: il valore minimo dei tipi con segno; il valore massimo dei tipi con segno; il valore massimo dei tipi senza segno. Lo standard attribuisce questi limiti in modo indipendente dalla rappresentazione dei valori negativi, ma tali limiti possono essere estesi in senso assoluto:

Macro-variabile	Valore standard che può essere superato in termini assoluti
<code>INT_FAST<math>n</math>_MIN</code>	$-(2^{n-1}-1)$
<code>INT_FAST<math>n</math>_MAX</code>	$2^{n-1}-1$
<code>UINT_FAST<math>n</math>_MAX</code>	$2^n-1$

<code>#define INT_FAST8_MIN</code>	<code>(-0x80)</code>
<code>#define INT_FAST16_MIN</code>	<code>(-0x80000000)</code>
<code>#define INT_FAST32_MIN</code>	<code>(-0x80000000)</code>
<code>#define INT_FAST64_MIN</code>	<code>(-0x8000000000000000LL)</code>
<code>#define INT_FAST8_MAX</code>	<code>0x7F</code>
<code>#define INT_FAST16_MAX</code>	<code>0x7FFFFFFF</code>
<code>#define INT_FAST32_MAX</code>	<code>0x7FFFFFFF</code>
<code>#define INT_FAST64_MAX</code>	<code>0x7FFFFFFFFFFFFFFFLL</code>
<code>#define UINT_FAST8_MAX</code>	<code>0xFF</code>
<code>#define UINT_FAST16_MAX</code>	<code>0xFFFFFFFFU</code>
<code>#define UINT_FAST32_MAX</code>	<code>0xFFFFFFFFU</code>
<code>#define UINT_FAST64_MAX</code>	<code>0xFFFFFFFFFFFFFFFFULL</code>

## 69.4.4 Tipi interi per rappresentare dei puntatori

«

Sono previsti due tipi opzionali interi, adatti a contenere il valore di un puntatore, garantendo che la conversione da e verso `'void *'` avvenga sempre correttamente. Per la precisione si tratta di `'intptr_t'` e `'uintptr_t'`, dove il primo è un intero con segno, mentre il secondo è senza segno.

```
typedef int                intptr_t;           // x86-32
typedef unsigned int      uintptr_t;         // x86-32
```

Le macro-variabili usate per definire i limiti di questi valori interi sono *INTPTR\_MIN*, *INTPTR\_MAX* e *UINTPTR\_MAX*, per indicare rispettivamente: il valore minimo con segno, il valore massimo con segno e il valore massimo senza segno. Lo standard attribuisce dei limiti riferiti ad architetture in grado di indirizzare al massimo con 16 bit e ovviamente vanno adattati alla realtà dell'architettura effettiva:

Macro-variabile	Valore standard che può essere superato in termini assoluti
INTPTR_MIN	$-(2^{15}-1)$
INTPTR_MAX	$2^{15}-1$
UINTPTR_MAX	$2^{16}-1$

```
#define INTPTR_MIN        (-0x80000000)
#define INTPTR_MAX        0x7FFFFFFF
#define UINTPTR_MAX       0xFFFFFFFFU
```

## 69.4.5 Tipi interi di rango massimo

Per poter rappresentare in modo indipendente dall'architettura degli interi di rango massimo, sono previsti due tipi specifici, richiesti espressamente dallo standard: `'intmax_t'` e `'uintmax_t'`. Le macro-variabili che definiscono i limiti sono: `INTMAX_MIN`, `INTMAX_MAX` e `UINTMAX_MAX`. Lo standard prescrive che si tratti di variabili con un rango di almeno 64 bit.

```
typedef long long int          intmax_t;          // x86-32
typedef unsigned long long int uintmax_t;        // x86-32
```

Macro-variabile	Valore standard che può essere superato in termini assoluti
<code>INTMAX_MIN</code>	$-(2^{63}-1)$
<code>INTMAX_MAX</code>	$2^{63}-1$
<code>UINTMAX_MAX</code>	$2^{64}-1$

Anche ai tipi interi di rango massimo sono associate delle macroistruzioni per facilitare la rappresentazione corretta dei valori costanti:

```
INTMAX_C (valore)
```

```
UINTMAX_C (valore)
```

In pratica, per indicare il valore costante 1234567890, precisando che va inteso come un tipo `'uintmax_t'`, si deve scrivere:

‘**UINTMAX\_C(1234567890)**’. Il valore costante in sé, può essere espresso in qualunque modo, purché sia ammissibile nel contesto comune.

```
#define INTMAX_C (VAL)    VAL ## LL           // x86-32
#define UINTMAX_C (VAL)  VAL ## ULL          // x86-32
```

A questo punto, anche la definizione dei valori minimi e massimi diventa più agevole:

```
// x86-32
#define INTMAX_MIN      (-INTMAX_C(0x8000000000000000))
#define INTMAX_MAX      (INTMAX_C(0x7FFFFFFFFFFFFFFFFF))
#define UINTMAX_MAX     (UINTMAX_C(0xFFFFFFFFFFFFFFFF))
```

## 69.4.6 Limiti per altri tipi interi

«

Altri tipi interi dichiarati al di fuori del file ‘`stdint.h`’ hanno i limiti definiti qui. Ne viene mostrata solo una tabella riepilogativa.

Macro-variabile	Descrizione
PTRDIFF_MIN PTRDIFF_MAX	Indicano il valore minimo e il valore massimo rappresentabile in una variabile ‘ <code>ptrdiff_t</code> ’.
SIG_ATOMIC_MIN SIG_ATOMIC_MAX	Indicano il valore minimo e il valore massimo rappresentabile in una variabile ‘ <code>sig_atomic_t</code> ’.
SIZE_MAX	Indica il valore massimo rappresentabile in una variabile ‘ <code>size_t</code> ’, la quale è destinata a contenere valori senza segno.
WCHAR_MIN WCHAR_MAX	Indicano il valore minimo e il valore massimo rappresentabile in una variabile ‘ <code>wchar_t</code> ’.

Macro-variabile	Descrizione
WINT_MIN WINT_MAX	Indicano il valore minimo e il valore massimo rappresentabile in una variabile <code>'wint_t'</code> .

L'esempio seguente riporta i valori usati in un sistema GNU/Linux con architettura x86-32, a parte il caso di `'wchar_t'` e `'wint_t'` che si intendono rispettivamente a 32 bit senza segno e 64 bit con segno:

<code>#define PTRDIFF_MIN</code>	<code>(-0x80000000)</code>	<code>// x86-32</code>
<code>#define PTRDIFF_MAX</code>	<code>0x7FFFFFFF</code>	<code>// x86-32</code>
<code>#define SIG_ATOMIC_MIN</code>	<code>(-0x80000000)</code>	<code>// x86-32</code>
<code>#define SIG_ATOMIC_MAX</code>	<code>0x7FFFFFFF</code>	<code>// x86-32</code>
<code>#define SIZE_MAX</code>	<code>0xFFFFFFFFU</code>	<code>// x86-32</code>
<code>#define WCHAR_MIN</code>	<code>0x00000000</code>	
<code>#define WCHAR_MAX</code>	<code>0xFFFFFFFFU</code>	<code>// x86-32</code>
<code>#define WINT_MIN</code>	<code>(-0x8000000000000000LL)</code>	<code>// x86-32</code>
<code>#define WINT_MAX</code>	<code>0x7FFFFFFFFFFFFFFFLL</code>	<code>// x86-32</code>

## 69.5 File «errno.h»

Il file `'errno.h'` della libreria standard definisce principalmente delle macro-variabili per rappresentare simbolicamente delle situazioni di errore. Queste macro-variabili si espandono in un numero intero e positivo, di tipo `'int'`, ma la corrispondenza tra l'errore simbolico rappresentato dalla macro-variabile e il numero in cui questa si deve espandere, dipende dalle convenzioni del sistema operativo (si veda eventualmente la realizzazione del file `'errno.h'` nei sorgenti



di `os32`, sezione [95.5](#), tenendo conto che lì si aggiungono delle funzioni non standard, con le quali è più facile l'individuazione della posizione del sorgente in cui l'errore si è manifestato).

Lo standard del linguaggio prescrive poche macro-variabili, da cui dipendono le librerie standard, mentre tutte le altre sono competenza delle convenzioni del sistema operativo.

Oltre alle macro-variabili che rappresentano le situazioni di errore previste, il file `'errno.h'` deve dichiarare *errno*, in qualità di espressione che si traduca in una variabile scalare. In pratica può trattarsi di una variabile esterna o di una macro-variabile che si traduce in qualunque cosa consenta di assegnarvi un valore. Il valore iniziale che si può leggere da *errno* è zero (con cui si intende l'assenza di qualunque tipo di situazione di errore) e viene modificato dalle funzioni che, di volta in volta, possono avere bisogno di annotare uno stato di errore.

Si osservi che i numeri che si vedono associati alle macro-variabili sono stati tratti, come esempio, dalla configurazione di un sistema GNU/Linux.

```
extern int errno;

#define EDOM      33
#define EILSEQ   84
#define ERANGE   34
```

Il nome *errno*, in un sistema che consenta l'esecuzione di programmi suddivisi in più thread, deve tradursi in un'espressione tale da rappresentare una variabile scalare individuale per ogni thread, in modo che i thread non possano interferire tra di loro a questo proposito. Evidentemente, l'esempio mostrato non offre questa



accortezza.

Tabella 69.31. Macro-variabili standard per la rappresentazione di situazioni di errore.

Macro-variabile	Descrizione
EDOM	Errore di dominio: l'argomento di una funzione matematica ha un valore al di fuori del dominio previsto.
EILSEQ	Errore di codifica: la sequenza dei byte che deve rappresentare una certa codifica contiene un errore.
ERANGE	Errore nell'intervallo di valori: il risultato di un'espressione matematica non può essere rappresentato nell'intervallo di valori previsto (ovvero nella variabile che deve riceverlo).

Il file di intestazione `errno.h` di un sistema POSIX è più articolato, in quanto contiene un elenco numeroso di macro-variabili. Valgono naturalmente le stesse considerazioni per la variabile globale *errno*, a proposito dei thread multipli. Nell'esempio successivo, i numeri associati alle varie macro-variabili non fanno riferimento ad alcun sistema operativo reale; va osservato inoltre che ogni sistema POSIX aggiunge propri tipi di errore, necessari per le proprie caratteristiche specifiche.

```
extern int errno;

#define E2BIG          1  // Argument list too long.
#define EACCES        2  // Permission denied.
#define EADDRINUSE    3  // Address in use.
#define EADDRNOTAVAIL 4  // Address not available.
#define EAFNOSUPPORT  5  // Address family not supported.
#define EAGAIN        6  // Resource unavailable, try
                        // again.
#define EALREADY      7  // Connection already in
```

```

// progress.
#define EBADF      8 // Bad file descriptor.
#define EBADMSG   9 // Bad message.
#define EBUSY    10 // Device or resource busy.
#define ECANCELED 11 // Operation canceled.
#define ECHILD   12 // No child processes.
#define ECONNABORTED 13 // Connection aborted.
#define ECONNREFUSED 14 // Connection refused.
#define ECONNRESET 15 // Connection reset.
#define EDEADLK  16 // Resource deadlock would occur.
#define EDESTADDRREQ 17 // Destination address required.
#define EDOM     18 // Mathematics argument out of
// domain of function.
#define EDQUOT   19 // Reserved.
#define EEXIST   20 // File exists.
#define EFAULT   21 // Bad address.
#define EFBIG    22 // File too large.
#define EHOSTUNREACH 23 // Host is unreachable.
#define EIDRM    24 // Identifier removed.
#define EILSEQ   25 // Illegal byte sequence.
#define EINPROGRESS 26 // Operation in progress.
#define EINTR    27 // Interrupted function.
#define EINVAL   28 // Invalid argument.
#define EIO      29 // I/O error.
#define EISCONN  30 // Socket is connected.
#define EISDIR   31 // Is a directory.
#define ELOOP    32 // Too many levels of symbolic
// links.
#define EMFILE   33 // Too many open files.
#define EMLINK   34 // Too many links.
#define EMSGSIZE 35 // Message too large.
#define EMULTIHOP 36 // Reserved.
#define ENAMETOOLONG 37 // Filename too long.
#define ENETDOWN 38 // Network is down.
```

```
#define ENETRESET          39 // Connection aborted by network.
#define ENETUNREACH       40 // Network unreachable.
#define ENFILE            41 // Too many files open in system.
#define ENOBUFS           42 // No buffer space available.
#define ENODATA           43 // No message is available on the
// stream head read queue.
#define ENODEV            44 // No such device.
#define ENOENT            45 // No such file or directory.
#define ENOEXEC           46 // Executable file format error.
#define ENOLCK            47 // No locks available.
#define ENOLINK           48 // Reserved.
#define ENOMEM            49 // Not enough space.
#define ENOMSG            50 // No message of the desired
// type.
#define ENOPROTOOPT       51 // Protocol not available.
#define ENOSPC            52 // No space left on device.
#define ENOSR             53 // No stream resources.
#define ENOSTR            54 // Not a stream.
#define ENOSYS            55 // Function not supported.
#define ENOTCONN          56 // The socket is not connected.
#define ENOTDIR           57 // Not a directory.
#define ENOTEMPTY         58 // Directory not empty.
#define ENOTSOCK          59 // Not a socket.
#define ENOTSUP           60 // Not supported.
#define ENOTTY            61 // Inappropriate I/O control
// operation.
#define ENXIO             62 // No such device or address.
#define EOPNOTSUPP        63 // Operation not supported on
// socket.
#define EOVERFLOW         64 // Value too large to be stored
// in data type.
#define EPERM             65 // Operation not permitted.
#define EPIPE             66 // Broken pipe.
#define EPROTO            67 // Protocol error.
```

```
#define EPROTONOSUPPORT 68 // Protocol not supported.
#define EPROTOTYPE      69 // Protocol wrong type for
                        // socket.
#define ERANGE          70 // Result too large.
#define EROFS           71 // Read-only file system.
#define ESPIPE          72 // Invalid seek.
#define ESRCH           73 // No such process.
#define ESTALE          74 // Reserved.
#define ETIME           75 // Stream ioctl() timeout.
#define ETIMEDOUT      76 // Connection timed out.
#define ETXTBSY        77 // Text file busy.
#define EWOULDBLOCK    78 // Operation would block (may
                        // be the same as EAGAIN).
#define EXDEV          79 // Cross-device link.
```

## 69.6 File «locale.h»

«

Il file ‘locale.h’ della libreria standard definisce delle macrovariabili, un tipo di struttura e alcune funzioni, relative alla gestione della configurazione locale del programma. Se non si fa uso di tale configurazione, il proprio programma opera in quella che è nota come «configurazione locale C», ovvero il minimo indispensabile.

Nell’ambito di un sistema operativo Unix o simile, la configurazione locale avviene principalmente attraverso l’impostazione di variabili di ambiente il cui nome inizia per ‘LC\_...’. Tuttavia, questa configurazione non viene ereditata automaticamente dal programma scritto in linguaggio C, perché questo deve acquisirla espressamente, se vuole.

La definizione della configurazione locale avviene attraverso una stringa contenente delle sigle che esprimono la lingua, la nazionalità e la codifica da utilizzare per rappresentare i caratteri. Per esem-

pio, `'it_CH.UTF-8'` rappresenta la lingua italiana, la nazionalità svizzera e la codifica UTF-8.

### 69.6.1 Impostazione della configurazione locale

Per modificare l'impostazione della configurazione locale del proprio programma, si utilizza la funzione *setlocale()* che richiede l'indicazione di un numero intero, a rappresentare la *categoria* nella quale intervenire. La categoria viene definita formalmente attraverso delle macro-variabili il cui nome inizia per *LC\_...* e si tratta degli stessi nomi usati nelle variabili di ambiente di un sistema Unix o simile. Viene proposto un esempio di dichiarazione delle macro-variabili indispensabili, ma l'associazione al numero varia molto da un sistema all'altro:

LC_ALL	0
LC_COLLATE	1
LC_CTYPE	2
LC_MONETARY	3
LC_NUMERIC	4
LC_TIME	5

La macro variabile successiva è un'estensione usata nei sistemi POSIX:

LC_MESSAGES	6
-------------	---

La funzione *setlocale()* con cui si cambia la configurazione locale ha il prototipo seguente:

```
char *setlocale (int category, const char *locale);
```

Si prevedono due situazioni diverse di utilizzo della funzione. Per cominciare può essere usata per interrogare la configurazione attuale, come nell'esempio seguente:



```
...  
char *p;  
p = setlocale (LC_COLLATE, NULL);  
...
```

Fornendo un puntatore nullo, al posto della stringa che deve indicare la configurazione locale, si ottiene un puntatore alla stringa che descrive quella attuale. In questo caso, se si utilizza la categoria **‘LC\_ALL’**, si ottiene una stringa che descrive tutte le altre categorie, ammesso che ci siano delle differenze. Se invece la funzione non è in grado di dare questa informazione, si ottiene semplicemente un puntatore nullo.

Naturalmente, la funzione serve anche per cambiare la configurazione locale, specificando in tal caso la stringa che la descrive. Per esempio, nel modo seguente si interviene nella categoria **‘LC\_NUMERIC’**:

```
...  
char *p;  
p = setlocale (LC_NUMERIC, "it_IT.UTF-8");  
...
```

Anche in questo caso si ottiene un puntatore che descrive la categoria scelta, ma se l’operazione fallisce, si ottiene invece il puntatore nullo.

Normalmente è più probabile che, nell’impostazione della configurazione locale si voglia indicare una modalità unica per tutte le categorie; pertanto, in tal caso va usato **‘LC\_ALL’**:

```
...  
p = setlocale (LC_ALL, "it_IT.UTF-8");  
...
```

Viene mostrato un esempio di programma completo, in cui si imposta prima la configurazione locale complessiva, poi se ne cambia una e quindi si interroga la situazione:

```
#include <stdio.h>  
#include <locale.h>  
  
int main (void)  
{  
    setlocale (LC_ALL, "it_IT.UTF-8");  
    setlocale (LC_MONETARY, "en_US.UTF-8");  
  
    printf ("LC_COLLATE:  \">%s\<"\           setlocale (LC_COLLATE, NULL));  
    printf ("LC_CTYPE:    \">%s\<"\           setlocale (LC_CTYPE, NULL));  
    printf ("LC_MONETARY: \">%s\<"\           setlocale (LC_MONETARY, NULL));  
    printf ("LC_NUMERIC:  \">%s\<"\           setlocale (LC_NUMERIC, NULL));  
    printf ("LC_TIME:     \">%s\<"\           setlocale (LC_TIME, NULL));  
    printf ("LC_ALL:      \">%s\<"\           setlocale (LC_ALL, NULL));  
#ifdef LC_MESSAGES  
    printf ("LC_MESSAGES: \">%s\<"\           setlocale (LC_MESSAGES, NULL));  
#endif  
  
    return 0;  
}
```

Ecco cosa si può ottenere:

```
LC_COLLATE: "it_IT.UTF-8"
LC_CTYPE: "it_IT.UTF-8"
LC_MONETARY: "en_US.UTF-8"
LC_NUMERIC: "it_IT.UTF-8"
LC_TIME: "it_IT.UTF-8"
LC_ALL: "LC_CTYPE=it_IT.UTF-8;LC_NUMERIC=it_IT.UTF-8;↵
↵LC_TIME=it_IT.UTF-8;LC_COLLATE=it_IT.UTF-8;LC_MONETARY=en_US.UTF
↵LC_MESSAGES=it_IT.UTF-8"
LC_MESSAGES: "it_IT.UTF-8"
```

Per fare sì che il programma erediti la configurazione locale dal contesto in cui si trova a funzionare (quindi dalla configurazione locale del sistema operativo), si può indicare la stringa nulla al posto della definizione:

```
...
p = setlocale (LC_ALL, "");
...
```

Pertanto, questo è il modo appropriato per iniziare la configurazione all'interno di un programma.

## 69.6.2 Composizione dei valori numerici

« Il modo in cui si rappresenta testualmente un valore numerico, con o senza indicazione della valuta (la moneta), dipende dalla configurazione locale. La funzione *localeconv()* restituisce il puntatore a una struttura che contiene i dettagli riguardo alle modalità di rappresentazione dei valori numerici, secondo la configurazione locale. L'utilizzo consentito di questa struttura si limita all'interrogazione dei valori, perché la modifica dipende dalla gestione della configurazione locale.



```

struct lconv {char *decimal_point;
              char *thousands_sep;
              char *grouping;
              char *mon_decimal_point;
              char *mon_thousands_sep;
              char *mon_grouping;
              char *positive_sign;
              char *negative_sign;
              char *currency_symbol;
              char frac_digits;
              char p_cs_precedes;
              char n_cs_precedes;
              char p_sep_by_space;
              char n_sep_by_space;
              char p_sign_posn;
              char n_sign_posn;
              char *int_curr_symbol;
              char int_frac_digits;
              char int_p_cs_precedes;
              char int_n_cs_precedes;
              char int_p_sep_by_space;
              char int_n_sep_by_space;
              char int_p_sign_posn;
              char int_n_sign_posn;
};

```

A titolo di esempio vengono descritti solo alcuni membri della struttura; per gli altri si deve consultare la documentazione dello standard:

Membro	Descrizione
decimal_point	Stringa contenente il carattere usato per separare la parte decimale in un numero per uso generale.

Membro	Descrizione
<code>thousand_sep</code>	Stringa contenente il carattere usato per separare le cifre della parte intera di un numero per uso generale.
<code>positive_sign</code> <code>negative_sign</code>	Stringa contenente il carattere usato per rappresentare il segno, positivo o negativo, di un numero usato per le valute.
<code>int_curr_symbol</code>	Stringa composta da quattro caratteri, di cui i primi tre indicano la sigla internazionale della valuta (' <b>USD</b> ', ' <b>EUR</b> ', ecc.) e il quarto è solo un carattere di separazione da usare tra tale sigla e il valore numerico a cui questa si riferisce.

I membri che rappresentano delle stringhe (puntatori a carattere), quando si riferiscono a dati facoltativi, possono essere vuoti (nel senso di stringhe nulle). I membri di tipo '**char**' vengono usati in modo numerico.

```
struct lconv *localeconv (void);
```

Come si vede dal prototipo, la funzione *localeconv()* serve esclusivamente per ottenere il puntatore alla struttura '**lconv**', da usare per la sua consultazione. Viene mostrato un esempio molto semplice per il suo utilizzo:

```
#include <stdio.h>
#include <locale.h>

int main (void)
{
    struct lconv *lc;
```

```
setlocale (LC_ALL, "it_IT.UTF-8");
lc = localeconv ();

printf ("decimal_point:\t\t\"%s\"\n",
        lc->decimal_point);
printf ("thousand_sep:\t\t\"%s\"\n",
        lc->thousands_sep);
printf ("grouping:\t\t\"%s\"\n",
        lc->grouping);
printf ("mon_decimal_point:\t\"%s\"\n",
        lc->mon_decimal_point);
printf ("mon_thousands_sep:\t\"%s\"\n",
        lc->mon_thousands_sep);
printf ("mon_grouping:\t\t\"%s\"\n",
        lc->mon_grouping);
printf ("positive_sign:\t\t\"%s\"\n",
        lc->positive_sign);
printf ("negative_sign:\t\t\"%s\"\n",
        lc->negative_sign);
printf ("currency_symbol:\t\"%s\"\n",
        lc->currency_symbol); // Multibyte.
printf ("frac_digits:\t\t%i\n",
        lc->frac_digits);
printf ("p_cs_precedes:\t\t%i\n",
        lc->p_cs_precedes);
printf ("n_cs_precedes:\t\t%i\n",
        lc->n_cs_precedes);
printf ("p_sep_by_space:\t\t%i\n",
        lc->p_sep_by_space);
printf ("n_sep_by_space:\t\t%i\n",
        lc->n_sep_by_space);
printf ("p_sign_posn:\t\t%i\n",
        lc->p_sign_posn);
printf ("n_sign_posn:\t\t%i\n",
```

```
        lc->n_sign_posn);
printf ("int_curr_symbol:\t\"%s\"\n",
        lc->int_curr_symbol);
printf ("int_frac_digit:\t\t%i\n",
        lc->int_frac_digit);
printf ("int_p_cs_precedes:\t\t%i\n",
        lc->int_p_cs_precedes);
printf ("int_n_cs_precedes:\t\t%i\n",
        lc->int_n_cs_precedes);
printf ("int_p_sep_by_space:\t\t%i\n",
        lc->int_p_sep_by_space);
printf ("int_n_sep_by_space:\t\t%i\n",
        lc->int_n_sep_by_space);
printf ("int_p_sign_posn:\t\t%i\n",
        lc->int_p_sign_posn);
printf ("int_n_sign_posn:\t\t%i\n",
        lc->int_n_sign_posn);

return 0;
}
```

Il risultato che si ottiene dovrebbe essere molto simile a quello seguente:

decimal_point:	","
thousand_sep:	""
grouping:	""
mon_decimal_point:	","
mon_thousands_sep:	."
mon_grouping:	""
positive_sign:	""
negative_sign:	"-"
currency_symbol:	"€"
frac_digits:	2

```

p_cs_precedes:      1
n_cs_precedes:      1
p_sep_by_space:     1
n_sep_by_space:     1
p_sign_posn:        1
n_sign_posn:        1
int_curr_symbol:    "EUR "
int_frac_digit:     2
int_p_cs_precedes:  1
int_n_cs_precedes:  1
int_p_sep_by_space: 1
int_n_sep_by_space: 1
int_p_sign_posn:    1
int_n_sign_posn:    4

```

Si osservi che, nell'esempio, la stringa a cui si accede tramite il membro `'currency_symbol'` è una sequenza «multibyte», nel senso che utilizza più byte per rappresentare un solo carattere.

## 69.7 File «ctype.h»

Il file `'ctype.h'` della libreria standard definisce alcune funzioni per la classificazione e la trasformazione dei caratteri (intesi come `'char'`). Gli esempi proposti qui riguardano esclusivamente l'insieme di caratteri corrispondente alla codifica ASCII e, di conseguenza, la configurazione locale `'C'`. Tuttavia va ricordato che il linguaggio C non impone che l'insieme di caratteri minimo sia descritto attraverso la codifica ASCII, mentre così è invece nello standard POSIX. (si veda eventualmente la realizzazione del file `'ctype.h'` nei sorgenti di `os32`, listato [95.1.5](#)).

Le funzioni di questo file hanno in comune il parametro, costituito da un valore intero di tipo `'int'`, usato per rappresentare il caratte-

re.<sup>1</sup> Le funzioni del tipo *is...()* restituiscono un valore intero diverso da zero (corrispondente a *Vero*) se la condizione riferita al carattere fornito si verifica. Le funzioni *to...()* restituiscono un valore intero, corrispondente al carattere fornito e trasformato nel modo richiesto, se ciò è possibile.

Listato 69.47. Prototipi delle funzioni dichiarate nel file ‘ctype.h’.

```
int isalnum (int c);
int isalpha (int c);
int isblank (int c);
int iscntrl (int c);
int isdigit (int c);
int isgraph (int c);
int islower (int c);
int isprint (int c);
int ispunct (int c);
int isspace (int c);
int isupper (int c);
int isxdigit (int c);
int tolower (int c);
int toupper (int c);
```

Listato 69.48. Prototipi aggiuntivi dello standard POSIX.

```
int isascii (int c); // POSIX
int toascii (int c); // POSIX
```

### 69.7.1 Funzioni «is...()»

«

Il gruppo di funzioni *is...()* restituisce un valore intero diverso da zero (corrispondente a *Vero*) se la condizione riferita al carattere fornito si verifica. Vengono proposte le varie soluzioni, affiancando la tabella ASCII con i caratteri validi evidenziati.

Listato 69.49. Funzione *isalnum()*.

<b>NUL</b> 0000	<b>DLE</b> 0010	<b>SP</b> 0020	<b>0</b> 0030	<b>@</b> 0040	<b>P</b> 0050	<b>'</b> 0060	<b>p</b> 0070
<b>SOH</b> 0001	<b>DC1</b> 0011	<b>!</b> 0021	<b>1</b> 0031	<b>A</b> 0041	<b>Q</b> 0051	<b>a</b> 0061	<b>q</b> 0071
<b>STX</b> 0002	<b>DC2</b> 0012	<b>"</b> 0022	<b>2</b> 0032	<b>B</b> 0042	<b>R</b> 0052	<b>b</b> 0062	<b>r</b> 0072
<b>ETX</b> 0003	<b>DC3</b> 0013	<b>#</b> 0023	<b>3</b> 0023	<b>C</b> 0043	<b>S</b> 0053	<b>c</b> 0063	<b>s</b> 0073
<b>EOT</b> 0004	<b>DC4</b> 0014	<b>\$</b> 0024	<b>4</b> 0034	<b>D</b> 0044	<b>T</b> 0054	<b>d</b> 0064	<b>t</b> 0074
<b>ENQ</b> 0005	<b>NAK</b> 0015	<b>%</b> 0025	<b>5</b> 0035	<b>E</b> 0045	<b>U</b> 0055	<b>e</b> 0065	<b>u</b> 0075
<b>ACK</b> 0006	<b>SYN</b> 0016	<b>&amp;</b> 0026	<b>6</b> 0036	<b>F</b> 0046	<b>V</b> 0056	<b>f</b> 0066	<b>v</b> 0076
<b>BEL</b> 0007	<b>ETB</b> 0017	<b>'</b> 0027	<b>7</b> 0037	<b>G</b> 0047	<b>W</b> 0057	<b>g</b> 0067	<b>w</b> 0077
<b>BS</b> 0008	<b>CAN</b> 0018	<b>(</b> 0028	<b>8</b> 0038	<b>H</b> 0048	<b>X</b> 0058	<b>h</b> 0068	<b>x</b> 0078
<b>HT</b> 0009	<b>EM</b> 0019	<b>)</b> 0029	<b>9</b> 0039	<b>I</b> 0049	<b>Y</b> 0059	<b>i</b> 0069	<b>y</b> 0079
<b>LF</b> 000A	<b>SUB</b> 001A	<b>*</b> 002A	<b>:</b> 003A	<b>J</b> 004A	<b>Z</b> 005A	<b>j</b> 006A	<b>z</b> 007A
<b>VT</b> 000B	<b>ESC</b> 001B	<b>+</b> 002B	<b>;</b> 003B	<b>K</b> 004B	<b>[</b> 005B	<b>k</b> 006B	<b>{</b> 007B
<b>FF</b> 000C	<b>FS</b> 001C	<b>,</b> 002C	<b>&lt;</b> 003C	<b>L</b> 004C	<b>\</b> 005C	<b>l</b> 006C	<b> </b> 007C
<b>CR</b> 000D	<b>GS</b> 001D	<b>-</b> 002D	<b>=</b> 003D	<b>M</b> 004D	<b>]</b> 005D	<b>m</b> 006D	<b>}</b> 007D
<b>SO</b> 000E	<b>RS</b> 001E	<b>.</b> 002E	<b>&gt;</b> 003E	<b>N</b> 004E	<b>^</b> 005E	<b>n</b> 006E	<b>~</b> 007E
<b>SI</b> 000F	<b>US</b> 001F	<b>/</b> 002F	<b>?</b> 003F	<b>O</b> 004F	<b>_</b> 005F	<b>o</b> 006F	<b>DEL</b> 007F

```
#include <ctype.h>
int
isalnum (int c)
{
    if (isalpha (c)
        || isdigit (c))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Listato 69.50. Funzione *isalpha()*.

<b>NUL</b> 0000	<b>DLE</b> 0010	<b>SP</b> 0020	<b>0</b> 0030	<b>@</b> 0040	<b>P</b> 0050	<b>'</b> 0060	<b>p</b> 0070
<b>SOH</b> 0001	<b>DC1</b> 0011	<b>!</b> 0021	<b>1</b> 0031	<b>A</b> 0041	<b>Q</b> 0051	<b>a</b> 0061	<b>q</b> 0071
<b>STX</b> 0002	<b>DC2</b> 0012	<b>"</b> 0022	<b>2</b> 0032	<b>B</b> 0042	<b>R</b> 0052	<b>b</b> 0062	<b>r</b> 0072
<b>ETX</b> 0003	<b>DC3</b> 0013	<b>#</b> 0023	<b>3</b> 0023	<b>C</b> 0043	<b>S</b> 0053	<b>c</b> 0063	<b>s</b> 0073
<b>EOT</b> 0004	<b>DC4</b> 0014	<b>\$</b> 0024	<b>4</b> 0034	<b>D</b> 0044	<b>T</b> 0054	<b>d</b> 0064	<b>t</b> 0074
<b>ENQ</b> 0005	<b>NAK</b> 0015	<b>%</b> 0025	<b>5</b> 0035	<b>E</b> 0045	<b>U</b> 0055	<b>e</b> 0065	<b>u</b> 0075
<b>ACK</b> 0006	<b>SYN</b> 0016	<b>&amp;</b> 0026	<b>6</b> 0036	<b>F</b> 0046	<b>V</b> 0056	<b>f</b> 0066	<b>v</b> 0076
<b>BEL</b> 0007	<b>ETB</b> 0017	<b>'</b> 0027	<b>7</b> 0037	<b>G</b> 0047	<b>W</b> 0057	<b>g</b> 0067	<b>w</b> 0077
<b>BS</b> 0008	<b>CAN</b> 0018	<b>(</b> 0028	<b>8</b> 0038	<b>H</b> 0048	<b>X</b> 0058	<b>h</b> 0068	<b>x</b> 0078
<b>HT</b> 0009	<b>EM</b> 0019	<b>)</b> 0029	<b>9</b> 0039	<b>I</b> 0049	<b>Y</b> 0059	<b>i</b> 0069	<b>y</b> 0079
<b>LF</b> 000A	<b>SUB</b> 001A	<b>*</b> 002A	<b>:</b> 003A	<b>J</b> 004A	<b>Z</b> 005A	<b>j</b> 006A	<b>z</b> 007A
<b>VT</b> 000B	<b>ESC</b> 001B	<b>+</b> 002B	<b>;</b> 003B	<b>K</b> 004B	<b>[</b> 005B	<b>k</b> 006B	<b>{</b> 007B
<b>FF</b> 000C	<b>FS</b> 001C	<b>,</b> 002C	<b>&lt;</b> 003C	<b>L</b> 004C	<b>\</b> 005C	<b>l</b> 006C	<b> </b> 007C
<b>CR</b> 000D	<b>GS</b> 001D	<b>-</b> 002D	<b>=</b> 003D	<b>M</b> 004D	<b>]</b> 005D	<b>m</b> 006D	<b>}</b> 007D
<b>SO</b> 000E	<b>RS</b> 001E	<b>.</b> 002E	<b>&gt;</b> 003E	<b>N</b> 004E	<b>^</b> 005E	<b>n</b> 006E	<b>~</b> 007E
<b>SI</b> 000F	<b>US</b> 001F	<b>/</b> 002F	<b>?</b> 003F	<b>O</b> 004F	<b>_</b> 005F	<b>o</b> 006F	<b>DEL</b> 007F

```

int
isalpha (int c)
{
    if (isupper (c)
        || islower (c))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

```



Listato 69.51. Funzione *isblank()*.

<b>NUL</b> 0000	<b>DLE</b> 0010	<b>SP</b> 0020	<b>0</b> 0030	<b>@</b> 0040	<b>P</b> 0050	<b>'</b> 0060	<b>p</b> 0070
<b>SOH</b> 0001	<b>DC1</b> 0011	<b>!</b> 0021	<b>1</b> 0031	<b>A</b> 0041	<b>Q</b> 0051	<b>a</b> 0061	<b>q</b> 0071
<b>STX</b> 0002	<b>DC2</b> 0012	<b>"</b> 0022	<b>2</b> 0032	<b>B</b> 0042	<b>R</b> 0052	<b>b</b> 0062	<b>r</b> 0072
<b>ETX</b> 0003	<b>DC3</b> 0013	<b>#</b> 0023	<b>3</b> 0023	<b>C</b> 0043	<b>S</b> 0053	<b>c</b> 0063	<b>s</b> 0073
<b>EOT</b> 0004	<b>DC4</b> 0014	<b>\$</b> 0024	<b>4</b> 0034	<b>D</b> 0044	<b>T</b> 0054	<b>d</b> 0064	<b>t</b> 0074
<b>ENQ</b> 0005	<b>NAK</b> 0015	<b>%</b> 0025	<b>5</b> 0035	<b>E</b> 0045	<b>U</b> 0055	<b>e</b> 0065	<b>u</b> 0075
<b>ACK</b> 0006	<b>SYN</b> 0016	<b>&amp;</b> 0026	<b>6</b> 0036	<b>F</b> 0046	<b>V</b> 0056	<b>f</b> 0066	<b>v</b> 0076
<b>BEL</b> 0007	<b>ETB</b> 0017	<b>'</b> 0027	<b>7</b> 0037	<b>G</b> 0047	<b>W</b> 0057	<b>g</b> 0067	<b>w</b> 0077
<b>BS</b> 0008	<b>CAN</b> 0018	<b>(</b> 0028	<b>8</b> 0038	<b>H</b> 0048	<b>X</b> 0058	<b>h</b> 0068	<b>x</b> 0078
<b>HT</b> 0009	<b>EM</b> 0019	<b>)</b> 0029	<b>9</b> 0039	<b>I</b> 0049	<b>Y</b> 0059	<b>i</b> 0069	<b>y</b> 0079
<b>LF</b> 000A	<b>SUB</b> 001A	<b>*</b> 002A	<b>:</b> 003A	<b>J</b> 004A	<b>Z</b> 005A	<b>j</b> 006A	<b>z</b> 007A
<b>VT</b> 000B	<b>ESC</b> 001B	<b>+</b> 002B	<b>;</b> 003B	<b>K</b> 004B	<b>[</b> 005B	<b>k</b> 006B	<b>{</b> 007B
<b>FF</b> 000C	<b>FS</b> 001C	<b>,</b> 002C	<b>&lt;</b> 003C	<b>L</b> 004C	<b>\</b> 005C	<b>l</b> 006C	<b> </b> 007C
<b>CR</b> 000D	<b>GS</b> 001D	<b>-</b> 002D	<b>=</b> 003D	<b>M</b> 004D	<b>]</b> 005D	<b>m</b> 006D	<b>}</b> 007D
<b>SO</b> 000E	<b>RS</b> 001E	<b>.</b> 002E	<b>&gt;</b> 003E	<b>N</b> 004E	<b>^</b> 005E	<b>n</b> 006E	<b>~</b> 007E
<b>SI</b> 000F	<b>US</b> 001F	<b>/</b> 002F	<b>?</b> 003F	<b>O</b> 004F	<b>_</b> 005F	<b>o</b> 006F	<b>DEL</b> 007F

```
#include <ctype.h>
int
isblank (int c)
{
    if (c == ' '
        || c == '\t' )
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Listato 69.52. Funzione *iscntrl()*.

<b>NUL</b> 0000	<b>DLE</b> 0010	<b>SP</b> 0020	<b>0</b> 0030	<b>@</b> 0040	<b>P</b> 0050	<b>'</b> 0060	<b>p</b> 0070
<b>SOH</b> 0001	<b>DC1</b> 0011	<b>!</b> 0021	<b>1</b> 0031	<b>A</b> 0041	<b>Q</b> 0051	<b>a</b> 0061	<b>q</b> 0071
<b>STX</b> 0002	<b>DC2</b> 0012	<b>"</b> 0022	<b>2</b> 0032	<b>B</b> 0042	<b>R</b> 0052	<b>b</b> 0062	<b>r</b> 0072
<b>ETX</b> 0003	<b>DC3</b> 0013	<b>#</b> 0023	<b>3</b> 0023	<b>C</b> 0043	<b>S</b> 0053	<b>c</b> 0063	<b>s</b> 0073
<b>EOT</b> 0004	<b>DC4</b> 0014	<b>\$</b> 0024	<b>4</b> 0034	<b>D</b> 0044	<b>T</b> 0054	<b>d</b> 0064	<b>t</b> 0074
<b>ENQ</b> 0005	<b>NAK</b> 0015	<b>%</b> 0025	<b>5</b> 0035	<b>E</b> 0045	<b>U</b> 0055	<b>e</b> 0065	<b>u</b> 0075
<b>ACK</b> 0006	<b>SYN</b> 0016	<b>&amp;</b> 0026	<b>6</b> 0036	<b>F</b> 0046	<b>V</b> 0056	<b>f</b> 0066	<b>v</b> 0076
<b>BEL</b> 0007	<b>ETB</b> 0017	<b>'</b> 0027	<b>7</b> 0037	<b>G</b> 0047	<b>W</b> 0057	<b>g</b> 0067	<b>w</b> 0077
<b>BS</b> 0008	<b>CAN</b> 0018	<b>(</b> 0028	<b>8</b> 0038	<b>H</b> 0048	<b>X</b> 0058	<b>h</b> 0068	<b>x</b> 0078
<b>HT</b> 0009	<b>EM</b> 0019	<b>)</b> 0029	<b>9</b> 0039	<b>I</b> 0049	<b>Y</b> 0059	<b>i</b> 0069	<b>y</b> 0079
<b>LF</b> 000A	<b>SUB</b> 001A	<b>*</b> 002A	<b>:</b> 003A	<b>J</b> 004A	<b>Z</b> 005A	<b>j</b> 006A	<b>z</b> 007A
<b>VT</b> 000B	<b>ESC</b> 001B	<b>+</b> 002B	<b>;</b> 003B	<b>K</b> 004B	<b>[</b> 005B	<b>k</b> 006B	<b>{</b> 007B
<b>FF</b> 000C	<b>FS</b> 001C	<b>,</b> 002C	<b>&lt;</b> 003C	<b>L</b> 004C	<b>\</b> 005C	<b>l</b> 006C	<b> </b> 007C
<b>CR</b> 000D	<b>GS</b> 001D	<b>-</b> 002D	<b>=</b> 003D	<b>M</b> 004D	<b>]</b> 005D	<b>m</b> 006D	<b>}</b> 007D
<b>SO</b> 000E	<b>RS</b> 001E	<b>.</b> 002E	<b>&gt;</b> 003E	<b>N</b> 004E	<b>^</b> 005E	<b>n</b> 006E	<b>~</b> 007E
<b>SI</b> 000F	<b>US</b> 001F	<b>/</b> 002F	<b>?</b> 003F	<b>O</b> 004F	<b>_</b> 005F	<b>o</b> 006F	<b>DEL</b> 007F

```
#include <ctype.h>
int
iscntrl (int c)
{
    if (((c >= 0x00)
        && (c <= 0x1F))
        || (c == 0x7F))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Listato 69.53. Funzione *isdigit()*.

<b>NUL</b> 0000	<b>DLE</b> 0010	<b>SP</b> 0020	<b>0</b> 0030	<b>@</b> 0040	<b>P</b> 0050	<b>'</b> 0060	<b>p</b> 0070
<b>SOH</b> 0001	<b>DC1</b> 0011	<b>!</b> 0021	<b>1</b> 0031	<b>A</b> 0041	<b>Q</b> 0051	<b>a</b> 0061	<b>q</b> 0071
<b>STX</b> 0002	<b>DC2</b> 0012	<b>"</b> 0022	<b>2</b> 0032	<b>B</b> 0042	<b>R</b> 0052	<b>b</b> 0062	<b>r</b> 0072
<b>ETX</b> 0003	<b>DC3</b> 0013	<b>#</b> 0023	<b>3</b> 0023	<b>C</b> 0043	<b>S</b> 0053	<b>c</b> 0063	<b>s</b> 0073
<b>EOT</b> 0004	<b>DC4</b> 0014	<b>\$</b> 0024	<b>4</b> 0034	<b>D</b> 0044	<b>T</b> 0054	<b>d</b> 0064	<b>t</b> 0074
<b>ENQ</b> 0005	<b>NAK</b> 0015	<b>%</b> 0025	<b>5</b> 0035	<b>E</b> 0045	<b>U</b> 0055	<b>e</b> 0065	<b>u</b> 0075
<b>ACK</b> 0006	<b>SYN</b> 0016	<b>&amp;</b> 0026	<b>6</b> 0036	<b>F</b> 0046	<b>V</b> 0056	<b>f</b> 0066	<b>v</b> 0076
<b>BEL</b> 0007	<b>ETB</b> 0017	<b>'</b> 0027	<b>7</b> 0037	<b>G</b> 0047	<b>W</b> 0057	<b>g</b> 0067	<b>w</b> 0077
<b>BS</b> 0008	<b>CAN</b> 0018	<b>(</b> 0028	<b>8</b> 0038	<b>H</b> 0048	<b>X</b> 0058	<b>h</b> 0068	<b>x</b> 0078
<b>HT</b> 0009	<b>EM</b> 0019	<b>)</b> 0029	<b>9</b> 0039	<b>I</b> 0049	<b>Y</b> 0059	<b>i</b> 0069	<b>y</b> 0079
<b>LF</b> 000A	<b>SUB</b> 001A	<b>*</b> 002A	<b>:</b> 003A	<b>J</b> 004A	<b>Z</b> 005A	<b>j</b> 006A	<b>z</b> 007A
<b>VT</b> 000B	<b>ESC</b> 001B	<b>+</b> 002B	<b>;</b> 003B	<b>K</b> 004B	<b>[</b> 005B	<b>k</b> 006B	<b>{</b> 007B
<b>FF</b> 000C	<b>FS</b> 001C	<b>,</b> 002C	<b>&lt;</b> 003C	<b>L</b> 004C	<b>\</b> 005C	<b>l</b> 006C	<b> </b> 007C
<b>CR</b> 000D	<b>GS</b> 001D	<b>-</b> 002D	<b>=</b> 003D	<b>M</b> 004D	<b>]</b> 005D	<b>m</b> 006D	<b>}</b> 007D
<b>SO</b> 000E	<b>RS</b> 001E	<b>.</b> 002E	<b>&gt;</b> 003E	<b>N</b> 004E	<b>^</b> 005E	<b>n</b> 006E	<b>~</b> 007E
<b>SI</b> 000F	<b>US</b> 001F	<b>/</b> 002F	<b>?</b> 003F	<b>O</b> 004F	<b>_</b> 005F	<b>o</b> 006F	<b>DEL</b> 007F

```
#include <ctype.h>
int
isdigit (int c)
{
    if ((c >= 0x30)
        && (c <= 0x39))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Listato 69.54. Funzione *isgraph()*.

<b>NUL</b> 0000	<b>DLE</b> 0010	<b>SP</b> 0020	<b>0</b> 0030	<b>@</b> 0040	<b>P</b> 0050	<b>'</b> 0060	<b>p</b> 0070
<b>SOH</b> 0001	<b>DC1</b> 0011	<b>!</b> 0021	<b>1</b> 0031	<b>A</b> 0041	<b>Q</b> 0051	<b>a</b> 0061	<b>q</b> 0071
<b>STX</b> 0002	<b>DC2</b> 0012	<b>"</b> 0022	<b>2</b> 0032	<b>B</b> 0042	<b>R</b> 0052	<b>b</b> 0062	<b>r</b> 0072
<b>ETX</b> 0003	<b>DC3</b> 0013	<b>#</b> 0023	<b>3</b> 0023	<b>C</b> 0043	<b>S</b> 0053	<b>c</b> 0063	<b>s</b> 0073
<b>EOT</b> 0004	<b>DC4</b> 0014	<b>\$</b> 0024	<b>4</b> 0034	<b>D</b> 0044	<b>T</b> 0054	<b>d</b> 0064	<b>t</b> 0074
<b>ENQ</b> 0005	<b>NAK</b> 0015	<b>%</b> 0025	<b>5</b> 0035	<b>E</b> 0045	<b>U</b> 0055	<b>e</b> 0065	<b>u</b> 0075
<b>ACK</b> 0006	<b>SYN</b> 0016	<b>&amp;</b> 0026	<b>6</b> 0036	<b>F</b> 0046	<b>V</b> 0056	<b>f</b> 0066	<b>v</b> 0076
<b>BEL</b> 0007	<b>ETB</b> 0017	<b>'</b> 0027	<b>7</b> 0037	<b>G</b> 0047	<b>W</b> 0057	<b>g</b> 0067	<b>w</b> 0077
<b>BS</b> 0008	<b>CAN</b> 0018	<b>(</b> 0028	<b>8</b> 0038	<b>H</b> 0048	<b>X</b> 0058	<b>h</b> 0068	<b>x</b> 0078
<b>HT</b> 0009	<b>EM</b> 0019	<b>)</b> 0029	<b>9</b> 0039	<b>I</b> 0049	<b>Y</b> 0059	<b>i</b> 0069	<b>y</b> 0079
<b>LF</b> 000A	<b>SUB</b> 001A	<b>*</b> 002A	<b>:</b> 003A	<b>J</b> 004A	<b>Z</b> 005A	<b>j</b> 006A	<b>z</b> 007A
<b>VT</b> 000B	<b>ESC</b> 001B	<b>+</b> 002B	<b>;</b> 003B	<b>K</b> 004B	<b>[</b> 005B	<b>k</b> 006B	<b>{</b> 007B
<b>FF</b> 000C	<b>FS</b> 001C	<b>,</b> 002C	<b>&lt;</b> 003C	<b>L</b> 004C	<b>\</b> 005C	<b>l</b> 006C	<b> </b> 007C
<b>CR</b> 000D	<b>GS</b> 001D	<b>-</b> 002D	<b>=</b> 003D	<b>M</b> 004D	<b>]</b> 005D	<b>m</b> 006D	<b>}</b> 007D
<b>SO</b> 000E	<b>RS</b> 001E	<b>.</b> 002E	<b>&gt;</b> 003E	<b>N</b> 004E	<b>^</b> 005E	<b>n</b> 006E	<b>~</b> 007E
<b>SI</b> 000F	<b>US</b> 001F	<b>/</b> 002F	<b>?</b> 003F	<b>O</b> 004F	<b>_</b> 005F	<b>o</b> 006F	<b>DEL</b> 007F

```
#include <ctype.h>
int
isgraph (int c)
{
    if ((c >= 0x21)
        && (c <= 0x7E))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Listato 69.55. Funzione *islower()*.

<b>NUL</b> 0000	<b>DLE</b> 0010	<b>SP</b> 0020	<b>0</b> 0030	<b>@</b> 0040	<b>P</b> 0050	<b>'</b> 0060	<b>p</b> 0070
<b>SOH</b> 0001	<b>DC1</b> 0011	<b>!</b> 0021	<b>1</b> 0031	<b>A</b> 0041	<b>Q</b> 0051	<b>a</b> 0061	<b>q</b> 0071
<b>STX</b> 0002	<b>DC2</b> 0012	<b>"</b> 0022	<b>2</b> 0032	<b>B</b> 0042	<b>R</b> 0052	<b>b</b> 0062	<b>r</b> 0072
<b>ETX</b> 0003	<b>DC3</b> 0013	<b>#</b> 0023	<b>3</b> 0023	<b>C</b> 0043	<b>S</b> 0053	<b>c</b> 0063	<b>s</b> 0073
<b>EOT</b> 0004	<b>DC4</b> 0014	<b>\$</b> 0024	<b>4</b> 0034	<b>D</b> 0044	<b>T</b> 0054	<b>d</b> 0064	<b>t</b> 0074
<b>ENQ</b> 0005	<b>NAK</b> 0015	<b>%</b> 0025	<b>5</b> 0035	<b>E</b> 0045	<b>U</b> 0055	<b>e</b> 0065	<b>u</b> 0075
<b>ACK</b> 0006	<b>SYN</b> 0016	<b>&amp;</b> 0026	<b>6</b> 0036	<b>F</b> 0046	<b>V</b> 0056	<b>f</b> 0066	<b>v</b> 0076
<b>BEL</b> 0007	<b>ETB</b> 0017	<b>'</b> 0027	<b>7</b> 0037	<b>G</b> 0047	<b>W</b> 0057	<b>g</b> 0067	<b>w</b> 0077
<b>BS</b> 0008	<b>CAN</b> 0018	<b>(</b> 0028	<b>8</b> 0038	<b>H</b> 0048	<b>X</b> 0058	<b>h</b> 0068	<b>x</b> 0078
<b>HT</b> 0009	<b>EM</b> 0019	<b>)</b> 0029	<b>9</b> 0039	<b>I</b> 0049	<b>Y</b> 0059	<b>i</b> 0069	<b>y</b> 0079
<b>LF</b> 000A	<b>SUB</b> 001A	<b>*</b> 002A	<b>:</b> 003A	<b>J</b> 004A	<b>Z</b> 005A	<b>j</b> 006A	<b>z</b> 007A
<b>VT</b> 000B	<b>ESC</b> 001B	<b>+</b> 002B	<b>;</b> 003B	<b>K</b> 004B	<b>[</b> 005B	<b>k</b> 006B	<b>{</b> 007B
<b>FF</b> 000C	<b>FS</b> 001C	<b>,</b> 002C	<b>&lt;</b> 003C	<b>L</b> 004C	<b>\</b> 005C	<b>l</b> 006C	<b> </b> 007C
<b>CR</b> 000D	<b>GS</b> 001D	<b>-</b> 002D	<b>=</b> 003D	<b>M</b> 004D	<b>]</b> 005D	<b>m</b> 006D	<b>}</b> 007D
<b>SO</b> 000E	<b>RS</b> 001E	<b>.</b> 002E	<b>&gt;</b> 003E	<b>N</b> 004E	<b>^</b> 005E	<b>n</b> 006E	<b>~</b> 007E
<b>SI</b> 000F	<b>US</b> 001F	<b>/</b> 002F	<b>?</b> 003F	<b>O</b> 004F	<b>_</b> 005F	<b>o</b> 006F	<b>DEL</b> 007F

```
#include <ctype.h>
int
islower (int c)
{
    if ((c >= 0x61)
        && (c <= 0x7A))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Listato 69.56. Funzione *isprint()*.

<b>NUL</b> 0000	<b>DLE</b> 0010	<b>SP</b> 0020	<b>0</b> 0030	<b>@</b> 0040	<b>P</b> 0050	<b>'</b> 0060	<b>p</b> 0070
<b>SOH</b> 0001	<b>DC1</b> 0011	<b>!</b> 0021	<b>1</b> 0031	<b>A</b> 0041	<b>Q</b> 0051	<b>a</b> 0061	<b>q</b> 0071
<b>STX</b> 0002	<b>DC2</b> 0012	<b>"</b> 0022	<b>2</b> 0032	<b>B</b> 0042	<b>R</b> 0052	<b>b</b> 0062	<b>r</b> 0072
<b>ETX</b> 0003	<b>DC3</b> 0013	<b>#</b> 0023	<b>3</b> 0023	<b>C</b> 0043	<b>S</b> 0053	<b>c</b> 0063	<b>s</b> 0073
<b>EOT</b> 0004	<b>DC4</b> 0014	<b>\$</b> 0024	<b>4</b> 0034	<b>D</b> 0044	<b>T</b> 0054	<b>d</b> 0064	<b>t</b> 0074
<b>ENQ</b> 0005	<b>NAK</b> 0015	<b>%</b> 0025	<b>5</b> 0035	<b>E</b> 0045	<b>U</b> 0055	<b>e</b> 0065	<b>u</b> 0075
<b>ACK</b> 0006	<b>SYN</b> 0016	<b>&amp;</b> 0026	<b>6</b> 0036	<b>F</b> 0046	<b>V</b> 0056	<b>f</b> 0066	<b>v</b> 0076
<b>BEL</b> 0007	<b>ETB</b> 0017	<b>'</b> 0027	<b>7</b> 0037	<b>G</b> 0047	<b>W</b> 0057	<b>g</b> 0067	<b>w</b> 0077
<b>BS</b> 0008	<b>CAN</b> 0018	<b>(</b> 0028	<b>8</b> 0038	<b>H</b> 0048	<b>X</b> 0058	<b>h</b> 0068	<b>x</b> 0078
<b>HT</b> 0009	<b>EM</b> 0019	<b>)</b> 0029	<b>9</b> 0039	<b>I</b> 0049	<b>Y</b> 0059	<b>i</b> 0069	<b>y</b> 0079
<b>LF</b> 000A	<b>SUB</b> 001A	<b>*</b> 002A	<b>:</b> 003A	<b>J</b> 004A	<b>Z</b> 005A	<b>j</b> 006A	<b>z</b> 007A
<b>VT</b> 000B	<b>ESC</b> 001B	<b>+</b> 002B	<b>;</b> 003B	<b>K</b> 004B	<b>[</b> 005B	<b>k</b> 006B	<b>{</b> 007B
<b>FF</b> 000C	<b>FS</b> 001C	<b>,</b> 002C	<b>&lt;</b> 003C	<b>L</b> 004C	<b>\</b> 005C	<b>l</b> 006C	<b> </b> 007C
<b>CR</b> 000D	<b>GS</b> 001D	<b>-</b> 002D	<b>=</b> 003D	<b>M</b> 004D	<b>]</b> 005D	<b>m</b> 006D	<b>}</b> 007D
<b>SO</b> 000E	<b>RS</b> 001E	<b>.</b> 002E	<b>&gt;</b> 003E	<b>N</b> 004E	<b>^</b> 005E	<b>n</b> 006E	<b>~</b> 007E
<b>SI</b> 000F	<b>US</b> 001F	<b>/</b> 002F	<b>?</b> 003F	<b>O</b> 004F	<b>_</b> 005F	<b>o</b> 006F	<b>DEL</b> 007F

```
#include <ctype.h>
int
isprint (int c)
{
    if ((c >= 0x20)
        && (c <= 0x7E))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Listato 69.57. Funzione *ispunct()*.

<b>NUL</b> 0000	<b>DLE</b> 0010	<b>SP</b> 0020	<b>0</b> 0030	<b>@</b> 0040	<b>P</b> 0050	<b>'</b> 0060	<b>p</b> 0070	
<b>SOH</b> 0001	<b>DC1</b> 0011	<b>!</b> 0021	<b>1</b> 0031	<b>A</b> 0041	<b>Q</b> 0051	<b>a</b> 0061	<b>q</b> 0071	
<b>STX</b> 0002	<b>DC2</b> 0012	<b>"</b> 0022	<b>2</b> 0032	<b>B</b> 0042	<b>R</b> 0052	<b>b</b> 0062	<b>r</b> 0072	
<b>ETX</b> 0003	<b>DC3</b> 0013	<b>#</b> 0023	<b>3</b> 0023	<b>C</b> 0043	<b>S</b> 0053	<b>c</b> 0063	<b>s</b> 0073	
<b>EOT</b> 0004	<b>DC4</b> 0014	<b>\$</b> 0024	<b>4</b> 0034	<b>D</b> 0044	<b>T</b> 0054	<b>d</b> 0064	<b>t</b> 0074	#include <ctype.h>
<b>ENQ</b> 0005	<b>NAK</b> 0015	<b>%</b> 0025	<b>5</b> 0035	<b>E</b> 0045	<b>U</b> 0055	<b>e</b> 0065	<b>u</b> 0075	int
<b>ACK</b> 0006	<b>SYN</b> 0016	<b>&amp;</b> 0026	<b>6</b> 0036	<b>F</b> 0046	<b>V</b> 0056	<b>f</b> 0066	<b>v</b> 0076	ispunct (int c)
<b>BEL</b> 0007	<b>ETB</b> 0017	<b>,</b> 0027	<b>7</b> 0037	<b>G</b> 0047	<b>W</b> 0057	<b>g</b> 0067	<b>w</b> 0077	{
<b>BS</b> 0008	<b>CAN</b> 0018	<b>(</b> 0028	<b>8</b> 0038	<b>H</b> 0048	<b>X</b> 0058	<b>h</b> 0068	<b>x</b> 0078	if (isgraph (c)
<b>HT</b> 0009	<b>EM</b> 0019	<b>)</b> 0029	<b>9</b> 0039	<b>I</b> 0049	<b>Y</b> 0059	<b>i</b> 0069	<b>y</b> 0079	&& (! isspace (c))
<b>LF</b> 000A	<b>SUB</b> 001A	<b>*</b> 002A	<b>:</b> 003A	<b>J</b> 004A	<b>Z</b> 005A	<b>j</b> 006A	<b>z</b> 007A	&& (! isalnum(c))
<b>VT</b> 000B	<b>ESC</b> 001B	<b>+</b> 002B	<b>;</b> 003B	<b>K</b> 004B	<b>[</b> 005B	<b>k</b> 006B	<b>{</b> 007B	{
<b>FF</b> 000C	<b>FS</b> 001C	<b>,</b> 002C	<b>&lt;</b> 003C	<b>L</b> 004C	<b>\</b> 005C	<b>l</b> 006C	<b> </b> 007C	return 1;
<b>CR</b> 000D	<b>GS</b> 001D	<b>-</b> 002D	<b>=</b> 003D	<b>M</b> 004D	<b>]</b> 005D	<b>m</b> 006D	<b>}</b> 007D	}
<b>SO</b> 000E	<b>RS</b> 001E	<b>.</b> 002E	<b>&gt;</b> 003E	<b>N</b> 004E	<b>^</b> 005E	<b>n</b> 006E	<b>~</b> 007E	else
<b>SI</b> 000F	<b>US</b> 001F	<b>/</b> 002F	<b>?</b> 003F	<b>O</b> 004F	<b>_</b> 005F	<b>o</b> 006F	<b>DEL</b> 007F	{

```

#include <ctype.h>
int
ispunct (int c)
{
    if (isgraph (c)
        && (! isspace (c))
        && (! isalnum(c)))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

```

Listato 69.58. Funzione *isspace()*.

<b>NUL</b> 0000	<b>DLE</b> 0010	<b>SP</b> 0020	<b>0</b> 0030	<b>@</b> 0040	<b>P</b> 0050	<b>'</b> 0060	<b>p</b> 0070
<b>SOH</b> 0001	<b>DC1</b> 0011	<b>!</b> 0021	<b>1</b> 0031	<b>A</b> 0041	<b>Q</b> 0051	<b>a</b> 0061	<b>q</b> 0071
<b>STX</b> 0002	<b>DC2</b> 0012	<b>"</b> 0022	<b>2</b> 0032	<b>B</b> 0042	<b>R</b> 0052	<b>b</b> 0062	<b>r</b> 0072
<b>ETX</b> 0003	<b>DC3</b> 0013	<b>#</b> 0023	<b>3</b> 0023	<b>C</b> 0043	<b>S</b> 0053	<b>c</b> 0063	<b>s</b> 0073
<b>EOT</b> 0004	<b>DC4</b> 0014	<b>\$</b> 0024	<b>4</b> 0034	<b>D</b> 0044	<b>T</b> 0054	<b>d</b> 0064	<b>t</b> 0074
<b>ENQ</b> 0005	<b>NAK</b> 0015	<b>%</b> 0025	<b>5</b> 0035	<b>E</b> 0045	<b>U</b> 0055	<b>e</b> 0065	<b>u</b> 0075
<b>ACK</b> 0006	<b>SYN</b> 0016	<b>&amp;</b> 0026	<b>6</b> 0036	<b>F</b> 0046	<b>V</b> 0056	<b>f</b> 0066	<b>v</b> 0076
<b>BEL</b> 0007	<b>ETB</b> 0017	<b>'</b> 0027	<b>7</b> 0037	<b>G</b> 0047	<b>W</b> 0057	<b>g</b> 0067	<b>w</b> 0077
<b>BS</b> 0008	<b>CAN</b> 0018	<b>(</b> 0028	<b>8</b> 0038	<b>H</b> 0048	<b>X</b> 0058	<b>h</b> 0068	<b>x</b> 0078
<b>HT</b> 0009	<b>EM</b> 0019	<b>)</b> 0029	<b>9</b> 0039	<b>I</b> 0049	<b>Y</b> 0059	<b>i</b> 0069	<b>y</b> 0079
<b>LF</b> 000A	<b>SUB</b> 001A	<b>*</b> 002A	<b>:</b> 003A	<b>J</b> 004A	<b>Z</b> 005A	<b>j</b> 006A	<b>z</b> 007A
<b>VT</b> 000B	<b>ESC</b> 001B	<b>+</b> 002B	<b>;</b> 003B	<b>K</b> 004B	<b>[</b> 005B	<b>k</b> 006B	<b>{</b> 007B
<b>FF</b> 000C	<b>FS</b> 001C	<b>,</b> 002C	<b>&lt;</b> 003C	<b>L</b> 004C	<b>\</b> 005C	<b>l</b> 006C	<b> </b> 007C
<b>CR</b> 000D	<b>GS</b> 001D	<b>-</b> 002D	<b>=</b> 003D	<b>M</b> 004D	<b>]</b> 005D	<b>m</b> 006D	<b>}</b> 007D
<b>SO</b> 000E	<b>RS</b> 001E	<b>.</b> 002E	<b>&gt;</b> 003E	<b>N</b> 004E	<b>^</b> 005E	<b>n</b> 006E	<b>~</b> 007E
<b>SI</b> 000F	<b>US</b> 001F	<b>/</b> 002F	<b>?</b> 003F	<b>O</b> 004F	<b>_</b> 005F	<b>o</b> 006F	<b>DEL</b> 007F

```
#include <ctype.h>
int
isspace (int c)
{
    if ( (c == ' ')
        || (c == '\f')
        || (c == '\n')
        || (c == '\r')
        || (c == '\t')
        || (c == '\v'))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```



Listato 69.59. Funzione *isupper()*.

<b>NUL</b> 0000	<b>DLE</b> 0010	<b>SP</b> 0020	<b>0</b> 0030	<b>@</b> 0040	<b>P</b> 0050	<b>'</b> 0060	<b>p</b> 0070
<b>SOH</b> 0001	<b>DC1</b> 0011	<b>!</b> 0021	<b>1</b> 0031	<b>A</b> 0041	<b>Q</b> 0051	<b>a</b> 0061	<b>q</b> 0071
<b>STX</b> 0002	<b>DC2</b> 0012	<b>"</b> 0022	<b>2</b> 0032	<b>B</b> 0042	<b>R</b> 0052	<b>b</b> 0062	<b>r</b> 0072
<b>ETX</b> 0003	<b>DC3</b> 0013	<b>#</b> 0023	<b>3</b> 0023	<b>C</b> 0043	<b>S</b> 0053	<b>c</b> 0063	<b>s</b> 0073
<b>EOT</b> 0004	<b>DC4</b> 0014	<b>\$</b> 0024	<b>4</b> 0034	<b>D</b> 0044	<b>T</b> 0054	<b>d</b> 0064	<b>t</b> 0074
<b>ENQ</b> 0005	<b>NAK</b> 0015	<b>%</b> 0025	<b>5</b> 0035	<b>E</b> 0045	<b>U</b> 0055	<b>e</b> 0065	<b>u</b> 0075
<b>ACK</b> 0006	<b>SYN</b> 0016	<b>&amp;</b> 0026	<b>6</b> 0036	<b>F</b> 0046	<b>V</b> 0056	<b>f</b> 0066	<b>v</b> 0076
<b>BEL</b> 0007	<b>ETB</b> 0017	<b>'</b> 0027	<b>7</b> 0037	<b>G</b> 0047	<b>W</b> 0057	<b>g</b> 0067	<b>w</b> 0077
<b>BS</b> 0008	<b>CAN</b> 0018	<b>(</b> 0028	<b>8</b> 0038	<b>H</b> 0048	<b>X</b> 0058	<b>h</b> 0068	<b>x</b> 0078
<b>HT</b> 0009	<b>EM</b> 0019	<b>)</b> 0029	<b>9</b> 0039	<b>I</b> 0049	<b>Y</b> 0059	<b>i</b> 0069	<b>y</b> 0079
<b>LF</b> 000A	<b>SUB</b> 001A	<b>*</b> 002A	<b>:</b> 003A	<b>J</b> 004A	<b>Z</b> 005A	<b>j</b> 006A	<b>z</b> 007A
<b>VT</b> 000B	<b>ESC</b> 001B	<b>+</b> 002B	<b>;</b> 003B	<b>K</b> 004B	<b>[</b> 005B	<b>k</b> 006B	<b>{</b> 007B
<b>FF</b> 000C	<b>FS</b> 001C	<b>,</b> 002C	<b>&lt;</b> 003C	<b>L</b> 004C	<b>\</b> 005C	<b>l</b> 006C	<b> </b> 007C
<b>CR</b> 000D	<b>GS</b> 001D	<b>-</b> 002D	<b>=</b> 003D	<b>M</b> 004D	<b>]</b> 005D	<b>m</b> 006D	<b>}</b> 007D
<b>SO</b> 000E	<b>RS</b> 001E	<b>.</b> 002E	<b>&gt;</b> 003E	<b>N</b> 004E	<b>^</b> 005E	<b>n</b> 006E	<b>~</b> 007E
<b>SI</b> 000F	<b>US</b> 001F	<b>/</b> 002F	<b>?</b> 003F	<b>O</b> 004F	<b>_</b> 005F	<b>o</b> 006F	<b>DEL</b> 007F

```
#include <ctype.h>
int
isupper (int c)
{
    if ((c >= 0x41)
        && (c <= 0x5A))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Listato 69.60. Funzione *isxdigit()*.

<b>NUL</b> 0000	<b>DLE</b> 0010	<b>SP</b> 0020	<b>0</b> 0030	<b>@</b> 0040	<b>P</b> 0050	<b>'</b> 0060	<b>p</b> 0070
<b>SOH</b> 0001	<b>DC1</b> 0011	<b>!</b> 0021	<b>1</b> 0031	<b>A</b> 0041	<b>Q</b> 0051	<b>a</b> 0061	<b>q</b> 0071
<b>STX</b> 0002	<b>DC2</b> 0012	<b>"</b> 0022	<b>2</b> 0032	<b>B</b> 0042	<b>R</b> 0052	<b>b</b> 0062	<b>r</b> 0072
<b>ETX</b> 0003	<b>DC3</b> 0013	<b>#</b> 0023	<b>3</b> 0023	<b>C</b> 0043	<b>S</b> 0053	<b>c</b> 0063	<b>s</b> 0073
<b>EOT</b> 0004	<b>DC4</b> 0014	<b>\$</b> 0024	<b>4</b> 0034	<b>D</b> 0044	<b>T</b> 0054	<b>d</b> 0064	<b>t</b> 0074
<b>ENQ</b> 0005	<b>NAK</b> 0015	<b>%</b> 0025	<b>5</b> 0035	<b>E</b> 0045	<b>U</b> 0055	<b>e</b> 0065	<b>u</b> 0075
<b>ACK</b> 0006	<b>SYN</b> 0016	<b>&amp;</b> 0026	<b>6</b> 0036	<b>F</b> 0046	<b>V</b> 0056	<b>f</b> 0066	<b>v</b> 0076
<b>BEL</b> 0007	<b>ETB</b> 0017	<b>'</b> 0027	<b>7</b> 0037	<b>G</b> 0047	<b>W</b> 0057	<b>g</b> 0067	<b>w</b> 0077
<b>BS</b> 0008	<b>CAN</b> 0018	<b>(</b> 0028	<b>8</b> 0038	<b>H</b> 0048	<b>X</b> 0058	<b>h</b> 0068	<b>x</b> 0078
<b>HT</b> 0009	<b>EM</b> 0019	<b>)</b> 0029	<b>9</b> 0039	<b>I</b> 0049	<b>Y</b> 0059	<b>i</b> 0069	<b>y</b> 0079
<b>LF</b> 000A	<b>SUB</b> 001A	<b>*</b> 002A	<b>:</b> 003A	<b>J</b> 004A	<b>Z</b> 005A	<b>j</b> 006A	<b>z</b> 007A
<b>VT</b> 000B	<b>ESC</b> 001B	<b>+</b> 002B	<b>;</b> 003B	<b>K</b> 004B	<b>[</b> 005B	<b>k</b> 006B	<b>{</b> 007B
<b>FF</b> 000C	<b>FS</b> 001C	<b>,</b> 002C	<b>&lt;</b> 003C	<b>L</b> 004C	<b>\</b> 005C	<b>l</b> 006C	<b> </b> 007C
<b>CR</b> 000D	<b>GS</b> 001D	<b>-</b> 002D	<b>=</b> 003D	<b>M</b> 004D	<b>]</b> 005D	<b>m</b> 006D	<b>}</b> 007D
<b>SO</b> 000E	<b>RS</b> 001E	<b>.</b> 002E	<b>&gt;</b> 003E	<b>N</b> 004E	<b>^</b> 005E	<b>n</b> 006E	<b>~</b> 007E
<b>SI</b> 000F	<b>US</b> 001F	<b>/</b> 002F	<b>?</b> 003F	<b>O</b> 004F	<b>_</b> 005F	<b>o</b> 006F	<b>DEL</b> 007F

```
#include <ctype.h>
int
isxdigit (int c)
{
    if (((c >= 0x30)
        && (c <= 0x39))
        || ((c >= 0x41)
            && (c <= 0x46))
        || ((c >= 0x61)
            && (c <= 0x66)))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Listato 69.61. Nello standard POSIX si aggiunge la funzione *isascii()*.

<b>NUL</b> 0000	<b>DLE</b> 0010	<b>SP</b> 0020	<b>0</b> 0030	<b>@</b> 0040	<b>P</b> 0050	<b>'</b> 0060	<b>p</b> 0070
<b>SOH</b> 0001	<b>DC1</b> 0011	<b>!</b> 0021	<b>1</b> 0031	<b>A</b> 0041	<b>Q</b> 0051	<b>a</b> 0061	<b>q</b> 0071
<b>STX</b> 0002	<b>DC2</b> 0012	<b>"</b> 0022	<b>2</b> 0032	<b>B</b> 0042	<b>R</b> 0052	<b>b</b> 0062	<b>r</b> 0072
<b>ETX</b> 0003	<b>DC3</b> 0013	<b>#</b> 0023	<b>3</b> 0023	<b>C</b> 0043	<b>S</b> 0053	<b>c</b> 0063	<b>s</b> 0073
<b>EOT</b> 0004	<b>DC4</b> 0014	<b>\$</b> 0024	<b>4</b> 0034	<b>D</b> 0044	<b>T</b> 0054	<b>d</b> 0064	<b>t</b> 0074
<b>ENQ</b> 0005	<b>NAK</b> 0015	<b>%</b> 0025	<b>5</b> 0035	<b>E</b> 0045	<b>U</b> 0055	<b>e</b> 0065	<b>u</b> 0075
<b>ACK</b> 0006	<b>SYN</b> 0016	<b>&amp;</b> 0026	<b>6</b> 0036	<b>F</b> 0046	<b>V</b> 0056	<b>f</b> 0066	<b>v</b> 0076
<b>BEL</b> 0007	<b>ETB</b> 0017	<b>'</b> 0027	<b>7</b> 0037	<b>G</b> 0047	<b>W</b> 0057	<b>g</b> 0067	<b>w</b> 0077
<b>BS</b> 0008	<b>CAN</b> 0018	<b>(</b> 0028	<b>8</b> 0038	<b>H</b> 0048	<b>X</b> 0058	<b>h</b> 0068	<b>x</b> 0078
<b>HT</b> 0009	<b>EM</b> 0019	<b>)</b> 0029	<b>9</b> 0039	<b>I</b> 0049	<b>Y</b> 0059	<b>i</b> 0069	<b>y</b> 0079
<b>LF</b> 000A	<b>SUB</b> 001A	<b>*</b> 002A	<b>:</b> 003A	<b>J</b> 004A	<b>Z</b> 005A	<b>j</b> 006A	<b>z</b> 007A
<b>VT</b> 000B	<b>ESC</b> 001B	<b>+</b> 002B	<b>;</b> 003B	<b>K</b> 004B	<b>[</b> 005B	<b>k</b> 006B	<b>{</b> 007B
<b>FF</b> 000C	<b>FS</b> 001C	<b>,</b> 002C	<b>&lt;</b> 003C	<b>L</b> 004C	<b>\</b> 005C	<b>l</b> 006C	<b> </b> 007C
<b>CR</b> 000D	<b>GS</b> 001D	<b>-</b> 002D	<b>=</b> 003D	<b>M</b> 004D	<b>]</b> 005D	<b>m</b> 006D	<b>}</b> 007D
<b>SO</b> 000E	<b>RS</b> 001E	<b>.</b> 002E	<b>&gt;</b> 003E	<b>N</b> 004E	<b>^</b> 005E	<b>n</b> 006E	<b>~</b> 007E
<b>SI</b> 000F	<b>US</b> 001F	<b>/</b> 002F	<b>?</b> 003F	<b>O</b> 004F	<b>_</b> 005F	<b>o</b> 006F	<b>DEL</b> 007F

```
#include <ctype.h>
int
isascii (int c)
{
    if ((c >= 0x00)
        && (c <= 0x7F))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

POSIX

## 69.7.2 macroistruzioni «is...()»

&lt;&lt;

In alternativa a delle funzioni vere e proprie, si possono realizzare semplicemente delle macroistruzioni per verificare le condizioni riferite al carattere. Il listato seguente è conforme a quanto già visto nella sezione precedente:

```
#define isblank(C) ((int) (C == ' ' || C == '\t'))
#define isspace(C) ((int) (C == ' ' \
                          || C == '\f' \
                          || C == '\n' \
                          || C == '\r' \
                          || C == '\t' \
                          || C == '\v'))
#define isdigit(C) ((int) (C >= '0' && C <= '9'))
#define isxdigit(C) ((int) ((C >= '0' && C <= '9') \
                             || (C >= 'A' && C <= 'F') \
                             || (C >= 'a' && C <= 'f'))))
#define isupper(C) ((int) (C >= 'A' && C <= 'Z'))
#define islower(C) ((int) (C >= 'a' && C <= 'z'))
#define iscntrl(C) ((int) ((C >= 0x00 && C <= 0x1F)
                          || C == 0x7F))
#define isgraph(C) ((int) (C >= 0x21 && C <= 0x7E))
#define isprint(C) ((int) (C >= 0x20 && C <= 0x7E))
#define isalpha(C) (isupper (C) || islower (C))
#define isalnum(C) (isalpha (C) || isdigit (C))
#define ispunct(C) (isgraph (C) && (!isspace (C))
                  && (!isalnum (C)))
#define isascii(C) ((int) (C >= 0x00 && C <= 0x7F))
```

### 69.7.3 Funzioni di conversione

Le due funzioni *tolower()* e *toupper()* si occupano di convertire un carattere, rispettivamente, in minuscolo o in maiuscolo.

Listato 69.63. Funzione *tolower()*.

<b>NUL</b> 0000	<b>DLE</b> 0010	<b>SP</b> 0020	<b>0</b> 0030	<b>@</b> 0040	<b>P</b> 0050	<b>'</b> 0060	<b>p</b> 0070
<b>SOH</b> 0001	<b>DC1</b> 0011	<b>!</b> 0021	<b>1</b> 0031	<b>A</b> 0041	<b>Q</b> 0051	<b>a</b> 0061	<b>q</b> 0071
<b>STX</b> 0002	<b>DC2</b> 0012	<b>"</b> 0022	<b>2</b> 0032	<b>B</b> 0042	<b>R</b> 0052	<b>b</b> 0062	<b>r</b> 0072
<b>ETX</b> 0003	<b>DC3</b> 0013	<b>#</b> 0023	<b>3</b> 0023	<b>C</b> 0043	<b>S</b> 0053	<b>c</b> 0063	<b>s</b> 0073
<b>EOT</b> 0004	<b>DC4</b> 0014	<b>\$</b> 0024	<b>4</b> 0034	<b>D</b> 0044	<b>T</b> 0054	<b>d</b> 0064	<b>t</b> 0074
<b>ENQ</b> 0005	<b>NAK</b> 0015	<b>%</b> 0025	<b>5</b> 0035	<b>E</b> 0045	<b>U</b> 0055	<b>e</b> 0065	<b>u</b> 0075
<b>ACK</b> 0006	<b>SYN</b> 0016	<b>&amp;</b> 0026	<b>6</b> 0036	<b>F</b> 0046	<b>V</b> 0056	<b>f</b> 0066	<b>v</b> 0076
<b>BEL</b> 0007	<b>ETB</b> 0017	<b>'</b> 0027	<b>7</b> 0037	<b>G</b> 0047	<b>W</b> 0057	<b>g</b> 0067	<b>w</b> 0077
<b>BS</b> 0008	<b>CAN</b> 0018	<b>(</b> 0028	<b>8</b> 0038	<b>H</b> 0048	<b>X</b> 0058	<b>h</b> 0068	<b>x</b> 0078
<b>HT</b> 0009	<b>EM</b> 0019	<b>)</b> 0029	<b>9</b> 0039	<b>I</b> 0049	<b>Y</b> 0059	<b>i</b> 0069	<b>y</b> 0079
<b>LF</b> 000A	<b>SUB</b> 001A	<b>*</b> 002A	<b>:</b> 003A	<b>J</b> 004A	<b>Z</b> 005A	<b>j</b> 006A	<b>z</b> 007A
<b>VT</b> 000B	<b>ESC</b> 001B	<b>+</b> 002B	<b>;</b> 003B	<b>K</b> 004B	<b>[</b> 005B	<b>k</b> 006B	<b>{</b> 007B
<b>FF</b> 000C	<b>FS</b> 001C	<b>,</b> 002C	<b>&lt;</b> 003C	<b>L</b> 004C	<b>\</b> 005C	<b>l</b> 006C	<b> </b> 007C
<b>CR</b> 000D	<b>GS</b> 001D	<b>-</b> 002D	<b>=</b> 003D	<b>M</b> 004D	<b>]</b> 005D	<b>m</b> 006D	<b>}</b> 007D
<b>SO</b> 000E	<b>RS</b> 001E	<b>.</b> 002E	<b>&gt;</b> 003E	<b>N</b> 004E	<b>^</b> 005E	<b>n</b> 006E	<b>~</b> 007E
<b>SI</b> 000F	<b>US</b> 001F	<b>/</b> 002F	<b>?</b> 003F	<b>O</b> 004F	<b>_</b> 005F	<b>o</b> 006F	<b>DEL</b> 007F

```
#include <ctype.h>
int
tolower (int c)
{
    if (isupper (c))
    {
        return
            (c + 0x20);
    }
    else
    {
        return c;
    }
}
```

Listato 69.64. Funzione *toupper()*.

<b>NUL</b> 0000	<b>DLE</b> 0010	<b>SP</b> 0020	<b>0</b> 0030	<b>@</b> 0040	<b>P</b> 0050	<b>'</b> 0060	<b>p</b> 0070
<b>SOH</b> 0001	<b>DC1</b> 0011	<b>!</b> 0021	<b>1</b> 0031	<b>A</b> 0041	<b>Q</b> 0051	<b>a</b> 0061	<b>q</b> 0071
<b>STX</b> 0002	<b>DC2</b> 0012	<b>"</b> 0022	<b>2</b> 0032	<b>B</b> 0042	<b>R</b> 0052	<b>b</b> 0062	<b>r</b> 0072
<b>ETX</b> 0003	<b>DC3</b> 0013	<b>#</b> 0023	<b>3</b> 0023	<b>C</b> 0043	<b>S</b> 0053	<b>c</b> 0063	<b>s</b> 0073
<b>EOT</b> 0004	<b>DC4</b> 0014	<b>\$</b> 0024	<b>4</b> 0034	<b>D</b> 0044	<b>T</b> 0054	<b>d</b> 0064	<b>t</b> 0074
<b>ENQ</b> 0005	<b>NAK</b> 0015	<b>%</b> 0025	<b>5</b> 0035	<b>E</b> 0045	<b>U</b> 0055	<b>e</b> 0065	<b>u</b> 0075
<b>ACK</b> 0006	<b>SYN</b> 0016	<b>&amp;</b> 0026	<b>6</b> 0036	<b>F</b> 0046	<b>V</b> 0056	<b>f</b> 0066	<b>v</b> 0076
<b>BEL</b> 0007	<b>ETB</b> 0017	<b>'</b> 0027	<b>7</b> 0037	<b>G</b> 0047	<b>W</b> 0057	<b>g</b> 0067	<b>w</b> 0077
<b>BS</b> 0008	<b>CAN</b> 0018	<b>(</b> 0028	<b>8</b> 0038	<b>H</b> 0048	<b>X</b> 0058	<b>h</b> 0068	<b>x</b> 0078
<b>HT</b> 0009	<b>EM</b> 0019	<b>)</b> 0029	<b>9</b> 0039	<b>I</b> 0049	<b>Y</b> 0059	<b>i</b> 0069	<b>y</b> 0079
<b>LF</b> 000A	<b>SUB</b> 001A	<b>*</b> 002A	<b>:</b> 003A	<b>J</b> 004A	<b>Z</b> 005A	<b>j</b> 006A	<b>z</b> 007A
<b>VT</b> 000B	<b>ESC</b> 001B	<b>+</b> 002B	<b>;</b> 003B	<b>K</b> 004B	<b>[</b> 005B	<b>k</b> 006B	<b>{</b> 007B
<b>FF</b> 000C	<b>FS</b> 001C	<b>,</b> 002C	<b>&lt;</b> 003C	<b>L</b> 004C	<b>\</b> 005C	<b>l</b> 006C	<b> </b> 007C
<b>CR</b> 000D	<b>GS</b> 001D	<b>-</b> 002D	<b>=</b> 003D	<b>M</b> 004D	<b>]</b> 005D	<b>m</b> 006D	<b>}</b> 007D
<b>SO</b> 000E	<b>RS</b> 001E	<b>.</b> 002E	<b>&gt;</b> 003E	<b>N</b> 004E	<b>^</b> 005E	<b>n</b> 006E	<b>~</b> 007E
<b>SI</b> 000F	<b>US</b> 001F	<b>/</b> 002F	<b>?</b> 003F	<b>O</b> 004F	<b>_</b> 005F	<b>o</b> 006F	<b>DEL</b> 007F

```
#include <ctype.h>
int
toupper (int c)
{
    if (islower (c))
    {
        return
            (c - 0x20);
    }
    else
    {
        return c;
    }
}
```

Lo standard POSIX aggiunge anche la funzione *toascii()* che si limita ad azzerare i bit più significativi, dopo il settimo.

Listato 69.65. Funzione *toascii()*.

```
#include <ctype.h>
int
toascii (int c)
{
    return (c & 0x7F);
}
```

#### 69.7.4 macroistruzioni di conversione

Anche le funzioni *toupper()*, *tolower()* e *toascii()* possono essere rappresentate agevolmente in forma di macroistruzioni. Il listato seguente è conforme a quanto già visto nella sezione precedente:

```
#define tolower(C)  (isupper (C) ? ((C) + 0x20) : (C))
#define toupper(C) (islower (C) ? ((C) - 0x20) : (C))
#define toascii(C) (C & 0x7F)
```

Ma nel caso dello standard POSIX, in questo caso vanno ancora aggiunte due macroistruzioni, a cui non fanno capo funzioni con lo stesso nome:

```
#define _tolower(C) (isupper (C) ? ((C) + 0x20) : (C))
#define _toupper(C) (islower (C) ? ((C) - 0x20) : (C))
```

#### 69.7.5 Esempio di utilizzo delle funzioni

Viene proposto un programma molto semplice che utilizza tutte le funzioni dichiarate nel file 'ctype.h', ma solo secondo lo standard C:

```
#include <stdio.h>
#include <ctype.h>
```



```
int
main (int argc, char *argv[])
{
    int c;
    for (c = 0; c <= 0x7F; c++)
        {
            printf ("%02x", c);
            printf ("\t"); if (iscntrl (c)) printf ("cntrl");
                               if (isprint (c)) printf ("print");
            printf ("\t"); if (isblank (c)) printf ("blank");
                               if (isgraph (c)) printf ("graph");
            printf ("\t"); if (isspace (c)) printf ("space");
                               if (ispunct (c)) printf ("punct");
                               if (isupper (c)) printf ("upper");
                               if (islower (c)) printf ("lower");
                               if (isdigit (c)) printf ("digit");
            printf ("\t"); if (isalnum (c)) printf ("alnum");
            printf ("\t"); if (isxdigit (c)) printf ("xdigit");
            printf ("\t"); if (isalpha (c)) printf ("alpha");
            printf ("\n");
        }
    printf ("\n");
    printf ("ASCII:\n");
    for (c = 0; c <= 0x7F; c++)
        {
            if (isprint (c)) printf ("%c", c);
        }
    printf ("\n");
    printf ("\n");
    printf ("to upper:\n");
    for (c = 0; c <= 0x7F; c++)
        {
            if (isprint (c)) printf ("%c", toupper (c));
```

```

    }
    printf ("\n");
    printf ("\n");
    printf ("to lower:\n");
    for (c = 0; c <= 0x7F; c++)
        {
            if (isprint (c)) printf ("%c", tolower (c));
        }
    printf ("\n");
    return 0;
}

```

Una volta compilato il programma, avviandolo si deve ottenere un testo come quello che si vede nell'estratto seguente:

```

...
44      print      graph      upper      alnum      xdigit      alpha
45      print      graph      upper      alnum      xdigit      alpha
46      print      graph      upper      alnum      xdigit      alpha
47      print      graph      upper      alnum                        alpha
48      print      graph      upper      alnum                        alpha
...

```

ASCII:

```

!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJ↵
↵KLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

```

to upper:

```

!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJ↵
↵KLMNOPQRSTUVWXYZ[\]^_`ABCDEFGHIJKLMNOPQRSTUVWXYZ{|}~

```

to lower:

```

!"#$%&'()*+,-./0123456789:;<=>?@abcdefghij↵
↵klmnopqrstuvwxyz[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

```

## 69.8 File «stdarg.h»

Il file ‘`stdarg.h`’ della libreria standard definisce principalmente delle macroistruzioni per gestire gli argomenti variabili passati a una funzione, assieme a un tipo di variabile, ‘`va_list`’, specifico per gestire il puntatore a tali parametri non dichiarati (si veda eventualmente la realizzazione del file ‘`stdarg.h`’ nei sorgenti di `os32`, listato [95.1.10](#)).

Tabella 69.70. macroistruzioni standard per la gestione di argomenti variabili.

Macroistruzione	Descrizione
<pre>void va_start (va_list <i>ap</i>,                <i>parametro_n</i>);</pre>	Inizializza la variabile <i>ap</i> , di tipo ‘ <code>va_list</code> ’, in modo che punti all’area di memoria immediatamente successiva al parametro indicato, il quale deve essere l’ultimo.
<pre><i>tipo</i> va_arg (va_list <i>ap</i>, <i>tipo</i>);</pre>	Restituisce il contenuto dell’area di memoria a cui punta <i>ap</i> , utilizzando il tipo indicato, incrementando contestualmente il puntatore in modo che, al termine, si trovi nell’area di memoria immediatamente successiva.
<pre>void va_copy (va_list <i>dst</i>,               va_list <i>org</i>);</pre>	Copia il puntatore <i>org</i> nella variabile <i>dst</i> .

Macroistruzione	Descrizione
<code>void va_end (va_list <i>ap</i>);</code>	Conclude l'utilizzo del puntatore <i>ap</i> .

## 69.8.1 Realizzazione

«

Il listato successivo è tutto ciò che serve per realizzare la libreria:

```
typedef unsigned char * va_list;

#define va_start(AP, LAST) \
    ((void) ((AP) = ((va_list) &(LAST)) + (sizeof (LAST))))

#define va_end(AP) ((void) ((AP) = 0))
#define va_copy(DEST, SRC) \
    ((void) ((DEST) = (va_list) (SRC)))

#define va_arg(AP, TYPE) \
    (((AP) = (AP) + (sizeof (TYPE))), \
     *((TYPE *) ((AP) - (sizeof (TYPE)))))
```

Delle macroistruzioni mostrate nell'esempio, la più difficile da interpretare potrebbe essere *va\_arg*, la quale deve restituire il valore dell'area di memoria puntata inizialmente, ma garantendo di lasciare il puntatore pronto per l'area successiva. In pratica, prima viene incrementato il puntatore per l'area successiva, quindi viene dereferenziato ricalcolando lo spazio necessario a raggiungere la posizione precedente. In altri termini è come scrivere:

```
...
va_list ap;
...
//
```

```

// va_start
//
ap = (va_list) &ultimo_parametro;
ap = ap + (sizeof (tipo_ultimo_parametro));
...
//
// va_arg
//
ap = ap + (sizeof tipo_successivo);
var = (tipo_successivo *)
      (ap - (sizeof (tipo_successivo)));
...
//
// va_end
//
ap = 0;
...

```

## 69.8.2 Esempio di utilizzo delle macro

Viene riproposto un programma molto semplice, già apparso in altri capitoli, per dimostrare l'utilizzo delle macroistruzioni dichiarate nel file 'stdarg.h'. «

```

#include <stdio.h>
#include <stdarg.h>

void
f (int w, ...)
{
    long double x;    // Dichiarare le variabili che servono
    long long int y;  // a contenere gli argomenti per i
    int z;            // quali mancano i parametri formali.
}

```

```
va_list ap;           // Dichiaro il puntatore agli argomenti.

va_start (ap, w);    // Posiziona il puntatore dopo la fine
                    // di «w».

x = va_arg (ap, long double); // Estrae l'argomento
                               // successivo portando
                               // avanti il puntatore
                               // di conseguenza.

printf ("w = %i; ", w); // Mostra il valore del primo
                        // parametro.
printf ("x = %Lf; ", x); // Mostra il valore
                        // dell'argomento successivo.

y = va_arg (ap, long long int); // Estrapola e mostra
printf ("y = %lli; ", y);       // il terzo argomento.

z = va_arg (ap, int);           // Estrapola e mostra
printf ("z = %i\n", z);        // il quarto argomento.

va_end (ap);                   // Conclude la scansione.

return;
}

int
main (int argc, char *argv[])
{
    f (10, (long double)12.34, (long long int)13, 14);
    return 0;
}
```

Avviando il programma di esempio si deve visualizzare il messaggio

seguinte:

```
w = 10; x = 12.340000; y = 13; z = 14
```

### 69.8.3 Promozione

Va ricordato che gli argomenti delle chiamate alle funzioni vengono adattati in modo tale da facilitare l'uso della pila dei dati. Pertanto, i valori che prevedono una rappresentazione in memoria troppo piccola, subiscono quella che è nota come «promozione».

La funzione che ha degli argomenti variabili, dovrebbe gestire solo valori che non possono subire una trasformazione di questo tipo, altrimenti, quando poi utilizza la macroistruzione *va\_arg* deve indicare un tipo adeguato alla promozione che si prevede sia applicata ai valori degli argomenti.

A questo proposito si può notare che nell'esempio di utilizzo che appare nella sezione [69.8.2](#), non si fa mai uso di tipi di dati di rango inferiore a `int`.

## 69.9 File «stdlib.h»

Il file `stdlib.h` della libreria standard definisce alcuni tipi di dati, varie funzioni di utilità generale e alcune macro-variabili. Viene proposto un esempio di questo file, e di alcune delle funzioni a cui si riferisce, indicando per le altre solo i prototipi (si veda eventualmente la realizzazione del file `stdlib.h` e di alcune delle sue funzioni, nei sorgenti di os32, sezione [95.19](#)).

Lo standard POSIX estende significativamente il contenuto del file `stdlib.h`, ma qui non si fa riferimento ad alcuna di tali estensioni.

## 69.9.1 Tipi di dati speciali



I tipi di dati che il file ‘`stdlib.h`’ definisce sono ‘`size_t`’ e ‘`wchar_t`’, già descritti nel file ‘`stddef.h`’ (sezione [69.13](#)), oltre a ‘`div_t`’, ‘`ldiv_t`’ e ‘`lldiv_t`’. I tipi ‘...`div_t`’ sono delle strutture il cui scopo è quello di contenere il risultato di una divisione, espresso come quoziente e resto. Questi tipi di dati si usano per contenere il valore restituito dalle funzioni *div()*, *ldiv()* e *lldiv()*. La distinzione tra i tre tipi deriva dalla capienza dei membri della struttura. Ecco come potrebbero essere dichiarati:

```
typedef struct {
    int quot;
    int rem;
} div_t;
typedef struct {
    long int quot;
    long int rem;
} ldiv_t;
typedef struct {
    long long int quot;
    long long int rem;
} lldiv_t;
```

## 69.9.2 Macro-variabili



Il file ‘`stdlib.h`’ dichiara nuovamente la macro-variabile *NULL*, come già avviene nel file ‘`stddef.h`’ (sezione [69.13](#)); inoltre definisce quelle seguenti:



Macro	Descrizione
EXIT_SUCCESS  EXIT_FAILURE	Rappresentano un numero intero che possa essere usato come argomento della funzione <i>exit()</i> , per rappresentare il successo o l'insuccesso dell'attività svolta. Normalmente, i valori in cui si espandono le due macro-variabili sono rispettivamente zero e uno.
RAND_MAX	Rappresenta il valore massimo che possa essere generato dalla funzione <i>rand()</i> e generalmente corrisponde al valore massimo che può assumere un numero intero di tipo <code>int</code> .
MB_CUR_MAX	Rappresenta la quantità massima di byte che possono essere utilizzati in una sequenza multibyte, in base alla configurazione locale. Questo valore è un intero che deve essere di tipo <code>size_t</code> e non può superare il limite rappresentato dalla macro-variabile <i>MB_LEN_MAX</i> (dichiarata nel file <code>limits.h</code> , descritto nella sezione <a href="#">69.3</a> ).

Merita un po' di attenzione la macro-variabile *MB\_CUR\_MAX*. La sequenza multibyte è una sequenza di byte che, in base alla configurazione locale, deve essere interpretata come un carattere singolo. Per esempio, questo meccanismo si utilizza nella codifica UTF-8 e in altre; ma proprio perché esistono più metodi alternativi, per quanto superati possano essere rispetto a UTF-8, la configurazione locale stabilisce le regole particolari per interpretare tali sequenze e i limiti rispetto a queste. Pertanto, la macro-variabile *MB\_CUR\_MAX*

dovrebbe espandersi in una funzione che restituisce il valore desiderato, in relazione alla configurazione locale che si trova a essere attiva in un certo momento. Per semplicità, nell'esempio che viene proposto si associa il valore di questa macro-variabile a quello massimo accettabile in assoluto.

```
#define EXIT_FAILURE    1
#define EXIT_SUCCESS    0

#define RAND_MAX        INT_MAX

#define MB_CUR_MAX      ((size_t) MB_LEN_MAX)    // [1]
//
// [1] Sarebbe meglio una funzione.
//
```

Nell'esempio proposto viene usata la macro-variabile ***MB\_LEN\_MAX***, pertanto, in questo modo si rende necessaria l'inclusione del file `'limits.h'` che deve contenere la sua dichiarazione.

### 69.9.3 Conversioni numeriche

«

Un gruppo di funzioni del file `'stdlib.h'` permette di convertire una stringa in un valore numerico. In particolare, le funzioni con nomi del tipo ***ato...()*** (*ASCII to ...*) non eseguono controlli particolari e non modificano la variabile ***errno*** (sezione 69.5); invece, le funzioni con nomi ***strto...()*** (*string to ...*) sono più sofisticate.

Le funzioni ***ato...()*** interpretano una stringa e convertono il suo contenuto in un numero intero o in un numero a virgola mobile. Le funzioni sono ***atoi()***, ***atol()***, ***atoll()*** e ***atof()***, che convertono rispettivamente in un tipo `'int'`, `'long int'`, `'long long int'` e `'double'`. Ecco i prototipi:

```
int          atoi  (const char *nptr);
long int     atol  (const char *nptr);
long long int atoll (const char *nptr);
double       atof  (const char *nptr);
```

Viene proposta una soluzione per queste funzioni di conversione:

```
#include <stdlib.h>
#include <ctype.h>
int
atoi (const char *nptr)
{
    int i;
    int sign = +1;
    int n;

    for (i = 0; isspace (nptr[i]); i++)
        ; // Si limita a saltare gli spazi iniziali.
    }

    if (nptr[i] == '+')
        {
            sign = +1;
            i++;
        }
    else if (nptr[i] == '-')
        {
            sign = -1;
            i++;
        }

    for (n = 0; isdigit (nptr[i]); i++)
        {
            n = (n * 10) + (nptr[i] - '0'); // Accumula il valore.
        }
}
```

```
    return sign * n;
}
```

Logicamente, le funzioni *atol()* e *atoll()* sono praticamente uguali, con la differenza che la variabile automatica *n* deve essere dello stesso tipo restituito dalla funzione; pertanto si passa alla soluzione proposta per la funzione *atof()*:

```
#include <stdlib.h>
#include <ctype.h>
double
atof (const char *nptr)
{
    int i;
    int sign = +1;
    double n;           // Il risultato sarà: n / d.
    double d;          //

    for (i = 0; isspace (nptr[i]); i++)
        ;           // Si limita a saltare gli spazi iniziali.
    }

    if (nptr[i] == '+')
    {
        sign = +1;
        i++;
    }
    else if (nptr[i] == '-')
    {
        sign = -1;
        i++;
    }
}
```

```

for (n = 0.0; isdigit (nptr[i]); i++)
{
    n = (n * 10.0) + (nptr[i] - '0');           // Accumula
                                                // il valore.
}

if (nptr[i] == '.')
{
    i++;
}

for (d = 1.0; isdigit (nptr[i]); i++)
{
    // La variabile "d" viene inizializzata in ogni caso.

    n = (n * 10.0) + (nptr[i] - '0');
    d = d * 10.0;           // Tiene conto di quanto dovrà essere
                            // diviso il risultato.
}

return sign * n / d;
}

```

Le funzioni *strto...()* sono più complesse rispetto a quelle *ato...()*. Per dare una descrizione sommaria, si può osservare che, oltre alla stringa da scandire ricevono un puntatore di puntatore all'ultimo elemento utile di tale stringa;<sup>2</sup> se poi questo è nullo, la scansione avviene normalmente nella stringa, entro il limite del carattere nullo di terminazione. Se fallisce il riconoscimento del valore da tradurre, il puntatore all'inizio della stringa viene copiato nell'area di memoria a cui punta il puntatore di puntatore, a meno che questo, inizialmen-

te, sia già nullo (potrebbe essere nullo il puntatore principale o il contenuto dell'area a cui punta).

In caso di errore nell'interpretazione del valore, queste funzioni utilizzano la variabile *errno* per annotare il tipo di problema riscontrato.

I valori che possono essere convertiti sono esprimibili in notazione decimale o esadecimale; inoltre, le funzioni che convertono in valori a virgola mobile, accettano una notazione esponenziale e delle parole chiave per rappresentare l'infinito e NaN (*Not a number*). Nel caso particolare delle funzioni che convertono in un numero intero, esiste un terzo parametro per specificare la base di numerazione attesa.

Vengono presentati solo i prototipi di queste funzioni:

float	strtof	(const char * restrict nptr, char ** restrict endptr);
double	strtod	(const char * restrict nptr, char ** restrict endptr);
long double	strtold	(const char * restrict nptr, char ** restrict endptr);

long int	strtol	(const char * restrict nptr, char ** restrict endptr, int base);
long long int	strtoll	(const char * restrict nptr, char ** restrict endptr, int base);
unsigned long int	strtoul	(const char * restrict nptr, char ** restrict endptr, int base);
unsigned long long int	strtoull	(const char * restrict nptr, char ** restrict endptr, int base);

Per una descrizione completa si vedano le pagine di manuale *strtof(3)*, *strtod(3)*, *strtold(3)*, *strtol(3)*, *strtoll(3)*, *strtoul(3)* e *strtoull(3)*, oltre alla documentazione standard citata alla fine del capitolo.

L'esempio seguente mostra l'uso delle funzioni *ato...()*:

```
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    int          i;
    long int     li;
    long long int lli;
    double       d;
    char         s[] = " -987654.3210";

    i = atoi (s);
    li = atol (s);
    lli = atoll (s);
    d = atof (s);

    printf ("\n%s\n" = %i, %li, %lli, %f\n", s, i, li, lli, d);

    return 0;
}
```

Il risultato che ci si attende di visualizzare è questo:

```
" -987654.3210" = -987654, -987654, -987654, -987654.321000
```

## 69.9.4 Funzioni per la generazione di numeri in modo pseudo-casuale

«

La libreria standard deve disporre, nel file ‘`stdlib.h`’, di due funzioni per la generazione di numeri pseudo-casuali. Si tratta precisamente di *rand()* che restituisce un numero intero casuale (di tipo ‘`int`’, ma sono ammessi solo valori positivi) e di *srand()* che serve a cambiare il «seme» di generazione di tali numeri. Lo standard prescrive anche che per uno stesso seme, la sequenza di numeri pseudo-casuali sia la stessa e che il seme predefinito iniziale sia pari a uno.

Nella descrizione dello standard si fa riferimento al fatto che il valore che può essere generato deve andare da zero a ‘`RAND_MAX`’, escludendo quindi valori negativi. Considerando che la funzione *rand()* restituisce un valore di tipo ‘`int`’ e che questo non può essere negativo, significa che ‘`RAND_MAX`’ deve essere inferiore o uguale al massimo numero positivo rappresentabile con il tipo ‘`int`’.

```
int rand (void);  
void srand (unsigned int seed);
```

Viene proposta una soluzione molto semplice e anche molto scadente sul piano della sequenza casuale generata. Tuttavia garantisce che il valore ottenuto vada effettivamente da zero a ‘`RAND_MAX`’ incluso:



```
#include <stdlib.h>
static unsigned int _srand = 1; // Il rango di «_srand» deve
                                // essere maggiore o uguale
                                // a quello di «RAND_MAX»
                                // e di «unsigned int».

int
rand (void)
{
    _srand = _srand * 1234567 + 12345;
    return _srand % ((unsigned int) RAND_MAX + 1);
}

void
srand (unsigned int seed)
{
    _srand = seed;
}
```

L'esempio seguente consente di verificare sommariamente il lavoro delle funzioni per la generazione di numeri casuali. Per la precisione, si vogliono ottenere valori che vanno da 0 a 99 inclusi:

```
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    int r;
    int i;
    int j;
    const int max = 99;

    srand (123);

    for (i = 0; i < 25; i++)
    {
```

```
    for (j = 0; j < 26; j++)
        {
            r = (rand () % (max + 1));
            printf ("%2d ", r);
        }
    printf ("\n");
}

return 0;
}
```

Si dovrebbe ottenere un risultato simile a quello seguente:

```
86 67 22 15 22 47 94 91 10 99  6 31 22 51 98 11 14 11 22  3
22 11 82  7 54 11 62 75 10 63 50 75 98 91 90 79 74 87 18 15
34  7 54 71 42 47 10 23 82 67 50 19 14 31 34 27 58 71 38 43
14 87 38 71 82 95 50 99  2  7 30  7 98 87  2 19 46 71 78 83
34 43 58 99 70 79 30 75 58 99 46 31 78 91 34 79 98 75 14 99
66 63 82 99  2 51 46 11 74 39 90 31 54 63 78 39 14 31 50 55
26 87  2 51 70 15 98 67 54 27 34 55 14  7 74 71 42  7 30 87
70 67 34 15 26 47 90 91 30 19 66 27 86 15 26 15  6 99 46  7
14 51 22 43 54 95 58  7 18 15 50 83 26 71  2 27 38 71 30  7
94 19  6 47 62 19 90 39 54 39 42 15 86 91 82 87 38 27 34 87
74 87 50 87 82 67 78 63 22  3 26 87 86 11 94 71 26 35 10 83
78 71 46 63 82 47 74 63 70 11 54 95 62 31 74 87 30 71 54 15
18 19 14 79 86 23 58  7 98 47 10 63 46 39 26 19 10 67 54 99
 6 47 70 11 26  7 66 23 10 51 66 31 58 91 74 87 22 35 74 11
 6 39 58 39 26 35 94 55 82 47 78 67 98 91 30 67 18 75 74 83
18 23 54 51 26 91 38 11 22 23 66 91 18 99 74 27 82 99 62 35
58 35 54 95  2 95 86 95 82 47 90 51 90 51 54 23 62 91 90 99
18 19 62 51 10 59 26 39 82 71 94 83 98 27 38  7 22 15 90 79
18 31 58 71 22  3 58 39 98 63 62 55 38 51 86 51  2  7 86 87
94 43 54 67 62 23 42 31 10  3 42 47 74 87 26 27 54 43 54 95
18 31 34 27 58 47 66 47 70 75 22 35 82  7 54  7 62 19 18 91
22 63 94 83 98 27 86 59 78 91 78 35 62 67 90 67 62 35 78 95
```

```

86 99 54 47 14  3 62 19 58 63 74 11 62 99 46 71 86  7 34  3
34 59 18  3 62 43 62 27 26 95 46  3 22 15 54 35  6 75 86 27
58 51 70 71 82 15 50 39 38 91 14 99 66 67 66 91 54 31 34  3
30 55 98 27 50  7 38 11 26 11 42 47 26 31 10 91 90 15 54 19
50 87 70 35 14 79 10 47 74 55 34 51 54 95 58 23 46 59 78 91
38  3 94 51 70  3 54 51 86 51 14 95 70 55 50 99 70 63 86 15
 6 35 98 67 74 91 54 99 30  7 94 35 10 43 54 55 50 11 58 75
26 27 78 99 98 55 94 79 26 83 34 87 38 47  6 55 74 23 10 11
26 87  6 59 10 71 86 31 78 55 30 39 66 47 46 11 30 47  2 11
74 55 42 59  2 59 42 63 78 71  2 19  6 83 30 23  2 31 54 47
62 31 26 67  6 67 70 63 14 91

```

### 69.9.5 Funzioni standard per la generazione di numeri pseudo-casuali

Il documento che descrive lo standard descrive una versione della funzione *rand()* che corrisponde al listato successivo. I valori usati nei calcoli sono tali da essere adatti a un contesto in cui i limiti degli interi sono quelli minimi previsti. «

Listato 69.89. Esempio di realizzazione delle funzioni *rand()* e *srand()*, tratto dal documento che descrive lo standard del linguaggio.

```

// RAND_MAX assumed to be 32767

static unsigned long int next = 1;

int
rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

```

```
void
srand(unsigned int seed)
{
    next = seed;
}
```

## 69.9.6 Amministrazione della memoria

&lt;&lt;

Un gruppo di funzioni dichiarate nel file ‘`stdlib.h`’ consente di utilizzare dinamicamente la memoria. Si tratta di *malloc()*, *calloc()*, *realloc()* e *free()*. Le prime tre funzioni restituiscono un puntatore di tipo ‘`void *`’ all’area di memoria allocata, oppure il puntatore nullo nel caso l’operazione di allocazione fallisca; la funzione *free()* libera un’area di memoria allocata, indicando come argomento il puntatore che inizialmente la rappresentava.

```
void *malloc (size_t size);
void *calloc (size_t nmemb, size_t size);
void *realloc (void *ptr, size_t size);
void free (void *ptr);
```

Rispetto ai prototipi mostrati, la funzione *malloc()* richiede l’allocazione di una quantità di byte espressa dal parametro *size*; *calloc()* richiede una quantità di *nmemb* elementi da *size* byte (pertanto serve solo a facilitare l’allocazione di uno spazio necessario a un array); *realloc()* richiede la riallocazione della memoria già allocata precedentemente a partire dall’indirizzo *ptr* per avere *size* byte, con l’intento di non perdere le informazioni precedenti (a meno che si tratti di una riduzione della dimensione); infine, *free()* si limita a deallocare la memoria a cui punta *ptr*.

Nel linguaggio C, la memoria deve essere allocata e liberata espressamente, in quanto non esiste alcun sistema automatico al riguardo.

La gestione della memoria dipende strettamente dal sistema operativo, pertanto la realizzazione delle funzioni non può essere generalizzata. Per i dettagli che riguardano il comportamento di queste funzioni nel proprio sistema operativo vanno consultate le pagine di manuale *malloc(3)*, *calloc(3)*, *realloc(3)* e *free(3)*.

### 69.9.7 Conclusione forzata del programma

Alcune funzioni si occupano di interrompere il funzionamento del programma al di fuori della conclusione naturale della funzione *main()*. In generale si possono distinguere i casi in cui la conclusione del programma viene gestita in modo gentile, oppure viene forzata brutalmente.

Per una conclusione corretta di un programma, è possibile predisporre un elenco di funzioni da eseguire automaticamente nel momento della conclusione. Ciò avviene attraverso la funzione *atexit()* che accumula un elenco di puntatori a funzione; successivamente, attraverso la chiamata alla funzione *exit()* si ottiene l'esecuzione delle funzioni dell'elenco, senza argomenti, secondo l'ordine di inserimento. Quindi, la funzione *exit()* conclude con la chiusura dei file e con la restituzione del valore passato come argomento.

Una conclusione brutale si ottiene con la funzione *\_Exit()*, che si limita a terminare il programma, ma senza fare altro, soprattutto senza garantire che i file aperti siano chiusi correttamente.

Per ottenere una conclusione brutale del funzionamento di un programma si può usare anche la funzione *abort()* che però è legata alla gestione dei segnali (sezione 69.15) e qui non viene spiegato il suo utilizzo.

```
int  atexit (void (*func) (void));
void exit  (int status);
void _Exit (int status);
void abort (void);
```

La funzione *atexit()* riceve come unico argomento il puntatore a una funzione, la quale non restituisce alcun valore (di tipo ‘**void**’) e non si attende alcun argomento (ancora il tipo ‘**void**’). La funzione *atexit()* restituisce un valore numerico da intendere come *Vero* o *Falso*, per comunicare il successo o l’insuccesso dell’operazione, dato che la quantità di puntatori a funzione che possono essere accumulati può avere un limite.

Per una descrizione completa dell’uso di queste funzioni si vedano le pagine di manuale *abort(3)*, *atexit(3)*, *exit(3)* e *\_Exit(3)*.

### 69.9.8 Funzioni di comunicazione con l’ambiente

«

Nel file ‘*stdlib.h*’ sono dichiarate due funzioni per interagire con il sistema operativo, *getenv()* e *system()*, dove la prima consente di interrogare le variabili di ambiente (nel senso inteso nei sistemi Unix ed equivalenti) e la seconda consente di eseguire dei comandi attraverso la shell.

Nel documento che descrive lo standard del linguaggio C, il concetto viene generalizzato, ma in pratica, il contesto da cui derivano queste funzioni è quello dei sistemi Unix.

```
char *getenv (const char *name);  
int  system (const char *string);
```

La funzione *getenv()* si aspetta di ricevere come argomento il nome di una variabile di ambiente (o di qualcosa di comparabile, nel contesto di un altro tipo di sistema operativo), restituendo il puntatore al contenuto di tale variabile. La funzione *system()* può essere usata indicando un puntatore nullo e in tal caso restituisce un valore diverso da zero se il sistema operativo è in grado di recepire dei comandi testuali. Se invece viene passata una stringa, la funzione tenta di farla eseguire come comando del sistema operativo: in un sistema Unix o equivalente si tratta di un comando che deve essere eseguito da `/bin/sh`. L'esito della funzione *system()* dipende da quello del comando impartito e generalmente si ottiene lo stesso valore restituito dal comando eseguito.

Si vedano le pagine di manuale *getenv(3)* e *system(3)*.

### 69.9.9 Funzioni di ricerca e riordino

Il file `stdlib.h` prevede la dichiarazione di due funzioni per il riordino degli array e per la ricerca all'interno di array ordinati. Si tratta precisamente delle funzioni *qsort()* e *bsearch()*, dove i nomi richiamano evidentemente gli algoritmi tradizionali noti come *quick sort* e *binary search*. «

Le funzioni della libreria standard generalizzano il problema dell'ordinamento e della ricerca utilizzando puntatori di tipo `void *` e scandendo la memoria a blocchi di una dimensione determinata. Ma dal momento che l'area di memoria da scandire non ha la personalità di un array di un qualche tipo, occorre fornire a entrambe queste

funzioni il puntatore a una funzione diversa, in grado di confrontare due valori nel contesto di proprio interesse.

```
void qsort (void *base,
           size_t nmemb,
           size_t size,
           int (*compar) (const void *, const void *));

void *bsearch (const void *key,
              const void *base,
              size_t nmemb,
              size_t size,
              int (*compar) (const void *, const void *));
```

Prima di descrivere il significato dei parametri delle due funzioni, conviene vedere un esempio in cui queste si utilizzano. Per la precisione viene scandito un piccolo array di elementi di tipo `int`: prima viene ordinato, poi si cerca un elemento al suo interno.

```
#include <stdio.h>
#include <stdlib.h>

int confronta (const void *a, const void *b)
{
    int x = *((int *) a);
    int y = *((int *) b);
    return x - y;
}

int main (void)
{
    int a[] = {3, 1, 5, 2};
    int cercato = 5;
    void *p;

    qsort (&a[0], 4, sizeof (int), confronta);
```



```
printf ("%i %i %i %i\n", a[0], a[1], a[2], a[3]);

p = bsearch (&cercato, &a[0], sizeof (int), 4, confronta);

printf ("%a[0] = %u; \"%i\" si trova in %u.\n",
        (unsigned int) &a[0], cercato, (unsigned int) p);

return 0;
}
```

Nell'esempio viene dichiarata la funzione *confronta()* che riceve due argomenti e restituisce un valore che può essere: minore, pari o maggiore di zero, se il primo argomento, rispetto al secondo, è minore, pari o maggiore. Questo è il modo in cui deve comportarsi la funzione da passare come argomento a *qsort()* e a *bsearch()*, tenendo conto che è da tali funzioni che riceve gli argomenti.

La funzione *qsort()* vuole ricevere il puntatore alla prima posizione in memoria da riordinare (il parametro *base*), la quantità degli elementi da riordinare (*nmemb*, ovvero *Number of memory blocks*), la dimensione di tali elementi (*size*) e la funzione da usare per la loro comparazione.

La funzione *bsearch()* vuole ricevere il puntatore alla chiave di ricerca (il parametro *key*), il puntatore alla prima posizione in memoria da scandire (*base*), la quantità degli elementi da scandire (*nmemb*), la dimensione di tali elementi (*size*) e la funzione da usare per la loro comparazione, tenendo conto che questa riceve la chiave di ordinamento come primo argomento.

L'esempio mostrato esegue un ordinamento crescente e il testo visualizzato che si ottiene deve essere simile a quello seguente:

```
1 2 3 5
```

```
&a[0] = 3218927260; "5" si trova in 3218927272.
```

È sufficiente invertire il risultato della funzione di comparazione per ottenere un ordinamento decrescente e per scandire un array ordinato in modo decrescente:

```
int confronta (const void *a, const void *b)
{
    int x = *((int *) a);
    int y = *((int *) b);
    return y - x;
}
```

In tal caso, il testo che viene emesso deve essere simile a quello seguente:

```
5 3 2 1
```

```
&a[0] = 3218593340; "5" si trova in 3218593340.
```

### 69.9.10 Funzioni per l'aritmetica con i numeri interi

«

Un gruppo di funzioni il cui nome termina per *...abs()* si occupa di calcolare il valore assoluto di un numero intero. Le funzioni sono precisamente: *abs()* per gli interi di tipo 'int', *labs()* per gli interi di tipo 'long int' e *llabs()* per gli interi di tipo 'long long int'.

```
int abs                (int j);
long int labs         (long int j);
long long int llabs   (long long int j);
```

Evidentemente, la realizzazione di queste funzioni è estremamente banale. Viene presentato solo il caso di *abs()*:

```
#include <stdlib.h>
int
abs (int j)
{
    if (j < 0)
        {
            return -j;
        }
    else
        {
            return j;
        }
}
```

Un gruppo di funzioni il cui nome termina per *...div()* si occupa di dividere due interi, calcolando il quoziente e il resto. Le funzioni sono precisamente: *div()* per gli interi di tipo ‘**int**’, *ldiv()* per gli interi di tipo ‘**long int**’ e *lldiv()* per gli interi di tipo ‘**long long int**’. Il risultato viene restituito in una variabile strutturata che contiene sia il quoziente, sia il resto: rispettivamente si tratta dei tipi ‘**div\_t**’, ‘**ldiv\_t**’ e ‘**lldiv\_t**’.

```
div_t   div    (int numer, int denom);
ldiv_t  ldiv   (long int numer, long int denom);
lldiv_t lldiv  (long long int numer, long long int denom);
```

I tre tipi creati appositamente per contenere il risultato di queste funzioni contengono i membri ‘**quot**’ e ‘**rem**’ che rappresentano, rispettivamente, il quoziente e il resto. Anche la realizzazione di queste funzioni è molto semplice banale. Viene presentato solo il caso di *div()*:

```
#include <stdlib.h>
div_t
div (int numer, int denom)
{
    div_t d;
    d.quot = numer / denom;
    d.rem = numer % denom;
    return d;
}
```

### 69.9.11 Funzioni per la gestione di caratteri estesi e sequenze multibyte

«

Il linguaggio C distingue tra una gestione dei caratteri basata sul byte, tale da consentire la gestione di un insieme minimo, come quello della codifica ASCII, e una gestione a byte multipli, o multibyte. Per esempio, la codifica UTF-8 è ciò che si intende per «multibyte», ma esistono anche altre codifiche che sfruttano questo meccanismo.

Quando il contesto richiede l'interpretazione dei byte secondo una codifica multibyte, è necessario stabilire un punto di riferimento per iniziare l'interpretazione e occorre poterne conservare lo stato quando la lettura di un carattere viene interrotta e ripresa a metà. Nella documentazione dello standard, nell'ambito delle sequenze multibyte, lo stato viene definito *shift state*.

Per gestire internamente la codifica universale, il C utilizza un tipo specifico, `wchar_t`, corrispondente a un intero di rango sufficiente a rappresentare tutti i caratteri che si intendono gestire. Di conseguenza, le stringhe letterali, precedute dalla lettera `L` (per esempio `L"àèìòùé"`), sono array di elementi `wchar_t`.

Le funzioni che riguardano la gestione di caratteri estesi e sequen-

ze multibyte del file `'stdlib.h'`, servono principalmente per convertire sequenze multibyte nel tipo `'wchar_t'` e viceversa. Tuttavia, occorre tenere presente che la configurazione locale deve essere tale da prevedere l'uso di caratteri da rappresentare attraverso sequenze multibyte, altrimenti le conversioni diventano prive di utilità.

Listato 69.102. Prototipi delle funzioni relative alla gestione multibyte.

```
int      mblen      (const char *s, size_t n);
int      mbtowc     (wchar_t *restrict pwc,
                    const char *restrict s,
                    size_t n);
int      wctomb     (char *s, wchar_t wc);
size_t   mbstowcs   (wchar_t *restrict pwcs,
                    const char *restrict s,
                    size_t n);
size_t   wcstombs   (char *restrict s,
                    const wchar_t *restrict pwcs,
                    size_t n);
```

### 69.9.12 Funzione «mblen()»

La funzione *mblen()* si usa normalmente per contare quanti byte sono presenti nella stringa *s* fornita come primo argomento, per comporre il primo carattere (multibyte) della stringa stessa, limitando la scansione a un massimo di *n* byte (il secondo argomento richiesto). Se al posto di indicare una stringa si fornisce il puntatore nullo, si ottiene un valore che può essere uno o zero, a seconda che sia prevista o meno una codifica multibyte con una gestione dello stato (*shift state*).

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
```

```
int main (void)
{
    int n;
    setlocale (LC_ALL, "en_US.UTF-8");
    n = mblen (NULL, 0);
    printf ("Gestione dello stato: %i\n", n);
    n = mblen ("€", 8);
    printf ("Il carattere %s richiede %i byte.\n", "€", n);
    return 0;
}
```

L'esempio mostrato dovrebbe chiarire alcune cose. La funzione richiede un argomento di tipo stringa di caratteri, perché un argomento di tipo **'char'** singolo, non consentirebbe di annotare una sequenza multibyte. Le sequenze multibyte sono stringhe normali, trattate come tali, salvo quando è necessario interpretare il loro contenuto; a questo proposito, si vede che la funzione *printf()* riceve una stringa multibyte e si limita a trattarla come una stringa normale.

Se la codifica in cui è scritto il sorgente è la stessa usata dal programma durante il suo funzionamento, si può ottenere il testo seguente:

```
Gestione dello stato: 0
Il carattere € richiede 3 byte.
```

Nell'uso normale della funzione *mblen()*, se la stringa che si fornisce contiene una sequenza multibyte errata o incompleta, il valore restituito è  $-1$ .

### 69.9.13 Funzioni «mbtowc()» e «wctomb()»

Le due funzioni *mbtowc()* e *wctomb()* si compensano a vicenda, fornendo il mezzo elementare di conversione dei caratteri da una sequenza multibyte a un numero intero di tipo ‘**wchar\_t**’ e viceversa. Quando a queste funzioni, al posto del puntatore alla stringa multibyte, si fornisce il puntatore nullo, si ottiene un funzionamento analogo a quello di *mblen()*, con un valore pari a uno se la configurazione locale prevede l’uso di sequenze multibyte e una gestione dello stato, oppure zero se questo problema non sussiste. Inoltre, per entrambe le funzioni, se la sequenza multibyte è errata o incompleta, si ottiene la restituzione del valore  $-1$ . Infine, se la conversione ha successo, si ottiene la quantità dei byte che compongono la sequenza multibyte (di origine o di destinazione, a seconda della funzione usata).

Viene mostrato un esempio molto semplice che dimostra l’uso delle due funzioni. In particolare viene convertita la sequenza multibyte che rappresenta la lettera «ä» in un numero ‘**wchar\_t**’, quindi il numero viene incrementato e riconvertito in una nuova sequenza multibyte, per ottenere il carattere «å».

```
#include <locale.h>
#include <stdlib.h>
#include <stdio.h>

int main (void)
{
    int n;
    wchar_t wc;
    char mb[20] = {}; // Inizializza l'array a zero.

    setlocale (LC_ALL, "en_US.UTF-8");
```

```

n = mbtowc (&wc, NULL, 8);
printf ("Gestione dello stato: %i\n", n);
n = wctomb (NULL, wc);
printf ("Gestione dello stato: %i\n", n);

n = mbtowc (&wc, "ä", 8);
printf ("Il carattere \"ä\" si rappresenta con %i byte ",
        n);
printf ("in una sequenza multibyte e con il numero %i ",
        (int) wc);
printf ("in una variabile di tipo \"wchar_t\".\n");

wc++;
n = wctomb (mb, wc);
printf ("Il carattere \"%s\" si rappresenta con %i byte ",
        mb, n);
printf ("in una sequenza multibyte e con il numero %i ",
        (int) wc);
printf ("in una variabile di tipo \"wchar_t\".\n");

return 0;
}

```

Si dovrebbe ottenere un testo come quello seguente:

```

Gestione dello stato: 0
Gestione dello stato: 0
Il carattere "ä" si rappresenta con 2 byte in una ↵
↳sequenza multibyte e con il numero 228 ↵
↳in una variabile di tipo "wchar_t".
Il carattere "å" si rappresenta con 2 byte in una ↵
↳sequenza multibyte e con il numero 229 ↵
↳in una variabile di tipo "wchar_t".

```

Per gli approfondimenti eventuali, si vedano le pagine di manuale



*mbtowc(3)* e *wctomb(3)*.

#### 69.9.14 Funzioni «*mbstowcs()*» e «*wcstombs()*»

Le funzioni *mbstowcs()* e '*wcstombs*' servono rispettivamente per convertire una stringa multibyte in un stringa estesa (un array di elementi '*wchar\_t*') e per fare l'opposto. Entrambe le funzioni richiedono tre argomenti: l'array di destinazione, l'array di origine e la quantità di elementi da utilizzare nell'array di destinazione. Entrambe le funzioni restituiscono un numero che esprime la quantità di elementi di destinazione convertiti, escluso ciò che costituisce il carattere nullo di terminazione. Entrambe restituiscono un valore pari a '*(size\_t) (-1)*' se la conversione produce un errore.<sup>3</sup>

Per convertire correttamente una stringa (multibyte o estesa), occorre che il numero di elementi di destinazione previsto includa anche il carattere nullo di terminazione. L'esempio seguente dovrebbe aiutare a comprendere il problema:

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    size_t n;
    wchar_t wca[] = {1, 2, 3, 4, 5, 6, 7};
    wchar_t wcb[] = {1, 2, 3, 4, 5, 6, 7};
    char mba[] = "*****";
    char mbb[] = "*****";

    setlocale (LC_ALL, "en_US.UTF-8");

    n = mbstowcs (wca, "ääâ", 3);
```

```
printf ("mbstowcs: %i: %i %i %i %i %i\n", n,
        wca[0], wca[1], wca[2], wca[3], wca[4]);

n = mbstowcs (wcb, "ääâ", 5);
printf ("mbstowcs: %i: %i %i %i %i %i\n", n,
        wcb[0], wcb[1], wcb[2], wcb[3], wcb[4]);

n = wcstombs (mba, L"ääâ", 6);
printf ("wcstombs: %i: \"%s\"\n", n, mba);

n = wcstombs (mbb, L"ääâ", 9);
printf ("wcstombs: %i: \"%s\"\n", n, mbb);

return 0;
}
```

Nell'esempio, la funzione *mbstowcs()* viene usata due volte, per convertire una stringa multibyte, composta da tre caratteri, se non si conta quello di terminazione. Nel primo caso, viene specificato che si vogliono convertire esattamente tre caratteri, ma questo significa che nell'array di destinazione rimane il contenuto originale a partire dal quarto elemento. In modo analogo, la funzione *wcstombs()* viene usata due volte per convertire una stringa estesa in una stringa multibyte. La stringa estesa si compone di tre caratteri che nella conversione vanno a occupare esattamente sei byte, con l'aggiunta eventuale del carattere nullo di terminazione (che sarebbe il settimo). Si può vedere che quando si chiede una conversione di sei elementi, la stringa ricevente mantiene il contenuto precedente nella parte restante. Ecco cosa si dovrebbe vedere eseguendo il programma:

```

mbstowcs: 3: 228 229 226 4 5
mbstowcs: 3: 228 229 226 0 5
wcstombs: 6: "ääâ*****"
wcstombs: 6: "ääâ"

```

Se al posto della destinazione (il primo argomento) viene posto il puntatore nullo, si ottiene la simulazione dell'operazione, senza memorizzare alcunché e senza tenere conto della quantità massima di elementi che si annota come ultimo argomento. Ciò ha lo scopo di contare quanti elementi servirebbero per produrre una conversione completa. L'esempio seguente modifica quello già visto, sfruttando questa funzionalità:

```

#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    size_t n;
    size_t max;
    wchar_t wca[] = {1, 2, 3, 4, 5, 6, 7};
    char mba[] = "*****";           // 15 byte total.

    setlocale (LC_ALL, "en_US.UTF-8");

    max = mbstowcs (NULL, "ääâ", 0);
    if (max <= 6)
    {
        n = mbstowcs (wca, "ääâ", max + 1);
        printf ("mbstowcs: %i: %i %i %i %i %i\n", n,
                wca[0], wca[1], wca[2], wca[3], wca[4]);
    }
}

```

```
max = wcstombs (NULL, L"ääâ", 0);
if (max <= 14)
{
    n = wcstombs (mba, L"ääâ", max + 1);
    printf ("wcstombs: %i: \"%s\"\n", n, mba);
}

return 0;
}
```

Ecco cosa si dovrebbe vedere eseguendo il programma:

```
mbstowcs: 3: 228 229 226 0 5
wcstombs: 6: "ääâ"
```

Per gli approfondimenti eventuali, si vedano le pagine di manuale *mbstowcs(3)* e *wcstombs(3)*.

## 69.10 File «inttypes.h»

«

Il file ‘inttypes.h’ della libreria standard serve principalmente a completare le funzionalità di ‘stdint.h’, per ciò che riguarda la gestione dei valori numerici interi, il cui rango è controllabile. Infatti, il problema principale nell’uso di interi definiti in modo alternativo allo standard del linguaggio C, privo di librerie, sta nell’uso appropriato degli specificatori di conversione nelle funzioni come *printf()* e *scanf()*. È proprio per risolvere questo problema che nel file ‘inttypes.h’ vanno definite, soprattutto, delle macrovariabili da usare in sostituzione degli specificatori di conversione basati sui tipi elementari (si veda eventualmente la realizzazione del file ‘inttypes.h’, ma senza le funzioni che lo riguardano, nei sorgenti di os32, listato [95.8](#)).<sup>4</sup>

Gli esempi proposti per descrivere la libreria che fa capo al file `'inttypes.h'` si riferiscono a quanto già definito nella sezione [69.4](#) a proposito del file `'stdint.h'`.

Inizialmente, il file `'inttypes.h'` deve includere `'stdint.h'`, inoltre dichiara il tipo `'wchar_t'`, già descritto nel file `'stddef.h'`:

```
#include <stdint.h>
typedef unsigned int wchar_t;
```

### 69.10.1 Divisione intera con interi di rango massimo

Nel file `'inttypes.h'` viene definito il tipo `'imaxdiv_t'` che va ad affiancarsi ai tipi `'div_t'`, `'ldiv_t'` e `'lldiv_t'`, definiti nel file `'stdlib.h'`. In pratica si tratta di una struttura il cui scopo è quello di contenere il risultato di una divisione, espresso come quoziente e resto, quando il tipo intero usato è quello massimo: «

```
typedef struct {
    intmax_t quot;
    intmax_t rem;
} imaxdiv_t;
```

Il tipo strutturato `'imaxdiv_t'` serve alle funzioni *imaxdiv()* e *uimaxdiv()*, le quali sono sostanzialmente equivalenti alle altre funzioni *..div()* del file `'stdlib.h'`:

```
imaxdiv_t imaxdiv (intmax_t numer, intmax_t denom);
```

```
#include <inttypes.h>
imaxdiv_t
imaxdiv (intmax_t numer, intmax_t denom)
{
    imaxdiv_t d;
    d.quot = numer / denom;
    d.rem = numer % denom;
    return d;
}
```

## 69.10.2 Macro-variabili in qualità di specificatori di conversione

«

Come accennato all'inizio del capitolo, per poter usare le funzioni *...printf()* e *...scanf()*, occorrono degli specificatori di conversione, ma non ne esistono per i tipi interi a rango controllato, pertanto, per questi, servono delle macro-variabili coerenti con il tipo relativo.

Le macro che iniziano per *PRI...* si usano come parte terminale di specificatori di conversione per la composizione dell'output (*...printf()*), mentre le macro che iniziano per *SCN...* sono adatte per l'interpretazione dell'input (*...scanf()*).

Le macro '*PRIn*' e '*SCNn*' terminano gli specificatori di conversione '*%...x*', per i tipi interi '*[u] intn\_t*'; le macro '*PRILEASTn*' e '*SCNLEASTn*' riguardano i tipi '*[u] int\_leastn\_t*'; le macro '*PRIFASTn*' e '*SCNFASTn*' riguardano i tipi '*[u] int\_fastn\_t*'; le macro '*PRIMAXn*' e '*SCNMAXn*' riguardano i tipi '*[u] intmax\_t*'; le macro '*PRIPTRn*' e '*SCNPTRn*' riguardano i tipi '*[u] intptr\_t*'.

L'esempio seguente dovrebbe dimostrare il significato di queste macro-variabili, attraverso l'uso di *printf()*:

```
#include <stdio.h>
#include <inttypes.h>
int
main (int argc, char *argv[])
{
    uint64_t num = INT64_C(1234567890);
    printf ("Il valore della variabile \"num\" "
           "corrisponde a "
           "%020" PRIu64 ".\n", num);
    return 0;
}
```

Il listato seguente mostra come possono essere dichiarate queste macro-variabili:

```
// Composizione dell'output.

#define PRId8           "d"
#define PRId16          "d"
#define PRId32          "d"
#define PRId64          "lld"

#define PRIdLEAST8      "d"
#define PRIdLEAST16     "d"
#define PRIdLEAST32     "d"
#define PRIdLEAST64     "lld"

#define PRIdFAST8       "d"
#define PRIdFAST16      "d"
#define PRIdFAST32      "d"
#define PRIdFAST64      "lld"

#define PRIdMAX         "lld"
```

```
#define PRIdPTR          "d"

#define PRIi8           "i"
#define PRIi16          "i"
#define PRIi32          "i"
#define PRIi64          "lli"

#define PRIiLEAST8      "i"
#define PRIiLEAST16     "i"
#define PRIiLEAST32     "i"
#define PRIiLEAST64     "lli"

#define PRIiFAST8       "i"
#define PRIiFAST16      "i"
#define PRIiFAST32      "i"
#define PRIiFAST64      "lli"

#define PRIiMAX         "lli"
#define PRIiPTR         "i"

#define PRIo8           "o"
#define PRIo16          "o"
#define PRIo32          "o"
#define PRIo64          "llo"

#define PRIoLEAST8      "o"
#define PRIoLEAST16     "o"
#define PRIoLEAST32     "o"
#define PRIoLEAST64     "llo"

#define PRIoFAST8       "o"
#define PRIoFAST16      "o"
#define PRIoFAST32      "o"
#define PRIoFAST64      "llo"
```



```
#define PRIoMAX          "llo"
#define PRIoPTR          "o"

#define PRIu8            "u"
#define PRIu16           "u"
#define PRIu32           "u"
#define PRIu64           "llu"

#define PRIuLEAST8       "u"
#define PRIuLEAST16      "u"
#define PRIuLEAST32      "u"
#define PRIuLEAST64      "llu"

#define PRIuFAST8        "u"
#define PRIuFAST16       "u"
#define PRIuFAST32       "u"
#define PRIuFAST64       "llu"

#define PRIuMAX          "llu"
#define PRIuPTR          "u"

#define PRIx8            "x"
#define PRIx16           "x"
#define PRIx32           "x"
#define PRIx64           "llx"

#define PRIxLEAST8       "x"
#define PRIxLEAST16      "x"
#define PRIxLEAST32      "x"
#define PRIxLEAST64      "llx"

#define PRIxFAST8        "x"
#define PRIxFAST16       "x"
```

```
#define PRIxFAST32      "x"
#define PRIxFAST64     "llx"

#define PRIxMAX        "llx"
#define PRIxPTR        "x"

#define PRIx8          "X"
#define PRIx16         "X"
#define PRIx32         "X"
#define PRIx64         "llx"

#define PRIxLEAST8     "X"
#define PRIxLEAST16   "X"
#define PRIxLEAST32   "X"
#define PRIxLEAST64   "llx"

#define PRIxFAST8      "X"
#define PRIxFAST16     "X"
#define PRIxFAST32     "X"
#define PRIxFAST64     "llx"

#define PRIxMAX        "llx"
#define PRIxPTR        "X"
```

*// Interpretazione dell'input.*

```
#define SCNd8          "hhd"
#define SCNd16         "hd"
#define SCNd32         "d"
#define SCNd64         "lld"

#define SCNdLEAST8     "hhd"
#define SCNdLEAST16   "hd"
#define SCNdLEAST32   "d"
```

```
#define SCNdLEAST64      "lld"

#define SCNdFAST8        "hhd"
#define SCNdFAST16       "d"
#define SCNdFAST32       "d"
#define SCNdFAST64       "lld"

#define SCNdMAX          "lld"
#define SCNdPTR          "d"

#define SCNi8           "hhi"
#define SCNi16          "hi"
#define SCNi32          "i"
#define SCNi64          "lli"

#define SCNiLEAST8      "hhi"
#define SCNiLEAST16     "hi"
#define SCNiLEAST32     "i"
#define SCNiLEAST64     "lli"

#define SCNiFAST8       "hhi"
#define SCNiFAST16      "i"
#define SCNiFAST32      "i"
#define SCNiFAST64      "lli"

#define SCNiMAX         "lli"
#define SCNiPTR         "i"

#define SCNo8           "hho"
#define SCNo16          "ho"
#define SCNo32          "o"
#define SCNo64          "llo"

#define SCNoLEAST8      "hho"
```

```
#define SCNoLEAST16    "ho"
#define SCNoLEAST32    "o"
#define SCNoLEAST64    "llo"

#define SCNoFAST8      "hho"
#define SCNoFAST16     "o"
#define SCNoFAST32     "o"
#define SCNoFAST64     "llo"

#define SCNoMAX        "llo"
#define SCNoPTR        "o"

#define SCNu8          "hhu"
#define SCNu16         "hu"
#define SCNu32         "u"
#define SCNu64         "llu"

#define SCNuLEAST8     "hhu"
#define SCNuLEAST16    "hu"
#define SCNuLEAST32    "u"
#define SCNuLEAST64    "llu"

#define SCNuFAST8      "hhu"
#define SCNuFAST16     "u"
#define SCNuFAST32     "u"
#define SCNuFAST64     "llu"

#define SCNuMAX        "llu"
#define SCNuPTR        "u"

#define SCNx8          "hhx"
#define SCNx16         "hx"
#define SCNx32         "x"
#define SCNx64         "llx"
```

```
#define SCNxLEAST8      "hhx"
#define SCNxLEAST16    "hx"
#define SCNxLEAST32    "x"
#define SCNxLEAST64    "llx"

#define SCNxFAST8      "hhx"
#define SCNxFAST16     "x"
#define SCNxFAST32     "x"
#define SCNxFAST64     "llx"

#define SCNxMAX        "llx"
#define SCNxPTR        "x"
```

### 69.10.3 Valore assoluto

Nel file `stdlib.h` si trovano dichiarate alcune funzioni per il calcolo del valore assoluto: ...*abs()*. Nel file `inttypes.h` si aggiunge la funzione *imaxabs()*, da usare per i valori interi massimi: <<

```
intmax_t imaxabs (intmax_t j);
```

```
#include <inttypes.h>
intmax_t
imaxabs (intmax_t j)
{
    if (j < 0)
    {
        return -j;
    }
    else
    {
        return j;
    }
}
```

#### 69.10.4 Conversione da stringa a numero intero



Per convertire una stringa contenente un valore numerico intero, quando si vuole fare riferimento all'intero di dimensione massima, si possono usare le funzioni *strtoimax()*, *strtouimax()*, *wcstoimax()* e *wcstouimax()*, dichiarate nel file 'inttypes.h'. Come il nome suggerisce, le prime due funzioni sono destinate alla conversione di stringhe «normali», mentre le altre sono specifiche per le stringhe estese.

Evidentemente si tratta di funzioni che si abbinano alle altre *strto...()* del file 'stdlib.h' e alle funzioni *wcsto...()* del file 'wchar.h'.

```

intmax_t  strtouimax (const char *restrict nptr,
                    char **restrict endptr, int base);
uintmax_t strtouimax (const char *restrict nptr,
                    char **restrict endptr, int base);
intmax_t  wcstouimax (const wchar_t *restrict nptr,
                    wchar_t **restrict endptr, int base);
uintmax_t wcstouimax (const wchar_t *restrict nptr,
                    wchar_t **restrict endptr, int base);

```

Come si vede, i parametri delle funzioni sono gli stessi; quello che cambia è il tipo di stringa, che nelle funzioni *strto...()* è normale, mentre nelle funzioni *wcsto...()* è di tipo esteso. Nel caso di funzioni *...touimax()* si ottiene un valore intero senza segno, mentre con le funzioni *...toimax()* si ottiene un valore intero con segno.

Il comportamento di queste funzioni è analogo a quello delle altre funzioni *strto...()* e *wcsto...()*, per ciò che riguarda l'interpretazione di valori interi, con la differenza che si fa riferimento al valore intero più grande. Il valore restituito è zero se non si può procedere alla conversione; se invece il valore è al di fuori dell'intervallo rappresentabile, a seconda dei casi si può avere il valore corrispondente a *INTMAX\_MAX*, *INTMAX\_MIN* o *UINTMAX\_MIN*, con l'aggiornamento della variabile *errno* al valore rappresentato da *ERANGE*.

## 69.11 File «iso646.h»

Il file 'iso646.h' della libreria standard definisce alcune macrovariabili da usare in sostituzione di simboli che potrebbero mancare nella propria tastiera, anche se ciò è comunque poco probabile.<sup>5</sup>

Macro	Corrispon- denza	Codice
and	&&	#define and &&
and_eq	&=	#define and_eq &=
bitand	&	#define bitand &
bitor		#define bitor
compl	~	#define compl ~
not	!	#define not !
not_eq	!=	#define not_eq !=
or		#define or
or_eq	=	#define or_eq  =
xor	^	#define xor ^
xor_eq	^=	#define xor_eq ^=

## 69.12 File «stdbool.h»

«

Il file ‘stdbool.h’ della libreria standard definisce alcune macrovariabili da usare per la gestione dei valori logici (*Vero* e *Falso*); in particolare consente di utilizzare il nome *bool* al posto di ‘\_Bool’ (si veda eventualmente la realizzazione di questo file nei sorgenti di os32, listato [95.1.11](#)).<sup>6</sup>.



```
#define bool      _Bool
#define true      1
#define false     0
#define __bool_true_false_are_defined 1
```

Come si può vedere, la macro-variabile `__bool_true_false_are_defined` consente di sapere se le macro-variabili `bool`, `true` e `false`, sono definite.

## 69.13 File «stddef.h»

Il file ‘`stddef.h`’ della libreria standard definisce alcuni tipi di dati e delle macro fondamentali (si veda eventualmente la realizzazione del file ‘`stddef.h`’ nei sorgenti di `os32`, listato [95.1.12](#)).<sup>7</sup>

```
typedef long int          ptrdiff_t;
typedef unsigned long int size_t;
typedef unsigned int      wchar_t;

#define NULL              0
#define offsetof(TYPE, MEMBER) \
    ((size_t) &((TYPE *)0)->MEMBER)
```

Di tutte le definizioni merita attenzione la macroistruzione ‘`offsetof`’ che serve a misurare lo scostamento di un membro di una struttura, per la quale è il caso di scomporre i suoi componenti:

- l’espressione ‘`((tipo_struttura *)0)`’ rappresenta un puntatore nullo trasformato, con un cast, in un puntatore nullo al tipo di struttura alla quale si sta facendo riferimento;
- l’espressione ‘`((tipo_struttura *)0)->nome_membro`’ rappresenta il contenuto del membro indicato, preso a partire dall’indirizzo zero;

- l'espressione `'& ( (tipo_struttura *) 0) ->nome_membro'` rappresenta l'indirizzo del membro indicato, preso a partire dall'indirizzo zero.

Pertanto, l'indirizzo del membro, relativo all'indirizzo zero, corrisponde anche al suo scostamento a partire dall'inizio della struttura. Così, tale valore viene convertito con un cast nel tipo `'size_t'`.

## 69.14 File «string.h»

«

Il file `'string.h'` della libreria standard definisce il tipo `'size_t'`, la macro-variabile `NULL` (come dal file `'stddef.h'`, descritto nella sezione [69.13](#)) e una serie di funzioni per il trattamento delle stringhe o comunque di sequenze di caratteri (si veda eventualmente la realizzazione del file `'string.h'` e di alcune delle sue funzioni nei sorgenti di `os32`, sezione [95.20](#))).

### 69.14.1 Copia

«

Seguono i prototipi delle funzioni disponibili per la copia:

```
void *memcpy (void *restrict dst,
              const void *restrict org, size_t n);
void *memmove (void *dst, const void *org, size_t n);
char *strcpy (char *restrict dst,
              const char *restrict org);
char *strncpy (char *restrict dst,
              const char *restrict org, size_t n);
```

POSIX

Lo standard POSIX aggiunge anche i prototipi seguenti:

```
void *memccpy (void *restrict dst, const void *restrict org,
              int c, size_t n);
char *strdup (const char *org);
```

## 69.14.1.1 Funzione «memcpy()» (memory copy)

La funzione *memcpy()* copia *n* caratteri a partire dall'indirizzo indicato da *org*, per riprodurli a partire dall'indirizzo *dst*, alla condizione che i due insiemi non risultino sovrapposti. La funzione restituisce l'indirizzo *dst*.

```
#include <string.h>
void *
memcpy (void *restrict dst, const void *restrict org,
        size_t n)
{
    unsigned char *d = (unsigned char *) dst;
    unsigned char *o = (unsigned char *) org;
    size_t i;
    for (i = 0; n > 0 && i < n; i++)
        {
            d[i] = o[i];
        }
    return dst;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove la variabile *y* viene sovrascritta dal contenuto di *x*, ma questo attraverso la copia dei byte (si intende che gli interi siano da 32 bit).

```
#include <stdio.h>
#include <string.h>

int
main (void)
{
    int    x = 0x12345678;
    int    y = 0xFFFFFFFF;
```

```

printf ("prima: %x\n", y);
memcpy (&y, &x, sizeof (int));
printf ("dopo:  %x\n", y);
return 0;
}

```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```

prima: ffffffff
dopo:  12345678

```

### 69.14.1.2 Funzione «memccpy()»

«

La funzione *memccpy()* appartiene allo standard POSIX e si distingue rispetto a *memcpy()* perché la copia termina al raggiungimento di un certo carattere (il parametro *c*), restituendo il puntatore alla posizione successiva nella destinazione.

POSIX

```

#include <string.h>
void *
memccpy (void *restrict dst, const void *restrict org,
         int c, size_t n)
{
    unsigned char *d = (unsigned char *) dst;
    unsigned char *o = (unsigned char *) org;
    size_t i;
    for (i = 0; n > 0 && i < n; i++)
        {
            d[i] = o[i];
            if (o[i] == (unsigned char) c)
                {
                    return (dst + i + 1);
                }
        }
}

```

```
    }  
    return NULL;  
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove l'array *y* viene sovrascritto dal contenuto di *x*, fino a quando si raggiunge il carattere '7'. Nell'esempio vengono anche visualizzati i valori dei puntatori di *y* e della posizione raggiunta all'interno di *y* alla fine della copia, sempre in forma di puntatore.

```
#include <stdio.h>  
#include <string.h>  
#include <inttypes.h>  
  
int  
main (void)  
{  
    char x[11] = { '0', '1', '2', '3', '4', '5',  
                  '6', '7', '8', '9', '\0' };  
    char y[11] = { 'a', 'b', 'c', 'd', 'e', 'f',  
                  'g', 'h', 'i', 'j', '\0' };  
  
    char *z;  
    printf ("prima: \"%s\" %" PRIuPTR "\n", y, (uintptr_t) y);  
    z = memccpy (y, x, (int) '7', (size_t) 11);  
    printf ("dopo:  \"%s\" %" PRIuPTR "\n", y, (uintptr_t) z);  
    return 0;  
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima: "abcdefghij" 3218609914
```

dopo: "01234567ij" 3218609922

Come si può vedere dal risultato, l'array *y* inizia a partire dal puntatore 3218609914 e, alla fine del trasferimento parziale dall'array *x*, che si ferma al carattere '7', la funzione restituisce il puntatore 3218609922 che individua la posizione successiva al carattere copiato, corrispondente in pratica al carattere 'i'.

### 69.14.1.3 Funzione «memmove()» (memory move)

«

La funzione *memmove()* opera in modo simile a *memcpy()*, con la differenza che le due aree di memoria coinvolte possono sovrapporsi. In pratica la copia avviene prima in un'area temporanea, quindi, dall'area temporanea viene ricopiata nella destinazione. La funzione restituisce l'indirizzo *dst*.

```
#include <string.h>
void *
memmove (void *dst, const void *org, size_t n)
{
    char temp[n];
    unsigned char *d = (unsigned char *) dst;
    unsigned char *o = (unsigned char *) org;
    size_t i;
    for (i = 0; i < n; i++)
        {
            temp[i] = o[i];
        }
    for (i = 0; n > 0 && i < n; i++)
        {
            d[i] = temp[i];
        }
    return dst;
}
```

Per osservare il comportamento della funzione si può riutilizzare lo stesso programma usato per *memcpy()*, con la modifica del nome della funzione chiamata. Il risultato atteso è lo stesso:

```
...
printf ("prima: %x\n", y);
memmove (&y, &x, sizeof (int));
printf ("dopo:  %x\n", y);
...
```

#### 69.14.1.4 Funzione «strcpy()» (string copy)

La funzione *strcpy()* copia la stringa *org* nell'array a cui punta *dst*, includendo anche il carattere zero di conclusione delle stringhe, alla condizione che le due stringhe non si sovrappongano. La funzione restituisce *dst*.

```
#include <string.h>
char *
strcpy (char *restrict dst, const char *restrict org)
{
    size_t i;
    for (i = 0; org[i] != 0; i++)
        {
            dst[i] = org[i];
        }
    dst[i] = 0;
    return dst;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove l'array *y* viene sovrascritto dal contenuto di *x*.

```
#include <stdio.h>
#include <string.h>
int main (void)
{
    char x[] = "abcdefghijklmnopqrstuvwxyz";
    char y[50] = "ciao";
    printf ("prima: %s\n", y);
    strcpy (y, x);
    printf ("dopo:  %s\n", y);
    return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima: ciao
dopo:  abcdefghijklmnopqrstuvwxyz
```

### 69.14.1.5 Funzione «strncpy()»

«

La funzione *strncpy()* agisce in modo analogo a quello di ‘*strcpy*’, con la differenza che la copia riguarda al massimo i primi *n* caratteri, includendo in questo anche il carattere nullo di terminazione delle stringhe. Se però la stringa *org* è più breve (in quanto si incontra il carattere di terminazione prima di *n* caratteri), i caratteri rimasti vengono copiati con un valore a zero nella destinazione. La funzione restituisce *dst*.

```
#include <string.h>
char *
strncpy (char *restrict dst, const char *restrict org,
        size_t n)
{
    size_t i;
```



```

for (i = 0; n > 0 && i < n && org[i] != 0; i++)
{
    dst[i] = org[i];
}
for ( ; n > 0 && i < n; i++)
{
    dst[i] = 0;
}
return dst;
}

```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove l'array *y* viene sovrascritto dal contenuto di *x*.

```

#include <stdio.h>
#include <string.h>
int main (void)
{
    char x[] = "abcdefghijklmnopqrstuvwxyz";
    char y[50] = "ciaociaociaociaociaociaociaociao";
    printf ("prima:   %s\n", y);
    strncpy (y, x, 10);
    printf ("durante: %s\n", y);
    strncpy (y, x, 27);
    printf ("dopo:    %s\n", y);
    return 0;
}

```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```

prima:   ciaociaociaociaociaociaociaociao
durante: abcdefghijaociaociaociaociaociao

```

dopo:        abcdefghijklmnopqrstuvwxyz

## 69.14.1.6 Funzione «strdup()»

«

La funzione *strdup()*, richiesta dallo standard POSIX, è simile a *strcpy()*, con la differenza che richiede solo l'indicazione della stringa da duplicare, mentre alloca autonomamente la memoria per produrre una copia. Pertanto, la funzione restituisce il puntatore alla stringa duplicata (e allocata) ed è poi possibile liberare la memoria attraverso la funzione *free()*.

POSIX

```
#include <string.h>
#include <stdlib.h>
#include <stdint.h>

char *
strdup (const char *org)
{
    char *d;
    int i;
    size_t size;

    for (i = 0; i < (SIZE_MAX - 1); i++)
    {
        if (org[i] == '\\0')
        {
            break;
        }
    }

    if (i == (SIZE_MAX - 1))
    {
        return NULL;
    }
}
```

```
size = i + 1;

d = malloc (size);

if (d == NULL)
{
    return NULL;
}

strcpy (d, org);

return d;
}
```

Nell'esempio proposto, si riutilizza la funzione *strcpy()* e la funzione *malloc()* per allocare la memoria necessaria. Viene anche usata la macro-variabile *SIZE\_MAX*, per dare un limite massimo alla scansione, nel caso la stringa di origine non contenga il carattere nullo di terminazione. Per questa ragione, diventa necessario includere i file 'stdint.h' e 'stdlib.h'.

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove l'array *o* viene copiato altrove, associandogli il puntatore *d*: se l'operazione ha successo, viene visualizzata la stringa a cui punta *d*, altrimenti si ottiene un messaggio di errore. Dovendo usare la funzione *free()* per liberare la memoria, si include anche il file 'stdlib.h'.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
int
main (void)
{
    char *d;
    char o[] = "abcdefghijklmnopqrstuvwxyz";

    d = strdup (o);

    if (d == NULL)
    {
        printf ("Non è possibile duplicare la stringa!\n");
    }
    else
    {
        printf ("%s\n", d);
        free (d);
    }
    return 0;
}
```

## 69.14.2 Concatenamento



Seguono i prototipi delle funzioni per il concatenamento:

```
char *strcat (char *restrict dst,
              const char *restrict org);
char *strncat (char *restrict dst,
              const char *restrict org, size_t n);
```

## 69.14.2.1 Funzione «strcat()» (string cat)

La funzione *strcat()* copia la stringa *org* a partire dalla fine della stringa *dst* (sovrascrivendo il carattere nullo preesistente), alla condizione che le due stringhe non siano sovrapposte. La funzione restituisce *dst*.

```
#include <string.h>
char *
strcat (char *restrict dst, const char *restrict org)
{
    size_t i;
    size_t j;
    for (i = 0; dst[i] != 0; i++)
        {
            ; // Si limita a cercare il carattere nullo.
        }
    for (j = 0; org[j] != 0; i++, j++)
        {
            dst[i] = org[j];
        }
    dst[i] = 0;
    return dst;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove la stringa *y* viene estesa con il contenuto di *x*.

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
    char x[] = "abcdefghijklmnopqrstuvwxyz";
```

```
char y[50] = "ciao";
printf ("prima: %s\n", y);
strcat (y, x);
printf ("dopo: %s\n", y);
return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima: ciao
dopo: ciaoabcdefghijklmnopqrstuvwxyz
```

### 69.14.2.2 Funzione «strncat()»

«

La funzione *strncat()* si comporta in modo analogo a *strcat()*, con la differenza che copia al massimo *n* caratteri, ammesso che la stringa *org* ne contenga abbastanza. In ogni caso, la stringa *dst* viene completata con il carattere nullo di terminazione.

```
#include <string.h>
char *
strncat (char *restrict dst, const char *restrict org,
         size_t n)
{
    size_t i;
    size_t j;
    for (i = 0; n > 0 && dst[i] != 0; i++)
        {
            ; // Si limita a cercare il carattere nullo.
        }
    for (j = 0; n > 0 && j < n && org[j] != 0; i++, j++)
        {
            dst[i] = org[j];
        }
}
```

```
    }  
    dst[i] = 0;  
    return dst;  
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove la stringa **y** viene estesa con il contenuto di **x**, in due fasi.

```
#include <stdio.h>  
#include <string.h>  
int main (void)  
{  
    char x[] = "abcdefghijklmnopqrstuvwxyz";  
    char y[50] = "ciao";  
    printf ("prima: %s\n", y);  
    strncat (y, x, 10);  
    printf ("durante: %s\n", y);  
    strncat (y, x, 40);  
    printf ("dopo: %s\n", y);  
    return 0;  
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima: ciao  
durante: ciaoabcdefghij  
dopo: ciaoabcdefghijklmnopqrstuvwxyz
```

### 69.14.3 Comparazione

«

Le funzioni di comparazione *memcmp()*, *strcmp()* e *strncmp()* confrontano due sequenze di caratteri, determinando se la prima sia maggiore, minore o uguale rispetto alla seconda, scandendo i caratteri progressivamente e arrestando l'analisi appena si incontra una differenza. Pertanto, il carattere che differisce è quello che determina l'ordine tra le due sequenze.

```
int      memcmp  (const void *s1, const void *s2, size_t n);
int      strcmp  (const char *s1, const char *s2);
int      strcoll (const char *s1, const char *s2);
int      strncmp (const char *s1, const char *s2, size_t n);
size_t  strxfrm (char *restrict dst,
                const char *restrict org, size_t n);
```

#### 69.14.3.1 Funzione «memcmp()» (memory compare)

«

La funzione *memcmp()* confronta i primi *n* caratteri delle aree di memoria a cui puntano *s1* e *s2*, restituendo: un valore pari a zero se le due sequenze si equivalgono; un valore maggiore di zero se la sequenza di *s1* è maggiore di *s2*; un valore minore di zero se la sequenza di *s1* è minore di *s2*.

```
#include <string.h>
int
memcmp (const void *s1, const void *s2, size_t n)
{
    unsigned char *a = (unsigned char *) s1;
    unsigned char *b = (unsigned char *) s2;
    size_t i;
    for (i = 0; n > 0 && i < n; i++)
    {
        if (a[i] > b[i])
```



```
        {
            return 1;
        }
    else if (a[i] < b[i])
        {
            return -1;
        }
    }
    return 0;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove le variabili  $x$  e  $y$  sono interi (che si presume siano a 32 bit) rappresentati in memoria invertendo l'ordine dei byte (*little endian*), pertanto il confronto avviene in modo inverso all'apparenza dei simboli.

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
    unsigned int  x = 0x123456FF;
    unsigned int  y = 0xEEEEEEEE;
    int  r;
    r = memcmp (&x, &y, sizeof (int));
    printf ("memcmp: %x %i %x\n", x, r, y);
    r = memcmp (&x, &x, sizeof (int));
    printf ("memcmp: %x %i %x\n", x, r, x);
    r = memcmp (&y, &x, sizeof (int));
    printf ("memcmp: %x %i %x\n", y, r, x);
    return 0;
}
```



```
    }  
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente:

```
#include <stdio.h>  
#include <string.h>  
int  
main (void)  
{  
    char x[] = "ciao";  
    char y[] = "ciao amore";  
    int r;  
    r = strcmp (x, y);  
    printf ("strcmp: %s %i %s\n", x, r, y);  
    r = strcmp (x, x);  
    printf ("strcmp: %s %i %s\n", x, r, x);  
    r = strcmp (y, x);  
    printf ("strcmp: %s %i %s\n", y, r, x);  
    return 0;  
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
strcmp: ciao -1 ciao amore  
strcmp: ciao 0 ciao  
strcmp: ciao amore 1 ciao
```

### 69.14.3.3 Funzione «strcoll()» (string collate compare)



La funzione *strcoll()* è analoga a *strcmp()*, con la differenza che la comparazione avviene sulla base della configurazione locale (la categoria ‘**LC\_COLLATE**’). Nel caso della configurazione locale ‘**C**’ la funzione si comporta esattamente come *strcmp()*.

### 69.14.3.4 Funzione «strncmp()»



La funzione *strncmp()* si comporta in modo analogo a *strcmp()*, con la differenza che la comparazione si arresta al massimo dopo *n* caratteri.

```
#include <string.h>
int
strncmp (const char *s1, const char *s2, size_t n)
{
    unsigned char *a = (unsigned char *) s1;
    unsigned char *b = (unsigned char *) s2;
    size_t i;
    for (i = 0; i < n ; i++)
    {
        if (a[i] > b[i])
        {
            return 1;
        }
        else if (a[i] < b[i])
        {
            return -1;
        }
        else if (a[i] == 0 && b[i] == 0)
        {
            return 0;
        }
    }
}
```

```
    return 0;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
    char x[] = "CIao";
    char y[] = "CIAO";
    int r;
    r = strncmp (x, y, 4);
    printf ("strncmp: %i %s %i %s\n", 4, x, r, y);
    r = strncmp (x, y, 2);
    printf ("strncmp: %i %s %i %s\n", 2, x, r, x);
    r = strncmp (y, x, 4);
    printf ("strncmp: %i %s %i %s\n", 4, y, r, x);
    return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
strncmp: 4 CIao 1 CIAO
strncmp: 2 CIao 0 CIao
strncmp: 4 CIAO -1 CIao
```

### 69.14.3.5 Funzione «strxfrm()» (string transform)



La funzione *strxfrm()* trasforma la stringa *org* sovrascrivendo la stringa *dst* in modo relativo alla configurazione locale. In pratica, la stringa trasformata che si ottiene può essere comparata con un'altra stringa trasformata nello stesso modo attraverso la funzione *strcmp()* ottenendo lo stesso esito che si avrebbe confrontando le stringhe originali con la funzione *strcmp()*.

Nella stringa di destinazione vengono messi non più di *n* caratteri, incluso il carattere nullo di terminazione. Se *n* è pari a zero, *dst* può essere un puntatore nullo. Le due stringhe non devono sovrapporsi.

La funzione *strxfrm()* restituisce la quantità di caratteri necessari a contenere la stringa *org* trasformata, senza però contare il carattere nullo di terminazione. Se *n* è zero e *dst* corrisponde al puntatore nullo, restituisce il valore che sarebbe necessario per trasformare la stringa *org* in tutta la sua lunghezza.

L'esempio seguente di tale funzione è valido solo per la configurazione locale 'C':

```
#include <string.h>
size_t
strxfrm (char *restrict dst, const char *restrict org,
         size_t n)
{
    size_t i;
    if (n == 0 && dst == NULL)
        {
            return strlen (org);
        }
    else
        {
```

```
    for (i = 0; i < n ; i++)
        {
            dst[i] = org[i];
            if (org[i] == 0)
                {
                    break;
                }
        }
    return i;
}
```

## 69.14.4 Ricerca

Seguono i prototipi delle funzioni utili per la ricerca all'interno di sequenze di byte, secondo lo standard C: «

```
void *memchr (const void *s, int c, size_t n);
char *strchr (const char *s, int c);
char *strrchr (const char *s, int c);
size_t strspn (const char *s, const char *accept);
size_t strcspn (const char *s, const char *reject);
char *strpbrk (const char *s, const char *accept);
char *strstr (const char *string, const char *substring);
char *strtok (char *restrict string,
              const char *restrict delim);
```

Lo standard POSIX aggiunge anche il prototipo seguente:

```
char *strtok_r (char *restrict string,
               const char *restrict delim,
               char **saveptr);
```

### 69.14.4.1 Funzione «memchr()» (memory character)



La funzione *memchr()* cerca un carattere a partire da una certa posizione in memoria, scandendo al massimo una quantità determinata di caratteri, restituendo il puntatore al carattere trovato. Se nell'ambito specificato non trova il carattere, restituisce il puntatore nullo.

```
#include <string.h>
void *
memchr (const void *s, int c, size_t n)
{
    unsigned char *a = (unsigned char *) s;
    unsigned char x = (unsigned char) c;
    size_t i;
    for (i = 0; n > 0 && i < n; i++)
        {
            if (a[i] == x)
                {
                    return (void *) (s + i);
                }
        }
    return NULL;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, in cui si scandisce il contenuto di una variabile di tipo 'int', intendendo che questa debba occupare uno spazio di 32 bit:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
```



```

int x = 0x13579BDF;
void *p;
p = memchr (&x, 0xDF, 4);
printf ("contenuto della variabile: %x\n", x);
printf ("indirizzo iniziale della variabile: 0x%x\n",
        (unsigned int) &x);
printf ("indirizzo di 0x%x all'interno della "
        "variabile: 0x%x\n",
        0xDF,
        (unsigned int) p);
return 0;
}

```

Avviando questo programma in un'architettura che inverte l'ordine dei byte (*little endian*) si deve ottenere un risultato simile a quello seguente:

```

contenuto della variabile: 13579bdf
indirizzo iniziale della variabile: 0xbff7f3bc
indirizzo di 0xdf all'interno della variabile: 0xbff7f3bc

```

## 69.14.4.2 Funzione «strchr()» (string character)

La funzione *strchr()* cerca un carattere all'interno di una stringa, restituendo il puntatore al carattere trovato, oppure il puntatore nullo se la ricerca fallisce. Nella scansione viene preso in considerazione anche il carattere nullo di terminazione della stringa.

```

#include <string.h>
char *
strchr (const char *s, int c)
{
    unsigned char *a = (unsigned char *) s;
    unsigned char x = (unsigned char) c;

```

```
size_t i;
for (i = 0; ; i++)
{
    if (a[i] == x)
    {
        return (char *) (s + i);
    }
    else if (a[i] == 0)
    {
        return NULL;
    }
}
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
    char *x = "ciao amore mio";
    char *p;

    p = strchr (x, 'a');
    printf ("La stringa \"%s\", collocata a partire ", x);
    printf ("dall'indirizzo %u, contiene il carattere '%c' ",
            (unsigned int) x, 'a');
    printf ("all'indirizzo %u.\n", (unsigned int) p);

    p = strchr (x, 0);
    printf ("La stringa \"%s\", collocata a partire ", x);
    printf ("dall'indirizzo %u, contiene il carattere 0x%x ",
            (unsigned int) x, 0);
}
```

```

printf ("all'indirizzo %u.\n", (unsigned int) p);

return 0;
}

```

Avviando questo programma si deve ottenere un risultato simile a quello seguente:

La stringa "ciao amore mio", collocata a partire ←  
 ↪dall'indirizzo 134516936, contiene il carattere ←  
 ↪'a' all'indirizzo 134516938.  
 La stringa "ciao amore mio", collocata a partire ←  
 ↪dall'indirizzo 134516936, contiene il carattere ←  
 ↪0x0 all'indirizzo 134516950.

### 69.14.4.3 Funzione «strrchr()» (string character)

La funzione *strrchr()* cerca un carattere all'interno di una stringa, restituendo il puntatore all'ultimo carattere corrispondente trovato, oppure il puntatore nullo se la ricerca fallisce. Nella scansione viene preso in considerazione anche il carattere nullo di terminazione della stringa. «

```

#include <string.h>
char *
strrchr (const char *string, int c)
{
    int i;
    //
    for (i = strlen (string); i >= 0; i--)
    {
        if (string[i] == (char) c)
        {
            break;
        }
    }
}

```

```
    }  
    //  
    if (i < 0)  
    {  
        return NULL;  
    }  
    else  
    {  
        return (string + i);  
    }  
}
```

Per verificare sommariamente il comportamento della funzione si può modificare leggermente l'esempio già apparso a proposito della funzione *strchr()*:

```
...  
    p = strrchr (x, 'a');  
...  
    p = strrchr (x, 0);  
...
```

Avviando questo programma si deve ottenere un risultato simile a quello seguente:

La stringa "ciao amore mio", collocata a partire ←  
↳ dall'indirizzo 134514088, contiene il carattere ←  
↳ 'a' all'indirizzo 134514093.

La stringa "ciao amore mio", collocata a partire ←  
↳ dall'indirizzo 134514088, contiene il carattere ←  
↳ 0x0 all'indirizzo 134514102.

## 69.14.4.4 Funzione «strspn()» (string span)

La funzione *strspn()* calcola la lunghezza massima iniziale della stringa *s*, composta esclusivamente da caratteri contenuti nella stringa *accept*, restituendo tale valore.

```
#include <string.h>
size_t
strspn (const char *s, const char *accept)
{
    size_t i;
    size_t j;
    int found;
    for (i = 0; s[i] != 0; i++)
        {
            for (j = 0, found = 0; accept[j] != 0; j++)
                {
                    if (s[i] == accept[j])
                        {
                            found = 1;
                            break;
                        }
                }
            if (!found)
                {
                    break;
                }
        }
    return i;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
```

```
#include <string.h>
int
main (void)
{
    char *x = "ciao amore mio";
    char *y = "abcdefghi";
    size_t n;

    n = strspn (x, y);
    printf ("La parte iniziale di \"%s\" ", x);
    printf ("che contiene i caratteri \"%s\" ", y);
    printf ("si compone di %i caratteri.\n", n);

    n = strspn (x, x);
    printf ("La parte iniziale di \"%s\" ", x);
    printf ("che contiene i caratteri \"%s\" ", x);
    printf ("si compone di %i caratteri.\n", n);

    return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

La parte iniziale di "ciao amore mio" che contiene i ↵  
↵caratteri "abcdefghi" si compone di 3 caratteri.

La parte iniziale di "ciao amore mio" che contiene i ↵  
↵caratteri "ciao amore mio" si compone di 14 caratteri.

#### 69.14.4.5 Funzione «strcspn()»

«

La funzione *strcspn()* si comporta in modo analogo a *strspn()*, con la differenza che l'insieme di caratteri contenuto nella stringa 'reject' non deve costituire l'insieme iniziale della stringa *s* che

si va a contare. Pertanto, il valore restituito è la quantità di caratteri iniziali della stringa *s* che non si trovano anche nell'insieme *reject*.

```
#include <string.h>
size_t
strcspn (const char *s, const char *reject)
{
    size_t i;
    size_t j;
    int found;
    for (i = 0; s[i] != 0; i++)
        {
            for (j = 0, found = 0; reject[j] != 0 || found; j++)
                {
                    if (s[i] == reject[j])
                        {
                            found = 1;
                            break;
                        }
                }
            if (found)
                {
                    break;
                }
        }
    return i;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
```

```

{
    char *x = "ciao amore mio";
    char *y = "mnopqrstuvwxyz";
    size_t n;

    n = strcspn (x, y);
    printf ("La parte iniziale di \"%s\" ", x);
    printf ("che non contiene i caratteri \"%s\" ", y);
    printf ("si compone di %i caratteri.\n", n);

    n = strcspn (x, x);
    printf ("La parte iniziale di \"%s\" ", x);
    printf ("che non contiene i caratteri \"%s\" ", x);
    printf ("si compone di %i caratteri.\n", n);

    return 0;
}

```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```

La parte iniziale di "ciao amore mio" che non contiene i ↵
↳ caratteri "mnopqrstuvwxyz" si compone di 3 caratteri.
La parte iniziale di "ciao amore mio" che non contiene i ↵
↳ caratteri "ciao amore mio" si compone di 0 caratteri.

```

#### 69.14.4.6 Funzione «strpbrk()» (string point break)

«

La funzione *strpbrk()* scandisce la stringa *s* alla ricerca del primo carattere che risulti contenuto nella stringa *accept*, restituendo il puntatore al carattere trovato, oppure, in mancanza di alcuna corrispondenza, il puntatore nullo.

```

#include <string.h>
char *

```



```
strpbrk (const char *s, const char *accept)
{
    size_t i;
    size_t j;
    for (i = 0; s[i] != 0; i++)
        {
            for (j = 0; accept[j] != 0; j++)
                {
                    if (s[i] == accept[j])
                        {
                            return (char *) (s + i);
                        }
                }
        }
    return NULL;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
    char *x = "ciao amore mio";
    char *y = "mnopqrstuvwxyz";
    char *p;

    p = strpbrk (x, y);
    printf ("La stringa \"%s\" che inizia all'indirizzo %u, ",
           x, (unsigned int) x);
    printf ("trova la prima corrispondenza con la "
           "stringa \"%s\" ", y);
    printf ("all'indirizzo %u.\n", (unsigned int) p);
}
```

```
    return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

La stringa "ciao amore mio" che inizia all'indirizzo ↵  
↵134516840, trova la prima corrispondenza con la ↵  
↵stringa "mnopqrstuvwxyz" all'indirizzo 134516843.

### 69.14.4.7 Funzione «strstr()»

«

La funzione *strstr()* cerca la stringa *substring* nella stringa *string* restituendo il puntatore alla prima corrispondenza trovata (nella stringa *string*). Se la corrispondenza non c'è, la funzione restituisce il puntatore nullo.

```
#include <string.h>
char *
strstr (const char *string, const char *substring)
{
    size_t i;
    size_t j;
    size_t k;
    int found;
    if (substring[0] == 0)
        {
            return (char *) string;
        }
    for (i = 0, j = 0, found = 0; string[i] != 0; i++)
        {
            if (string[i] == substring[0])
                {
```

```

        for (k = i, j = 0;
            string[k] == substring[j] && string[k] != 0
            && substring[j] != 0;
            j++, k++)
        {
            ;
        }
        if (substring[j] == 0)
        {
            found = 1;
        }
    }
    if (found)
    {
        return (char *) (string + i);
    }
}
return NULL;
}

```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```

#include <stdio.h>
#include <string.h>
int
main (void)
{
    char *x = "ciao amore mio";
    char *y = "amore";
    char *p;

    p = strstr (x, y);
    printf ("La stringa \"%s\" che inizia all'indirizzo %u, ",
           x, (unsigned int) x);
}

```

```

printf ("contiene la stringa \"%s\" ", y);
printf ("all'indirizzo %u.\n", (unsigned int) p);

p = strstr (x, "");
printf ("La stringa \"%s\" che inizia all'indirizzo %u, ",
        x, (unsigned int) x);
printf ("contiene la stringa \"%s\" ", "");
printf ("all'indirizzo %u.\n", (unsigned int) p);

p = strstr (x, "baba");
printf ("La stringa \"%s\" che inizia all'indirizzo %u, ",
        x, (unsigned int) x);
printf ("contiene la stringa \"%s\" ", "baba");
printf ("all'indirizzo %u.\n", (unsigned int) p);

return 0;
}

```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```

La stringa "ciao amore mio" che inizia all'indirizzo ←
↪134517000, contiene la stringa "amore" all'indirizzo ←
↪134517005.
La stringa "ciao amore mio" che inizia all'indirizzo ←
↪134517000, contiene la stringa "" all'indirizzo 134517000.
La stringa "ciao amore mio" che inizia all'indirizzo ←
↪134517000, contiene la stringa "baba" all'indirizzo 0.

```

#### 69.14.4.8 Funzione «strtok()» (string token)

«

La funzione *strtok()* serve a suddividere una stringa in unità, definite *token*, specificando un elenco di caratteri da intendere come delimitatori, in una seconda stringa. La funzione va usata in fasi successive, fornendo solo inizialmente la stringa da suddividere che continua poi

a essere utilizzata se al suo posto viene fornito il puntatore nullo. La funzione restituisce, di volta in volta, il puntatore alla sottostringa contenente l'unità individuata, oppure il puntatore nullo, se non può trovarla.

La funzione deve tenere memoria di un puntatore in un'area di memoria persistente (quello che nei commenti viene definito «puntatore statico») e deve isolare le unità modificando la stringa originale, inserendo il carattere nullo di terminazione alla fine delle unità individuate.

Quando la funzione viene chiamata indicando al posto della stringa da scandire il puntatore nullo, l'insieme dei delimitatori può essere diverso da quello usato nelle fasi precedenti.

```
#include <string.h>
char *
strtok (char *restrict string, const char *restrict delim)
{
    static char *next = NULL;
    size_t i = 0;
    size_t j;
    int found_token;
    int found_delim;
    //
    // Se la stringa fornita come argomento è un puntatore
    // nullo, occorre avvalersi del puntatore statico. Se
    // però questo è nullo a sua volta, la scansione non può
    // avvenire.
    //
    if (string == NULL)
    {
        if (next == NULL)
        {
            return NULL;
        }
    }
}
```

```
    }
    else
    {
        string = next;
    }
}
//
// Se la stringa fornita come argomento è vuota, la
// scansione non può avvenire.
//
if (string[0] == 0)
{
    next = NULL;
    return NULL;
}
else
{
    if (delim[0] == 0)
    {
        return string;
    }
}
//
// Trova la prossima unità (token).
//
for (i = 0, found_token = 0, j = 0;
     string[i] != 0 && (!found_token);
     i++)
{
    //
    // Cerca tra i delimitatori.
    //
    for (j = 0, found_delim = 0; delim[j] != 0; j++)
    {
```

```
        if (string[i] == delim[j])
            {
                found_delim = 1;
            }
    }
    //
    // Se il carattere attuale della stringa non è
    // un delimitatore, si tratta dell'inizio di una
    // nuova unità (token).
    //
    if (!found_delim)
        {
            found_token = 1;
            break;
        }
    }
    //
    // Se è stata trovata una unità (token) viene aggiustato
    // il puntatore che rappresenta la stringa. Se invece
    // non è stata trovata l'unità, vuol dire che non ce ne
    // possono essere altre.
    //
    if (found_token)
        {
            string += i;
        }
    else
        {
            next = NULL;
            return NULL;
        }
    //
    // Cerca la fine dell'unità trovata.
    //
```

```
for (i = 0, found_delim = 0; string[i] != 0; i++)
{
    for (j = 0; delim[j] != 0; j++)
    {
        if (string[i] == delim[j])
        {
            found_delim = 1;
            break;
        }
    }
    if (found_delim)
    {
        break;
    }
}
//
// Se è stato trovato un delimitatore, allora il carattere
// corrispondente nella stringa deve essere azzerato.
// Se invece la stringa originale è terminata per conto
// proprio, allora non è possibile continuare la ricerca
// in una fase successiva, perché non ci possono essere
// altre unità.
//
if (found_delim)
{
    string[i] = 0;
    next = &string[i+1];
}
else
{
    next = NULL;
}
//
// A questo punto, la stringa attuale rappresenta
```



```
// l'unità trovata.  
//  
return string;  
}
```

Per comprendere lo scopo della funzione viene utilizzato lo stesso esempio che appare nel documento *ISO/IEC 9899:TC2*, al paragrafo 7.21.5.7, con qualche piccola modifica per poterlo rendere un programma autonomo:

```
#include <stdio.h>  
#include <string.h>  
int  
main (void)  
{  
    char str[] = "?a???b,,,#c";  
    char *t;  
  
    t = strtok (str, "?");           // t punta all'unità "a"  
    printf ("strtok: \"%s\"\n", t);  
    t = strtok (NULL, ",");         // t punta all'unità "??b"  
    printf ("strtok: \"%s\"\n", t);  
    t = strtok (NULL, "#,");        // t punta all'unità "c"  
    printf ("strtok: \"%s\"\n", t);  
    t = strtok (NULL, "?");         // t è un puntatore nullo  
    printf ("strtok: \"%s\"\n", t);  
  
    return 0;  
}
```

Avviando il programma si ottiene quanto già descritto dai commenti inseriti nel codice:

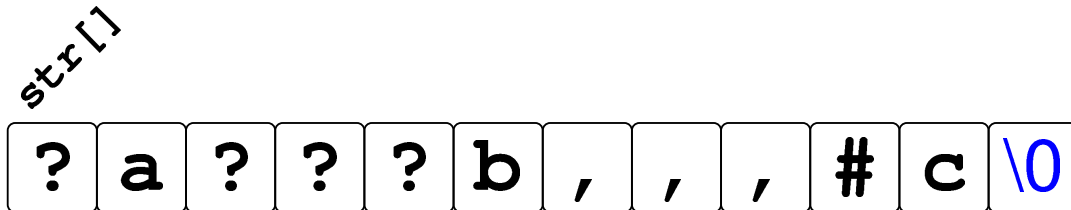
```
strtok: "a"
```

```

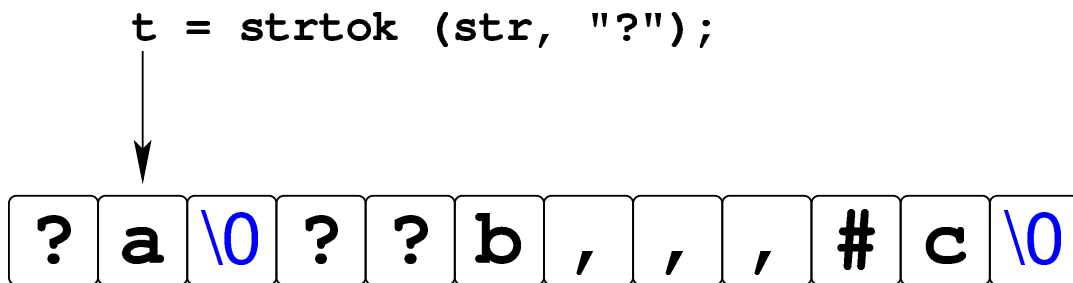
strtok: "??b"
strtok: "c"
strtok: "(null)"

```

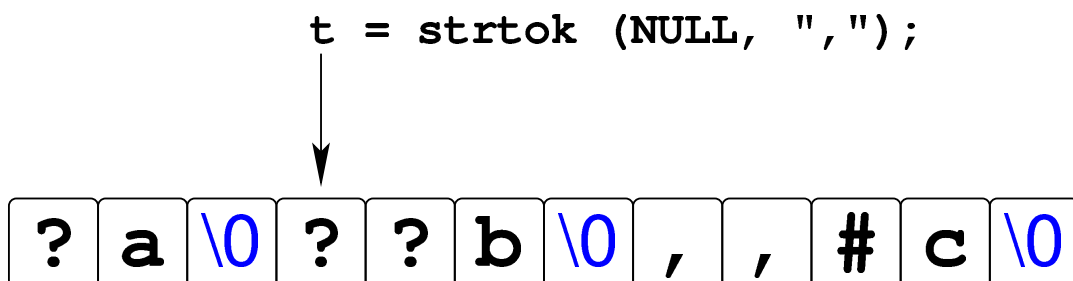
Ciò che avviene nell'esempio può essere schematizzato dalle figure successive. Inizialmente la stringa 'str' ha in memoria l'aspetto seguente:



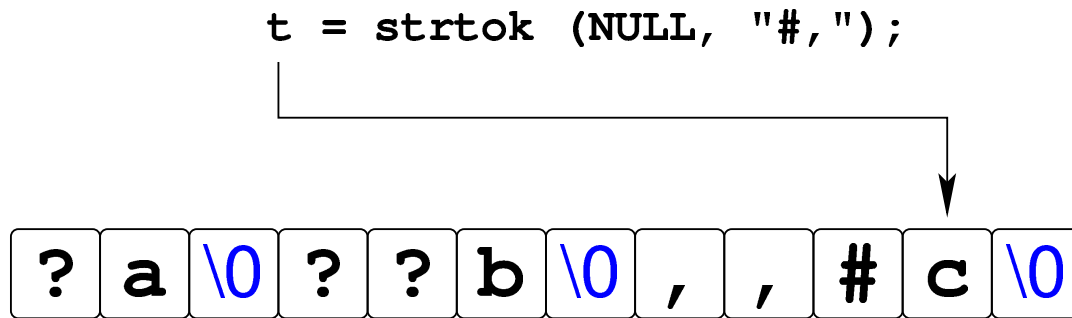
Dopo la prima chiamata della funzione *strtok()* la stringa risulta alterata e il puntatore ottenuto raggiunge la lettera 'a':



Dopo la seconda chiamata della funzione, in cui si usa il puntatore nullo per richiedere una scansione ulteriore della stringa originale, si ottiene un nuovo puntatore che, questa volta, inizia a partire dal quarto carattere, rispetto alla stringa originale, dal momento che il terzo è già stato sovrascritto da un carattere nullo:



La penultima chiamata della funzione *strtok()* raggiunge la lettera ‘c’ che è anche alla fine della stringa originale:



L’ultimo tentativo di chiamata della funzione non può dare alcun esito, perché la stringa originale si è già conclusa.

Va tenuto in considerazione che la funzione *strtok()*, dovendo mantenere in memoria la posizione trovata dell’ultima scansione eseguita, da una chiamata a quella successiva, non è «rientrante», pertanto non si presta per i programmi che si suddividono in più thread.

#### 69.14.4.9 Funzione «strtok\_r()»

La funzione *strtok\_r()*, richiesta dallo standard POSIX, salva il puntatore interno alla stringa che viene scandita, esternamente, in modo da poter essere usata in un contesto in cui più thread operano simultaneamente. In pratica si aggiunge un terzo parametro, costituito da un puntatore a puntatore a carattere.

Il puntatore a carattere (‘**char \***’), il cui puntatore viene fornito come terzo argomento della funzione, deve essere dichiarato prima della chiamata della funzione. La prima volta che viene chiamata la funzione *strtok\_r()* non conta quale valore abbia effettivamente tale

variabile di tipo `'char *'`, perché è la funzione stessa che lo inzializza, ma nelle chiamate successive (quando al posto della stringa si dà alla funzione il valore `'NULL'`). Quando termina la scansione della stringa con le chiamate di `strtok_r()`, la variabile `'char *'` di cui si passa il puntatore, può essere utilizzata per altri scopi, o per altre scansioni.

La soluzione seguente, per la realizzazione della funzione `strtok_r()`, può essere confrontato con quella relativa alla funzione `strtok()`, per comprendere il senso e l'utilizzo dell'ultimo parametro.

```
char *
strtok_r (char *restrict string, const char *restrict delim,
          char **saveptr)
{
    size_t i = 0;
    size_t j;
    int found_token;
    int found_delim;
    //
    // Se la stringa fornita come argomento è un puntatore
    // nullo, occorre avvalersi del puntatore "*saveptr".
    // Se però questo è nullo a sua volta, la scansione non
    // può avvenire.
    //
    if (string == NULL)
    {
        if (*saveptr == NULL)
        {
            return NULL;
        }
        else
        {
            string = *saveptr;
        }
    }
}
```

```
    }  
    //  
    // Se la stringa fornita come argomento è vuota, la  
    // scansione non può avvenire.  
    //  
    if (string[0] == 0)  
    {  
        *saveptr = NULL;  
        return NULL;  
    }  
    else  
    {  
        if (delim[0] == 0)  
        {  
            return string;  
        }  
    }  
    //  
    // Trova la prossima unità (token).  
    //  
    for (i = 0, found_token = 0, j = 0;  
        string[i] != 0 && (!found_token);  
        i++)  
    {  
        //  
        // Cerca tra i delimitatori.  
        //  
        for (j = 0, found_delim = 0; delim[j] != 0; j++)  
        {  
            if (string[i] == delim[j])  
            {  
                found_delim = 1;  
            }  
        }  
    }
```

```
//  
// Se il carattere attuale della stringa non è  
// un delimitatore, si tratta dell'inizio di una  
// nuova unità (token).  
//  
if (!found_delim)  
    {  
        found_token = 1;  
        break;  
    }  
}  
//  
// Se è stata trovata una unità (token) viene aggiustato  
// il puntatore che rappresenta la stringa. Se invece  
// non è stata trovata l'unità, vuol dire che non ce ne  
// possono essere altre.  
//  
if (found_token)  
    {  
        string += i;  
    }  
else  
    {  
        *saveptr = NULL;  
        return NULL;  
    }  
//  
// Cerca la fine dell'unità trovata.  
//  
for (i = 0, found_delim = 0; string[i] != 0; i++)  
    {  
        for (j = 0; delim[j] != 0; j++)  
            {  
                if (string[i] == delim[j])
```

```
        {
            found_delim = 1;
            break;
        }
    }
    if (found_delim)
    {
        break;
    }
}
//
// Se è stato trovato un delimitatore, allora il carattere
// corrispondente nella stringa deve essere azzerato.
// Se invece la stringa originale è terminata per conto
// proprio, allora non è possibile continuare la ricerca
// in una fase successiva, perché non ci possono essere
// altre unità.
//
if (found_delim)
{
    string[i] = 0;
    *saveptr = &string[i+1];
}
else
{
    *saveptr = NULL;
}
//
// A questo punto, la stringa attuale rappresenta
// l'unità trovata.
//
return string;
}
```

Per dimostrare il lavoro della funzione, viene utilizzato lo stesso esempio già usato a proposito di *strtok()*, con poche piccole modifiche:

```
int
main (void)
{
    char str[] = "?a???b,,,#c";
    char *t;
    char *save;

    t = strtok_r (str, "?", &save);    // t punta all'unità "a"
    printf ("strtok: \"%s\"\n", t);
    t = strtok_r (NULL, ",", &save);  // t punta all'unità "??b"
    printf ("strtok: \"%s\"\n", t);
    t = strtok_r (NULL, "#", &save);  // t punta all'unità "c"
    printf ("strtok: \"%s\"\n", t);
    t = strtok_r (NULL, "?", &save);  // t è un puntatore nullo
    printf ("strtok: \"%s\"\n", t);

    return 0;
}
```

Avviando il programma si ottiene esattamente la stessa cosa dell'esempio già visto:

```
strtok: "a"
strtok: "??b"
strtok: "c"
strtok: "(null)"
```



## 69.14.5 Funzioni varie

Seguono i prototipi delle funzioni descritte nelle sezioni successive. Questi appartengono allo standard C:

```
void *memset      (void *s, int c, size_t n);
char *strerror    (int errnum);
size_t strlen     (const char *s);
```

Il prototipo successivo viene aggiunto dallo standard POSIX:

```
int  strerror_r  (int errnum, char *s, size_t n);
```

POSIX

### 69.14.5.1 Funzione «memset()» (memory set)

La funzione *memset()* consente di inizializzare una certa area di memoria con la ripetizione di un certo carattere. Per la precisione, viene usato il valore del parametro *c*, tradotto in un carattere senza segno, copiandolo per *n* volte a partire dall'indirizzo a cui punta *s*. La funzione restituisce *s*.

```
#include <string.h>
void *
memset (void *s, int c, size_t n)
{
    unsigned char *a = (unsigned char *) s;
    unsigned char x  = (unsigned char) c;
    size_t i;
    for (i = 0; n > 0 && i < n; i++)
        {
            a[i] = x;
        }
    return s;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
    int x = 0x12345678;
    printf ("prima: 0x%x\n", x);
    memset (&x, 0xFF, 2);
    printf ("dopo: 0x%x\n", x);

    char X[] = "ciao amore mio";
    printf ("prima: \"%s\"\n", X);
    memset (X, 'Q', 5);
    printf ("dopo: \"%s\"\n", X);

    return 0;
}
```

Avviando questo programma in un elaboratore con architettura a 32 bit e inversione dei byte (*little endian*) si deve ottenere il risultato seguente:

```
prima: 0x12345678
dopo: 0x1234ffff
prima: "ciao amore mio"
dopo: "QQQQQamore mio"
```

## 69.14.5.2 Funzione «strerror()» (string error)

La funzione *strerror()* serve a tradurre il numero fornito come argomento in un puntatore da cui inizia una stringa contenente una spiegazione. In altri termini, serve a trasformare un numero in una descrizione di un tipo di errore. Qui viene mostrata una soluzione priva di utilità, anche se risponde alle richieste delle specifiche:

```
#include <string.h>
char *
strerror (int errnum)
{
    static char answare[] = "Unknown error";
    return answare;
}
```

L'esempio successivo può servire a dimostrare il senso di questa funzione:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
    printf ("%s\n", strerror (0));
    printf ("%s\n", strerror (1));
    printf ("%s\n", strerror (2));
    printf ("%s\n", strerror (3));
    printf ("%s\n", strerror (4));
    return 0;
}
```

Utilizzando questo programma compilato con le librerie di un sistema GNU si potrebbero vedere i messaggi seguenti:

```
Success
Operation not permitted
No such file or directory
No such process
Interrupted system call
```

La stringa a cui punta la funzione può essere condivisa da altre chiamate successive della stessa, pertanto, in un programma con thread multipli, è possibile che avvenga la sovrascrittura, a meno di disporre di un elenco separato di tutti i tipi di messaggio di errore.

### 69.14.5.3 Funzione «`strerror_r()`» (string error)

«

La funzione *strerror\_r()* viene aggiunta dallo standard POSIX e consente di tradurre un errore numerico in stringa, fornendo alla funzione il puntatore iniziale della stringa da produrre e la lunghezza massima che questa può raggiungere, garantendo l'indipendenza tra thread multipli.

POSIX

Per quanto riguarda l'utilizzo, la differenza fondamentale rispetto a *strerror()* sta nel fatto che restituisce un valore intero, pari a zero, se l'operazione ha avuto successo, oppure  $-1$  in caso di problemi, aggiornando di conseguenza anche la variabile *errno*. L'esempio successivo può servire a dimostrare il senso di questa funzione:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
    char msg[100];

    if (!strerror_r (1, msg, 100))
```

```
{
    printf ("%s\n", msg);
}
if (!strerror_r (2, msg, 100))
{
    printf ("%s\n", msg);
}
if (!strerror_r (3, msg, 100))
{
    printf ("%s\n", msg);
}
if (!strerror_r (4, msg, 100))
{
    printf ("%s\n", msg);
}
return 0;
}
```

Utilizzando questo programma compilato con le librerie di un sistema GNU si potrebbero vedere i messaggi seguenti:

```
Success
Operation not permitted
No such file or directory
No such process
Interrupted system call
```

#### 69.14.5.4 Funzione «strlen()» (string length)

La funzione *strlen()* calcola la lunghezza di una stringa, escludendo dal conteggio il carattere nullo di terminazione: «

```
#include <string.h>
size_t
```

```
strlen (const char *s)
{
    size_t i;
    for (i = 0; s[i] != 0 ; i++)
        {
            ; // Eseguo solo il conteggio.
        }
    return i;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
    size_t lunghezza;
    char stringa[] = "ciao amore";
    lunghezza = strlen (stringa);
    printf ("la frase \"%s\" si compone di %i caratteri\n",
            stringa,
            lunghezza);
    return 0;
}
```

Avviando il programma si deve vedere il risultato seguente:

la frase "ciao amore" si compone di 10 caratteri

## 69.15 File «signal.h»

Il file ‘signal.h’ della libreria standard definisce principalmente delle funzioni per la gestione dei segnali che riguardano il programma. Assieme alle funzioni definisce anche delle macro-variabili per classificare i segnali e per fare riferimento a delle funzioni predefinite, destinate astrattamente al trattamento dei segnali (si veda eventualmente la realizzazione del file ‘signal.h’ e di alcune delle sue funzioni nei sorgenti di os32, sezione [95.17](#)).


Dal punto di vista del programmatore, l’uso delle funzioni di questo file di intestazione può essere abbastanza semplice, ma la comprensione di come siano organizzate nel file ‘signal.h’ diventa invece difficile.

Nello standard POSIX, la questione dei segnali è particolarmente complessa. Nel capitolo viene considerato solo il fatto che i segnali standard sono in numero maggiore, tralasciando sostanzialmente il resto.

Qui vengono proposti due modi alternativi di scrivere il file ‘signal.h’ che dovrebbero essere disponibili presso [allegati/c/include/signal.h](#) e [allegati/c/include/signal-bis.h](#).

### 69.15.1 Dichiarazione contorta

Per la gestione dei segnali ci sono due funzioni che vengono dichiarate nel file ‘signal.h’: **signal()** e **raise()**. La funzione **raise()** serve ad azionare un segnale, come dire che serve ad attivare manualmente un allarme interno al programma, specificato da un numero particolare che ne definisce il tipo. Il programma contiene sempre una procedura predefinita che stabilisce ciò che deve essere fatto in

presenza di un certo allarme, ma il programmatore può ridefinire la procedura attraverso l'uso della funzione *signal()*, con la quale si associa l'avvio di una funzione particolare in presenza di un certo segnale. Il modello sintattico seguente rappresenta, in modo  estremamente semplificato, l'uso della funzione *signal()*:

```
signal (n_segnaie, funzione_da_associare)
```

Logicamente la funzione che si associa a un certo numero di segnale viene indicata negli argomenti della chiamata come puntatore a funzione. La funzione che viene passata come argomento è un gestore di segnale e deve avere una certa forma:

```
void gestore (n_segnaie)
```

In pratica, quando viene creata l'associazione tra segnale e funzione che deve gestirlo, la funzione in questione deve avere un parametro tale da poter rappresentare il numero del segnale che la riguarda e non restituisce alcun valore (pertanto è di tipo '**void**').


 Avendo determinato questo, il modello della funzione *signal()* può essere precisato un po' di più:

```
signal (n_segnaie, void (*gestore)(int))
```

Ciò significa che il secondo argomento della funzione *signal()* è un puntatore a una funzione ('*gestore ()*') con un parametro di tipo '**int**', la quale non restituisce alcunché ('**void**').

Ma non è ancora stato specificato cosa deve restituire la funzione



***signal()***: un puntatore a una funzione che ha un parametro di tipo ‘**int**’ e che a sua volta non restituisce alcunché. In pratica, ***signal()*** deve restituire il puntatore a una funzione che ha le stesse caratteristiche di quella del proprio secondo parametro. A questo punto, si arriva al prototipo completo, ma molto difficile da interpretare a prima vista: 

```
void (*signal (n_segnale, void (*gestore) (int))) (int);
```

Per ovviare a questo problema di comprensibilità, anche se lo standard non lo prescrive, di norma, nel file ‘`signal.h`’ si dichiara un tipo speciale, in qualità di puntatore a funzione con le caratteristiche del gestore di segnale:

```
...  
typedef void (*sig
```

```
handler_t) (int);  
...
```


Così facendo, la funzione ***signal()*** può essere dichiarata in modo più gradevole:

```
sig
```

```
handler_t signal (n_segnale, sig
```

```
handler_t gestore);
```

## 69.15.2 Tipo speciale

A parte il caso di ‘***sig*handler\_t**’ che non fa parte dello standard del linguaggio, il file ‘`include.h`’ definisce il tipo ‘***sig*\_atomic\_t**’, il cui uso non viene precisato dai documenti ufficiali. Si chiarisce solo che deve trattarsi di un valore intero, possibilmente di tipo volatile, a cui si possa accedere attraverso una sola 

istruzione elementare del linguaggio macchina (in modo tale che la lettura o la modifica del suo contenuto non possa essere sospesa a metà da un'interruzione di qualunque genere).

```
typedef int sig_atomic_t;
```

Nell'esempio, il tipo '**sig\_atomic\_t**' viene dichiarato come equivalente al tipo '**int**', supponendo che l'accesso alla memoria per un tipo intero normale corrisponda a un'operazione «atomica» nel linguaggio macchina. A ogni modo, il tipo a cui corrisponde '**sig\_atomic\_t**' può dipendere da altri fattori, mentre l'unico vincolo nel rango è quello di poter contenere i valori rappresentati dalle macro-variabili **SIG...**, le quali individuano mnemonicamente i segnali.

Il programmatore che deve memorizzare un segnale in una variabile, potrebbe usare per questo il tipo '**sig\_atomic\_t**'.

### 69.15.3 Denominazione dei segnali

«

Un gruppo di macro-variabili definisce l'elenco dei segnali gestibili. Lo standard del linguaggio ne prescrive solo una quantità minima, mentre il sistema operativo può richiederne degli altri. Teoricamente l'associazione del numero al nome simbolico del segnale è libera, ma in pratica la concordanza con altri standard prescrive il rispetto di un minimo di uniformità.

```
#define SIGINT      2
#define SIGILL     4
#define SIGABRT    6
#define SIGFPE     8
#define SIGSEGV   11
#define SIGTERM   15
```

Tabella 69.208. Denominazione dei segnali indispensabili al linguaggio C.

Denominazione	Significato mnemonico	Descrizione
SIGABRT	<i>abort</i>	Deriva da una terminazione anomala che può essere causata espressamente dall'uso della funzione <i>abort()</i> .
SIGFPE	<i>floating point exception</i>	Viene provocato da un'operazione aritmetica errata, come la divisione per zero o uno straripamento del risultato.
SIGILL	<i>illegal</i>	Istruzione «illegale».
SIGINT	<i>interrupt</i>	Deriva dalla ricezione di una richiesta interattiva di attenzione, quale può essere quella di un'interruzione.
SIGSEGV	<i>segmentation violation</i>	Deriva da un accesso alla memoria non valido, per esempio oltre i limiti fissati.
SIGTERM	<i>termination</i>	Indica la ricezione di una richiesta di terminazione del funzionamento del programma.

#### 69.15.4 Segnali secondo POSIX

Lo standard POSIX prescrive un insieme minimo di segnali più numeroso, attribuendo anche un'azione predefinita a carico del sistema operativo.



Tabella 69.209. Denominazione dei segnali indispensabili allo standard POSIX.

Denominazione	Azione predefinita	Descrizione
SIGABRT	terminazione anomala	Conclusione prematura del processo elaborativo.
SIGALRM	terminazione normale	Segnale di allarme da un orologio programmabile.
SIGBUS	terminazione anomala	Accesso errato alla memoria.
SIGCHLD	ignorato	Conclusione, sospensione o continuazione di un processo figlio.
SIGCONT	continuazione	Ripresa dell'esecuzione di un processo, se risulta sospeso.
SIGFPE	terminazione anomala	Operazione aritmetica errata.
SIGHUP	terminazione normale	Aggancio (interruzione del collegamento con il terminale).
SIGILL	terminazione anomala	Istruzione illegale.
SIGINT	terminazione normale	Segnale di interruzione dal terminale.
SIGKILL	terminazione normale	Eliminazione del processo elaborativo (senza la possibilità che il segnale sia catturato o ignorato).

Denominazione	Azione pre-definita	Descrizione
SIGPIPE	terminazione normale	Scrittura verso un condotto ( <i>pipe</i> ) senza che alcun processo stia leggendo da lì.
SIGQUIT	terminazione anomala	Segnale di abbandono ( <i>quit</i> ) dal terminale.
SIGSEGV	terminazione anomala	Riferimento errato alla memoria.
SIGSTOP	sospensione	Sospensione dell'esecuzione (senza la possibilità che il segnale sia catturato o ignorato).
SIGTERM	terminazione normale	Richiesta di conclusione del funzionamento.
SIGSTOP	sospensione	Segnale di stop dal terminale.
SIGTTIN	sospensione	Processo sullo sfondo che attende di poter leggere (dalla tastiera).
SIGTTOU	sospensione	Processo sullo sfondo che attende di poter scrivere (sullo schermo del terminale).
SIGUSR1	terminazione normale	Segnale 1, il cui scopo è definibile dall'utente.
SIGUSR2	terminazione normale	Segnale 2, il cui scopo è definibile dall'utente.
SIGPOLL	terminazione normale	<i>Pollable event.</i>

Denominazione	Azione predefinita	Descrizione
SIGPROF	terminazione normale	<i>Profiling timer expired.</i>
SIGSYS	terminazione anomala	Chiamata di sistema errata.
SIGTRAP	terminazione anomala	Trappola scattata per il tracciamento del codice.
SIGURG	ignorato	Informazione urgente disponibile da un socket.
SIGVTALRM	terminazione normale	<i>Virtual timer expired.</i>
SIGXCPU	terminazione anomala	Limite del tempo di CPU superato.
SIGXFSZ	terminazione anomala	Limite della dimensione dei file superato.

L'azione predefinita è quella che deve essere svolta dal sistema operativo se il segnale non viene catturato o non viene ignorato dal programma che lo riceve (tenendo conto che per '**SIGKILL**' e '**SIGSTOP**' i programmi non possono intervenire). Una «terminazione normale» implica la conclusione normale del processo elaborativo, a parte il fatto che il valore restituito dal processo dipende dal segnale stesso; una «terminazione anomala» implica di solito qualcosa di più, di solito si tratta dello scarico della memoria del processo in un file (*core dump*), per consentire un'analisi di quanto accaduto; la

«sospensione» rappresenta un arresto temporaneo, in attesa di un segnale di «continuazione»; un segnale «ignorato» indica che lo stato del processo non viene cambiato, ma ciò non significa che il segnale in sé sia privo di conseguenze.

Il segnale '**SIGCHLD**' che formalmente viene indicato come privo di effetti, nella tradizione Unix ha un ruolo molto importante e relativamente complesso, per la gestione della dipendenza dei processi. In un sistema Unix i processi hanno una dipendenza gerarchica, trattata secondo un albero genealogico, dove ogni processo ha un genitore. Dato che la conclusione di un processo produce un valore che dovrebbe essere raccolto dal genitore che lo ha avviato (oppure che lo ha adottato, nel caso il genitore vero sia defunto nel frattempo), quando un processo muore (termina di funzionare per qualunque motivo), il genitore riceve un segnale '**SIGCHLD**': se il genitore è in attesa del valore di uscita del processo defunto, lo raccoglie e le tracce residue di tale processo possono essere distrutte definitivamente; altrimenti, se il segnale non viene catturato il processo defunto viene eliminato senza comunicare tale valore al genitore.

### 69.15.5 Gestori fittizi di segnali

Lo standard prescrive di definire tre macro-variabili che devono espandersi in un puntatore a quel tipo di funzione che deve essere in grado di gestire le azioni da compiere in relazione alla ricezione di un certo segnale. Tuttavia, questo puntatore non deve essere rivolto a una funzione vera, ma averne solo la forma. In pratica, si usano dei valori interi con un valore assoluto molto piccolo e si esegue un cast per trasformarli in puntatori a funzione, come già accennato.

Per ottenere questo risultato, si possono dichiarare le macro-

variabili in due modi equivalenti, con la differenza che il secondo è probabilmente più difficile da interpretare:

```
typedef void (*sighandler_t) (int); // Il tipo
                                     // «sighandler_t» è un
                                     // puntatore a funzione
                                     // per la gestione dei
                                     // segnali con parametro
                                     // «int» che
                                     // restituisce «void».

//
// Funzioni non dichiarabili
//
#define SIG_ERR ((sighandler_t) -1) // Trasforma un numero
#define SIG_DFL ((sighandler_t) 0) // intero in un tipo
#define SIG_IGN ((sighandler_t) 1) // «sighandler_t»,
                                     // ovvero un puntatore
                                     // a funzione che però
                                     // non esiste realmente.
```

```
//
// Funzioni non dichiarabili
//
#define SIG_ERR ((void (*) (int)) -1) // Trasforma un numero
#define SIG_DFL ((void (*) (int)) 0) // intero in un
#define SIG_IGN ((void (*) (int)) 1) // puntatore a una
                                     // funzione che ha un
                                     // parametro «int» e
                                     // restituisce «void».
```

Lo standard sottolinea il fatto che il numero trasformato in puntatore non deve poter corrispondere all'indirizzo di alcuna funzione reale; pertanto i valori usati possono essere solo molto bassi o molto alti (in termini di valore assoluto), contando sul fatto che a tali indirizzi



non ci possano essere funzioni reali. In pratica, non deve succedere che venga dichiarata una funzione per la gestione di un segnale che finisca per avere proprio tali indirizzi, perché se così fosse, non verrebbe avviata, ma al suo posto verrebbe considerata l'azione che una di queste macro-variabili simboleggia.

Tabella 69.212. Macro-variabili per la gestione predefinita dei segnali.

Denominazione	Significato mnemonico	Descrizione
SIG_DFL	<i>default</i>	Indica simbolicamente che l'azione da compiere alla ricezione del segnale deve essere quella predefinita.
SIG_IGN	<i>ignore</i>	Indica simbolicamente che alla ricezione del segnale si procede come se nulla fosse accaduto.
SIG_ERR	<i>error</i>	Rappresenta un risultato errato nell'uso della funzione <i>signal()</i> .

## 69.15.6 Funzioni

La funzione *signal()* viene usata per associare un «gestore di segnale», costituito dal puntatore a una funzione, a un certo segnale; tutto questo allo scopo di attivare automaticamente quella tale funzione al verificarsi di un certo evento che si manifesta tramite un certo segnale.

La funzione *signal()* restituisce un puntatore alla funzione che precedentemente si doveva occupare di quel segnale. Se invece l'operazione fallisce, *signal()* esprime questo errore restituendo il valo-

re *SIG\_ERR*, spiegando così il motivo per cui questo debba avere l'apparenza di un puntatore a funzione.

Per la stessa ragione per cui esiste *SIG\_ERR*, le macro-variabili *SIG\_DFL* e *SIG\_IGN* vanno usate come gestori di segnali, rispettivamente, per ottenere il comportamento predefinito o per far sì che i segnali siano ignorati semplicemente.

In linea di principio si può ritenere che nel proprio programma esista una serie iniziale di dichiarazioni implicite per cui si associano tutti i segnali gestibili a *SIG\_DFL*:

```
...  
signal (segnale, SIG_DFL);  
...
```

In base al fatto che sia stata dichiarato o meno il tipo '**sighandler\_t**', la funzione potrebbe avere i prototipi seguenti:

```
sighandler_t signal (int sig, sighandler_t handler);
```

```
void (*signal (int sig, void (*handler) (int))) (int);
```

L'altra funzione da considerare è *raise()*, con la quale si attiva volontariamente un segnale, dal quale poi dovrebbero o potrebbero sortire delle conseguenze, come stabilito in una fase precedente attraverso *signal()*. La funzione *raise()* è molto semplice:

```
int raise (int sig);
```

La funzione richiede come argomento il numero del segnale da attivare e restituisce un valore pari a zero in caso di successo, altrimenti restituisce un valore diverso da zero. Naturalmente, a seconda del-

l'azione che viene intrapresa all'interno del programma, a seguito della ricezione del segnale, può darsi che dopo questa funzione non venga eseguito altro, pertanto non è detto che possa essere letto il valore che la funzione potrebbe restituire.

### 69.15.7 Esempio

Viene proposto un esempio che serve a dimostrare il meccanismo di provocazione e intercettazione dei segnali: «

```
#include <stdio.h>
#include <signal.h>

void sig_generic_handler (int sig)
{
    printf ("Ho intercettato il segnale n. %i.\n", sig);
}

void sigfpe_handler (int sig)
{
    printf ("Attenzione: ho intercettato il segnale "
           "SIGFPE (%i)\n"
           "          e devo concludere il "
           "funzionamento!\n", sig);
    exit (sig);
}

void sigterm_handler (int sig)
{
    printf ("Attenzione: ho intercettato il segnale "
           "SIGTERM (%i),\n"
           "          però non intendo rispettarlo.\n",
           sig);
}
```

```
void sigint_handler (int sig)
{
    printf ("Attenzione: ho intercettato il segnale "
           "SIGINT (%i),\n"
           "           però non intendo rispettarlo.\n",
           sig);
}

int main (void)
{
    signal (SIGFPE,  sigfpe_handler);
    signal (SIGTERM, sigterm_handler);
    signal (SIGINT,  sigint_handler);
    signal (SIGILL,  sig_generic_handler);
    signal (SIGSEGV, sig_generic_handler);

    int c;
    int x;

    printf ("[0][Invio] divisione per zero\n");
    printf ("[c][Invio] provoca un segnale SIGINT\n");
    printf ("[t][Invio] provoca un segnale SIGTERM\n");
    printf ("[q][Invio] conclude il funzionamento\n");
    while (1)
    {
        c = getchar();
        if (c == '0')
        {
            printf ("Sto per eseguire una divisione per "
                   "zero:\n");
            x = x / 0;
        }
        else if (c == 'c')
        {

```

```
        raise (SIGINT);
    }
    else if (c == 't')
    {
        raise (SIGTERM);
    }
    else if (c == 'q')
    {
        return 0;
    }
}
return 0;
}
```

All'inizio del programma vengono definite delle funzioni per il trattamento delle situazioni che hanno provocato un certo segnale. Nella funzione *main()*, prima di ogni altra cosa, si associano tali funzioni ai segnali principali, quindi si passa a un ciclo senza fine, nel quale possono essere provocati dei segnali premendo un certo tasto, come suggerito da un breve menù. Per esempio è possibile provocare la condizione che si verifica tentando di dividere un numero per zero:

```
[0][Invio] divisione per zero
[c][Invio] provoca un segnale SIGINT
[t][Invio] provoca un segnale SIGTERM
[q][Invio] conclude il funzionamento
```

**0** [Invio]

Sto per eseguire una divisione per zero:

Attenzione: ho intercettato il segnale SIGFPE (8)  
e devo concludere il funzionamento!

La divisione per zero fa scattare il segnale '**SIGFPE**' che viene in-

tercettato dalla funzione *sigfpe\_handler()*, la quale però non può far molto e così conclude anche il funzionamento del programma.

Attraverso il menù è possibile provocare anche un segnale ‘**SIGINT**’ e un segnale ‘**SIGTERM**’, ma per questo è più interessante provare con i mezzi che dovrebbe offrire il sistema operativo:

```
[0][Invio] divisione per zero
[c][Invio] provoca un segnale SIGINT
[t][Invio] provoca un segnale SIGTERM
[q][Invio] conclude il funzionamento
```

**[Ctrl c]** [Invio]

Attenzione: ho intercettato il segnale SIGINT (2),  
però non intendo rispettarlo.

Utilizzando un sistema operativo Unix o simile, da un altro terminale, o da un'altra console, è possibile inviare un segnale specifico al programma:

```
$ kill n_processo [Invio]
```

Attenzione: ho intercettato il segnale SIGTERM (15),  
però non intendo rispettarlo.

```
$ kill -s 4 n_processo [Invio]
```

Ho intercettato il segnale n. 4.

```
$ kill -s 11 n_processo [Invio]
```

Ho intercettato il segnale n. 11.

Secondo l'esempio, i segnali 4 e 11 sono, rispettivamente, ‘**SIGILL**’ e ‘**SIGSEGV**’.

[ **¶** ] [ *Invio* ]

## 69.16 File «time.h»

Il file ‘time.h’ della libreria standard definisce principalmente delle funzioni per il trattamento delle informazioni data-orario. Non è stabilito in che modo venga rappresentato il tempo internamente alle funzioni, anche se di norma si tratta di un valore intero che esprime una quantità di secondi o di frazioni di secondo (si veda eventualmente la realizzazione del file ‘time.h’ e di alcune delle sue funzioni nei sorgenti di os32, sezione [95.29](#)).

### 69.16.1 Il tempo di CPU

La funzione *clock()* consente di ottenere il tempo di utilizzo del microprocessore (CPU), espresso virtualmente in cicli di CPU. In pratica, viene definita la macro-variabile *CLOCKS\_PER\_SEC*, contenente il valore che esprime convenzionalmente la quantità di cicli di CPU per secondo; quindi, il valore restituito dalla funzione *clock()* si traduce in secondi dividendolo per *CLOCKS\_PER\_SEC*. Il valore restituito dalla funzione *clock()* e l’espressione in cui si traduce la macro-variabile *CLOCKS\_PER\_SEC* sono di tipo ‘clock\_t’:

```
typedef long int clock_t;      // Unità di tempo convenzionale
                               // che rappresenta un ciclo
                               // virtuale di CPU.

#define CLOCKS_PER_SEC 1000000L // Valore convenzionale di
                               // 1 s, in termini di
                               // cicli virtuali di CPU.

clock_t clock (void);         // Tempo di utilizzo della
                               // CPU.
```

La funzione *clock()* restituisce il tempo di CPU espresso in unità `'clock_t'`, utilizzato dal processo elaborativo a partire dall'avvio del programma. Se la funzione non è in grado di dare questa indicazione, allora restituisce il valore `-1`, o più precisamente `'(clock_t) (-1)'`.

Per valutare l'intervallo di tempo di utilizzo della CPU, da una certa posizione del programma, a un'altra, occorre memorizzare i valori ottenuti dalla funzione e poi procedere a una sottrazione.

Per comprendere il significato della funzione *clock()*, del tipo `'clock_t'` e della macro-variabile *CLOCKS\_PER\_SEC*, viene proposto un esempio molto semplice, ma completo, dove si intende che il tipo `'clock_t'` sia intero e sia contenibile in una variabile di tipo `'long int'`:

```
#include <stdio.h>
#include <time.h>

int
main (int argc, char *argv[])
{
    clock_t t0;
    clock_t t1;
    long int i;
    long int x;

    t0 = clock ();
    printf ("Tempo iniziale: %li/%li\n",
           (long int) t0, (long int) CLOCKS_PER_SEC);

    for (i = 0; i < 10000000; i++)
    {
        x = i * 123;
```



```
    }

    t1 = clock ();
    printf ("Tempo finale: %li/%li\n",
           (long int) t1, (long int) CLOCKS_PER_SEC);

    return 0;
}
```

Avviando questo programma si potrebbe leggere un risultato simile al testo seguente, dove si vede un valore di ‘**CLOCKS\_PER\_SEC**’ pari a 1000000:

```
Tempo iniziale: 0/1000000
Tempo finale: 20000/1000000
```

## 69.16.2 Rappresentazione interna del tempo

Generalmente, nei sistemi Unix si tratta il tempo come una quantità di secondi trascorsi a partire da un’epoca di riferimento, che tradizionalmente coincide con l’ora zero del giorno 1 gennaio 1970. Da questo concetto deriva il tipo ‘**time\_t**’ della libreria, che, secondo lo standard, rappresenta la quantità di unità di tempo trascorsa a partire da un’epoca di riferimento. <<

```
typedef long int time_t; // Unità di tempo convenzionale
                        // per le informazioni data-orario.
```

Ammesso che si tratti di un numero intero, così come viene ipotizzato dall’esempio proposto, il rango costituisce il limite alle date rappresentabili. Pertanto, se il tipo ‘**time\_t**’ viene dichiarato come numero intero con segno, a 32 bit, per rappresentare una quantità di secondi (come nella tradizione Unix), significa che si possono rap-

presentare al massimo 24 855 giorni, pari a circa 68 anni.<sup>8</sup> Se l'epoca di riferimento è il 1970, si può arrivare al massimo al 2038.

### 69.16.3 Rappresentazione strutturata del tempo

«

La libreria standard prescrive che sia definito il tipo '**struct tm**', con il quale è possibile rappresentare tutte le informazioni relative a un certo tempo, secondo le convenzioni umane. Lo standard prescrive con precisione i membri minimi della struttura e l'intervallo di valori che possono contenere:

```
struct tm {
    int tm_sec;      // Secondi:           da 0 a 60.
    int tm_min;     // Minuti:           da 0 a 59.
    int tm_hour;    // Ora:             da 0 a 23.
    int tm_mday;    // Giorno del mese: da 1 a 31.
    int tm_mon;     // Mese dell'anno: da 0 a 11.
    int tm_year;    // Anno dal 1900.
    int tm_wday;    // Giorno della settimana: da 0 a 6
                    // con lo zero corrispondente alla
                    // domenica.
    int tm_yday;    // Giorno dell'anno: da 0 a 365.
    int tm_isdst;   // Ora estiva. Contiene un valore
                    // positivo se è in vigore l'ora estiva;
                    // zero se l'ora è quella «normale»
                    // ovvero quella invernale;
                    // un valore negativo se l'informazione
                    // non è disponibile.
};
```

Si può osservare che il mese viene rappresentato con valori che vanno da 0 a 11, pertanto gennaio si indica con lo zero e dicembre con il numero 11; inoltre, l'intervallo ammesso per i secondi consente di rappresentare un secondo in più, dato che l'intervallo corretto sa-

rebbe da 0 a 59; infine, il fatto che i giorni dell'anno vadano da 0 (il primo) a 365 (l'ultimo), significa che negli anni normali i valori vanno da 0 a 364, mentre negli anni bisestili si arriva a contare fino a 365.

#### 69.16.4 Funzioni per l'elaborazione di valori legati al tempo

Un gruppo di funzioni dichiarate nel file `'time.h'` ha lo scopo di elaborare in qualche modo le informazioni legate al tempo ed eventualmente di convertirle in formati diversi. Queste funzioni trattano il tempo in forma di variabili di tipo `'time_t'` o di tipo `'struct tm'`.<sup>9</sup>

La variabile di tipo `'time_t'` che viene usata in queste funzioni potrebbe esprimere un valore riferito al tempo universale (UT), mentre le funzioni che la utilizzano dovrebbero tenere conto del fuso orario, in base alle informazioni che può offrire il sistema operativo.

##### 69.16.4.1 Funzione «time()»

La funzione *time()* determina il tempo attuale secondo il calendario del sistema operativo, restituendolo nella forma del tipo `'time_t'`. La funzione richiede un parametro, costituito da un puntatore di tipo `'time_t *'`: se questo puntatore è valido, la stessa informazione che viene restituita viene anche memorizzata nell'indirizzo indicato da tale puntatore.

```
time_t time (time_t *timer);
```

In pratica, se è possibile, l'informazione data-orario raccolta dalla funzione, viene anche memorizzata in *\*timer*.

Se la funzione non può fornire l'informazione richiesta, allora restituisce il valore  $-1$ , o più precisamente: `(time_t) (-1)`.

### 69.16.4.2 Funzione «difftime()»

«

La funzione *difftime()* calcola la differenza tra due date, espresse in forma `'time_t'` e restituisce l'intervallo in secondi, in una variabile in virgola mobile, di tipo `'double'`:

```
double difftime (time_t time1, time_t time0);
```

Per la precisione, viene eseguito *time1-time0* e di conseguenza va il segno del risultato.

### 69.16.4.3 Funzione «mktime()»

«

La funzione *mktime()* riceve come argomento il puntatore a una variabile strutturata di tipo `'struct tm'`, contenente le informazioni sull'ora locale, e determina il valore di quella data secondo la rappresentazione interna, di tipo `'time_t'`:

```
time_t mktime (struct tm *timeptr);
```

La funzione tiene in considerazione solo alcuni membri della struttura; per la precisione, non considera il giorno della settimana e il giorno dell'anno; inoltre, ammette anche valori al di fuori degli intervalli stabiliti per i vari membri della struttura; infine, considera un valore negativo per il membro *timeptr->tm\_isdst* come la richiesta di determinare se sia o meno in vigore l'ora estiva per la data indicata.

Se la funzione non è in grado di restituire un valore rappresentabile nel tipo `'time_t'`, o comunque se non può eseguire il suo compito, restituisce il valore  $-1$ , o più precisamente `(time_t) (-1)`.

Se invece tutto procede regolarmente, la funzione provvede anche a correggere i valori dei vari membri della struttura e a ricalcolare il giorno della settimana e dell'anno.

L'esempio successivo mostra la dichiarazione di una variabile strutturata di tipo '**struct tm**', assegnando ai suoi membri dei valori non corretti. Con l'aiuto della funzione *mktime()* si ricostruisce la data secondo le convenzioni comuni:

```
#include <stdio.h>
#include <time.h>

int
main (int argc, char *argv[])
{
    struct tm t;
    time_t tx;

    t.tm_year = 107;      // 2007 - 1900
    t.tm_mon  = 5;
    t.tm_mday = 33;
    t.tm_hour = 0;
    t.tm_min  = 0;
    t.tm_sec  = 60;
    t.tm_isdst = -1;

    printf ("%i/%i/%i %i:%i:%i\n",
            t.tm_year + 1900, t.tm_mon + 1, t.tm_mday,
            t.tm_hour, t.tm_min, t.tm_sec);

    tx = mktime (&t);

    if (tx == (time_t) (-1))
    {
        printf ("Errore! %li\n", (long int) tx);
    }
}
```

```
    }
else
    {
        printf ("%i/%i/%i %i:%i:%i\n",
                t.tm_year + 1900, t.tm_mon + 1, t.tm_mday,
                t.tm_hour, t.tm_min, t.tm_sec);

        printf ("giorno della settimana: %i\n", t.tm_wday);
        printf ("giorno dell'anno: %i\n", t.tm_yday + 1);
        printf ("ora estiva: %i\n", t.tm_isdst);
    }
return 0;
}
```

Eseguendo questo programma di esempio si dovrebbe ottenere il testo seguente:

```
2007/6/33 0:0:60
2007/7/3 0:1:0
giorno della settimana: 2
giorno dell'anno: 184
ora estiva: 1
```

#### 69.16.4.4 Funzioni «*gmtime()*» e «*localtime()*»

«

Le funzioni *gmtime()* e *localtime()* hanno in comune il fatto di ricevere come argomento il puntatore di tipo '**time\_t \***', a un'informazione data-orario, per restituire il puntatore a una variabile strutturata di tipo '**struct tm \***'. In altri termini, le due funzioni convertono una data espressa nella forma del tipo '**time\_t**', in una data suddivisa nella struttura '**struct tm**':

```
struct tm *gmtime      (const time_t *timer);
struct tm *localtime  (const time_t *timer);
```

Nell'ambito di queste funzioni, è ragionevole supporre che l'informazione di tipo `'time_t'` a cui fanno riferimento, sia espressa in termini di tempo universale e che le funzioni stesse abbiano la possibilità di stabilire il fuso orario e la modalità di regolazione dell'ora estiva.

In ogni caso, la differenza tra le due funzioni sta nel fatto che *gmtime()* traduce il tempo a cui punta il suo argomento in una struttura contenente la data tradotta secondo il tempo coordinato universale, mentre *localtime()* la traduce secondo l'ora locale.

Va osservato che queste funzioni restituiscono un puntatore a un'area di memoria che può essere sovrascritta da altre chiamate alle stessi funzioni o a funzioni simili.

## 69.16.5 Conversione in stringa

Un piccolo gruppo di funzioni del file `'time.h'` è destinato alla conversione dei valori data-orario in stringhe, per l'interpretazione umana.

### 69.16.5.1 Funzione «*asctime()*»

La funzione *asctime()* converte un'informazione data-orario, espressa nella forma di una struttura `'struct tm'`, in una stringa che esprime l'ora locale, usando però una rappresentazione fissa in lingua inglese:

```
char *asctime (const struct tm *timeptr);
```

In pratica, dal momento che la data e l'orario vanno espressi secondo le convenzioni della lingua inglese, lo standard stesso descrive completamente questa funzione e il listato seguente è tratto letteralmente da tale definizione:

```
#include <time.h>
char *asctime(const struct tm *timeptr)
{
    static const char wday_name[7][3] = {
        "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
    };
    static const char mon_name[12][3] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    };
    static char result[26];
    sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %i\n",
        wday_name[timeptr->tm_wday],
        mon_name[timeptr->tm_mon],
        timeptr->tm_mday, timeptr->tm_hour,
        timeptr->tm_min, timeptr->tm_sec,
        1900 + timeptr->tm_year);
    return result;
}
```

## 69.16.5.2 Funzione «ctime()»

«

La funzione *ctime()* converte un'informazione data-orario, espressa nella forma del tipo `'time_t'` in una stringa che esprime l'ora locale, usando però una rappresentazione fissa in lingua inglese:

```
char *ctime (const time_t *timer);
```

Il comportamento di questa funzione è tale da generare una stringa analoga a quella della funzione *asctime()*, tanto che la si potrebbe esprimere così:



```
char *ctime (const time_t *timer)
{
    return asctime (localtime (timer));
}
```

Oppure, come macroistruzione, così:

```
#define ctime(t) (asctime (localtime (t)));
```

### 69.16.5.3 Funzione «strftime()»

La funzione *strftime()* si occupa di interpretare il contenuto di una struttura di tipo ‘**struct tm**’ e di tradurlo in un testo, secondo una stringa di composizione libera. In altri termini, questa funzione si comporta in modo simile a *printf()*, dove l’input è costituito dalla struttura contenente le informazioni data-orario. «

```
size_t strftime (char * restrict s,
                size_t maxsize,
                const char * restrict format,
                const struct tm * restrict timeptr);
```

Dal modello del prototipo della funzione, si vede che questa restituisce un valore numerico di tipo ‘**size\_t**’. Questo valore rappresenta la quantità di elementi<sup>10</sup> che sono stati scritti nella stringa di destinazione, rappresentata dal primo parametro. Dal computo di questi elementi è escluso il carattere nullo di terminazione, benché venga comunque aggiunto dalla funzione.

La funzione richiede, nell’ordine: un array di caratteri da utilizzare per comporre il testo; la dimensione massima di questo array; la stringa di composizione, contenente del testo costante e degli specificatori di conversione; il puntatore alla struttura contenente le informazioni data-orario da usare nella conversione.

La funzione termina il proprio lavoro con successo solo se può scrivere nell'array di destinazione il testo composto secondo le indicazioni della stringa di composizione, includendo anche il carattere nullo di terminazione. Se ciò non avviene, il valore restituito dalla funzione è zero e il contenuto dell'array di destinazione è imprecisato.

Il listato successivo mostra un programma completo che dimostra il funzionamento di *strftime()*. Va osservato che la conversione eseguita da tale funzione è sensibile alla configurazione locale; precisamente dipende dalla categoria **'LC\_TIME'**:

```
#include <stdio.h>
#include <locale.h>
#include <time.h>

int
main (int argc, char *argv[])
{
    char s[100];
    time_t t      = time (NULL);
    struct tm *tp = localtime (&t);
    int dim;
    setlocale (LC_ALL, "it_IT.UTF-8");
    dim = strftime (s, 100, "Ciao amore: sono "
                  "le %H:%M del %d %B %Y.", tp);
    printf ("%d: %s\n", dim, s);
    return 0;
}
```

Ecco cosa si potrebbe ottenere eseguendo questo programma:

```
45: Ciao amore: sono le 09:32 del 27 giugno 2012.
```

Nella tabella successiva vengono elencati gli specificatori di conversione principali. Sono ammissibili delle varianti, con l'aggiunta di modificatori, che però non vengono descritte. Per esempio è ammissibile l'uso degli specificatori '%Ec' e '%Od', per indicare rispettivamente una variante di '%c' e '%d'.

Tabella 69.244. Specificatori di conversione usati dalla funzione *strftime()*.

Specificatore	Corrispondenza
%C	<i>century</i> Il secolo, ottenuto dividendo l'anno per 100 e ignorando i decimali.
%y %Y	<i>year</i> L'anno: nel primo caso si mostrano solo le ultime due cifre, mentre nel secondo si mostrano tutte.
%b   %h %B	Rispettivamente, il nome abbreviato e il nome per esteso del mese.
%m	<i>month</i> Il numero del mese, da 01 a 12.
%d %e	<i>day</i> Il giorno del mese, in forma numerica, da 1 a 31, utilizzando sempre due cifre: nel primo caso si aggiunge eventualmente uno zero; nel secondo si aggiunge eventualmente uno spazio.
%a %A	Rispettivamente, il nome abbreviato e il nome per esteso del giorno della settimana.

Specificatore	Corrispondenza
%H	<i>hour</i>
%L	L'ora, espressa rispettivamente a 24 ore e a 12 ore.
%p	La sigla da usare, secondo la configurazione locale, per specificare che si tratta di un'ora antimeridiana o pomeridiana. Nella convenzione inglese si ottengono, per esempio, le sigle «AM» e «PM».
%r	L'ora espressa a 12 ore, completa dell'indicazione se trattasi di ora antimeridiana o pomeridiana, secondo le convenzioni locali.
%R	L'ora e i minuti, equivalente a '%H:%M'.
%M	<i>minute</i> I minuti, da 00 a 59.
%S	<i>second</i> I secondi, espresso con valori da 00 a 60.
%T	<i>time</i> L'ora, i minuti e i secondi, equivalente a '%H:%M:%S'.
%z	<i>time zone</i> La rappresentazione del fuso orario, nel primo caso come distanza dal tempo coordinato universale (UTC), mentre nel secondo si usa una rappresentazione conforme alla configurazione locale.
%Z	
%j	<i>julian</i> Il giorno dell'anno, usando sempre tre cifre numeriche: da 001 a 366.

Specificatore	Corrispondenza
%g %G	L'anno a cui appartiene la settimana secondo lo standard ISO 8601: nel primo caso si mostrano solo le ultime due cifre, mentre nel secondo si ha l'anno per esteso. Secondo lo standard ISO 8601 la settimana inizia con lunedì e la prima settimana dell'anno è quella che include il 4 gennaio.
%V	Il numero della settimana secondo lo standard ISO 8601. I valori vanno da 01 a 53. Secondo lo standard ISO 8601 la settimana inizia con lunedì e la prima settimana dell'anno è quella che include il 4 gennaio.
%u %w	Il giorno della settimana, espresso in forma numerica, dove, rispettivamente, si conta da 1 a 7, oppure da 0 a 6. Zero e sette rappresentano la domenica; uno è il lunedì.
%U %W	Il numero della settimana, contando, rispettivamente, dalla prima domenica o dal primo lunedì di gennaio. Si ottengono cifre da 00 a 53.
%x	La data, rappresentata secondo le convenzioni locali.
%X	L'ora, rappresentata secondo le convenzioni locali.
%c	La data e l'ora, rappresentate secondo le convenzioni locali.
%D	<i>date</i> La data, rappresentata come '%m/%d/%Y'.
%F	La data, rappresentata come '%Y-%m-%d'.
%n	Viene rimpiazzato dal codice di interruzione di riga.
%t	Viene rimpiazzato da una tabulazione orizzontale.

Specificatore	Corrispondenza
%%	Viene rimpiazzato da un carattere di percentuale.

## 69.17 File «stdio.h»

«

Il file ‘`stdio.h`’ della libreria standard è quello che fornisce le funzioni più importanti e in generale è il più complesso da realizzare, in quanto dipende strettamente dal meccanismo di gestione dei file del sistema operativo (si veda eventualmente la realizzazione del file ‘`stdio.h`’ e di alcune delle sue funzioni nei sorgenti di `os32`, sezione [95.18](#)). L’elemento più delicato che viene definito qui è il tipo di dati ‘**FILE**’, da cui dipende quasi tutto il resto.

Alle complicazioni che esistevano già alla nascita del linguaggio, nei primi sistemi Unix, si aggiungono attualmente quelle relative alla distinzione tra file di testo e file binari, oltre che quelle relative alla gestione dei caratteri multibyte, per cui la lettura o la scrittura attraverso un flusso di dati deve tenere conto dello stato di completamento di tali informazioni.

Il file ‘`stdio.h`’ definisce le funzioni principali per l’accesso ai file e una serie di funzioni per la lettura e scrittura di dati formattati (si vedano *`print()`*, *`scanf()`* e altre analoghe), ma altre funzioni realizzate espressamente per caratteri e stringhe estese (formate da elementi ‘`wchar_t`’) si trovano nel file ‘`wchar.h`’.

I file proposti che si basano sugli esempi del capitolo sono incompleti, in quanto manca la dichiarazione del tipo ‘**FILE**’ e del tipo ‘`fpos_t`’.

## 69.17.1 Tipi

Il file `'stdio.h'`, oltre a `'size_t'` che fa già parte del file `'stddef.h'`, e di `'va_list'` che fa già parte del file `'stdarg.h'`, dichiara due tipi di dati a uso specifico per la gestione dei file: `'FILE'` e `'fpos_t'`, realizzati normalmente attraverso delle strutture.

Il tipo `'fpos_t'` serve a rappresentare tutte le informazioni necessarie a specificare univocamente le posizioni interne a un file, per gli scopi delle funzioni `fgetpos()` e `fsetpos()`. Il tipo `'FILE'` deve poter esprimere tutte le informazioni necessarie a controllare un flusso di file (ovvero le operazioni su un file aperto), in particolare le posizioni correnti, il puntatore alla memoria tampone (*buffer*), l'indicatore di errore e di fine file.

```
typedef struct { /* omissis */ } fpos_t;  
  
typedef struct { /* omissis */ } FILE;
```

L'organizzazione effettiva delle strutture che costituiscono i tipi `'fpos_t'` e `'FILE'` dipende strettamente dal sistema operativo (nel contesto particolare della propria architettura); pertanto, per poterne approfondire le caratteristiche, occorre prima uno studio dettagliato delle funzionalità del sistema operativo stesso.

Alcune funzioni aggiunte dallo standard POSIX utilizzano anche il tipo `'off_t'`, che è descritto nel file di intestazione `'sys/types.h'`.

## 69.17.2 Macro-variabili varie

«

Il file ‘stdio.h’ dichiara la macro-variabile *NULL*, come già avviene nel file ‘stddef.h’, assieme ad altre macro-variabili a uso delle funzioni dichiarate al proprio interno. Quelle più semplici sono descritte nella tabella 69.248. L’esempio proposto della dichiarazione di tali macro-variabili è molto approssimativo:

```
#define _IOFBF          0 // Input-output fully buffered.
#define _IOLBF          1 // Input-output line buffered.
#define _IONBF          2 // Input-output with no buffering.

#define EOF             (-1)

#define FOPEN_MAX       10
#define FILENAME_MAX    254
#define L_tmpnam        FILENAME_MAX

#define SEEK_SET        0 // Dall'inizio.
#define SEEK_CUR        1 // Dalla posizione corrente.
#define SEEK_END        2 // Dalla fine del file.

#define TMP_MAX          100000 // Si ipotizza di usare nomi
                                // da «TMP00000.tmp» a
                                // «TMP99999.tmp».
```

POSIX

La porzione successiva riguarda le estensioni POSIX:

```
#define L_ctermid       14
#define P_tmpdir        "/tmp"
```



Tabella 69.248. Macro-variabili comuni per le funzioni di ‘stdio.h’.

Denominazione	Significato mnemonico	Descrizione
_IOFBF	<i>input output fully buffered</i>	Indica simbolicamente la richiesta di utilizzo di una memoria tampone a blocchi.
_IOLBF	<i>input output line buffered</i>	Indica simbolicamente la richiesta di utilizzo di una memoria tampone gestita a righe di testo.
_IONBF	<i>input output with no buffering</i>	Indica simbolicamente la richiesta di non utilizzare alcuna memoria tampone.
BUFSIZE	<i>buffer size</i>	Rappresenta la dimensione predefinita della memoria tampone.
EOF	<i>end of file</i>	È un numero intero di tipo ‘ <b>int</b> ’, negativo, che rappresenta il raggiungimento della fine del file. È in pratica ciò che si ottiene leggendo oltre la fine del file.
FOPEN_MAX	<i>file open max</i>	Il numero di file che un processo elaborativo può aprire simultaneamente, in base alle limitazioni poste dal sistema operativo.

Denominazione	Significato mnemonico	Descrizione
FILENAME_MAX		La dimensione di un array di elementi <b>'char'</b> , tale da essere abbastanza grande da contenere il nome del file più lungo (incluse le eventuali sequenze multibyte) che il sistema consenta di gestire.
L_tmpnam	<i>temporary name</i>	La dimensione di un array di elementi <b>'char'</b> , tale da essere abbastanza grande da contenere il nome di un file temporaneo generato dalla funzione <i>tmpnam()</i> .
SEEK_CUR	<i>seek current</i>	Indica di eseguire un posizionamento a partire dalla posizione corrente del file.
SEEK_END		Indica di eseguire un posizionamento a partire dalla fine di un file.
SEEK_SET		Indica di eseguire un posizionamento a partire dall'inizio di un file.
TMP_MAX		Rappresenta la quantità massima di nomi di file differenti che possono essere generati dalla funzione <i>tmpnam()</i> .

Tabella 69.249. Macro-variabili aggiunte dalle estensioni POSIX.

Denominazione	Significato mnemonico	Descrizione
L_ctermid	<i>character terminal identity</i>	La dimensione di un array di elementi <b>'char'</b> , tale da essere abbastanza grande da contenere il nome del file di dispositivo restituito dalla funzione <i>ctermid()</i> .
P_tmpdir	<i>temporary directory</i>	Definisce il percorso di una directory temporanea, da scegliere quando ciò che viene specificato con la funzione <i>tempnam()</i> non è appropriato.

### 69.17.3 Ipotesi di gestione del tipo «FILE»

Pur non essendo necessario che sia così, si può ipotizzare che per ogni file che possa essere aperto simultaneamente, sia disponibile un elemento di tipo **'FILE'** organizzato in un array. In tal caso, potrebbe essere dichiarato come nell'esempio seguente, già nel file `'stdio.h'`, anche se il nome usato per l'array è puramente indicativo: «

```
...
FILE _stream[FOPEN_MAX];
...
```

L'uso della macro-variabile *FOPEN\_MAX* garantisce che siano predisposti esattamente tutti gli elementi necessari alla gestione simultanea del limite di file previsti.

## 69.17.4 Flussi standard

«

Lo standard del linguaggio C prescrive che i nomi dei flussi standard previsti siano delle macro-variabili, tali da espandersi in espressioni che rappresentino puntatori di tipo ‘**FILE \***’, diretti ai flussi standard rispettivi. Nel caso del compilatore GNU C i puntatori sono già definiti con lo stesso nome dei flussi e, nel file ‘stdio.h’ vi si fa riferimento in qualità di variabili esterne (in quanto dichiarate nella libreria precompilata):

```
extern FILE *stdin;      // Si ipotizza che la libreria C
                        // definisca già i puntatori ai
extern FILE *stdout;    // flussi standard, usando
extern FILE *stderr;    // i nomi predefiniti.
                        //
#define stdin  stdin     // In questo caso, è facile definire
#define stdout stdout    // le macro che fanno riferimento
#define stderr stderr    // ai flussi standard.
```

Diversamente, nell’ipotesi in cui si gestisca un array di elementi ‘**FILE**’, si potrebbe supporre che i primi tre elementi siano usati per i flussi standard e in tal caso le dichiarazioni delle macro-variabili potrebbero essere fatte così:

```
...
#define stdin  (&_stream[0])
#define stdout (&_stream[1])
#define stderr (&_stream[2])
...
```

## 69.17.5 Funzioni per la rimozione e la ridenominazione dei file

Le funzioni *remove()* e *rename()* consentono, rispettivamente di eliminare o di rinominare un file. Il file in questione viene individuato da una stringa, il cui contenuto deve conformarsi alle caratteristiche del sistema operativo. Le due funzioni hanno in comune il fatto di restituire un valore intero (di tipo ‘**int**’), dove il valore zero rappresenta il completamento con successo dell’operazione, mentre un valore differente indica un fallimento.

```
int remove (const char *filename);  
int rename (const char *old, const char *new);
```

La sintassi per l’uso della funzione *remove()* è evidente dal suo prototipo, in quanto si attende un solo argomento che è costituito dal nome del file da eliminare; nel caso della funzione *rename()*, invece, il primo argomento è il nome del file preesistente e il secondo è quello che si vuole attribuirgli.

È importante ribadire che il comportamento delle due funzioni dipende dal sistema operativo. Per esempio, la ridenominazione può provocare la cancellazione di un file preesistente con lo stesso nome che si vorrebbe attribuire a un altro, oppure potrebbe limitarsi a fallire. In un sistema Unix o simile, molto dipende dalla configurazione dei permessi.

## 69.17.6 Funzioni per la gestione dei file temporanei

Le funzioni *tmpfile()* e *tmpnam()* servono per facilitare la creazione di file temporanei. La prima crea automaticamente un file di cui non si conosce il nome e la collocazione, aprendolo in aggiorna-

mento (modalità **'wb+'**); la seconda si limita a generare un nome che potrebbe essere usato per creare un file temporaneo:

```
FILE *tmpfile (void);  
char *tmpnam (char *s);
```

L'uso della funzione *tmpfile()* è evidente, in quanto non richiede argomenti e restituisce il puntatore al file creato; la seconda richiede l'indicazione di un array di caratteri da poter modificare, restituendo comunque il puntatore all'inizio dello stesso array. In ogni caso va chiarito che il file creato con la funzione **'tmpfile'**, una volta chiuso, viene rimosso automaticamente.

Le due funzioni devono essere in grado di poter generare un numero di nomi differente pari almeno al valore rappresentato da **'TMP\_MAX'**, rimanendo il fatto che non possano essere aperti più di **'FOPEN\_MAX'** file e che non possono essere generati file con nomi già esistenti.

Se si utilizza la funzione *tmpnam()*, l'array di caratteri che costituisce il primo argomento (*s*), viene usato dalla funzione per scriverci il nome del file temporaneo, restituendone poi il puntatore; tale array deve avere una dimensione di almeno **'L\_tmpnam'** elementi, come si vede nell'esempio seguente:

```
#include <stdio.h>  
int main (void)  
{  
    char t[L_tmpnam];  
    char *p;  
    p = tmpnam (t);  
    printf ("%s %s\n", t, p);  
    return 0;  
}
```

Se la funzione *tmpnam()* riceve come argomento il puntatore nullo, il nome del file temporaneo viene scritto in un'area di memoria statica che viene sovrascritta a ogni chiamata successiva della funzione stessa.

Entrambe le funzioni, se non possono eseguire il loro compito, restituiscono un puntatore nullo.

Le estensioni POSIX aggiungono anche la funzione *tempnam()*, la quale ha un comportamento simile a quello di *tmpnam()*, in quanto non crea il file, ma restituisce il percorso del file che potrebbe essere creato:

```
char *tempnam (const char *dir, const char *prefix);
```

La funzione *tempnam()* richiede l'indicazione di una stringa contenente il percorso di una directory, in cui si vuole sia creato un file temporaneo. Se la stringa indica una directory inadatta (perché non esiste o non è accessibile o non gli si può scrivere) oppure si indica il puntatore nullo, si fa riferimento a quanto descritto dalla macro-variabile *P\_tmpdir*. Se anche la directory a cui si riferisce la macro-variabile *P\_tmpdir* dà dei problemi, è possibile che la funzione decida in qualche modo dove sia possibile collocare un file temporaneo.

Il secondo argomento della funzione è una stringa che rappresenta un prefisso da usare per il nome del file temporaneo. Tale prefisso può essere lungo al massimo cinque caratteri. Se si vuole omettere tale prefisso, basta indicare il puntatore nullo.

Se tutto va bene, la funzione alloca dello spazio in memoria per la stringa che deve contenere il percorso di un file temporaneo che si potrebbe creare (ma senza crearlo) e ne restituisce il puntatore.

Quando tale informazione non serve più, la memoria allocata può essere liberata con la funzione *free()*. Se invece la funzione fallisce nel suo compito, restituisce il puntatore nullo e aggiorna la variabile *errno*.

### 69.17.7 Funzioni per l'apertura e la chiusura dei flussi di file

«

Le funzioni *fopen()*, *freopen()* e *fclose()*, consentono di aprire e chiudere i file, gestendoli attraverso un puntatore al *flusso di file* loro associato (*stream*). Il puntatore in questione è di tipo '**FILE \***'.

```
FILE *fopen    (const char *restrict filename,
               const char *restrict io_mode);

FILE *freopen (const char *restrict filename,
               const char *restrict io_mode,
               FILE *restrict stream);

int  fclose   (FILE *stream);
```

Quando viene aperto un file, gli si associa una variabile strutturata di tipo '**FILE**', contenente tutte le informazioni che servono a gestirne l'accesso. Questa variabile deve rimanere univoca e vi si accede normalmente attraverso un puntatore ('**FILE \***'). Dal momento che per il linguaggio C un file aperto è un flusso, la variabile strutturata che contiene le informazioni necessarie a gestirne l'accesso viene identificata come il flusso stesso, pertanto nei prototipi la variabile che contiene il puntatore di tipo '**FILE \***' viene denominata generalmente *stream*.



Dal momento che non è compito del programmatore dichiarare la variabile di tipo **'FILE'**, in pratica ci si riferisce al flusso di file sempre solo attraverso un puntatore a quella variabile. Pertanto, è più propriamente il puntatore a tale variabile che rappresenta il flusso di file.

L'apertura di un file, oltre che l'indicazione del nome del file, richiede di specificare la modalità, ovvero il tipo di accesso che si intende gestire. Sono previste le modalità elencate nella tabella 69.258.

Tabella 69.258. Modalità di accesso ai file.

Sigla	Mnemonico	Descrizione
r	<i>read</i>	Accesso in sola lettura di un file di testo.
w	<i>write</i>	Accesso a un file di testo in scrittura, che implica la creazione del file all'apertura, ovvero il suo troncamento a zero, se esiste già.
a	<i>append</i>	Accesso a un file di testo in aggiunta, che implica la creazione del file all'apertura, ovvero la sua estensione se esiste già.
rb wb ab	<i>binary</i>	Accesso in lettura, scrittura o aggiunta, ma di tipo binario.

Sigla	Mnemonico	Descrizione
r+ w+ a+	<i>update</i>	Accesso a un file di testo in lettura, scrittura o aggiunta, assieme alla modalità di aggiornamento. In pratica, con la lettura è consentita anche la scrittura; con la scrittura e l'aggiunta è consentita anche la riletture.
rb+   r+b wb+   w+b ab+   a+b		Accesso a un file binario in lettura, scrittura o aggiunta, assieme alla modalità di aggiornamento. In pratica, con la lettura è consentita anche la scrittura; con la scrittura e l'aggiunta è consentita anche la riletture. Si può osservare che il segno '+' può essere messo indifferentemente in mezzo o alla fine.

La funzione *fopen()* apre il file indicato come primo argomento (una stringa), con la modalità specificata nel secondo (un'altra stringa), restituendo il puntatore al flusso che consente di accedervi (se l'operazione fallisce, la funzione restituisce il puntatore nullo). La modalità di accesso viene espressa attraverso le sigle elencate nella tabella 69.258.

La funzione *freopen()* consente di associare un file differente a un flusso già esistente, cambiando anche la modalità di accesso, cosa che viene fatta normalmente per ridirigere i flussi standard. I primi due argomenti della funzione sono gli stessi di *fopen()*, con l'aggiunta alla fine del puntatore al flusso che si vuole ridirigere. La

funzione restituisce il puntatore al flusso ridiretto se l'operazione ha successo, altrimenti produce soltanto il puntatore nullo. Se nel primo argomento, al posto di indicare il nome del file, si mette un puntatore nullo, la chiamata della funzione serve solo per modificare la modalità di accesso a un file già aperto, senza ridirigerne il flusso. Va osservato che il cambiamento della modalità di accesso, in ogni caso, dipende dal sistema operativo e non è detto che si possano applicare tutte le combinazioni.

La funzione *fclose()* permette di chiudere il flusso indicato come argomento, restituendo un valore numerico pari a zero se l'operazione ha successo, oppure il valore corrispondente alla macro-variabile *EOF* in caso contrario. La chiusura di un flusso implica la scrittura di dati rimasti in sospeso (in una memoria tampone). Un flusso già chiuso non deve essere chiuso nuovamente.

Dal momento che lo standard POSIX introduce il concetto di descrittore di file, per poter associare un flusso di file a un file già aperto come descrittore, si usa la funzione *fdopen()*, mentre per fare il contrario, si usa la funzione *fileno()*:

```
FILE *fdopen (int fdn, const char *io_mode);  
int fileno (FILE *stream);
```

La funzione *fdopen()* richiede l'indicazione del numero del descrittore e della modalità di accesso, la quale deve essere compatibile con quanto già definito a proposito del descrittore stesso. L'associazione tra flusso di file e descrittore comporta inizialmente l'azzeramento dell'indicatore di errore e di quello di fine file; inoltre, se viene chiuso il flusso di file, si ottiene automaticamente la chiusura del descrittore relativo.

La funzione *fileno()* restituisce il numero di descrittore associato a un flusso di file già aperto. Se però l'operazione fallisce, restituisce il valore  $-1$  e aggiorna la variabile *errno*.

### 69.17.8 Funzioni per la gestione della memoria tampone

«

Le funzioni *setvbuf()* e *setbuf()* consentono di attribuire una memoria tampone (*buffer*) a un certo flusso di dati (un file già aperto), mentre *fflush()* consente di richiedere espressamente lo scarico della memoria in modo che le operazioni sospese di scrittura siano portate a termine completamente.

```
int  setvbuf (FILE *restrict stream, char *restrict buffer,
            int  buf_mode, size_t size);

void setbuf  (FILE *restrict stream, char *restrict buffer);

int  fflush  (FILE *stream);
```

La funzione *setvbuf()* permette di attribuire una memoria tampone a un file che è appena stato aperto e per il quale non è ancora stato eseguito alcun accesso. Il primo argomento della funzione è il puntatore al flusso relativo e il secondo è il puntatore all'inizio dell'array di caratteri da usare come memoria tampone. Se al posto del riferimento alla memoria tampone si indica un puntatore nullo, si intende che la funzione debba allocare automaticamente lo spazio necessario; se invece l'array viene fornito, è evidente che deve rimanere disponibile per tutto il tempo in cui il flusso rimane aperto.

Il terzo argomento atteso dalla funzione *setvbuf()* è un numero che esprime la modalità di funzionamento della memoria tampone. Questo numero viene fornito attraverso l'indicazione di una tra le macrovariabili *\_IOFBF*, *\_IOLBF* e *\_IONBF*. Il quarto argomento indica

la dimensione dell'array da usare come memoria tampone: se l'array viene fornito effettivamente, si tratta della dimensione che può essere utilizzata; altrimenti è la dimensione richiesta per l'allocazione automatica.

La funzione *setvbuf()* restituisce zero se l'operazione richiesta è eseguita con successo; in caso contrario restituisce un valore differente.

La funzione *setbuf()* è una semplificazione di *setvbuf()* che non restituisce alcun valore, dove al posto di indicare la modalità di gestione della memoria tampone, si intende implicitamente quella corrispondente alla macro-variabile *\_IOFBF* (pertanto si tratta di una gestione completa della memoria tampone), mentre al posto di indicare la dimensione dell'array che costituisce la memoria tampone si intende il valore corrispondente alla macro-variabile *BUFSIZ*. In pratica, è come utilizzare la funzione *setvbuf()* così:

```
(void) setvbuf (stream, buffer, _IOFBF, BUFSIZ);
```

La funzione '**fflush**' si usa per i file aperti in scrittura, allo scopo di aggiornare i file se ci sono dati sospesi nella memoria tampone che devono ancora essere trasferiti effettivamente. La funzione si attende come argomento il puntatore al flusso per il quale eseguire questo aggiornamento, ma se si fornisce il puntatore nullo (la macro-variabile *NULL*), si ottiene l'aggiornamento di tutti i file aperti in scrittura. A parte questo, la funzione non altera lo stato del flusso.

La funzione *fflush()* restituisce zero se riesce a completare con successo il proprio compito, altrimenti restituisce il valore corrispondente a *EOF* e aggiorna la variabile individuata dall'espressione

*errno* in modo da poter risalire al tipo di errore che si è presentato.

La funzione *fflush()* interviene solo nella memoria tampone gestita internamente al programma, ma bisogna tenere presente che il sistema operativo potrebbe gestire un'altra memoria del genere, per il cui scarico occorre eventualmente intervenire con funzioni specifiche del sistema stesso.

### 69.17.9 Funzioni per la composizione dell'output

«

Alcune funzioni del file 'stdio.h' sono realizzate con lo scopo principale di comporre una stringa attraverso l'inserzione di componenti di vario genere, convertendo i dati in modo da poterli rappresentare in forma «tipografica», nel senso di sequenza di caratteri che hanno una rappresentazione grafica.

Queste funzioni hanno in comune una stringa contenente degli *specificatori di conversione*, caratterizzati dal fatto che iniziano con il simbolo di percentuale ('%') e dalla presenza di un elenco indefinito di argomenti, il cui valore viene utilizzato in sostituzione degli specificatori di conversione. Il modo in cui si esprime uno specificatore di conversione può essere complesso, pertanto viene mostrato un modello sintattico che descrive la sua struttura:

% [*simbolo*] [*n\_ampiezza*] [*.n\_precision*] [hh | h | l | ll | j | z | t | L] *tipo*

La prima cosa da individuare in uno specificatore di conversione è il tipo di argomento che viene interpretato e, di conseguenza, il genere di rappresentazione che se ne vuole produrre. Il tipo viene espresso

da una lettera alfabetica, alla fine dello specificatore di conversione. La tabella successiva riepiloga i tipi principali.

Tabella 69.261. Tipi di conversione principali.

Simbolo	Tipo di argomento	Conversione applicata
%...d %...i	int	Numero intero con segno da rappresentare in base dieci.
%...u	unsigned int	Numero intero senza segno da rappresentare in base dieci.
%...o	unsigned int	Numero intero senza segno da rappresentare in ottale (senza lo zero iniziale che viene usato spesso per caratterizzare un tale tipo di rappresentazione).
%...x %...X	unsigned int	Numero intero senza segno da rappresentare in esadecimale (senza il prefisso '0x' o '0X' che viene usato spesso per caratterizzare un tale tipo di rappresentazione).
%...c	int	Un carattere singolo, dopo la conversione in 'unsigned char'.
%...s	char *	Una stringa.
%...f	double	Un numero a virgola mobile, da rappresentare in notazione decimale fissa: <b>[ - ] iii . dddddd</b>

Simbolo	Tipo di argomento	Conversione applicata
%...e %...E	double	Un numero a virgola mobile, da rappresentare in notazione esponenziale: $\begin{bmatrix} - \\ \end{bmatrix} i . dddddd e \pm xx$ $\begin{bmatrix} - \\ \end{bmatrix} i . dddddd E \pm xx$
%...g %...G	double	Un numero a virgola mobile, rappresentato in notazione decimale fissa o in notazione esponenziale, a seconda di quale si presti meglio in base ai vincoli posti da altri componenti dello specificatore di conversione.
%p	void *	Un puntatore generico rappresentato in qualche modo in forma grafica.
%n	int *	Questo specificatore non esegue alcuna conversione e si limita a memorizzare un valore intero (di tipo 'int') nella variabile a cui punta l'argomento. Per la precisione, viene memorizzata la quantità di caratteri generati fino a quel punto dalla conversione.
%%		Questo specificatore si limita a produrre un carattere di percentuale ('%') che altrimenti non sarebbe rappresentabile.

Nel modello sintattico che descrive lo specificatore di conversione, si vede che subito dopo il segno di percentuale può apparire un simbolo



(*flag*). I simboli principali che possono essere utilizzati sono descritti nella tabella successiva.

Tabella 69.262. Alcuni simboli, o *flag*.

Simbolo	Corrispondenza
%+... %#+... %+0 <i>ampiezza</i> ... %#+0 <i>ampiezza</i> ...	Il segno «+» fa sì che i numeri con segno lo mostrino anche se è positivo. Può combinarsi con lo zero e il cancelletto.
%0 <i>ampiezza</i> ... %+0 <i>ampiezza</i> ... %#0 <i>ampiezza</i> ... %#+0 <i>ampiezza</i> ...	Lo zero fa sì che siano inseriti degli zeri a sinistra per allineare a destra il valore, nell'ambito dell'ampiezza specificata. Può combinarsi con il segno «+» e il cancelletto.
% <i>ampiezza</i> ... % <i> </i> <i>ampiezza</i> ...	In mancanza di uno zero iniziale, in presenza dell'indicazione dell'ampiezza, il valore viene allineato a destra usando degli spazi. È possibile esprimere esplicitamente l'intenzione di usare gli spazi mettendo proprio uno spazio, ma in generale non è richiesto. Se si mette lo spazio letteralmente, questo non è poi compatibile con lo zero, mentre le combinazioni con gli altri simboli sono ammissibili.

Simbolo	Corrispondenza
% - <i>ampiezza</i> ... % - + <i>ampiezza</i> ... % # - <i>ampiezza</i> ... % # - + <i>ampiezza</i> ...	Il segno meno, usato quando la conversione prevede l'uso di una quantità fissa di caratteri con un valore che appare di norma allineato a destra, fa sì che il risultato sia allineato a sinistra. Il segno meno si può combinare il segno «+» e il cancelletto.
% #...	Il cancelletto richiede una modalità di rappresentazione alternativa, ammesso che questa sia prevista per il tipo di conversione specificato. È compatibile con gli altri simboli, ammesso che il suo utilizzo serva effettivamente per ottenere una rappresentazione alternativa.

Subito prima della lettera che definisce il tipo di conversione, possono apparire una o due lettere che modificano la lunghezza del valore da interpretare (per lunghezza si intende qui la quantità di byte usati per rappresentarlo). Per esempio, '%...Lf' indica che la conversione riguarda un valore di tipo '**long double**'. Tra questi specificatori della lunghezza del dato in ingresso ce ne sono alcuni che indicano un rango inferiore a quello di '**int**', come per esempio '%...hhd' che si riferisce a un numero intero della dimensione di un '**signed char**'; in questi casi occorre comunque considerare che nella trasmissione degli argomenti alle funzioni interviene sempre la promozione a intero, pertanto viene letto il dato della dimensione specificata, ma viene «consumato» il risultato ottenuto dalla promozione. La tabella successiva riepiloga i modificatori di lunghezza principali.

Tabella 69.263. Alcuni modificatori della lunghezza del dato in ingresso.

Simbolo	Tipo	Simbolo	Tipo
%...hhd %...hhi	signed char	%...hhu %...hho %...hhx   %...hhX	unsigned char
%...hd %...hi	short int	%...hu %...ho %...hx   %...hX	unsigned short int
%...ld %...li	long int	%...lu %...lo %...lx   %...lX	unsigned long int
%...lc	wint_t	%...ls	wchar_t *
%...lld %...lli	long long int	%...llu %...llo %...llx   %...llX	unsigned long long int

Simbolo	Tipo	Simbolo	Tipo
%...jd %...ji	intmax_t	%...ju %...jo %...jx   %...jX	uintmax_t
%...zd %...zi	size_t	%...zu %...zo %...zx   %...zX	size_t
%...td %...ti	ptrdiff_t	%...tu %...to %...tx   %...tX	ptrdiff_t
%...Le   %...LE %...Lf   %...LF %...Lg   %...LG	long double		

I modificatori di lunghezza si possono utilizzare anche con il tipo ‘%...n’. In tal caso, si intende che il puntatore sia del tipo specificato dalla lunghezza. Per esempio, ‘%tn’ richiede di memorizzare la quantità di byte composta fino a quel punto in una variabile di tipo ‘ptrdiff\_t’, a cui si accede tramite il puntatore fornito.

Tra il simbolo (*flag*) e il modificatore di lunghezza può apparire un numero che rappresenta l’ampiezza da usare nella trasformazione

ed eventualmente la precisione: ‘*ampiezza* [*.precisione*]’. Il concetto parte dalla rappresentazione dei valori in virgola mobile, dove l’ampiezza indica la quantità complessiva di caratteri da usare e la precisione indica quanti di quei caratteri usare per il punto decimale e le cifre successive, ma si applica anche alle stringhe.

In generale, per quanto riguarda la rappresentazione di valori numerici, la parte intera viene sempre espressa in modo completo, anche se l’ampiezza indicata è inferiore; ai numeri interi la precisione non si applica; per i numeri in virgola mobile con rappresentazione esponenziale, la precisione riguarda le cifre decimali che precedono l’esponente; per le stringhe la precisione specifica la quantità di caratteri da considerare, troncando il resto.

In un altro capitolo, la tabella 67.26 riporta un elenco di esempi di utilizzo della funzione *printf()* dove si può valutare l’effetto dell’indicazione dell’ampiezza e della precisione.

L’ampiezza, o la precisione, o entrambe, potrebbero essere indicate da un asterisco, come per esempio ‘%\*.\*f’. L’asterisco usato in questo modo indica che il valore corrispondente (ampiezza, precisione o entrambe) viene tratto dagli argomenti come intero (**int**). Pertanto, per tornare all’esempio composto come ‘%\*.\*f’, dagli argomenti viene prelevato un intero che rappresenta l’ampiezza, un altro intero che rappresenta la precisione, quindi si preleva un valore **double** che è quanto va rappresentato secondo l’ampiezza e la precisione richieste.

## 69.17.9.1 Funzioni che ricevono gli argomenti direttamente

«

Un gruppo di funzioni per la composizione dell'output riceve direttamente gli argomenti variabili che servono agli specificatori di conversione:

```
int sprintf (char *restrict s, const char *restrict format,  
            ...);  
int snprintf (char *restrict s, size_t n,  
             const char *restrict format, ...);  
int fprintf (FILE *restrict stream,  
            const char *restrict format, ...);  
int printf (const char *restrict format, ...);
```

Tutte le funzioni di questo gruppo hanno in comune la stringa di composizione, costituita dal parametro *format*, e gli argomenti successivi che sono in quantità e qualità indeterminata, in quanto per la loro interpretazione contano gli specificatori di conversione inseriti nella stringa di composizione. Inoltre, tutte queste funzioni restituiscono la quantità di caratteri prodotti dall'elaborazione della stringa di composizione. Va osservato che il conteggio riguarda solo i caratteri e non include, eventualmente, il carattere nullo di terminazione di stringa che viene usato per le funzioni *sprintf()* e *snprintf()*. Se durante il procedimento di composizione si verifica un errore, queste funzioni possono restituire un valore negativo.

La funzione *sprintf()* produce il risultato della composizione memorizzandolo a partire dal puntatore indicato come primo parametro (*s*) e aggiungendo il carattere nullo di terminazione. La funzione *snprintf()*, invece, produce al massimo *n*-1 caratteri, aggiungendo sempre il carattere nullo di terminazione.

La funzione *fprintf()* scrive il risultato della composizione attraverso il flusso di file *stream*, mentre *printf()* lo scrive attraverso lo standard output.

### 69.17.9.2 Funzioni che ricevono gli argomenti da un'altra funzione

A fianco delle funzioni descritte nella sezione precedente, un gruppo analogo svolge le stesse operazioni, ma ricevendo gli argomenti variabili per riferimento. In pratica si tratta di ciò che serve quando gli argomenti variabili sono stati ottenuti da un'altra funzione e non da una chiamata diretta.

```
int vsprintf (char *restrict s,  
             const char *restrict format, va_list arg);  
int vsnprintf (char *restrict s, size_t n,  
             const char *restrict format, va_list arg);  
int vfprintf (FILE *restrict stream,  
            const char *restrict format, va_list arg);  
int vprintf (const char *restrict format, va_list arg);
```

Il funzionamento è conforme a quello delle funzioni che non hanno la lettera 'v' iniziale; per esempio, *vsprintf()* si comporta conformemente a *sprintf()*. Per comprendere la differenza si potrebbe dimostrare la realizzazione ipotetica della funzione *printf()* avvalendosi di *vprintf()*:

```
int printf (const char *restrict format, ...)
{
    va_list arg;
    va_start (arg, format);
    int count;
    count = vprintf (format, arg);
    va_end (arg);
    return count;
}
```

### 69.17.10 Funzioni per l'interpretazione dell'input

«

Un piccolo gruppo di funzioni del file `'stdio.h'` è specializzato nell'interpretazione di una stringa, dalla quale si vanno a estrapolare dei componenti da collocare in variabili di tipo opportuno. In altri termini, da una stringa che rappresenta un valore espresso attraverso caratteri grafici, si vuole estrarre il valore e assegnare a una certa variabile.

Il meccanismo è opposto a quello usato dalle funzioni del tipo `'...printf()'` e anche in questo caso si parte da una stringa contenente principalmente degli specificatori di conversione, seguita da un numero indefinito di argomenti. Gli specificatori delle funzioni che interpretano l'input sono simili a quelli usati per la composizione dell'output, ma non possono essere equivalenti in tutto. Sinteticamente si possono descrivere così:

```
% [*] [n_ampiezza] [hh|h|l|ll|j|z|t|L] tipo
```

Come si può vedere, all'inizio può apparire un asterisco, il cui scopo è quello di annullare l'assegnamento del valore a una variabile. In



pratica, con l'asterisco il dato corrispondente allo specificatore viene interpretato, ma poi non viene salvato.

Successivamente può apparire un numero che rappresenta l'ampiezza del dato da interpretare, in byte, il cui scopo è quello di limitare la lettura fino a un certo carattere (inteso come **'char'**, pertanto le sequenze multibyte contano per più di una unità singola).

Dopo può apparire una sigla, composta da una o più lettere, il cui scopo è quello di modificare la dimensione predefinita della variabile di destinazione. In altri termini, senza questo modificatore si intende che la variabile ricevente debba essere di una certa grandezza, ma con l'aggiunta del «modificatore di lunghezza» si precisa invece qualcosa di diverso. In pratica, il modificatore di lunghezza usato da queste funzioni è equivalente a quello delle funzioni di composizione dell'output.

Al termine dello specificatore di conversione appare una lettera che dichiara come deve essere interpretato il dato in ingresso e, in mancanza del modificatore di lunghezza, indica anche la dimensione della variabile ricevente.

Tabella 69.267. Tipi di conversione principali.

Simbolo	Tipo di argomento	Conversione applicata
<code>%...d</code>	<code>int *</code>	Numero intero con segno rappresentato in base dieci.
<code>%...i</code>	<code>int *</code>	Numero intero con segno rappresentato in base dieci o in base otto, avendo come prefisso uno zero, oppure in base sedici, avendo come prefisso <b>'0x'</b> o <b>'0X'</b> .

Simbolo	Tipo di argomento	Conversione applicata
%...u	unsigned int *	Numero intero senza segno rappresentato in base dieci.
%...o	unsigned int *	Numero intero senza segno rappresentato in ottale (con o senza lo zero iniziale).
%...x	unsigned int *	Numero intero senza segno rappresentato in esadecimale (con o senza il prefisso '0x' o '0X').
%...c	char *	Interpreta un solo carattere, o più caratteri se si specifica l'ampiezza. Nella lettura contano anche gli spazi o qualunque altro carattere e non viene aggiunto il carattere nullo di terminazione.
%...s	char *	Interpreta una sequenza di caratteri che non siano spazi, aggiungendo alla fine il carattere nullo di terminazione.
%...a %...e %...f %...g	double *	Un numero a virgola mobile rappresentato in notazione decimale fissa o in notazione esponenziale: $\begin{bmatrix} - \end{bmatrix} iii . dddddd$ $\begin{bmatrix} - \end{bmatrix} i . dddddd e \pm xx$ $\begin{bmatrix} - \end{bmatrix} i . dddddd E \pm xx$
%p	void *	Interpreta il valore di un puntatore che sia rappresentato nello stesso modo in cui farebbe la funzione 'printf("%p", <i>puntatore</i> )'.

Simbolo	Tipo di argomento	Conversione applicata
<code>%n</code>	<code>int *</code>	Questo specificatore non esegue alcuna conversione e si limita a memorizzare la quantità di caratteri ( <b>'char'</b> ) letti fino a quel punto.
<code>%... [...]</code>	<code>char *</code>	Interpreta una stringa non vuota contenente solo i caratteri elencati tra parentesi quadre, aggiungendo alla fine il carattere nullo di terminazione. Se tra i caratteri si cerca anche la parentesi quadra chiusa, questa va messa all'inizio dell'elenco: <code>'%... [...]...'</code> .
<code>%... [^...]</code>	<code>char *</code>	Interpreta una stringa non vuota contenente solo caratteri diversi da quelli elencati tra parentesi quadre, aggiungendo alla fine il carattere nullo di terminazione. Se tra i caratteri da escludere si vuole indicare anche la parentesi quadra chiusa, questa va messa all'inizio dell'elenco: <code>'%... [^]...'</code> .
<code>%%</code>		Interpreta un carattere di percentuale tra i dati in ingresso, ma senza memorizzare alcunché.

Tabella 69.268. Alcuni modificatori della lunghezza del dato in uscita.

Simbolo	Tipo	Simbolo	Tipo
%...hhd %...hhi	signed char *	%...hhu %...hho %...hhx %...hhn	unsigned char *
%...hd %...hi	short int *	%...hu %...ho %...hx %...hn	unsigned short int *
%...ld %...li	long int *	%...lu %...lo %...lx %...ln	unsigned long int *
		%...lc %...ls %...lc %...l [...]	wchar_t *

Simbolo	Tipo	Simbolo	Tipo
%lld %lli	long long int *	%llu %llo %llx %lln	unsigned long long int *
%jd %ji	intmax_t *	%ju %jo %jx %jn	uintmax_t *
%zd %zi	size_t *	%zu %zo %zx %zn	size_t *
%td %ti	ptrdiff_t *	%tu %to %tx %tn	ptrdiff_t *

Simbolo	Tipo	Simbolo	Tipo
%...Le			
%...Lf	long double *		
%...Lg			

A proposito dell'interpretazione di caratteri e di stringhe, va precisato cosa accade quando si usa il modificatore 'l' (elle). Se nello specificatore di conversione appare un valore numerico che esprime un'ampiezza, questa indica una quantità di caratteri, in ingresso, da intendersi come byte. Utilizzando gli specificatori '%...lc' e '%...ls', la quantità di questi caratteri continua a riferirsi a byte, ma si interpretano le sequenze multibyte in ingresso per generare caratteri di tipo 'wchar\_t'.

Il documento che descrive lo standard del linguaggio afferma che la stringa di conversione è composta da direttive, ognuna delle quali è formata da: uno o più spazi (spazi veri e propri o caratteri di tabulazione orizzontale); un carattere multibyte diverso da '%' e diverso dai caratteri che rappresentano spazi, oppure uno specificatore di conversione.

[*spazi*] *carattere\_multibyte* | %...

Dalla sequenza multibyte che costituisce i dati in ingresso da interpretare, vengono eliminati automaticamente gli spazi iniziali e finali (tutto ciò che si può considerare spazio, anche il codice di interruzione di riga), quando all'inizio o alla fine non ci sono corrispondenze con specificatori di conversione che possono interpretarli.

Quando la direttiva di interpretazione inizia con uno o più spazi orizzontali, significa che si vogliono ignorare gli spazi a partire dalla posizione corrente nella lettura dei dati in ingresso; inoltre, la presenza di un carattere che non fa parte di uno specificatore di conversione indica che quello stesso carattere deve essere incontrato nell'interpretazione dei dati in ingresso, altrimenti il procedimento di lettura e valutazione si deve interrompere. Se due specificatori di conversione appaiono adiacenti, i dati in ingresso corrispondenti possono essere separati da spazi orizzontali o da spazi verticali (il codice di interruzione di riga).

Purtroppo, la sintassi per la scrittura delle stringhe di conversione non è molto soddisfacente e diventa difficile spiegarne il comportamento, a meno di rimanere fermi su esempi molto semplici.

### 69.17.10.1 Funzioni che ricevono gli argomenti direttamente

Un gruppo di funzioni per l'interpretazione dell'input riceve direttamente gli argomenti variabili che servono agli specificatori di conversione:

```
int fscanf (FILE *restrict stream,  
            const char *restrict format, ...);  
int sscanf (const char *restrict s,  
            const char *restrict format, ...);  
int scanf  (const char *restrict format, ...);
```

Tutte le funzioni di questo gruppo hanno in comune la stringa di conversione, costituita dal parametro *format*, e gli argomenti suc-

cessivi che sono puntatori di tipo indeterminato, in quanto per la loro interpretazione contano gli specificatori di conversione inseriti nella stringa. Inoltre, tutte queste funzioni restituiscono la quantità di valori assegnati alle variabili rispettive, oppure il valore corrispondente alla macro-variabile *EOF* nel caso si verifichi un errore prima di qualunque conversione.

La funzione *sscanf()* scandisce il contenuto della stringa indicata come primo parametro (*s*); la funzione *fscanf()* scandisce l'input proveniente dal flusso di file indicato come primo argomento (*stream*), mentre la funzione *scanf()* scandisce direttamente lo standard input.

## 69.17.10.2 Funzioni che ricevono gli argomenti da un'altra funzione

«

A fianco delle funzioni descritte nella sezione precedente, un gruppo analogo svolge le stesse operazioni, ma ricevendo gli argomenti variabili per riferimento. In pratica si tratta di ciò che serve quando gli argomenti variabili sono stati ottenuti da un'altra funzione e non da una chiamata diretta.

```
int vfscanf (FILE *restrict stream,
            const char *restrict format,
            va_list arg);
int vsscanf (const char *restrict s,
            const char *restrict format,
            va_list arg);
int vscanf (const char *restrict format,
            va_list arg);
```

Il funzionamento è conforme a quello delle funzioni che non hanno la lettera 'v' iniziale; per esempio, *vscanf()* si comporta conformemente a *scanf()*. Per comprendere la differenza si potrebbe dimo-



strare la realizzazione ipotetica della funzione *scanf()* avvalendosi di *vscanf()*:

```
int scanf (const char *restrict format, ...);
{
    va_list arg;
    va_start (arg, format);
    int count;
    count = vscanf (format, arg);
    va_end (arg);
    return count;
}
```

### 69.17.11 Funzioni per la lettura e la scrittura di un carattere alla volta

Le funzioni *fgetc()* e *getc()* leggono un carattere (*char*) attraverso il flusso di file indicato come argomento: <<

```
int fgetc (FILE *stream);

#define getc(STREAM) (fgetc (STREAM))
```

Lo standard prescrive che la funzione *getc()* sia in realtà una macroistruzione, così come si ipotizza nella dichiarazione appena mostrata. A questo proposito occorre tenere presente che, se si usa *getc()*, l'espressione usata per individuare il flusso di file potrebbe essere valutata più di una volta.

Il carattere letto da *fgetc()* viene interpretato senza segno e trasformato in un intero (pertanto deve risultare essere di segno positivo). Se viene tentata la lettura oltre la fine del file, la funzione restituisce il valore rappresentato da *EOF* e memorizza questa condizione nella variabile strutturata che rappresenta il flusso di file. Se invece si

verifica un errore di lettura, viene impostato il contenuto dell'indicatore di errore relativo al flusso di file e la funzione restituisce sempre il valore *EOF*.

Secondo lo standard, la funzione *getchar()* è equivalente a '*getc (stdin)*', senza specificare altro. Ciò può significare ragionevolmente che se *getc()* è una macroistruzione, anche *getchar()* dovrebbe esserlo, altrimenti potrebbe trattarsi di una funzione vera e propria:

```
#define getchar (getc (stdin))
```

La funzione *ungetc()* ha lo scopo di annullare l'effetto della lettura dell'ultimo carattere, ma il modo in cui viene gestita la cosa rende la questione molto delicata:

```
int ungetc (int c, FILE *stream);
```

Semplificando il problema, la funzione *ungetc()* rimanda indietro il carattere *c* nel flusso di file *stream* dal quale è appena stata eseguita una lettura. Tuttavia, non è garantito che il carattere in questione sia effettivamente quello che è stato letto per ultimo, ma la fase successiva di lettura deve fornire per primo tale carattere.

Si comprende intuitivamente che, se si eseguono operazioni di spostamento della posizione corrente relativa al flusso di file in questione, il carattere rimandato indietro con la funzione *ungetc()* debba essere dimenticato, soprattutto se questo non corrispondeva a quello che effettivamente era stato letto per ultimo in quel momento.

L'uso della funzione *ungetc()* implica un aggiornamento della posizione corrente relativa al flusso di file, ma questa modifica, in presenza di file di testo che non siano realizzati secondo lo standard tradizionale dei sistemi Unix, implica che l'entità di questa modifica

non possa essere predeterminabile.<sup>11</sup>

La funzione *ungetc()* può fallire nel suo intento e lo standard prescrive che sia «garantita» la possibilità di rimandare indietro almeno un carattere. Se la funzione riesce a eseguire l'operazione, restituisce il valore positivo corrispondente al carattere rinviato; altrimenti restituisce il valore della macro-variabile *EOF*.

Le funzioni *fputc()*, *putc()* e *putchar()* eseguono l'operazione inversa, rispettivamente, di *fgetc()*, *getc()* e *getchar()*; anche in questo caso vale il fatto che *putc()* possa essere realizzata come macroistruzione:

```
int fputc (int c, FILE *stream);

#define putc (CHAR, STREAM) (fputc ((CHAR), (STREAM)))

#define putchar (CHAR) (putc ((CHAR), stdout))
```

La funzione *fputc()* scrive un carattere (fornito come numero intero positivo) attraverso il flusso di file indicato; *putc()* fa lo stesso, ma potrebbe essere una macroistruzione; *putchar()* scrive attraverso lo standard output.

Se la scrittura fallisce, le funzioni (o le macroistruzioni) restituiscono il valore *EOF*; diversamente restituiscono il valore positivo corrispondente al carattere scritto.

Per come sono state proposte queste funzioni, non c'è differenza nell'uso di *getc()* al posto di *fgetc()*, così come tra *putc()* e *fputc()*. Evidentemente, se la propria libreria può esprimere le macroistruzioni *getc()* e *putc()* richiamando funzioni del sistema operativo (funzioni che dovrebbero essere richiamate anche da *fgetc()* e *fputc()*), si può risparmiare un livello di chiamate per accelerare leggermente l'esecuzione del programma.

### 69.17.12 Funzioni per l'input e l'output di file di testo

«

Le funzioni *fgets()* e *fputs()* sono utili per l'accesso a file di testo, quando si vuole indicare il flusso di file:

```
char *fgets (char *restrict s, int n,  
             FILE *restrict stream);  
  
int   fputs (const char *restrict s, FILE *restrict stream);
```

La funzione *fgets()* legge al massimo  $n-1$  caratteri (nel senso di elementi '**char**') attraverso il flusso di file *stream*, copiandoli in memoria a partire dall'indirizzo *s* e aggiungendo alla fine il carattere nullo di terminazione delle stringhe. La lettura si esaurisce prima di  $n-1$  caratteri se viene incontrato il codice di interruzione di riga, il quale viene rappresentato nella stringa a cui punta *s*, ovvero se si raggiunge la fine del file. In ogni caso, la stringa *s* viene terminata correttamente con il carattere nullo.

La funzione *fputs()* restituisce la stringa *s* se la lettura avviene con successo, ovvero se ha prodotto almeno un carattere; altrimenti, il contenuto dell'array a cui punta *s* non viene modificato e la funzione restituisce il puntatore nullo. Se si creano errori imprevisti, la

funzione potrebbe restituire il puntatore nullo, ma senza garantire che l'array *s* sia rimasto intatto.

La funzione *fputs()* serve a copiare la stringa a cui punta *s* nel file rappresentato dal flusso di file *stream*. La copia della stringa avviene escludendo però il carattere nullo di terminazione. Va osservato che questa funzione, pur essendo contrapposta evidentemente a *fgets()*, **non conclude la riga** del file, ovvero, non aggiunge il codice di interruzione di riga. Per ottenere la conclusione della riga di un file di testo, occorre inserire nella stringa, espressamente, il carattere '\n'.

La funzione *fputs()* restituisce il valore rappresentato da *EOF* se l'operazione di scrittura produce un errore; altrimenti restituisce un valore positivo qualunque.

Le funzioni *gets()* e *puts()* sono utili per l'accesso a file di testo, quando si vogliono utilizzare i flussi standard. In linea di massima, assomigliano a *fgets()* e *fputs()*, ma il funzionamento non è perfettamente conforme a quelle:

```
char *gets (char *s);  
  
int  puts (const char *s);
```

Il funzionamento di *gets()* è perfettamente conforme a quello di *fgets()*, con la sola differenza che il flusso di file da cui si leggono i caratteri è lo standard input. Nel caso di *puts()*, a parte il fatto che si usa lo standard output per la scrittura, occorre sottolineare che alla fine della stringa **viene accodata la scrittura del codice di interruzione di riga**.

## 69.17.13 Funzioni per l'input e output diretto



Le funzioni *fread()* e *fwrite()* consentono di leggere e scrivere attraverso un flusso di file aperto, il quale deve essere specificato espressamente tra gli argomenti. Lo standard prescrive che queste funzioni si avvalgano rispettivamente di *fgetc()* e di *fputc()*.

```
size_t fread (void *restrict ptr,  
             size_t size,  
             size_t nmemb,  
             FILE *restrict stream);  
  
size_t fwrite (const void *restrict ptr,  
             size_t size,  
             size_t nmemb,  
             FILE *restrict stream);
```

Le due funzioni (*fread()* e *fwrite()*) hanno praticamente gli stessi argomenti, usati in modo analogo. La lettura e la scrittura avviene a blocchi da *size* byte, ripetuta per *nmemb* volte, attraverso il flusso di file specificato come *stream*. La lettura implica la memorizzazione dei caratteri in forma di elementi 'unsigned char', a partire dall'indirizzo indicato dal puntatore *ptr*; la scrittura copia nello stesso modo i caratteri a partire dal puntatore *ptr*, verso il flusso di file.

L'aggiornamento della posizione corrente interna al file a cui si riferisce il flusso avviene esattamente come per le funzioni *fgetc()* e *fputc()*.

Il valore restituito dalle funzioni *fread()* e *fwrite()* rappresenta la quantità di blocchi, ovvero la quantità di elementi *nmemb* che sono stati copiati con successo. Pertanto, se si ottiene un valore inferiore

a *nmemb*, significa che l'operazione è stata interrotta a causa di un errore.

### 69.17.14 Funzioni per il posizionamento

Sono previste diverse funzioni per modificare la posizione corrente dei flussi di file. Le funzioni più semplici per iniziare sono *fseek()*, *ftell()* e *rewind()*:

```
int      fseek   (FILE *stream, long int offset, int whence);
long int ftell   (FILE *stream);
void     rewind  (FILE *stream);
```

Lo standard POSIX prevede anche le funzioni *fseeko()* e *ftello()*, equivalenti alle funzioni *fseek()* e *ftell()* dello standard C, con la differenza che lo scostamento, fornito come argomento o restituito dalla funzione, è di tipo '*off\_t*', al posto di essere di tipo '*long int*':

```
int      fseeko (FILE *stream, off_t offset, int whence);
off_t    ftello (FILE *stream);
```

In generale, le funzioni *fseek()* e *fseeko()* spostano la posizione corrente relativa al flusso di file *stream*, nella nuova posizione determinata dai parametri *whence* e *offset*. Il parametro *whence* viene fornito attraverso una macro-variabile che può essere *SEEK\_SET*, *SEEK\_CUR* o *SEEK\_END*, indicando rispettivamente l'inizio del file, la posizione corrente o la fine del file. Dalla posizione indicata dal parametro *whence* viene aggiunta, algebricamente, la quantità di byte indicata dal parametro *offset*.

Quanto descritto a proposito del posizionamento con le funzioni *fseek()* e *fseeko()* riguarda i file che vengono gestiti in modo binario, perché con i file di testo è opportuno avere maggiore accortezza:



il valore del parametro *offset* deve essere zero, oppure quanto restituito in precedenza dalle funzioni *ftell()* o *ftello()* per lo stesso flusso di file, ma in tal caso, ovviamente, il parametro *whence* deve corrispondere a *SEEK\_SET*.

Le funzioni *fseek()* e *fseeko()* restituiscono zero se possono eseguire l'operazione, altrimenti danno un risultato diverso; nel caso di *fseeko()*, quando si presenta un errore, il risultato restituito è precisamente  $-1$ , e in più viene aggiornata anche la variabile *errno*.

Le funzioni *ftell()* e *ftello()* restituiscono la posizione corrente del flusso di file indicato come argomento. Questo valore può essere usato con *fseek()* o *fseeko()* rispettivamente, al posto dello scostamento (il parametro *offset*), indicando come posizione di riferimento l'inizio del file, ovvero *SEEK\_SET*. Se le funzioni *ftell()* e *ftello()* non riescono a fornire la posizione, restituiscono il valore  $-1$  (tradotto rispettivamente in 'long int' e 'off\_t') e annotano il fatto nella variabile *errno*.

La funzione *rewind()* si limita a riposizionare il flusso di file all'inizio. In pratica è come utilizzare la funzione *fseek()* specificando uno scostamento pari a zero a partire da *SEEK\_SET*, ignorando il valore restituito:

```
(void) fseek (stream, 0L, SEEK_SET)
```

Va osservato che il riposizionamento di un flusso di file implica l'azzeramento dell'indicatore di fine file, se questo risulta impostato, e la cancellazione dei caratteri che eventualmente fossero stati rimandati indietro con la funzione *ungetc()*.

Le funzioni *fseek()* e *ftell()* sono utili particolarmente per i file bi-



nari ed eventualmente per i file di testo con una rappresentazione dei caratteri tradizionale. Ma quando il file di testo contiene anche caratteri espressi attraverso sequenze multibyte, il posizionamento al suo interno dovrebbe tenere anche conto del progresso nell'interpretazione di queste sequenze. Pertanto, esistono altre due funzioni per leggere la posizione e ripristinarla in un secondo momento:

```
int fgetpos (FILE *restrict stream, fpos_t *restrict pos);
int fsetpos (FILE *stream, const fpos_t *pos);
```

Entrambe le funzioni che appaiono nei due prototipi restituiscono zero se l'operazione è stata compiuta con successo, altrimenti restituiscono un valore differente. Nel caso particolare di *fsetpos()*, se si verifica un errore, questo viene annotato nella variabile *errno*.

Le due funzioni richiedono come primo argomento il flusso di file a cui ci si riferisce; come secondo argomento richiedono il puntatore a una variabile di tipo '**fpos\_t**'. La funzione *fgetpos()* memorizza nella variabile a cui punta il parametro *pos* le informazioni sulla posizione corrente del file, assieme allo stato di interpretazione relativo alle sequenze multibyte; la funzione *fsetpos()*, per converso, utilizza la variabile a cui punta *pos* per ripristinare la posizione memorizzata, assieme allo stato di avanzamento dell'interpretazione di una sequenza multibyte.

### 69.17.15 Accesso esclusivo ai flussi di file

Lo standard POSIX, introducendo la gestione dei thread multipli, inserisce nel file '`stdio.h`' alcune funzioni per controllare l'accesso esclusivo ai flussi di file. Va tenuto a mente che si tratta di un controllo che riguarda esclusivamente il processo elaborativo in corso, e non l'accesso ai file da parte di processi differenti.



```
void flockfile (FILE *stream);  
int  trylockfile (FILE *stream);  
void funlockfile (FILE *stream);
```

La funzione *flockfile()* cerca di ottenere un accesso esclusivo a un file, individuato da un puntatore che rappresenta il flusso di file relativo. Se il file risulta già impegnato, rimane in attesa, fino a quando si libera. La funzione *trylockfile()*, invece, non rimane in attesa e restituisce l'esito della sua azione: zero se è riuscita a ottenere l'accesso esclusivo, un numero diverso se invece non c'è riuscita. Quando poi un thread che aveva ottenuto l'accesso esclusivo a un flusso di file, non ne ha più bisogno, lo libera con la funzione *funlockfile()*.

Le funzioni *getc()*, *putc()*, *getchar()* e *putchar()*, quando sono presenti le estensioni per la gestione dei thread multipli, e di conseguenza anche per l'accesso esclusivo ai flussi di file, si comportano rispettando tali vincoli. Eventualmente, se per qualche ragione si vogliono usare queste funzionalità, ignorando espressamente tali vincoli, sono disponibili funzioni equivalenti, il cui nome termina per **'\_unlocked'**:

```
int getc_unlocked (FILE *stream);  
int putc_unlocked (int c, FILE *stream);  
int getchar_unlocked (void);  
int putchar_unlocked (int c);
```

Lo standard POSIX prescrive comunque che queste siano usate solo da thread che hanno già ottenuto un accesso esclusivo al flusso relativo, in modo da non compromettere la gestione controllata di tali accessi. Pertanto, in tal modo queste funzioni consentono semplicemente un'esecuzione più rapida, dal momento che non vengono eseguiti tutti i controlli necessari.

## 69.17.16 Condotti

Lo standard POSIX introduce il concetto di «condotto», ovvero di *pipe*, attraverso il quale è possibile inviare lo standard output di un processo elaborativo, verso lo standard input di un altro. Per attuare questo meccanismo, è necessario che un processo sia in grado di avviare un altro processo, attraverso un comando da dare alla shell, e da questo processo viene poi letto lo standard output o scritto lo standard input.

Dal punto di vista del processo che crea il condotto, il processo secondario avviato è trattato come se fosse un file, dove però si può solo leggere o scrivere, ma non si possono fare entrambe le cose.

```
FILE *popen (const char *command, const char *type);  
int  pclose (FILE *stream);
```

Il condotto viene aperto con la funzione *popen()*, la quale assomiglia a *fopen()*, ma invece dell'indicazione del percorso del file da aprire, richiede il comando da eseguire attraverso la shell `"/bin/sh"` (nota come la shell POSIX). Per quanto riguarda il tipo di accesso, va osservato che può trattarsi soltanto di lettura o scrittura, pertanto si può scegliere solo tra la stringa `"r"` o `"w"`.

La lettura o la scrittura in un flusso di file associato a un condotto avviene nel modo consueto, ma la chiusura richiede l'uso della funzione *pclose()*.

## 69.17.17 Informazioni sul terminale

Lo standard POSIX prevede che il sistema operativo abbia una gestione dei file di dispositivo per rappresentare i vari componenti fisici dell'elaboratore. Nel file `'stdio.h'` inserisce la funzione *ctermid()*,

con lo scopo di conoscere il percorso del file di dispositivo del terminale associato al processo elaborativo.

```
char *ctermid (char *s);
```

La funzione si aspetta di ricevere come argomento il puntatore a una stringa modificabile, in cui scrivere il percorso del terminale. Se però viene fornito un puntatore nullo, l'informazione viene annotata in un'area di memoria che può essere statica (e quindi riutilizzata a ogni chiamata della funzione). Il percorso del terminale, annotato dalla funzione, può utilizzare al massimo la quantità di caratteri definita dalla macro-variabile *L\_ctermid*; pertanto, se si definisce un array di caratteri da usare per tale annotazione, deve essere almeno quella dimensione.

La funzione restituisce sempre il puntatore a una stringa, che può essere nulla se non è possibile determinare il terminale. Pertanto la funzione non prevede l'indicazione di errori.

### 69.17.18 Gestione degli errori

«

Un gruppo di funzioni di 'stdio.h' consente di verificare ed eventualmente azzerare lo stato degli indicatori di errore riferiti a un certo flusso di file:

```
void clearerr (FILE *stream);  
int feof (FILE *stream);  
int ferror (FILE *stream);  
void perror (const char *s);
```

La funzione *clearerr()* azzerà gli indicatori di errore e di fine file per il flusso di file indicato come argomento, senza restituire alcunché.

La funzione *feof()* controlla lo stato dell'indicatore di fine file per il flusso di file indicato. Se questo non è attivo restituisce zero,

altrimenti restituisce un valore diverso da zero.

La funzione *ferror()* controlla lo stato dell'indicatore di errore per il flusso di file indicato. Se questo non è attivo restituisce zero, altrimenti restituisce un valore diverso da zero.

La funzione *perror()* prende in considerazione la variabile *errno* e cerca di tradurla in un messaggio testuale da emettere attraverso lo standard error (con tanto di terminazione della riga, in modo da riposizionare a capo il cursore). Se il parametro *s* corrisponde a una stringa non vuota, il testo di questa viene posto anteriormente al messaggio, separandolo con due punti e uno spazio (': '). Il contenuto del messaggio è lo stesso che si otterrebbe con la funzione *strerror()*, fornendo come argomento la variabile *errno*.

### 69.17.19 Realizzazione di «vsnprintf()» e altre collegate

Le funzioni per la composizione dell'output che possono essere realizzate senza avere definito la gestione dei file, sono quelle che si limitano a produrre una stringa. La funzione che va realizzata per prima è *vsnprintf()*, in quanto *snprintf()* si può limitare a richiamarla. Naturalmente, anche *vsprintf()* e *sprintf()* possono avvalersi della stessa *vsnprintf()*, ponendo un limite massimo abbastanza grande alla stringa da generare. Nella sezione [95.18.42](#) è disponibile un esempio di realizzazione parziale di *vsnprintf()*. L'esempio seguente mostra come si ottiene *snprintf()*, una volta che è disponibile *vsnprintf()*:

```
#include <stdio.h>
int
snprintf (char *restrict string, size_t n,
          const char *restrict format, ...)
{
```

```

va_list ap;
va_start (ap, format);
return vsnprintf (string, n, format, ap);
}

```

Eventualmente, per realizzare le funzioni *vsprintf()* e *sprintf()*, secondo le limitazioni già descritte, sono sufficienti due macroistruzioni:

```

#define vsprintf(s, format, arg) \
    (vsnprintf (s, SIZE_MAX, format, arg))
#define sprintf(s, ...) \
    (snprintf (s, SIZE_MAX, __VA_ARGS__))

```

## 69.18 Riferimenti

«

- Wikipedia, *C standard library*, [http://en.wikipedia.org/wiki/C\\_standard\\_library](http://en.wikipedia.org/wiki/C_standard_library)
- ISO/IEC 9899:TC2, <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf>
- Wikipedia, *assert.h*, *limits.h*, *stdint.h*, *errno.h*, *locale.h*, *ctype.h*, *stdarg.h*, *stdlib.h*, *inttypes.h*, *iso646.h*, *stdbool.h*, *stddef.h*, *string.h*, *signal.h*, *time.h*, *stdio.h*

<http://en.wikipedia.org/wiki/Assert.h> , <http://en.wikipedia.org/wiki/Limits.h> , <http://en.wikipedia.org/wiki/Stdint.h> , <http://en.wikipedia.org/wiki/Errno.h> , <http://en.wikipedia.org/wiki/Locale.h> , <http://en.wikipedia.org/wiki/Ctype.h> , <http://en.wikipedia.org/wiki/Stdarg.h> , <http://en.wikipedia.org/wiki/Stdlib.h> , <http://en.wikipedia.org/wiki/Inttypes.h> , <http://en.wikipedia.org/wiki/Iso646.h> , <http://en.wikipedia.org/wiki/Signal.h> , <http://en.wikipedia.org/wiki/Time.h> , <http://en.wikipedia.org/wiki/Stdio.h>

[Stdbool.h](http://en.wikipedia.org/wiki/Stdbool.h) , <http://en.wikipedia.org/wiki/Stddef.h> , <http://en.wikipedia.org/wiki/String.h> , <http://en.wikipedia.org/wiki/Signal.h> , <http://en.wikipedia.org/wiki/Time.h> , <http://en.wikipedia.org/wiki/Stdio.h>

- The Open Group, *The Single UNIX® Specification, Version 2, assert.h, limits.h, stdint.h, errno.h, locale.h, ctype.h, stdarg.h, stdlib.h, inttypes.h, iso646.h, stdbool.h, stddef.h, string.h, signal.h, time.h, stdio.h*

<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/assert.h.html> ,

<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/limits.h.html> ,

<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdint.h.html> ,

<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/errno.h.html> ,

<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/locale.h.html> ,

<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/ctype.h.html> ,

<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdarg.h.html> ,

<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdlib.h.html> ,

<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/inttypes.h.html> ,

<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/iso646.h.html> ,

<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdbool.h.html> ,  
<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stddef.h.html> ,  
<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/string.h.html> ,  
<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/signal.h.html> ,  
<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/time.h.html> ,  
<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdio.h.html>

- Steven Pemberton, *Enquire: Everything you wanted to know about your C Compiler and Machine, but didn't know who to ask*, <http://homepages.cwi.nl/~steven/enquire.html>

<sup>1</sup> Il carattere viene convertito da `'unsigned char'` a `'int'`.

<sup>2</sup> Si tratta di un puntatore di puntatore, solo perché si deve poter alterare ciò a cui punta, ma questo tipo di valore è, a sua volta, un puntatore.

<sup>3</sup> Il tipo `'size_t'`, restituito dalle funzioni, è un intero senza segno; pertanto, in condizioni normali, ovvero con una rappresentazione dei valori negativi con il complemento a due, la conversione di `-1` si traduce nel valore massimo rappresentabile.

<sup>4</sup> Lo standard POSIX non estende il contenuto del file `'inttypes.h'`.

<sup>5</sup> Lo standard POSIX non estende il file `'iso646.h'`.



- <sup>6</sup> Lo standard POSIX non estende il file `'stdbool.h'`
- <sup>7</sup> Lo standard POSIX non estende il file `'stddef.h'`.
- <sup>8</sup> Il valore positivo massimo è  $(2^{31})-1$ , il quale, diviso per la quantità di secondi di un giorno (86400) dà 24855 che, diviso 365, dà circa 68 anni.
- <sup>9</sup> Il caso della funzione *clock()* e del tipo `'clock_t'` è stato considerato a parte.
- <sup>10</sup> Si tratta di byte: se il testo copiato è costituito da sequenze multi-byte, i byte sono in quantità maggiore rispetto ai caratteri tipografici che si ottengono.
- <sup>11</sup> L'arretramento di un carattere nella posizione corrente di un file di testo non è detto corrisponda alla sottrazione di una unità, perché bisogna tenere in considerazione il modo in cui un file di testo è strutturato nel proprio sistema operativo.



## Libreria POSIX



70.1	File «sys/types.h» .....	1312
70.2	File «sys/stat.h» .....	1315
70.2.1	Macro-variabili per la definizione del contenuto di un valore di tipo «mode_t» 1316	
70.2.2	Struttura «stat» .....	1319
70.2.3	Prototipi di funzione .....	1320
70.3	File «strings.h» .....	1321
70.3.1	Funzione «ffs()» .....	1321
70.3.2	Funzioni «strcasecmp()» e «strncasecmp()» ...	1322
70.4	File «fcntl.h» .....	1322
70.4.1	Tipi di dati derivati .....	1323
70.4.2	Opzioni per la funzione «fcntl()» 1323	
70.4.3	Gestione del blocco dei file .....	1326
70.4.4	Funzioni .....	1327
70.5	File «unistd.h» .....	1329
70.5.1	Denominazione dei descrittori di file .....	1330
70.5.2	Verifica dei permessi di accesso .....	1330
70.5.3	Funzioni 1331	

70.5.4	Funzioni di servizio per la gestione di file e directory	1333			
70.6	File «dirent.h»	1342			
70.6.1	Struttura «dirent»	1343			
70.6.2	Tipo «DIR»	1343			
70.6.3	Prototipi di funzioni	1344			
70.7	File «termios.h»	1345			
70.7.1	Tipi speciali	1345			
70.7.2	Tipo «struct termios»	1346			
70.7.3	Codici di controllo	1347			
70.7.4	Indicatori per il membro «c_iflag»	1347			
70.7.5	Indicatori per il membro «c_oflag»	1348			
70.7.6	Indicatori per il membro «c_cflag»	1348			
70.7.7	Indicatori per il membro «c_lflag»	1349			
70.7.8	Funzioni	1349			
70.8	Riferimenti	1350			
access()	1331 1333	alarm()	1331	blkcnt_t	1312
blksize_t	1312	BRKINT	1347	cc_t	1345
chdir()	1331	chmod()	1320	chown()	1331
clock_t	1312	close()	1331	closedir()	1344
confstr()	1331	creat()	1327	dev_t	1312
DIR	1343	dirent.h	1342	dup()	1331
dup2()	1331	ECHO	1349	ECHOE	1349
ECHOK	1349	ECHONL	1349	execl()	1331
execle()	1331	execlp()	1331	execv()	1331
execve()	1331	execvp()	1331	fchmod()	1320
fchown()	1331	fcntl()	1327	fcntl.h	1322
FD_CLOEXEC					

1323 ffs() 1321 fork() 1331 fpathconf() 1331 fstat()  
1320 ftruncate() 1331 F\_DUPFD 1323 F\_GETFD 1323  
F\_GETFL 1323 F\_GETLK 1323 F\_GETOWN 1323 F\_OK 1330  
F\_RDLCK 1326 F\_SETFD 1323 F\_SETFL 1323 F\_SETLK 1323  
F\_SETLKW 1323 F\_SETOWN 1323 F\_UNLCK 1326 F\_WRLCK  
1326 getcwd() 1331 1337 getegid() 1331 geteuid()  
1331 getgid() 1331 getgroups() 1331 gethostname()  
1331 getlogin() 1331 getlogin\_r() 1331 getopt()  
1331 getpgrp() 1331 getpid() 1331 getppid() 1331  
getuid() 1331 gid\_t 1312 ICANON 1349 ICRNL 1347 id\_t  
1312 IEXTEN 1349 IGNBRK 1347 IGNCR 1347 IGNPAR 1347  
INLCR 1347 ino\_t 1312 INPCK 1347 isatty() 1331 ISIG  
1349 ISTRIP 1347 IXOFF 1347 IXON 1347 link() 1331 1340  
lseek() 1331 lstat() 1320 mkdir() 1320 mkfifo() 1320  
mknod() 1320 mode\_t 1312 1316 NCCS 1346 nlink\_t 1312  
NOFLSH 1349 off\_t 1312 open() 1327 opendir() 1344  
OPOST 1348 O\_ACCMODE 1323 O\_APPEND 1323 O\_CREAT 1323  
O\_DSYNC 1323 O\_EXCL 1323 O\_NOCTTY 1323 O\_NONBLOCK  
1323 O\_RDONLY 1323 O\_RDWR 1323 O\_RSYNC 1323 O\_SYNC  
1323 O\_TRUNC 1323 O\_WRONLY 1323 PARMRK 1347  
pathconf() 1331 pause() 1331 pid\_t 1312 pipe() 1331  
pthread\_t 1312 read() 1331 readdir() 1344  
readlink() 1331 rewinddir() 1344 rmdir() 1331 R\_OK  
1330 setegid() 1331 seteuid() 1331 setgid() 1331  
setpgrp() 1331 setsid() 1331 setuid() 1331 size\_t  
1312 sleep() 1331 speed\_t 1345 ssize\_t 1312 stat()  
1320 stat.h 1315 STDERR\_FILENO 1330 STDIN\_FILENO  
1330 STDOUT\_FILENO 1330 strcasecmp() 1322  
strings.h 1321 strncasecmp() 1322 structure stat

[1319](#) struct dirent [1343](#) struct termios [1346](#)  
[st\\_atime](#) [1319](#) [st\\_blksize](#) [1319](#) [st\\_blocks](#) [1319](#)  
[st\\_ctime](#) [1319](#) [st\\_dev](#) [1319](#) [st\\_gid](#) [1319](#) [st\\_ino](#) [1319](#)  
[st\\_mode](#) [1319](#) [st\\_mtime](#) [1319](#) [st\\_nlink](#) [1319](#) [st\\_rdev](#)  
[1319](#) [st\\_size](#) [1319](#) [st\\_uid](#) [1319](#) [symlink\(\)](#) [1331](#)  
[sys/types.h](#) [1312](#) [sysconf\(\)](#) [1331](#) [S\\_IFBLK](#) [1316](#)  
[S\\_IFCHR](#) [1316](#) [S\\_IFDIR](#) [1316](#) [S\\_IFIFO](#) [1316](#) [S\\_IFLNK](#) [1316](#)  
[S\\_IFMT](#) [1316](#) [S\\_IFREG](#) [1316](#) [S\\_IFSOCK](#) [1316](#) [S\\_IRGRP](#) [1316](#)  
[S\\_IROTH](#) [1316](#) [S\\_IRUSR](#) [1316](#) [S\\_IRWXG](#) [1316](#) [S\\_IRWXO](#) [1316](#)  
[S\\_IRWXU](#) [1316](#) [S\\_ISBLK\(\)](#) [1316](#) [S\\_ISCHR\(\)](#) [1316](#)  
[S\\_ISDIR\(\)](#) [1316](#) [S\\_ISFIFO\(\)](#) [1316](#) [S\\_ISGID](#) [1316](#)  
[S\\_ISLNK\(\)](#) [1316](#) [S\\_ISREG\(\)](#) [1316](#) [S\\_ISSOCK\(\)](#) [1316](#)  
[S\\_ISUID](#) [1316](#) [S\\_ISVTX](#) [1316](#) [S\\_IWGRP](#) [1316](#) [S\\_IWOTH](#) [1316](#)  
[S\\_IWUSR](#) [1316](#) [S\\_IXGRP](#) [1316](#) [S\\_IXOTH](#) [1316](#) [S\\_IXUSR](#) [1316](#)  
[tcflag\\_t](#) [1345](#) [tcgetattr\(\)](#) [1349](#) [tcgetpgrp\(\)](#) [1331](#)  
[TCSADRAIN](#) [1349](#) [TCSAFLUSH](#) [1349](#) [TCSANOW](#) [1349](#)  
[tcsetattr\(\)](#) [1349](#) [tcsetpgrp\(\)](#) [1331](#) [termios.h](#) [1345](#)  
[time\\_t](#) [1312](#) [TOSTOP](#) [1349](#) [ttyname\(\)](#) [1331](#) [ttyname\\_r\(\)](#)  
[1331](#) [uid\\_t](#) [1312](#) [umask\(\)](#) [1320](#) [unistd.h](#) [1329](#) [unlink\(\)](#)  
[1331](#) [1340](#) [VEOF](#) [1347](#) [VEOL](#) [1347](#) [VERASE](#) [1347](#) [VINTR](#) [1347](#)  
[VKILL](#) [1347](#) [VMIN](#) [1347](#) [VQUIT](#) [1347](#) [VSTART](#) [1347](#) [VSTOP](#)  
[1347](#) [VSUSP](#) [1347](#) [VTIME](#) [1347](#) [write\(\)](#) [1331](#) [W\\_OK](#) [1330](#)  
[X\\_OK](#) [1330](#) [\\_exit\(\)](#) [1331](#)

In generale, la libreria offerta da un compilatore del linguaggio C si può estendere in modo imprecisato verso le definizioni dello standard POSIX. Per esempio, è normale che una libreria C includa le funzionalità relative alla gestione delle espressioni regolari, definite dallo standard POSIX. Pertanto, non esiste propriamente una libreria

C e una POSIX, va quindi verificato con il proprio compilatore cosa offrono effettivamente le librerie disponibili, specificando eventualmente, in fase di compilazione, l'inclusione di questa o quella libreria precompilata per la gestione di quella certa funzionalità POSIX.

Nei capitoli successivi vengono descritti alcuni dei file di intestazione previsti dallo standard POSIX, che a loro volta non sono già presi in considerazione dallo standard del linguaggio C. In certi casi viene mostrato come potrebbero essere realizzati questi file (gli esempi dovrebbero essere disponibili a partire da [allegati/c/](#)).

Tabella 70.1. Alcuni file di intestazione dello standard POSIX che non si trovano già nello standard del linguaggio C.

Intestazione	Descrizione	Riferimenti
<code>sys/types.h</code>	Tipi di dati derivati.	sezione <a href="#">70.1</a>
<code>sys/stat.h</code>	Definizione dei dati restituiti dalla funzione <i>stat()</i> , necessari alla qualificazione delle caratteristiche dei file, e di alcune funzioni relative alla questione.	sezione <a href="#">70.2</a>
<code>strings.h</code>	Funzioni per il trattamento delle stringhe e simili che non sono già incluse in <code>'string.h'</code> .	sezione <a href="#">70.3</a>
<code>fcntl.h</code>	Opzioni per il controllo dei file, gestiti in qualità di descrittori.	sezione <a href="#">70.4</a>
<code>unistd.h</code>	Macro-variabili standard e molte funzioni sulla gestione dei file.	sezione <a href="#">70.5</a>
<code>dirent.h</code>	Gestione delle directory, in qualità di flussi, attraverso puntatori di tipo <code>'DIR *</code> .	sezione <a href="#">70.6</a>

Intestazione	Descrizione	Riferimenti
<code>termios.h</code>	Configurazione dei dispositivi di terminale.	sezione <a href="#">70.7</a>

## 70.1 File «`sys/types.h`»

«

Il file ‘`sys/types.h`’ viene usato dallo standard POSIX per definire tutti i tipi di dati derivati, inclusi alcuni che già fanno parte dello standard C puro e semplice (si veda eventualmente la realizzazione di questo file nei sorgenti di `os32`, sezione [95.26](#)). La tabella successiva ne descrive solo una parte.

Tipo derivato	Descrizione
<code>blkcnt_t</code>	Numero intero con segno. Quantità di blocchi, riferita ai file.
<code>blksize_t</code>	Numero intero con segno. Dimensione in blocchi di un file.
<code>clock_t</code>	Numero intero. Unità di tempo che rappresenta un ciclo virtuale di CPU. È già definito nel file ‘ <code>time.h</code> ’.
<code>dev_t</code>	Numero intero. Numero identificativo di un dispositivo.
<code>gid_t</code>	Numero intero senza segno. Numero identificativo del gruppo a cui appartiene un file.
<code>ino_t</code>	Numero intero senza segno. Numero di inode, ovvero il numero identificativo di un file.
<code>mode_t</code>	Numero intero. Attributo di accesso di un file.
<code>nlink_t</code>	Numero intero senza segno. Quantità di collegamenti fisici riferiti a un file.



Tipo derivato	Descrizione
off_t	Numero intero con segno. Scostamento relativo al contenuto di un file.
pid_t	Numero intero con segno. Numero che identifica un processo elaborativo. Il segno è necessario perché il valore -1 rappresenta un errore nella creazione di un processo.
pthread_t	Numero intero. Numero identificativo di un thread.
size_t	Numero intero senza segno. Usato per esprimere la grandezza di oggetti rappresentati in memoria (variabili, distanza tra puntatori e simili). Nello standard C, questo tipo derivato viene definito nel file 'stddef.h'.
ssize_t	Numero intero con segno ( <i>signed size_t</i> ). Usato per esprimere una quantità di byte, se positivo, oppure un errore se negativo.
time_t	Numero intero con segno. Usato per esprimere un tempo in secondi, trascorso a partire dal giorno 1 gennaio 1970. Nello standard C, questo tipo derivato viene definito nel file 'time.h'.
uid_t	Numero intero senza segno. Numero identificativo dell'utente a cui appartiene un file.

Tipo derivato	Descrizione
id_t	Numero intero senza segno. Numero identificativo generico, in grado di ospitare il valore dei tipi <code>'pid_t'</code> , <code>'uid_t'</code> e <code>'gid_t'</code> . Va tenuto in considerazione il fatto che un numero di identificazione di un processo deve essere positivo, salvo il caso in cui un valore negativo serva per indicare un errore; pertanto, il tipo <code>'id_t'</code> può essere definito come privo di segno.

L'esempio successivo mostra come potrebbero essere dichiarati questi tipi derivati, limitatamente ai casi descritti nella tabella:

```

typedef long int      blkcnt_t;
typedef long int      blksize_t;
typedef long int      clock_t;
typedef unsigned long int dev_t;
typedef unsigned int  id_t;
typedef unsigned long int ino_t;
typedef unsigned int  gid_t;
typedef unsigned int  mode_t;
typedef unsigned int  nlink_t;
typedef long long int off_t;
typedef int           pid_t;
typedef unsigned long int pthread_t;
typedef unsigned long int size_t;
typedef long int      ssize_t;
typedef long int      time_t;
typedef unsigned int  uid_t;

```

## 70.2 File «sys/stat.h»

Il file ‘`sys/stat.h`’ viene usato dallo standard POSIX principalmente per definire un insieme di macro-variabili che individuano le caratteristiche fondamentali di un file (tipo di file e permessi), per definire il tipo ‘`struct stat`’ che serve a rappresentare lo stato di un file, per dichiarare il prototipo di alcune funzioni che hanno a che fare con queste informazioni (si veda eventualmente la realizzazione del file ‘`sys/stat.h`’ e di alcune delle sue funzioni nei sorgenti di `os32`, sezione [95.25](#)).

È importante considerare il file ‘`sys/stat.h`’ assieme a ‘`fcntl.h`’.

Nel file ‘`sys/stat.h`’ si fa riferimento a un insieme di tipi derivati, dichiarati nel file ‘`sys/types.h`’. Per semplicità, l’esempio propone la sua inclusione iniziale:

```
#include <sys/types.h> // dev_t
                        // off_t
                        // blkcnt_t
                        // blksize_t
                        // ino_t
                        // mode_t
                        // nlink_t
                        // uid_t
                        // gid_t
                        // time_t
```

## 70.2.1 Macro-variabili per la definizione del contenuto di un valore di tipo «mode\_t»

«

Il tipo derivato ‘**mode\_t**’ serve a rappresentare il tipo di un file (o di una directory) e i permessi disponibili. Questo tipo si traduce generalmente in un intero a 16 bit. Trattandosi di un valore numerico, queste informazioni sono distinte a gruppi di bit, selezionabili attraverso una maschera. Pertanto, tra le macro-variabili che distinguono le varie caratteristiche associabili a una variabile di tipo ‘**mode\_t**’, alcune vanno usate come maschera, per distinguere un certo insieme di informazioni, dalle altre.

```
//  
// Tipo di file.  
//  
#define S_IFMT 0170000 // Maschera del tipo di file.  
//  
#define S_IFBLK 0060000 // File di dispositivo a blocchi.  
#define S_IFCHR 0020000 // File di dispositivo a caratteri.  
#define S_IFIFO 0010000 // File FIFO.  
#define S_IFREG 0100000 // File puro e semplice  
// (regular file).  
#define S_IFDIR 0040000 // Directory.  
#define S_IFLNK 0120000 // Collegamento simbolico.  
#define S_IFSOCK 0140000 // Socket di dominio Unix.  
//  
// Permessi di accesso dell'utente proprietario del file.  
//  
#define S_IRWXU 0000700 // Maschera che rappresenta  
// simultaneamente tutti i  
// permessi per l'utente  
// proprietario.  
#define S_IRUSR 0000400 // Permesso di lettura.  
#define S_IWUSR 0000200 // Permesso di scrittura.
```

```
#define S_IXUSR 0000100 // Permesso di esecuzione  
                        // o attraversamento.  
  
//  
// Permessi di accesso del gruppo proprietario del file.  
//  
#define S_IRWXG 0000070 // Maschera che rappresenta  
                        // simultaneamente tutti i  
                        // permessi per il gruppo  
                        // proprietario.  
  
#define S_IRGRP 0000040 // Permesso di lettura.  
#define S_IWGRP 0000020 // Permesso di scrittura.  
#define S_IXGRP 0000010 // Permesso di esecuzione  
                        // o attraversamento.  
  
//  
// Permessi di accesso degli altri utenti.  
//  
#define S_IRWXO 0000007 // Maschera che rappresenta  
                        // simultaneamente tutti i  
                        // permessi per il gruppo  
                        // proprietario.  
  
#define S_IROTH 0000004 // Permesso di lettura.  
#define S_IWOTH 0000002 // Permesso di scrittura.  
#define S_IXOTH 0000001 // Permesso di esecuzione  
                        // o attraversamento.  
  
//  
// Permessi aggiuntivi: S-bit.  
// In questo caso non c'è una maschera che li includa tutti.  
//  
#define S_ISUID 0004000 // S-UID.  
#define S_ISGID 0002000 // S-GID.  
#define S_ISVTX 0001000 // Sticky.
```



```
// [3] File FIFO.
// [4] File puro e semplice.
// [5] È una directory.
// [6] Collegamento simbolico.
// [7] Socket di dominio Unix.
```

## 70.2.2 Struttura «stat»

Il file ‘sys/stat.h’ include la dichiarazione del tipo derivato ‘**struct stat**’, con lo scopo di contenere le informazioni disponibili su di un file. La struttura deve contenere almeno i membri che appaiono nell’esempio successivo:

```
//
// Struttura «stat».
//
struct stat {
    dev_t      st_dev;      // File di dispositivo contenente
                        // il file.
    ino_t      st_ino;     // Numero di serie del file
                        // (inode).
    mode_t     st_mode;    // Tipo e permessi del file.
    nlink_t    st_nlink;   // Collegamenti fisici associati
                        // al file.
    uid_t      st_uid;     // Numero identificativo
                        // dell'utente proprietario.
    gid_t      st_gid;     // Numero identificativo del
                        // gruppo proprietario.
    dev_t      st_rdev;    // Numero del file dispositivo, se
                        // si tratta di un tale tipo di
                        // file.
    off_t      st_size;    // Se si tratta di un file vero
                        // e proprio, misura la dimensione
                        // del file; se si tratta di un
```

```

// collegamento simbolico, misura
// la dimensione del percorso che
// questo rappresenta; per altri
// casi il significato di questo
// campo non è precisato.
time_t    st_atime; // Data dell'ultimo accesso.
time_t    st_mtime; // Data dell'ultima modifica del
                // contenuto.
time_t    st_ctime; // Data dell'ultima modifica dello
                // stato.
blksize_t st_blksize; // Dimensione ottimale del blocco
                // per le operazioni di I/O.
blkcnt_t  st_blocks; // Blocchi da 512 byte allocati
                // per il file.
};

```

### 70.2.3 Prototipi di funzione



Il file ‘sys/stat.h’ include la dichiarazione di alcuni prototipi di funzione, come si vede nell’esempio seguente:

```

//
// Prototipi di funzione.
//
int    chmod  (const char *path, mode_t mode);
int    fchmod (int fdn, mode_t mode);
int    fstat  (int fdn, struct stat *buf);
int    lstat  (const char *restrict file,
                struct stat *restrict buf);
int    mkdir  (const char *path, mode_t mode);
int    mkfifo (const char *path, mode_t mode);
int    mknod  (const char *path, mode_t mode, dev_t dev);
int    stat   (const char *restrict path,
                struct stat *restrict buf);
mode_t umask  (mode_t mask);

```



## 70.3 File «strings.h»

Il file di intestazione ‘strings.h’ contiene i prototipi di alcune funzioni, legate prevalentemente alla scansione delle stringhe. Dal momento che viene usato il tipo derivato ‘**size\_t**’, questo viene definito attraverso l’inclusione del file ‘**stddef.h**’.

```
#include <stddef.h>

int ffs      (int i);
int strcasecmp (const char *s1, const char *s2);
int strncasecmp (const char *s1, const char *s2, size_t n);
```

Lo standard prevede anche altri prototipi di funzioni ormai superate, che rimangono solo per compatibilità con il passato.

### 70.3.1 Funzione «ffs()»

La funzione *ffs()* serve a scandire i bit di un valore numerico intero, alla ricerca del primo bit a uno, partendo dalla posizione meno significativa, restituendo l’indice di tale bit, considerando il bit meno significativo avente indice uno. Pertanto, se il valore da scandito è pari a zero (non ha alcun bit a uno), la funzione restituisce zero.

Al di fuori dello standard, è probabile trovare delle altre funzioni simili a questa, per la scansione degli interi di tipo ‘**long int**’ e di tipo ‘**long long int**’. In tal caso, i nomi delle funzioni ulteriori possono essere *ffsl()* e *ffsll()*.

### 70.3.2 Funzioni «`strcasecmp()`» e «`strncasecmp()`»

«

Le funzioni *strcasecmp()* e *strncasecmp()* servono a confrontare due stringhe, ignorando la differenza tra maiuscole e minuscole. Nel caso di *strncasecmp()* il confronto è limitato a una certa quantità massima di caratteri.

Se la configurazione locale è quella POSIX, il confronto avviene come se le due stringhe venissero convertite preventivamente in caratteri minuscoli; ma nel caso sia attiva una configurazione locale differente, lo standard non specifica in che modo la comparazione abbia luogo.

Il valore restituito dalle due funzioni dipende dal confronto lessicografico delle due stringhe. Se sono uguali (a parte la differenza tra maiuscole e minuscole), il risultato è zero; se la prima stringa è lessicograficamente precedente rispetto alla seconda, il valore restituito è inferiore a zero; se la prima stringa è lessicograficamente successiva alla seconda, il valore ottenuto è superiore a zero.

### 70.4 File «`fcntl.h`»

«

Il file di intestazione ‘`fcntl.h`’ riguarda la parte fondamentale della gestione dei file, attraverso i descrittori; precisamente si considerano la creazione, l’apertura e l’attribuzione di opzioni di funzionamento, mentre altre questioni sono gestite attraverso le definizioni contenute nel file ‘`unistd.h`’ (si veda eventualmente la realizzazione del file ‘`fcntl.h`’ e di alcune delle sue funzioni nei sorgenti di `os32`, sezione [95.6](#)).

## 70.4.1 Tipi di dati derivati

Il file di intestazione ‘fcntl.h’ utilizza alcuni tipi di dati derivati, già definiti nel file ‘sys/types.h’:

```
#include <sys/types.h> // mode_t
                        // off_t
                        // pid_t
```

## 70.4.2 Opzioni per la funzione «fcntl()»

Nel file di intestazione ‘fcntl.h’ si definiscono varie macro-variabili, di cui un primo insieme riguarda, quasi in modo esclusivo, l’uso della funzione *fcntl()*.

```
//
// Valori per il secondo argomento della funzione fcntl().
//
#define F_DUPFD          0 // Duplicate file descriptor.
#define F_GETFD         1 // Get file descriptor flags.
#define F_SETFD         2 // Set file descriptor flags.
#define F_GETFL         3 // Get file status flags.
#define F_SETFL         4 // Set file status flags.
#define F_GETLK         5 // Get record locking information.
#define F_SETLK         6 // Set record locking information.
#define F_SETLKW        7 // Set record locking information;
                        // wait if blocked.
#define F_GETOWN        8 // Set owner of socket.
#define F_SETOWN        9 // Get owner of socket.
//
// Flag da impostare con:
// fcntl (fd, F_SETFD, ...);
//
#define FD_CLOEXEC      1 // Chiude il descrittore del file
```

```
// nel momento dell'esecuzione di
// una funzione del gruppo
// 'exec()'.
```

Le macro-variabili *F\_DUPFD*, *F\_GETFD*, *F\_SETFD*, *F\_GETFL*, *F\_SETFL*, *F\_GETLK*, *F\_SETLK*, *F\_SETLKW*, *F\_GETOWN* e *F\_SETOWN*, servono per dichiarare il tipo di comando da dare alla funzione *fcntl()*, attraverso il suo secondo parametro:

```
int fcntl (int fdn, int cmd, ...);
```

La macro-variabile *FD\_CLOEXEC* riguarderebbe un insieme di indicatori associati a un descrittore di file (*fd\_flags*), di cui però ne esiste uno solo, rappresentato dalla macro-variabile stessa. Utilizzando la funzione *fcntl()* e specificando il comando *F\_GETFD* è possibile ottenere lo stato di questi indicatori (ovvero solo *FD\_CLOEXEC*), mentre con il comando *F\_SETFD* è possibile modificare questo stato. Quando l'indicatore *FD\_CLOEXEC* risulta attivo per un certo descrittore di file, se viene eseguita la sostituzione del processo con l'ausilio di una funzione del gruppo *exec...()*, il descrittore in questione viene chiuso, mentre diversamente rimarrebbe aperto.

Nel file di intestazione 'fcntl.h' vengono dichiarate anche delle macro-variabili per definire la modalità di accesso a un file, da usare prevalentemente con la funzione *open()*:

```
//
// Indicatori per la creazione dei file, da usare nel
// parametro «oflag» della funzione open().
//
#define O_CREAT      000010 // Crea il file se non esiste già.
```

```
#define O_EXCL      000020 // Indicatore di accesso  
                        // esclusivo.  
#define O_NOCTTY   000040 // Non assegna un terminale di  
                        // controllo.  
#define O_TRUNC    000100 // Indicatore di troncamento.  
//  
// Indicatori dello stato dei file,  
// usati nelle funzioni open() e fcntl().  
//  
#define O_APPEND   000200 // Scrittura in aggiunta.  
#define O_DSYNC    000400 // Scrittura sincronizzata  
                        // dei dati.  
#define O_NONBLOCK 001000 // Modalità non bloccante.  
#define O_RSYNC    002000 // Lettura sincronizzata.  
#define O_SYNC     004000 // Scrittura sincronizzata  
                        // dei file.  
  
//  
// Maschera per la selezione delle sole modalità principali  
// di accesso ai file.  
//  
#define O_ACCMODE  000003 // Seleziona gli ultimi due bit,  
                        // che in questo caso individuano  
                        // le modalità di accesso  
                        // principali (lettura, scrittura  
                        // ed entrambe) dalle altre  
                        // modalità che sono già state  
                        // descritte sopra.  
  
//  
// Modalità principali di accesso ai file, secondo la  
// tradizione.  
//  
//#define O_RDONLY 000000 // Apertura in sola lettura.  
//#define O_WRONLY 000001 // Apertura in sola scrittura.  
//#define O_RDWR  000002 // Apertura in lettura e
```



```

// da "l_start".
off_t    l_start; // Scostamento che individua
              // l'inizio dell'area bloccata.
              // Lo scostamento può essere
              // positivo o negativo.
off_t    l_len;   // Dimensione dell'area bloccata:
              // se si indica zero, significa che
              // questa raggiunge la fine
              // del file.
pid_t    l_pid;   // Il numero del processo
              // elaborativo che blocca
              // l'area.
};

```

#### 70.4.4 Funzioni

Sono presenti anche i prototipi delle funzioni *creat()*, *fcntl()* e *open()*: «

```

//
// Prototipi di funzione.
//
int creat (const char *file, mode_t mode);
int fcntl (int fdn, int cmd, ...);
int open  (const char *file, int oflag, ...);

```

Per l'uso delle funzioni *open()* e *creat()* si veda la sezione [68.5](#).

La funzione *fcntl()* esegue un'operazione, definita dal parametro *cmd*, sul descrittore richiesto come primo parametro (*fdn*). A seconda del tipo di operazione richiesta, potrebbero essere necessari degli argomenti ulteriori, i quali però non possono essere formalizzati in modo esatto nel prototipo della funzione. Il valore del secondo pa-

rametro che rappresenta l'operazione richiesta, va fornito in forma di costante simbolica, come descritto nell'elenco seguente, nel quale però sono descritti solo alcuni dei comandi possibili.

Sintassi	Descrizione
<pre>fcntl (<i>fdn</i>, F_DUPFD,       (int) <i>fdn_min</i>)</pre>	<p>Richiede la duplicazione del descrittore di file <i>fdn</i>, in modo tale che la copia abbia il numero di descrittore minore possibile, ma maggiore o uguale a quello indicato come argomento <i>fdn_min</i>.</p>
<pre>fcntl (<i>fdn</i>, F_GETFD) fcntl (<i>fdn</i>, F_SETFD,       (int) <i>fd_flags</i>)</pre>	<p>Rispettivamente, legge o imposta, gli indicatori del descrittore di file <i>fdn</i> (eventualmente noti come <i>file descriptor flags</i> o solo <i>fd flags</i>). Per il momento, è possibile impostare un solo indicatore, <b>FD_CLOEXEC</b>, pertanto, al posto di <i>fd_flags</i> si può mettere solo la costante <b>FD_CLOEXEC</b>.</p>
<pre>fcntl (<i>fdn</i>, F_GETFL) fcntl (<i>fdn</i>, F_SETFL,       (int) <i>fl_flags</i>)</pre>	<p>Rispettivamente, legge o imposta, gli indicatori dello stato del file, relativi al descrittore <i>fdn</i> (eventualmente noti come <i>file flags</i> o solo <i>fl flags</i>). Per impostare questi indicatori, vanno combinate delle costanti simboliche: <b>O_RDONLY</b>, <b>O_WRONLY</b>, <b>O_RDWR</b>, <b>O_CREAT</b>, <b>O_EXCL</b>, <b>O_NOCTTY</b>, <b>O_TRUNC</b>.</p>



Il significato del valore restituito dalla funzione dipende dal tipo di operazione richiesta, come sintetizzato dalla tabella successiva, relativa ai soli comandi già apparsi. In generale, anche per gli altri comandi, un risultato erraneo viene comunque evidenziato dalla restituzione di un valore negativo.

Operazione richiesta	Significato del valore restituito
F_DUPFD	Si ottiene il numero del descrittore prodotto dalla copia, oppure $-1$ in caso di errore.
F_GETFD	Si ottiene il valore degli indicatori del descrittore ( <i>fd flags</i> ), oppure $-1$ in caso di errore.
F_GETFL	Si ottiene il valore degli indicatori del file ( <i>fl flags</i> ), oppure $-1$ in caso di errore.

## 70.5 File «unistd.h»

Il file di intestazione ‘unistd.h’ raccoglie un po’ di tutto ciò che riguarda le estensioni POSIX, pertanto è frequente il suo utilizzo (si veda eventualmente la realizzazione del file ‘unistd.h’ e di alcune delle sue funzioni nei sorgenti di os32, sezione [95.30](#)).

Nel file ‘unistd.h’ si distingue la presenza di un elenco numeroso di macro-variabili con prefissi *\_POSIX\_...*, *\_POSIX2\_...* e *\_XOPEN\_...*, con lo scopo di dichiarare le caratteristiche del sistema e della libreria. Per l’interrogazione delle caratteristiche o delle limitazioni del sistema, si utilizzano però delle funzioni apposite, costituite precisamente da *pathconf()* e *sysconf()*, le quali utilizzano un proprio insieme di macro-variabili per individuare le caratteristiche da interrogare. Nel caso di *pathconf()* si aggiungono macro-variabili con prefisso *\_PC\_...*; per *sysconf()* le macro-variabili hanno il prefisso *\_SC\_...*.

Nei prototipi di funzione si utilizzano diversi tipi derivati, già dichiarati nei file ‘sys/types.h’ e ‘inttypes.h’:

```
#include <sys/types.h> // size_t, ssize_t, uid_t, gid_t,
                        // off_t, pid_t, useconds_t
#include <inttypes.h> // intptr_t
```

Nel file deve anche essere dichiarato il valore per la macro-variabile **NULL**; in questo caso viene incorporato il file ‘stddef.h’:

```
#include <stddef.h> // NULL
```

### 70.5.1 Denominazione dei descrittori di file

«

Per dare un nome ai descrittori dei flussi standard, nel file ‘unistd.h’ si dichiarano tre macro-variabili, il cui valore è stabilito necessariamente:

```
#define STDIN_FILENO 0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2
```

### 70.5.2 Verifica dei permessi di accesso

«

La funzione **access()** consente di verificare l’accessibilità di un file. Per questo richiede l’indicazione del percorso e di un valore che rappresenta i tipi di accesso che si vogliono considerare. Questi sono rappresentati dall’unione di quattro possibili macro-variabili:

```
#define R_OK 04 // Accessibilità in lettura.
#define W_OK 02 // Accessibilità in scrittura.
#define X_OK 01 // Accessibilità in esecuzione o
                // attraversamento.
#define F_OK 00 // Esistenza del file.
```

I valori che questa macro-variabili possono avere devono essere tali da consentire la combinazione con l'operatore ' | ' (OR, bit per bit), così da poter verificare simultaneamente tutti gli aspetti dell'accesso al file.

### 70.5.3 Funzioni

Segue l'elenco dei prototipi delle funzioni principali del file 'unistd.h':

```
int          access      (const char *path, int mode);
unsigned int alarm      (unsigned int seconds);
int          chdir      (const char *path);
int          chown      (const char *path, uid_t owner,
                        gid_t group);
int          close      (int fdn);
size_t      confstr     (int name, char *buffer,
                        size_t length);
int          dup         (int old_fdn);
int          dup2       (int old_fdn, int new_fdn);
int          execl      (const char *path,
                        const char *arg, ...);
int          execl     (const char *path,
                        const char *arg, ...,
                        char *const envp[]);
int          execlp     (const char *path,
                        const char *arg, ...);
int          execv      (const char *path,
                        char *const argv[]);
int          execve     (const char *path,
                        char *const argv[],
                        char *const envp[]);
int          execvp     (const char *path,
                        char *const argv[]);
```

```
void      _exit      (int status);
int       fchown     (int fdn, uid_t owner,
                    gid_t group);
pid_t     fork       (void);
long      fpathconf  (int fdn, int name);
int       ftruncate  (int fdn, off_t length);
char      *getcwd    (char *buffer, size_t size);
gid_t     getegid    (void);
uid_t     geteuid    (void);
gid_t     getgid     (void);
int       getgroups  (int size, gid_t list[]);
int       gethostname (char *name, size_t length);
char      *getlogin  (void);
int       getlogin_r (char *buffer, size_t size);
int       getopt     (int argv, char *const argv[],
                    const char *optstring);
pid_t     getpgrp    (void);
pid_t     getpid     (void);
pid_t     getppid    (void);
uid_t     getuid     (void);
int       isatty     (int fdn);
int       link       (const char *old_path,
                    const char *new_path);
off_t     lseek      (int fdn, off_t offset,
                    int whence);
long      pathconf   (const char *path, int name);
int       pause      (void);
int       pipe       (int fdn[2]);
ssize_t   read       (int fdn, void *buffer,
                    size_t length);
ssize_t   readlink   (const char *restrict path,
                    char *restrict buffer,
                    size_t size);
int       rmdir      (const char *path);
```

```

int      setegid      (gid_t egid);
int      seteuid      (uid_t euid);
int      setgid       (gid_t gid);
int      setpgid      (pid_t pid, pid_t pgid);
pid_t    setsid       (void);
int      setuid        (uid_t uid);
unsigned sleep        (unsigned int seconds);
int      symlink       (const char *old_path,
                        const char *new_path);

long     sysconf       (int name);
pid_t    tcgetpgrp     (int fdn);
int      tcsetpgrp     (int fdn, pid_t pgrp);
char     *ttyname      (int fdn);
int      ttyname_r     (int fdn, char *buffer,
                        size_t length);

int      unlink        (const char *path);
ssize_t  write         (int fdn, const void *buffer,
                        size_t length);

```

## 70.5.4 Funzioni di servizio per la gestione di file e directory

Nelle sezioni seguenti si descrivono solo alcune delle funzioni destinate alla gestione di file e directory, il cui prototipo appare nel file ‘unistd.h’.

### 70.5.4.1 Funzione «access()»

La funzione *access()* consente di verificare l’accessibilità di un file, il cui percorso viene fornito tramite una stringa. L’accessibilità viene valutata in base a delle opzioni, con cui è possibile specificare a cosa si è interessati in modo preciso.

```
int access (const char *percorso, int modalità);
```

Il secondo parametro della funzione è un numero intero fornito normalmente attraverso l'indicazione di una macro-variabile che rappresenta simbolicamente il tipo di accesso che si intende verificare. Si può utilizzare **F\_OK** per verificare l'esistenza del file o della directory; in alternativa, si possono usare le macro-variabili **R\_OK**, **W\_OK** e **X\_OK**, sommabili assieme attraverso l'operatore OR binario, per la verifica dell'accessibilità in lettura, in scrittura e in esecuzione o attraversamento. Per esempio, scrivendo '**R\_OK|W\_OK|X\_OK**' si vuole verificare che il file o la directory sia accessibile con tutti i tre permessi attivi.

Tabella 70.23. Macro-variabili usate per descrivere la modalità di accesso del secondo parametro della funzione *access()*.

Macro-variabile	Descrizione
F_OK	Si richiede la verifica dell'esistenza del file o della directory. Questa opzione va usata da sola e non può essere sommata alle altre.
R_OK	Si richiede la verifica dell'accessibilità in lettura del file o della directory.
W_OK	Si richiede la verifica dell'accessibilità in scrittura del file o della directory.
X_OK	Si richiede la verifica dell'accessibilità in esecuzione del file o in attraversamento della directory.

La funzione restituisce il valore zero se il file o la directory risultano accessibili nel modo richiesto, altrimenti restituisce il valore -1 e si può verificare il tipo di errore valutando il contenuto della variabile

***errno***.

L'esempio seguente descrive completamente l'uso della funzione. Si può osservare che per valutare il successo dell'operazione, l'esito restituito dalla funzione viene invertito; inoltre, il contenuto della variabile ***errno*** viene considerato con l'aiuto della funzione ***perror()***. Il file dell'esempio dovrebbe essere disponibile presso [allegati/c/esempio-posix-unistd-access.c](#) .

```
#include <stdio.h>
#include <unistd.h>

int
main (void)
{
    const char *file_name = "/tmp/test";

    if (! access (file_name, F_OK))
    {
        printf ("Il file o la directory "
                "\"%s\" esiste.\n", file_name);
        if (! access (file_name, R_OK))
        {
            printf ("Il file o la directory "
                    "\"%s\" ", file_name);
            printf ("è accessibile in lettura.\n");
        }
    }
    else
    {
        printf ("Il file o la directory "
                "\"%s\" ", file_name);
        printf ("non è accessibile in lettura.\n");
        perror (NULL);
    }
    if (! access (file_name, W_OK))
```

```
    {
        printf ("Il file o la directory "
               "\"%s\" ", file_name);
        printf ("è accessibile in scrittura.\n");
    }
else
    {
        printf ("Il file o la directory "
               "\"%s\" ", file_name);
        printf ("non è accessibile in scrittura.\n");
        perror (NULL);
    }
if (! access (file_name, X_OK))
    {
        printf ("Il file o la directory "
               "\"%s\" ", file_name);
        printf ("è accessibile in esecuzione o "
               "attraversamento.\n");
    }
else
    {
        printf ("Il file o la directory "
               "\"%s\" ", file_name);
        printf ("non è accessibile in esecuzione "
               "o attraversamento.\n");
        perror (NULL);
    }
}
else
    {
        printf ("Il file o la directory "
               "\"%s\" non esiste.\n", file_name);
        perror (NULL);
    }
}
```



```
    return (0);  
}
```

## 70.5.4.2 Funzione «getcwd()»

La funzione *getcwd()* (*get current working directory*) consente di annotare in una stringa il percorso della directory corrente. «

```
char *getcwd (char *buffer, size_t max);
```

Come si può vedere dal prototipo della funzione, occorre predisporre prima un array di caratteri, da usare come stringa, in cui la funzione va a scrivere il percorso trovato (con tanto di carattere nullo di terminazione). Come secondo parametro della funzione va indicata la dimensione massima dell'array, oltre la quale la scrittura non può andare.

La funzione restituisce il puntatore alla stringa contenente il percorso, ma se si verifica un errore restituisce il puntatore nullo e aggiorna la variabile *errno* con la specificazione della causa di tale errore.

L'esempio seguente descrive l'uso della funzione in modo molto semplice; in particolare, in caso di errore si usa la funzione *perror()* per visualizzarne una descrizione. Il file dell'esempio dovrebbe essere disponibile presso [allegati/c/esempio-posix-unistd-getcwd.c](#).

```
#include <stdio.h>  
#include <unistd.h>  
  
int  
main (void)
```

```
{
    char cwd[500];
    char *result;

    result = getcwd (cwd, 500);

    if (result == NULL)
    {
        perror (NULL);
    }
    else
    {
        printf ("%s\n", cwd);
    }
    return (0);
}
```

### 70.5.4.3 Funzione «chdir()»



La funzione *chdir()* (*change directory*) consente di cambiare la directory corrente del processo elaborativo.

```
int chdir (const char *path);
```

La funzione richiede come unico parametro la stringa che descrive il percorso da raggiungere. La funzione restituisce zero se l'operazione si conclude con successo, oppure il valore  $-1$  in caso di errore, ma in tal caso viene modificata anche la variabile *errno* con l'indicazione più precisa dell'errore verificatosi.

L'esempio seguente mostra il comportamento della funzione, ma va osservato che l'effetto riguarda esclusivamente il processo in fun-

zione e non si riflette al processo che lo genera a sua volta. Per questa ragione il programma di esempio visualizza la posizione corrente raggiunta. Il file dell'esempio dovrebbe essere disponibile presso [allegati/c/esempio-posix-unistd-chdir.c](#) .

```
#include <stdio.h>
#include <unistd.h>

int
main (void)
{
    const char *path = "/tmp/test";
    char cwd[500] = {'\0'};

    if (! chdir (path))
        {
            getcwd (cwd, 500);
            printf ("Il processo passa alla "
                   "directory \"%s\".\n", cwd);
        }
    else
        {
            printf ("Non è possibile passare alla "
                   "directory \"%s\"!\n",
                   path);
            perror (NULL);
        }

    return (0);
}
```

#### 70.5.4.4 Funzioni «link()» e «unlink()»

«

La funzione *link()* consente la creazione di un nuovo collegamento fisico a partire da un file o da una directory già esistente, tenendo conto che la facoltà di creare un collegamento fisico a partire da una directory è una funzione privilegiata e il sistema operativo potrebbe non ammetterla in ogni caso. Per collegamento fisico si intende il riferimento contenuto in una directory, verso un certo file o una certa sottodirectory individuati numericamente da un numero inode. La funzione *link()* produce una sorta di copia del file, nel senso che si predispone un riferimento aggiuntivo alla stessa cosa, senza la duplicazione dei dati relativi. Per converso, la funzione *unlink()* elimina il riferimento a un file o a una directory, cosa che coincide con la cancellazione del file o della directory, se si tratta dell'ultimo riferimento esistente a tale oggetto nel file system. Anche in questo caso, va considerato in modo particolare l'eliminazione del riferimento a una directory: il sistema operativo può impedirlo se si tratta di una directory non vuota.

```
int link (const char *p1, const char *p2);
```

```
int unlink (const char *p);
```

I parametri delle due funzioni sono stringhe che descrivono il percorso di un file o di una directory. Nel caso di *unlink()* si indica solo il percorso da rimuovere, mentre con *link()* se ne indicano due: l'origine e la destinazione che si vuole creare.

Le due funzioni restituiscono un valore intero pari a zero se tutto

va bene, altrimenti restituiscono il valore  $-1$ , modificando anche il contenuto della variabile *errno* con un'informazione più precisa sull'accaduto.

Gli esempi seguenti mostrano il comportamento delle due funzioni. I file degli esempi dovrebbero essere disponibili presso [allegati/c/esempio-posix-unistd-link.c](#) e [allegati/c/esempio-posix-unistd-unlink.c](#).

```
#include <stdio.h>
#include <unistd.h>

int
main (void)
{
    const char *old = "/tmp/test";
    const char *new = "/tmp/test.link";

    if (! link (old, new))
        {
            printf ("Creato il collegamento \"%s\".\n", new);
        }
    else
        {
            printf ("Non è possibile creare il "
                    "collegamento \"%s\"!\n",
                    new);
            perror (NULL);
        }

    return (0);
}
```

```
#include <stdio.h>
#include <unistd.h>
```

```
int
main (void)
{
    const char *name = "/tmp/test";

    if (! unlink (name))
        {
            printf ("Cancellato il file o la "
                    "directory \"%s\".\n", name);
        }
    else
        {
            printf ("Non è possibile cancellare il file "
                    "o la directory ");
            printf ("\">%s\"!\n", name);
            perror (NULL);
        }

    return (0);
}
```

## 70.6 File «dirent.h»



Il file di intestazione ‘dirent.h’ raccoglie ciò che serve per la gestione delle directory, attraverso flussi individuati da puntatori di tipo ‘**DIR \***’ (si veda eventualmente la realizzazione del file ‘dirent.h’ e di alcune delle sue funzioni nei sorgenti di os32, sezione [95.4](#)). La gestione di tali flussi può avvenire attraverso i descrittori di file, così come già avviene per i flussi individuati da puntatori di tipo ‘**FILE \***’, ma ciò è una facoltà, non una necessità realizzativa.

## 70.6.1 Struttura «dirent»

Il file `'dirent.h'` prevede la dichiarazione di un tipo derivato, denominato `'struct dirent'`, da usare per rappresentare una voce singola di una directory. I membri necessari di questa struttura sono `d_ino`, per rappresentare il numero di inode della voce, e `d_name[]` per contenere il nome del file relativo, completo di carattere nullo di terminazione delle stringhe. La definizione del tipo `'struct dirent'` potrebbe essere realizzata nel modo seguente:

```
#include <sys/types.h> // ino_t
#include <limits.h>    // NAME_MAX
//
struct dirent {
    ino_t  d_ino;           // Numero di inode
    char   d_name[NAME_MAX+1]; // NAME_MAX + '\0'
} __attribute__((packed));
```

## 70.6.2 Tipo «DIR»

Nel file `'dirent.h'` è definito il tipo derivato `'DIR'`, con il quale si intende fare riferimento a un flusso che individua una directory aperta. Se la gestione di tali flussi avviene attraverso i descrittori di file, ci deve poter essere il riferimento al numero del descrittore relativo. Quello che segue è un esempio di una tale struttura, seguita dalla dichiarazione di un array per il contenimento delle informazioni su tutte le directory aperte del processo:

```

typedef struct {
    int          fdn;          // Numero del descrittore di file.
    struct dirent dir;        // Last directory item read.
} DIR;

extern DIR _directory_stream[]; // Array di directory che
                                // però non è previsto
                                // espressamente dallo
                                // standard.

```

Nella struttura di tipo ‘**DIR**’ di questo esempio, viene inclusa una struttura di tipo ‘**struct dirent**’, per permettere alla funzione *readdir()* di annotare l’ultima voce letta da una certa directory.

### 70.6.3 Prototipi di funzioni



La gestione delle directory in forma di flussi di tipo ‘**DIR \***’ richiede alcune funzioni specifiche, di cui si trovano i prototipi nel file ‘`dirent.h`’:

```

int          closedir (DIR *dp);
DIR          *opendir (const char *name);
struct dirent *readdir (DIR *dp);
void         rewinddir (DIR *dp);

```

La funzione *opendir()* apre un flusso corrispondente a una directory indicata tramite il suo percorso, restituendo il puntatore relativo. Una volta aperta una directory, si possono leggere le sue voci con la funzione *readdir()*, la quale restituisce il puntatore di una variabile strutturata di tipo ‘**struct dirent**’, all’interno della quale è possibile trarre i dati della voce letta: negli esempi di questo capitolo, tali informazioni sono incorporate nella struttura ‘**DIR**’ che rappresenta la directory aperta. Ogni lettura fa avanzare alla voce successiva della directory e, se necessario, si può riposizionare l’indice di lettura



alla prima voce, con la funzione *rewinddir()*. Per chiudere un flusso aperto, si usa la funzione *closedir()*.

## 70.7 File «termios.h»

La gestione essenziale del terminale a caratteri è abbastanza complessa e si sintetizza con le definizioni del file ‘termios.h’ (si veda eventualmente la realizzazione del file ‘termios.h’ e di alcune delle sue funzioni nei sorgenti di os32, sezione [95.28](#)). Lo standard prevede due modalità di inserimento: canonica e non canonica. Negli esempi di questo capitolo ci si sofferma su ciò che serve nel file ‘termios.h’ per gestire la modalità canonica, ovvero quella tradizionale per cui il dispositivo del terminale fornisce dati, solo dopo l’inserimento completo di una riga, confermato con un codice di interruzione di riga o con un altro codice che concluda comunque l’inserimento.

### 70.7.1 Tipi speciali

Nel file ‘termios.h’ vengono definiti alcuni tipi speciali per variabili scalari, che potrebbero essere descritti nel modo seguente:

```
typedef unsigned int tcflag_t;
typedef unsigned int speed_t;
typedef unsigned int cc_t;
```

Il tipo ‘**tcflag\_t**’ viene usato nelle strutture di tipo ‘**struct termios**’, la cui definizione viene fatta nello stesso file ‘termios.h’; il tipo ‘**speed\_t**’ serve per contenere il valore di una velocità di comunicazione del terminale; il tipo ‘**cc\_t**’ serve per rappresentare un carattere di controllo, per la gestione del terminale.

## 70.7.2 Tipo «struct termios»

&lt;&lt;

Nel file `'termios.h'` viene definita la struttura `'struct termios'`, allo scopo di annotare tutte le informazioni sulla modalità di funzionamento di un certo dispositivo di terminale. Nell'esempio seguente si vedono solo i membri obbligatori, ma va considerata l'aggiunta di informazioni legate alla velocità di comunicazione, se il terminale ne deve fare uso. La definizione della struttura `'struct termios'` richiede anche la dichiarazione della macro-variabile *NCCS*, come si vede nell'esempio.

```
#define NCCS      11          // Dimensione dell'array 'c_cc[]'.  
//  
struct termios {  
    tcflag_t c_iflag;  
    tcflag_t c_oflag;  
    tcflag_t c_cflag;  
    tcflag_t c_lflag;  
    cc_t      c_cc[NCCS];  
};
```

Il membro *c\_iflag* contiene gli indicatori che descrivono la modalità di inserimento di dati attraverso il terminale; il membro *c\_oflag* contiene indicatori per la modalità di elaborazione dei dati in uscita, prima della loro lettura; il membro *c\_cflag* contiene opzioni di controllo; il membro *lflag* contiene opzioni «locali» (qui, in particolare, si definisce se il terminale debba funzionare in modalità canonica o meno); l'array *c\_cc[]* contiene i codici di caratteri da interpretare in modo speciale, per il controllo del funzionamento del terminale.

### 70.7.3 Codici di controllo

Durante l’inserimento di dati attraverso il terminale, alcuni codici possono assumere un significato particolare. Il valore numerico di questi codici è annotato nell’array `c_cc[]` che è membro della struttura di tipo ‘`struct termios`’; per farvi riferimento, l’indice da usare nell’array `c_cc[]` deve essere indicato attraverso una meta-variabile:

```
#define VEOF      0      // carattere EOF
#define VEOL     1      // carattere EOL
#define VERASE   2      // carattere ERASE
#define VINTR    3      // carattere INTR
#define VKILL    4      // carattere KILL
#define VMIN     5      // valore MIN
#define VQUIT   6      // carattere QUIT
#define VSTART   7      // carattere START
#define VSTOP    8      // carattere STOP
#define VSUSP   9      // carattere SUSP
#define VTIME   10     // valore TIME
```

Due valori di questo elenco fanno eccezione: ‘`VMIN`’ e ‘`VTIME`’, in quanto rappresentano invece un’informazione di tipo differente, necessaria per la gestione non canonica del terminale.

### 70.7.4 Indicatori per il membro «`c_iflag`»

Il membro `c_iflag` di una struttura di tipo ‘`struct termios`’ può contenere degli indicatori indipendenti sulla configurazione per l’inserimento. Tali indicatori vanno forniti attraverso macro-variabili definite nel file ‘`termios.h`’, di cui segue un esempio:

```

#define BRKINT      1  // Invia un segnale di interruzione
                    // in caso di ricevimento di un
                    // carattere INTR.

#define ICRNL      2  // Converta <CR> in <NL>.

#define IGNBRK     4  // Ignora il carattere INTR.

#define IGNCR      8  // Ignora il carattere <CR>.

#define IGNPAR    16  // Ignora i caratteri con errori di
                    // parità.

#define INLCR     32  // Converta <NL> in <CR>.

#define INPCK     64  // Abilita il controllo di parità.

#define ISTRIP    128 // Azzera l'ottavo bit dei caratteri.

#define IXOFF     256 // Abilita il controllo start/stop in
                    // ingresso.

#define IXON      512 // Abilita il controllo start/stop in
                    // uscita.

#define PARMRK   1024 // Segnala gli errori di parità.

```

### 70.7.5 Indicatori per il membro «c\_oflag»

«

Il membro *c\_oflag* di una struttura di tipo `'struct termios'` può contenere degli indicatori indipendenti sulla configurazione per l'output. Tali indicatori vanno forniti attraverso macro-variabili definite nel file `'termios.h'`. Di questi indicatori ne è obbligatorio solo uno: **OPOST**, che ha lo scopo di abilitare una forma imprecisata di elaborazione dell'output.

```

#define OPOST      1  // post-process output

```

### 70.7.6 Indicatori per il membro «c\_cflag»

«

Il membro *c\_cflag* di una struttura di tipo `'struct termios'` può contenere degli indicatori indipendenti sulla configurazione per il controllo del terminale. Tale configurazione qui viene omessa.

## 70.7.7 Indicatori per il membro «c\_lflag»

Il membro *c\_lflag* di una struttura di tipo ‘`struct termios`’ può contenere degli indicatori indipendenti sulla configurazione «locale» del terminale. Tali indicatori vanno forniti attraverso macro-variabili definite nel file ‘`termios.h`’, di cui segue un esempio:

```
#define ECHO      1 // Abilita l'eco del terminale.
#define ECHOE    2 // Visualizza la cancellazione di un
                  // carattere.
#define ECHOK    4 // Visualizza l'eliminazione di una
                  // riga.
#define ECHONL   8 // Visualizza l'effetto del codice di
                  // interruzione di riga.
#define ICANON   16 // Modalità di inserimento canonica.
#define IEXTEN  32 // Modalità di inserimento estesa
                  // (non canonica).
#define ISIG     64 // Abilita i segnali.
#define NOFLSH  128 // Disabilita lo scarico della memoria
                  // tampone dopo un'interruzione o una
                  // conclusione forzata.
#define TOSTOP  256 // Invia il segnale SIGTTOU per l'output
                  // sullo sfondo.
```

L'indicatore *ICANON* consente di ottenere un funzionamento del terminale in modalità canonica.

## 70.7.8 Funzioni

Quanto descritto fino a questo punto sul file ‘`termios.h`’ è ciò che poi serve per l'uso di alcune funzioni, il cui scopo è quello di configurare o interrogare la configurazione di un terminale. Nell'esempio seguente appaiono solo alcuni prototipi, assieme alla dichiarazione di alcune macro-variabili necessarie per la funzione *tcsetattr()*.

```
#define TCSANOW 1 // Cambia gli attributi immediatamente.
#define TCSADRAIN 2 // Cambia gli attributi quando l'output
// è stato utilizzato.
#define TCSAFLUSH 3 // Cambia gli attributi quando l'output
// è stato utilizzato e scarica l'input
// ancora sospeso.

int tcgetattr (int fdn, struct termios *termios_p);
int tcsetattr (int fdn, int action,
               struct termios *termios_p);
```

Le due funzioni mostrate nell'esempio richiedono l'indicazione del numero di descrittore associato a un terminale aperto. Il parametro *termios\_p* è un puntatore a una struttura con le informazioni sulla configurazione del terminale: la funzione *tcgetattr()* serve a ottenere la configurazione attuale, mentre la funzione *tcsetattr()* serve a modificarla. Il parametro *action* di *tcsetattr()* richiede di precisare la tempestività di tale modifica attraverso una delle macro-variabili elencate poco sopra.

## 70.8 Riferimenti

«

- Wikipedia, *C POSIX library*, [http://en.wikipedia.org/wiki/C\\_POSIX\\_library](http://en.wikipedia.org/wiki/C_POSIX_library)
- The Open Group, *The Single UNIX® Specification, Version 2, System Interface & Headers Issue 5*, <http://pubs.opengroup.org/onlinepubs/007908799/xshix.html>
- Free Software Foundation, *The GNU C Library*, <http://www.gnu.org/software/libc/manual/>
- The Open Group, *The Single UNIX® Specification, Version 2, sys/types.h, sys/stat.h strings.h fcntl.h unistd.h dirent.h termios.h*

<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/sys/types.h.html> ,  
<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/sys/stat.h.html> ,  
<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/strings.h.html> ,  
<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/fcntl.h.html> ,  
<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/unistd.h.html> ,  
<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/dirent.h.html> ,  
<http://pubs.opengroup.org/onlinepubs/000095399/basedefs/termios.h.html>

- Pagina di manuale *fcntl(2)*
- The Open Group, *The Single UNIX® Specification, Version 2*, *fcntl*, <http://pubs.opengroup.org/onlinepubs/000095399/functions/fcntl.html>
- The Open Group, *The Single UNIX® Specification, Version 2*, *General Terminal Interface*, [http://pubs.opengroup.org/onlinepubs/000095399/basedefs/xbd\\_chap11.html](http://pubs.opengroup.org/onlinepubs/000095399/basedefs/xbd_chap11.html)

<sup>1</sup> Di norma, il programmatore non accede direttamente a variabili di tipo ‘**struct flock**’, perché per la gestione dei blocchi si usano semplicemente le funzioni appropriate.





# Tabelle riepilogative della libreria C e POSIX



File «stdarg.h»	.....	1359
File «limits.h»	.....	1360
File «stdint.h»		1361
File «inttypes.h»		1365
File «ctype.h»	.....	1375
File «stdlib.h»	..	1379
File «string.h»	.....	1390
File «time.h»	.....	1399
File «stdio.h» per la gestione dei file e degli errori		1406
File «stdio.h» per la composizione dell'output	...	1421
File «stdio.h» per l'interpretazione dell'input	.....	1428
File «assert.h»	.....	1435
File «stddef.h»	.....	1435
File «locale.h»	.....	1436
File «regex.h»	.....	1437
File «sys/stat.h»		1442

abort 1379 abs() 1379 asctime() 1399 assert() 1435  
 assert.h 1435 atexit() 1379 atof() 1379 atoi() 1379  
 atol() 1379 atoll() 1379 bsearch() 1379 BUFSIZ 1406  
 calloc() 1379 CHAR\_BIT 1360 CHAR\_MAX 1360 CHAR\_MIN  
 1360 chmod() 1442 clearerr() 1406 clock() 1399  
 ctime() 1399 ctype.h 1375 difftime() 1399 div() 1379  
 EOF 1406 exit() 1379 fchmod() 1442 fclose() 1406  
 feof() 1406 ferror() 1406 fflush() 1406 fgetc() 1406  
 fgetpos() 1406 fgets() 1406 FILENAME\_MAX 1406  
 fopen() 1406 FOPEN\_MAX 1406 fprintf() 1421 fputc()  
 1406 fputs() 1406 fread() 1406 free() 1379 freopen()  
 1406 fscanf() 1428 fseek() 1406 fsetpos() 1406  
 fstat() 1442 ftell() 1406 fwrite() 1406 gets() 1406  
 gmtime() 1399 INT16\_C() 1361 INT16\_MAX 1361  
 INT16\_MIN 1361 int16\_t 1361 INT32\_C() 1361  
 INT32\_MAX 1361 INT32\_MIN 1361 int32\_t 1361  
 INT64\_C() 1361 INT64\_MAX 1361 INT64\_MIN 1361  
 int64\_t 1361 INT8\_C() 1361 INT8\_MAX 1361 INT8\_MIN  
 1361 int8\_t 1361 INTMAX\_C() 1361 INTMAX\_MAX 1361  
 INTMAX\_MIN 1361 intmax\_t 1361 INTPTR\_MAX 1361  
 INTPTR\_MIN 1361 intptr\_t 1361 inttypes.h 1365  
 INT\_FAST16\_MAX 1361 INT\_FAST16\_MIN 1361  
 int\_fast16\_t 1361 INT\_FAST32\_MAX 1361  
 INT\_FAST32\_MIN 1361 int\_fast32\_t 1361  
 INT\_FAST64\_MAX 1361 INT\_FAST64\_MIN 1361  
 int\_fast64\_t 1361 INT\_FAST8\_MAX 1361  
 INT\_FAST8\_MIN 1361 int\_fast8\_t 1361  
 INT\_LEAST16\_MAX 1361 INT\_LEAST16\_MIN 1361  
 int\_least16\_t 1361 INT\_LEAST32\_MAX 1361

INT\_LEAST32\_MIN [1361](#) int\_least32\_t [1361](#)  
 INT\_LEAST64\_MAX [1361](#) INT\_LEAST64\_MIN [1361](#)  
 int\_least64\_t [1361](#) INT\_LEAST8\_MAX [1361](#)  
 INT\_LEAST8\_MIN [1361](#) int\_least8\_t [1361](#) INT\_MAX [1360](#)  
 INT\_MIN [1360](#) isalnum() [1375](#) isalpha() [1375](#)  
 isblank() [1375](#) iscntrl() [1375](#) isdigit() [1375](#)  
 isgraph() [1375](#) islower() [1375](#) isprint() [1375](#)  
 ispunct() [1375](#) isspace() [1375](#) isupper() [1375](#)  
 isxdigit() [1375](#) labs() [1379](#) ldiv() [1379](#) limits.h  
[1360](#) llabs() [1379](#) lldiv() [1379](#) LLONG\_MAX [1360](#)  
 LLONG\_MIN [1360](#) locale.h [1436](#) localtime() [1399](#)  
 LONG\_MAX [1360](#) LONG\_MIN [1360](#) lstat() [1442](#) L\_tmpnam  
[1406](#) malloc() [1379](#) mblen() [1379](#) mbstowcs() [1379](#)  
 mbtowc() [1379](#) MB\_LEN\_MAX [1360](#) memchr() [1390](#)  
 memcmp() [1390](#) memcpy() [1390](#) memmove() [1390](#)  
 memset() [1390](#) mkdir() [1442](#) mkfifo() [1442](#) mknod()  
[1442](#) mktime() [1399](#) offsetof() [1435](#) perror() [1406](#)  
 PRId16 [1365](#) PRId32 [1365](#) PRId64 [1365](#) PRId8 [1365](#)  
 PRIdFAST16 [1365](#) PRIdFAST32 [1365](#) PRIdFAST64 [1365](#)  
 PRIdFAST8 [1365](#) PRIdLEAST16 [1365](#) PRIdLEAST32 [1365](#)  
 PRIdLEAST64 [1365](#) PRIdLEAST8 [1365](#) PRIdMAX [1365](#)  
 PRIdPTR [1365](#) PRIi16 [1365](#) PRIi32 [1365](#) PRIi64 [1365](#)  
 PRIi8 [1365](#) PRIiFAST16 [1365](#) PRIiFAST32 [1365](#)  
 PRIiFAST64 [1365](#) PRIiFAST8 [1365](#) PRIiLEAST16 [1365](#)  
 PRIiLEAST32 [1365](#) PRIiLEAST64 [1365](#) PRIiLEAST8 [1365](#)  
 PRIiMAX [1365](#) PRIiPTR [1365](#) printf() [1421](#) PRIo16 [1365](#)  
 PRIo32 [1365](#) PRIo64 [1365](#) PRIo8 [1365](#) PRIoFAST16 [1365](#)  
 PRIoFAST32 [1365](#) PRIoFAST64 [1365](#) PRIoFAST8 [1365](#)  
 PRIoLEAST16 [1365](#) PRIoLEAST32 [1365](#) PRIoLEAST64 [1365](#)

PRIoLEAST8 1365 PRIoMAX 1365 PRIoPTR 1365 PRIu16  
 1365 PRIu32 1365 PRIu64 1365 PRIu8 1365 PRIuFAST16  
 1365 PRIuFAST32 1365 PRIuFAST64 1365 PRIuFAST8 1365  
 PRIuLEAST16 1365 PRIuLEAST32 1365 PRIuLEAST64 1365  
 PRIuLEAST8 1365 PRIuMAX 1365 PRIuPTR 1365 PRIx16  
 1365 PRIx16 1365 PRIx32 1365 PRIx32 1365 PRIx64 1365  
 PRIx64 1365 PRIx8 1365 PRIx8 1365 PRIxFAST16 1365  
 PRIxFAST16 1365 PRIxFAST32 1365 PRIxFAST32 1365  
 PRIxFAST64 1365 PRIxFAST64 1365 PRIxFAST8 1365  
 PRIxFAST8 1365 PRIxLEAST16 1365 PRIxLEAST16 1365  
 PRIxLEAST32 1365 PRIxLEAST32 1365 PRIxLEAST64 1365  
 PRIxLEAST64 1365 PRIxLEAST8 1365 PRIxLEAST8 1365  
 PRIxMAX 1365 PRIxMAX 1365 PRIxPTR 1365 PRIxPTR 1365  
 PTRDIFF\_MAX 1361 PTRDIFF\_MIN 1361 ptrdiff\_t 1361  
 putchar() 1406 puts() 1406 qsort() 1379 rand() 1379  
 realloc() 1379 regcomp() 1437 regerror() 1437  
 regex.h 1437 regexec() 1437 regex\_t 1437 regfree()  
 1437 regmatch\_t 1437 1437 regoff\_t 1437 remove()  
 1406 rename() 1406 rewind() 1406 re\_sub 1437 rm\_se  
 1437 rm\_so 1437 scanf() 1428 SCHAR\_MAX 1360  
 SCHAR\_MIN 1360 SCNd16 1365 SCNd32 1365 SCNd64 1365  
 SCNd8 1365 SCNdFAST16 1365 SCNdFAST32 1365  
 SCNdFAST64 1365 SCNdFAST8 1365 SCNdLEAST16 1365  
 SCNdLEAST32 1365 SCNdLEAST64 1365 SCNdLEAST8 1365  
 SCNdMAX 1365 SCNdPTR 1365 SCNi16 1365 SCNi32 1365  
 SCNi64 1365 SCNi8 1365 SCNiFAST16 1365 SCNiFAST32  
 1365 SCNiFAST64 1365 SCNiFAST8 1365 SCNiLEAST16  
 1365 SCNiLEAST32 1365 SCNiLEAST64 1365 SCNiLEAST8  
 1365 SCNiMAX 1365 SCNiPTR 1365 SCNo16 1365 SCNo32

[1365](#)    [SCNo64](#) [1365](#)    [SCNo8](#) [1365](#)    [SCNoFAST16](#) [1365](#)  
[SCNoFAST32](#) [1365](#)    [SCNoFAST64](#) [1365](#)    [SCNoFAST8](#) [1365](#)  
[SCNoLEAST16](#) [1365](#) [SCNoLEAST32](#) [1365](#) [SCNoLEAST64](#) [1365](#)  
[SCNoLEAST8](#) [1365](#)    [SCNoMAX](#) [1365](#)    [SCNoPTR](#) [1365](#)    [SCNu16](#)  
[1365](#) [SCNu32](#) [1365](#) [SCNu64](#) [1365](#) [SCNu8](#) [1365](#) [SCNuFAST16](#)  
[1365](#) [SCNuFAST32](#) [1365](#) [SCNuFAST64](#) [1365](#) [SCNuFAST8](#) [1365](#)  
[SCNuLEAST16](#) [1365](#) [SCNuLEAST32](#) [1365](#) [SCNuLEAST64](#) [1365](#)  
[SCNuLEAST8](#) [1365](#)    [SCNuMAX](#) [1365](#)    [SCNuPTR](#) [1365](#)    [SCNx16](#)  
[1365](#) [SCNx32](#) [1365](#) [SCNx64](#) [1365](#) [SCNx8](#) [1365](#) [SCNxFAST16](#)  
[1365](#) [SCNxFAST32](#) [1365](#) [SCNxFAST64](#) [1365](#) [SCNxFAST8](#) [1365](#)  
[SCNxLEAST16](#) [1365](#) [SCNxLEAST32](#) [1365](#) [SCNxLEAST64](#) [1365](#)  
[SCNxLEAST8](#) [1365](#) [SCNxMAX](#) [1365](#) [SCNxPTR](#) [1365](#)    [SEEK\\_CUR](#)  
[1406](#)    [SEEK\\_END](#) [1406](#)    [SEEK\\_SET](#) [1406](#)    [setbuf\(\)](#) [1406](#)  
[setlocale\(\)](#) [1436](#)    [setvbuf\(\)](#) [1406](#)    [SHRT\\_MAX](#) [1360](#)  
[SHRT\\_MIN](#) [1360](#)    [SIG\\_ATOMIC\\_MAX](#) [1361](#)    [SIG\\_ATOMIC\\_MIN](#)  
[1361](#)    [sig\\_atomic\\_t](#) [1361](#)    [SIZE\\_MAX](#) [1361](#)    [size\\_t](#) [1361](#)  
[snprintf\(\)](#) [1421](#)    [sprintf\(\)](#) [1421](#)    [srand\(\)](#) [1379](#)  
[sscanf\(\)](#) [1428](#)    [stat\(\)](#) [1442](#)    [stat.h](#) [1442](#)    [stdarg.h](#) [1359](#)  
[stddef.h](#) [1435](#)    [stdint.h](#) [1361](#)    [stdio.h](#) [1406](#) [1421](#) [1428](#)  
[stdlib.h](#) [1379](#)    [strcat\(\)](#) [1390](#)    [strchr\(\)](#) [1390](#)    [strcmp\(\)](#)  
[1390](#)    [strcoll\(\)](#) [1390](#)    [strcpy\(\)](#) [1390](#)    [strcspn\(\)](#) [1390](#)  
[strerror\(\)](#) [1390](#)    [strftime\(\)](#) [1399](#)    [string.h](#) [1390](#)  
[strlen\(\)](#) [1390](#)    [strncat\(\)](#) [1390](#)    [strncmp\(\)](#) [1390](#)  
[strncpy\(\)](#) [1390](#)    [strpbrk\(\)](#) [1390](#)    [strrchr\(\)](#) [1390](#)  
[strspn\(\)](#) [1390](#)    [strstr\(\)](#) [1390](#)    [strtod\(\)](#) [1379](#)    [strtof\(\)](#)  
[1379](#)    [strtok\(\)](#) [1390](#)    [strtol\(\)](#) [1379](#)    [strtold\(\)](#) [1379](#)  
[strtoll\(\)](#) [1379](#)    [strtoul\(\)](#) [1379](#)    [strtoull\(\)](#) [1379](#)  
[strxfrm\(\)](#) [1390](#)    [st\\_atime](#) [1442](#)    [st\\_blksize](#) [1442](#)  
[st\\_blocks](#) [1442](#)    [st\\_ctime](#) [1442](#)    [st\\_dev](#) [1442](#)    [st\\_gid](#) [1442](#)

st\_ino 1442 st\_mode 1442 st\_mtime 1442 st\_nlink 1442  
st\_rdev 1442 st\_size 1442 st\_uid 1442 S\_IFBLK 1442  
S\_IFCHR 1442 S\_IFDIR 1442 S\_IFIFO 1442 S\_IFLNK 1442  
S\_IFMT 1442 S\_IFREG 1442 S\_IFSOCK 1442 S\_IRGRP 1442  
S\_IROTH 1442 S\_IRUSR 1442 S\_IRWXG 1442 S\_IRWXO 1442  
S\_IRWXU 1442 S\_ISBLK() 1442 S\_ISCHR() 1442  
S\_ISDIR() 1442 S\_ISFIFO() 1442 S\_ISGID 1442  
S\_ISLNK() 1442 S\_ISREG() 1442 S\_ISSOCK() 1442  
S\_ISUID 1442 S\_ISVTX 1442 S\_IWGRP 1442 S\_IWOTH 1442  
S\_IWUSR 1442 S\_IXGRP 1442 S\_IXOTH 1442 S\_IXUSR 1442  
time() 1399 time.h 1399 tmpfile() 1406 tmpnam() 1406  
TMP\_MAX 1406 tolower() 1375 toupper() 1375  
UCHAR\_MAX 1360 UINT16\_C() 1361 UINT16\_MAX 1361  
uint16\_t 1361 UINT32\_C() 1361 UINT32\_MAX 1361  
uint32\_t 1361 UINT64\_C() 1361 UINT64\_MAX 1361  
uint64\_t 1361 UINT8\_C() 1361 UINT8\_MAX 1361  
uint8\_t 1361 UINTMAX\_C() 1361 UINTMAX\_MAX 1361  
uintmax\_t 1361 UINTPTR\_MAX 1361 uintptr\_t 1361  
UINT\_FAST16\_MAX 1361 uint\_fast16\_t 1361  
UINT\_FAST32\_MAX 1361 uint\_fast32\_t 1361  
UINT\_FAST64\_MAX 1361 uint\_fast64\_t 1361  
UINT\_FAST8\_MAX 1361 uint\_fast8\_t 1361  
UINT\_LEAST16\_MAX 1361 uint\_least16\_t 1361  
UINT\_LEAST32\_MAX 1361 uint\_least32\_t 1361  
UINT\_LEAST64\_MAX 1361 uint\_least64\_t 1361  
UINT\_LEAST8\_MAX 1361 uint\_least8\_t 1361  
1360 ULLONG\_MAX 1360 ULONG\_MAX 1360 umask() 1442  
ungetc() 1406 USHRT\_MAX 1360 va\_arg() 1359  
va\_copy() 1359 va\_end() 1359 va\_list 1359

va\_start() 1359      vfprintf() 1421      vfscanf() 1428  
 vprintf() 1421      vscanf() 1428      vsnprintf() 1421  
 vsprintf() 1421      vsscanf() 1428      WCHAR\_MAX 1361  
 WCHAR\_MIN 1361      wchar\_t 1361      wcstombs() 1379  
 wctomb() 1379      WINT\_MAX 1361      WINT\_MIN 1361      wint\_t  
 1361      \_Exit() 1379      \_IOFBF 1406      \_IOLBF 1406      \_IONBF 1406

## File «stdarg.h»

Macroistruzione	Descrizione
<pre>void va_start (va_list <i>ap</i>,                <i>parametro_n</i>);</pre>	<p>Inizializza la variabile <i>ap</i>, di tipo 'va_list', in modo che punti all'area di memoria immediatamente successiva al parametro indicato, il quale deve essere l'ultimo.</p>
<pre><i>tipo</i> va_arg (va_list <i>ap</i>, <i>tipo</i>);</pre>	<p>Restituisce il contenuto dell'area di memoria a cui punta <i>ap</i>, utilizzando il tipo indicato, incrementando contestualmente il puntatore in modo che, al termine, si trovi nell'area di memoria immediatamente successiva.</p>
<pre>void va_copy (va_list <i>dst</i>,               va_list <i>org</i>);</pre>	<p>Copia il puntatore <i>org</i> nella variabile <i>dst</i>.</p>
<pre>void va_end (va_list <i>ap</i>);</pre>	<p>Conclude l'utilizzo del puntatore <i>ap</i>.</p>

## File «limits.h»

«

Macro-variabile	Descrizione
CHAR_BIT	Quantità di bit utilizzata per rappresentare il tipo <code>'char'</code> , con o senza segno. In altri termini è l'unità di memorizzazione più piccola con cui si può gestire l'insieme di caratteri minimo. Di norma si tratta di 8 bit.
SCHAR_MIN SCHAR_MAX	Il valore minimo e il valore massimo rappresentabile in una variabile <code>'signed char'</code> .
UCHAR_MAX	Il valore massimo rappresentabile in una variabile <code>'unsigned char'</code> . Il valore minimo è zero.
CHAR_MIN CHAR_MAX	Il valore minimo e il valore massimo rappresentabile in una variabile <code>'char'</code> . Questi valori dipendono dal fatto che il tipo <code>'char'</code> sia da intendere equivalente a un tipo <code>'unsigned char'</code> o <code>'signed char'</code> , da cui ereditano i limiti.
MB_LEN_MAX	La quantità massima di byte che possono essere usati per rappresentare un carattere multibyte, qualunque sia la configurazione locale.
SHRT_MIN SHRT_MAX	Il valore minimo e il valore massimo rappresentabile in una variabile <code>'short int'</code> .
USHRT_MAX	Il valore massimo rappresentabile in una variabile <code>'unsigned short int'</code> . Il valore minimo è zero.
INT_MIN INT_MAX	Il valore minimo e il valore massimo rappresentabile in una variabile <code>'int'</code> .
UINT_MAX	Il valore massimo rappresentabile in una variabile <code>'unsigned int'</code> . Il valore minimo è zero.



Macro-variabile	Descrizione
LONG_MIN LONG_MAX	Il valore minimo e il valore massimo rappresentabile in una variabile <b>'long int'</b> .
ULONG_MAX	Il valore massimo rappresentabile in una variabile <b>'unsigned long int'</b> . Il valore minimo è zero.
LLONG_MIN LLONG_MAX	Il valore minimo e il valore massimo rappresentabile in una variabile <b>'long long int'</b> .
ULLONG_MAX	Il valore massimo rappresentabile in una variabile <b>'unsigned long long int'</b> . Il valore minimo è zero.

## File «stdint.h»

Con segno	Senza segno	Descrizione
int8_t	uint8_t	Tipo intero, facoltativo, il cui rango è prestabilito esattamente.
int16_t	uint16_t	
int32_t	uint32_t	
int64_t	uint64_t	



Con segno	Senza segno	Descrizione
INT8_MIN		Limiti minimi e massimi dei tipi ' <b>int<math>n</math>_t</b> ' e ' <b>uint<math>n</math>_t</b> '.
INT8_MAX		
INT16_MIN	UINT8_MAX	
INT16_MAX	UINT16_MAX	
INT32_MIN	UINT32_MAX	
INT32_MAX	UINT64_MAX	
INT64_MIN		
INT64_MAX		
int_least8_t	uint_least8_t	Tipo intero con un rango minimo stabilito.
int_least16_t	uint_least16_t	
int_least32_t	uint_least32_t	
int_least64_t	uint_least64_t	

Con segno	Senza segno	Descrizione
INT_LEAST8_MIN INT_LEAST8_MAX INT_LEAST16_MIN INT_LEAST16_MAX INT_LEAST32_MIN INT_LEAST32_MAX INT_LEAST64_MIN INT_LEAST64_MAX	UINT_LEAST8_MAX UINT_LEAST16_MAX UINT_LEAST32_MAX UINT_LEAST64_MAX	Limiti minimi e massimi dei tipi <code>'int_leastn_t'</code> e <code>'uint_leastn_t'</code> .
INT8_C( <i>val</i> ) INT16_C( <i>val</i> ) INT32_C( <i>val</i> ) INT64_C( <i>val</i> )	UINT8_C( <i>val</i> ) UINT16_C( <i>val</i> ) UINT32_C( <i>val</i> ) UINT64_C( <i>val</i> )	Macroistruzione per attribuire l'estensione che definisce il tipo corretto a un valore costante, da intendere secondo il tipo <code>'int_leastn_t'</code> o <code>'uint_leastn_t'</code> .
int_fast8_t int_fast16_t int_fast32_t int_fast64_t	uint_fast8_t uint_fast16_t uint_fast32_t uint_fast64_t	Tipo intero con un rango minimo stabilito, con caratteristiche ottimali per la velocità elaborativa.

Con segno	Senza segno	Descrizione
INT_FAST8_MIN INT_FAST8_MAX INT_FAST16_MIN INT_FAST16_MAX INT_FAST32_MIN INT_FAST32_MAX INT_FAST64_MIN INT_FAST64_MAX	UINT_FAST8_MAX UINT_FAST16_MAX UINT_FAST32_MAX UINT_FAST64_MAX	Limiti minimi e massimi dei tipi <code>'int_fastn_t'</code> e <code>'uint_fastn_t'</code> .
<code>intptr_t</code>	<code>uintptr_t</code>	Tipo facoltativo intero capace di contenere il valore di un puntatore, convertibile da e verso <code>'void *'</code> .
INTPTR_MIN INTPTR_MAX	UINTPTR_MAX	Limiti minimi e massimi dei tipi <code>'intptr_t'</code> e <code>'uintptr_t'</code> .
<code>intmax_t</code>	<code>uintmax_t</code>	Tipo intero di rango massimo.
INTMAX_MIN INTMAX_MAX	UINTMAX_MAX	Limiti minimi e massimi dei tipi <code>'intmax_t'</code> e <code>'uintmax_t'</code> .

Con segno	Senza segno	Descrizione
INTMAX_C ( <i>val</i> )	UINTMAX_C ( <i>val</i> )	Macroistruzione per attribuire l'estensione che definisce il tipo corretto a un valore costante, da intendere secondo il tipo <code>'intmax_t'</code> o <code>'uintmax_t'</code> .
PTRDIFF_MIN PTRDIFF_MAX		Limiti minimi e massimi del tipo <code>'ptrdiff_t'</code> .
SIG_ATOMIC_MIN SIG_ATOMIC_MAX		Limiti minimi e massimi del tipo <code>'sig_atomic_t'</code> .
	SIZE_MAX	Limite massimo del tipo <code>'size_t'</code> (senza segno).
WCHAR_MIN WCHAR_MAX		Limiti minimi e massimi del tipo <code>'wchar_t'</code> .
WINT_MIN WINT_MAX		Limiti minimi e massimi del tipo <code>'wint_t'</code> .

File «inttypes.h»



Macro-variabili per la composizione dell'output	Macro-variabili per l'interpretazione dell'input	Esempi schematici
PRId8 PRId16 PRId32 PRId64	SCNd8 SCNd16 SCNd32 SCNd64	<pre>int32_t i = INT32_MAX; ... printf ("i = %010" PRId32 "\n", i); ... scanf ("% " SCNd32, &amp;i);</pre>
PRIi8 PRIi16 PRIi32 PRIi64	SCNi8 SCNi16 SCNi32 SCNi64	<pre>int32_t i = INT32_MAX; ... printf ("i = %010" PRIi32 "\n", i); ... scanf ("% " SCNi32, &amp;i);</pre>
PRIdLEAST8 PRIdLEAST16 PRIdLEAST32 PRIdLEAST64	SCNdLEAST8 SCNdLEAST16 SCNdLEAST32 SCNdLEAST64	<pre>int_least32_t i = INT_LEAST32_MAX; ... printf ("i = %010" PRIdLEAST32 "\n", i); ... scanf ("% " SCNdLEAST32, &amp;i);</pre>

Macro-variabili per la composizione dell'output	Macro-variabili per l'interpretazione dell'input	Esempi schematici
PRIiLEAST8 PRIiLEAST16 PRIiLEAST32 PRIiLEAST64	SCNiLEAST8 SCNiLEAST16 SCNiLEAST32 SCNiLEAST64	<pre> int_least32_t i = INT_LEAST32_MAX; ... printf ("i = %010" PRIiLEAST32 "\n", i); ... scanf ("%i" SCNiLEAST32, &amp;i); </pre>
PRIdFAST8 PRIdFAST16 PRIdFAST32 PRIdFAST64	SCNdFAST8 SCNdFAST16 SCNdFAST32 SCNdFAST64	<pre> int_fast32_t i = INT_FAST32_MAX; ... printf ("i = %010" PRIdFAST32 "\n", i); ... scanf ("%i" SCNdFAST32, &amp;i); </pre>
PRIiFAST8 PRIiFAST16 PRIiFAST32 PRIiFAST64	SCNiFAST8 SCNiFAST16 SCNiFAST32 SCNiFAST64	<pre> int_fast32_t i = INT_FAST32_MAX; ... printf ("i = %010" PRIiFAST32 "\n", i); ... scanf ("%i" SCNiFAST32, &amp;i); </pre>

Macro-variabili per la composizione dell'output	Macro-variabili per l'interpretazione dell'input	Esempi schematici
PRIiMAX PRIiPTR PRIiMAX PRIiPTR	SCNiMAX SCNiPTR SCNiMAX SCNiPTR	<pre> intmax_t i = INTMAX_MAX; ... printf ("i = %020" PRIiMAX "\n", i); ... scanf ("%i" SCNiMAX, &amp;i); </pre>
PRIo8 PRIo16 PRIo32 PRIo64	SCNo8 SCNo16 SCNo32 SCNo64	<pre> uint32_t i = UINT32_MAX; ... printf ("i = %011" PRIo32 "\n", i); ... scanf ("%i" SCNo32, &amp;i); </pre>
PRIoLEAST8 PRIoLEAST16 PRIoLEAST32 PRIoLEAST64	SCNoLEAST8 SCNoLEAST16 SCNoLEAST32 SCNoLEAST64	<pre> uint_least32_t i = UINT_LEAST32_MAX; ... printf ("i = %011" PRIoLEAST32 "\n", i); ... scanf ("%i" SCNoLEAST32, &amp;i); </pre>



Macro-variabili per la composizione dell'output	Macro-variabili per l'interpretazione dell'input	Esempi schematici
PRIoFAST8 PRIoFAST16 PRIoFAST32 PRIoFAST64	SCNoFAST8 SCNoFAST16 SCNoFAST32 SCNoFAST64	<pre> uint_fast32_t i = UINT_FAST32_MAX; ... printf ("i = %011" PRIoFAST32 "\n", i); ... scanf ("%k" SCNoFAST32, &amp;i); </pre>
PRIoMAX PRIoPTR	SCNoMAX SCNoPTR	<pre> uintmax_t i = INTMAX_MAX; ... printf ("i = %022" PRIoMAX "\n", i); ... scanf ("%k" SCNoMAX, &amp;i); </pre>
PRIu8 PRIu16 PRIu32 PRIu64	SCNu8 SCNu16 SCNu32 SCNu64	<pre> uint32_t i = UINT32_MAX; ... printf ("i = %010" PRIu32 "\n", i); ... scanf ("%k" SCNu32, &amp;i); </pre>
PRIuLEAST8 PRIuLEAST16 PRIuLEAST32 PRIuLEAST64	SCNuLEAST8 SCNuLEAST16 SCNuLEAST32 SCNuLEAST64	<pre> uint_least32_t i = UINT_LEAST32_MAX; ... printf ("i = %010" PRIuLEAST32 "\n", i); ... scanf ("%k" SCNuLEAST32, &amp;i); </pre>

Macro-variabili per la composizione dell'output	Macro-variabili per l'interpretazione dell'input	Esempi schematici
PRIuFAST8 PRIuFAST16 PRIuFAST32 PRIuFAST64	SCNuFAST8 SCNuFAST16 SCNuFAST32 SCNuFAST64	<pre> uint_fast32_t i = UINT_FAST32_MAX; ... printf ("i = %010" PRIuFAST32 "\n", i); ... scanf ("%u" SCNuFAST32, &amp;i); </pre>
PRIuMAX PRIuPTR	SCNuMAX SCNuPTR	<pre> uintmax_t i = INTMAX_MAX; ... printf ("i = %022" PRIuMAX "\n", i); ... scanf ("%u" SCNuMAX, &amp;i); </pre>
PRIx8 PRIx16 PRIx32 PRIx64	SCNx8 SCNx16 SCNx32 SCNx64	<pre> uint32_t i = UINT32_MAX; ... printf ("i = %08" PRIx32 "\n", i); ... scanf ("%x" SCNx32, &amp;i); </pre>

Macro-variabili per la composizione dell'output	Macro-variabili per l'interpretazione dell'input	Esempi schematici
PRIxLEAST8 PRIxLEAST8 PRIxLEAST16 PRIxLEAST16 PRIxLEAST32 PRIxLEAST32 PRIxLEAST64 PRIxLEAST64	SCNxLEAST8 SCNxLEAST16 SCNxLEAST32 SCNxLEAST64	<pre> uint_least32_t i = UINT_LEAST32_MAX; ... printf ("i = %08" PRIxLEAST32 "\n", i); ... scanf ("% " SCNxLEAST32, &amp;i); </pre>
PRIxFAST8 PRIxFAST8 PRIxFAST16 PRIxFAST16 PRIxFAST32 PRIxFAST32 PRIxFAST64 PRIxFAST64	SCNxFAST8 SCNxFAST16 SCNxFAST32 SCNxFAST64	<pre> uint_fast32_t i = UINT_FAST32_MAX; ... printf ("i = %08" PRIxFAST32 "\n", i); ... scanf ("% " SCNxFAST32, &amp;i); </pre>
PRIxMAX PRIxMAX PRIxPTR PRIxPTR	SCNxMAX SCNxPTR	<pre> uintmax_t i = UINTMAX_MAX; ... printf ("i = %016" PRIxMAX "\n", i); ... scanf ("% " SCNxMAX, &amp;i); </pre>

Funzione	Descrizione
<pre>intmax_t imaxabs (intmax_t <i>j</i>);</pre>	<p>Restituisce il valore assoluto del numero passato come argomento.</p>
<pre>imaxdiv_t imaxdiv (intmax_t <i>numer</i>,                     intmax_t <i>denom</i>);</pre>	<p>Restituisce il risultato della divisione dei due argomenti, in una struttura contenente il risultato intero e il resto della divisione.</p>

Funzione	Descrizione
<pre> intmax_t strtouimax     (const char *restrict <i>s</i>,      char **restrict <i>p</i>,      int <i>base</i>); uintmax_t strtouimax     (const char *restrict <i>s</i>,      char **restrict <i>p</i>,      int <i>base</i>); </pre>	<p>Converte la stringa fornita come primo argomento in un numero intero, come si vede dal modello sintattico, interpretando la stringa come numero espresso nella base rappresentata dal parametro <i>base</i>. La conversione avviene fino a dove è possibile riconoscere caratteri che compongono un valore valido; se il secondo argomento è un puntatore a un puntatore valido (un puntatore a un'area di memoria che può contenere a sua volta un puntatore dal tipo 'char'), al suo interno viene memorizzato l'indirizzo finale della scansione, a partire dal quale si trovano caratteri non decifrabili, oppure dove si trova il carattere nullo di terminazione della stringa.</p>

Funzione	Descrizione
<pre> intmax_t wcstoimax     (const wchar_t *restrict <i>wcs</i>,      wchar_t **restrict <i>p</i>,      int <i>base</i>);  uintmax_t wcstouimax     (const wchar_t *restrict <i>wcs</i>,      wchar_t **restrict <i>p</i>,      int <i>base</i>); </pre>	<p>Converte la stringa estesa fornita come primo argomento in un numero intero, come si vede dal modello sintattico, interpretando la stringa estesa come numero espresso nella base rappresentata dal parametro <i>base</i>. La conversione avviene fino a dove è possibile riconoscere caratteri estesi che compongono un valore valido; se il secondo argomento è un puntatore a un puntatore valido (un puntatore a un'area di memoria che può contenere a sua volta un puntatore dal tipo 'wchar_t'), al suo interno viene memorizzato l'indirizzo finale della scansione, a partire dal quale si trovano caratteri estesi non decifrabili, oppure dove si trova il carattere nullo di terminazione della stringa.</p>

## File «ctype.h»



Funzione	Descrizione
<pre>int isalnum (int c);</pre>	L'argomento rappresentato dal parametro <i>c</i> è un carattere senza segno convertito in un intero, oppure l'equivalente di 'EOF'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere alfabetico o numerico. Equivale alla corrispondenza con <i>isalpha()</i> o con <i>isdigit()</i> .
<pre>int isalpha (int c);</pre>	L'argomento rappresentato dal parametro <i>c</i> è un carattere senza segno convertito in un intero, oppure l'equivalente di 'EOF'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere alfabetico. Equivale alla corrispondenza con <i>isupper()</i> o con <i>islower()</i> .
<pre>int isblank (int c);</pre>	L'argomento rappresentato dal parametro <i>c</i> è un carattere senza segno convertito in un intero, oppure l'equivalente di 'EOF'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere per la spaziatura orizzontale delle parole.

Funzione	Descrizione
<pre>int iscntrl (int c);</pre>	<p>L'argomento rappresentato dal parametro <i>c</i> è un carattere senza segno convertito in un intero, oppure l'equivalente di <b>'EOF'</b>. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere di controllo.</p>
<pre>int isdigit (int c);</pre>	<p>L'argomento rappresentato dal parametro <i>c</i> è un carattere senza segno convertito in un intero, oppure l'equivalente di <b>'EOF'</b>. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere che rappresenta una cifra decimale.</p>
<pre>int isgraph (int c);</pre>	<p>L'argomento rappresentato dal parametro <i>c</i> è un carattere senza segno convertito in un intero, oppure l'equivalente di <b>'EOF'</b>. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere che ha una rappresentazione grafica, escluso lo spazio.</p>
<pre>int islower (int c);</pre>	<p>L'argomento rappresentato dal parametro <i>c</i> è un carattere senza segno convertito in un intero, oppure l'equivalente di <b>'EOF'</b>. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere alfabetico minuscolo.</p>



Funzione	Descrizione
<pre>int isprint (int c);</pre>	<p>L'argomento rappresentato dal parametro <i>c</i> è un carattere senza segno convertito in un intero, oppure l'equivalente di 'EOF'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere che ha una rappresentazione grafica, incluso lo spazio.</p>
<pre>int ispunct (int c);</pre>	<p>L'argomento rappresentato dal parametro <i>c</i> è un carattere senza segno convertito in un intero, oppure l'equivalente di 'EOF'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere di punteggiatura.</p>
<pre>int isspace (int c);</pre>	<p>L'argomento rappresentato dal parametro <i>c</i> è un carattere senza segno convertito in un intero, oppure l'equivalente di 'EOF'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere di spaziatura, sia orizzontale, sia verticale, incluso il salto pagina e il ritorno a carrello.</p>

Funzione	Descrizione
<pre>int isupper (int c);</pre>	<p>L'argomento rappresentato dal parametro <i>c</i> è un carattere senza segno convertito in un intero, oppure l'equivalente di 'EOF'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere alfabetico maiuscolo.</p>
<pre>int isxdigit (int c);</pre>	<p>L'argomento rappresentato dal parametro <i>c</i> è un carattere senza segno convertito in un intero, oppure l'equivalente di 'EOF'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere che rappresenta una cifra esadecimale (espressa indifferentemente con lettere minuscole o maiuscole).</p>
<pre>int tolower (int c);</pre>	<p>L'argomento rappresentato dal parametro <i>c</i> è un carattere senza segno convertito in un intero, oppure l'equivalente di 'EOF'. La funzione restituisce il carattere fornito come argomento, dopo la conversione in minuscolo, ammesso che ci possa essere una conversione.</p>

Funzione	Descrizione
<pre>int toupper (int <i>c</i>);</pre>	<p>L'argomento rappresentato dal parametro <i>c</i> è un carattere senza segno convertito in un intero, oppure l'equivalente di 'EOF'. La funzione restituisce il carattere fornito come argomento, dopo la conversione in maiuscolo, ammesso che ci possa essere una conversione.</p>

## File «stdlib.h»

Funzione	Descrizione
<pre>int atoi (const char *<i>s</i>); long int atol (const char *<i>s</i>); long long int atoll (const char *<i>s</i>); double atof (const char *<i>s</i>);</pre>	<p>Converte la stringa fornita come argomento in un numero intero o in virgola mobile, come si vede dal modello sintattico.</p>



Funzione	Descrizione
<pre>float strtod (const char *restrict <i>s</i>,               char **restrict <i>p</i>); double strtod     (const char *restrict <i>s</i>,      char **restrict <i>p</i>); long double strtold     (const char *restrict <i>s</i>,      char **restrict <i>p</i>);</pre>	<p>Converte la stringa fornita come primo argomento in un numero in virgola mobile, come si vede dal modello sintattico. La conversione avviene fino a dove è possibile riconoscere caratteri che compongono un valore valido; se il secondo argomento è un puntatore a un puntatore valido (un puntatore a un'area di memoria che può contenere a sua volta un puntatore dal tipo '<b>char</b>'), al suo interno viene memorizzato l'indirizzo finale della scansione, a partire dal quale si trovano caratteri non decifrabili, oppure dove si trova il carattere nullo di terminazione della stringa.</p>

Funzione	Descrizione
<pre> long int strtol     (const char *restrict <i>s</i>,      char **restrict <i>p</i>,      int <i>base</i>);  long long int strtoll     (const char *restrict <i>s</i>,      char **restrict <i>p</i>,      int <i>base</i>);  unsigned long int strtoul     (const char *restrict <i>s</i>,      char **restrict <i>p</i>,      int <i>base</i>);  unsigned long long int strtoull     (const char *restrict <i>s</i>,      char **restrict <i>p</i>,      int <i>base</i>); </pre>	<p>Converte la stringa fornita come primo argomento in un numero intero, come si vede dal modello sintattico, interpretando la stringa come numero espresso nella base rappresentata dal parametro <i>base</i>. La conversione avviene fino a dove è possibile riconoscere caratteri che compongono un valore valido; se il secondo argomento è un puntatore a un puntatore valido (un puntatore a un'area di memoria che può contenere a sua volta un puntatore dal tipo '<b>char</b>'), al suo interno viene memorizzato l'indirizzo finale della scansione, a partire dal quale si trovano caratteri non decifrabili, oppure dove si trova il carattere nullo di terminazione della stringa.</p>
<pre> void srand (unsigned int <i>seed</i>); </pre>	<p>Modifica il seme per la generazione di numeri casuali attraverso la funzione <i>rand()</i>.</p>

Funzione	Descrizione
<pre>int rand (void);</pre>	<p>Restituisce il numero casuale successivo.</p>
<pre>void *malloc (size_t <i>size</i>);</pre>	<p>Richiede l'allocazione di memoria di almeno <i>size</i> byte, restituendo il puntatore all'inizio della stessa se l'operazione ha successo, oppure il puntatore nullo se l'allocazione fallisce.</p>
<pre>void *calloc (size_t <i>nmemb</i>,               size_t <i>size</i>);</pre>	<p>Richiede l'allocazione di memoria di almeno <i>nmemb</i> elementi da <i>size</i> byte ciascuno, restituendo il puntatore all'inizio della stessa se l'operazione ha successo, oppure il puntatore nullo se l'allocazione fallisce.</p>

Funzione	Descrizione
<pre>void *realloc (void *<i>ptr</i>, size_t <i>size</i>);</pre>	<p>Richiede la riallocazione della memoria già allocata precedentemente a partire dall'indirizzo <i>ptr</i>, in modo da avere almeno <i>size</i> byte, recuperando il contenuto precedente, per ciò che è possibile. La riallocazione può avvenire in corrispondenza di un indirizzo differente da quello originale, ma può fallire, restituendo così solo il puntatore nullo.</p>
<pre>void free (void *<i>ptr</i>);</pre>	<p>Libera la memoria allocata precedente a partire dall'indirizzo <i>ptr</i>.</p>
<pre>int atexit (void (*<i>func</i>) (void));</pre>	<p>Accumula in un elenco il puntatore a una funzione che non richiede argomenti, da eseguire, assieme alle altre dell'elenco, quando viene chiamata la funzione <i>exit()</i>. La funzione <i>atexit()</i> restituisce un valore numerico da intendere come <i>Vero</i> o <i>Falso</i>, a indicare se l'operazione ha avuto successo o meno.</p>

Funzione	Descrizione
<pre>void exit (int <i>status</i>);</pre>	<p>Conclude il funzionamento del programma, ma prima esegue le funzioni accumulate con l'ausilio di <i>atexit()</i>, quindi chiude i file e infine passa il valore ricevuto come argomento in modo tale che sia restituito dal programma stesso.</p>
<pre>void _Exit (int <i>status</i>);</pre>	<p>Conclude il funzionamento del programma in modo brutale, senza occuparsi di nulla, a parte il far sì che il programma restituisca il valore indicato come argomento.</p>
<pre>void abort (void);</pre>	<p>Produce l'emissione del segnale '<b>SIGABRT</b>' (<i>abort</i>) che porta alla morte del processo elaborativo.</p>
<pre>char *getenv (const char *<i>name</i>);</pre>	<p>Restituisce il puntatore all'inizio della stringa che rappresenta il contenuto della variabile di ambiente indicata per nome, come argomento.</p>



Funzione	Descrizione
<pre>int system (const char *<i>string</i>);</pre>	<p>Esegue il comando indicato come argomento, attraverso il sistema operativo, restituendo il valore di uscita del comando stesso.</p>
<pre>void qsort (void *<i>base</i>,             size_t <i>nmemb</i>,             size_t <i>size</i>,             int (*<i>compar</i>)               (const void *,                const void *));</pre>	<p>Riordina un array che inizia dall'indirizzo <i>base</i>, essendo composto da <i>size</i> elementi da <i>nmemb</i> byte ognuno, utilizzando per il confronto la funzione <i>compar</i>.</p>
<pre>void *bsearch (const void *<i>key</i>,                const void *<i>base</i>,                size_t <i>nmemb</i>,                size_t <i>size</i>,                int (*<i>compar</i>)                  (const void *,                   const void *));</pre>	<p>Scandisce un array che inizia dall'indirizzo <i>base</i>, essendo composto da <i>size</i> elementi da <i>nmemb</i> byte ognuno, il quale risulta già ordinato secondo la funzione <i>compar</i>, alla ricerca della corrispondenza con il valore <i>*key</i>, la cui dimensione deve essere sempre di <i>nmemb</i> byte.</p>
<pre>int abs (int <i>j</i>);  long int labs (long int <i>j</i>);  long long int llabs   (long long int <i>j</i>);</pre>	<p>Restituisce il valore assoluto di <i>j</i>.</p>

Funzione	Descrizione
<pre>div_t div (int <i>numeratore</i> ,            int <i>denominatore</i> ); ldiv_t ldiv (long int <i>numeratore</i> ,              long int <i>denominatore</i> ); lldiv_t lldiv (long long int <i>numeratore</i> ,  long long int <i>denominatore</i> );</pre>	<p>Restituisce il risultato della divisione dei due argomenti, in una struttura contenente il risultato intero e il resto della divisione.</p>
<pre>int mblen (const char *<i>s</i> , size_t <i>n</i> );</pre>	<p>Restituisce la lunghezza in byte del primo carattere multibyte contenuto nella stringa fornita come primo argomento. La scansione termina comunque se raggiunge la quantità di byte indicata dal secondo argomento. Se la stringa contiene una sequenza multibyte errata o incompleta, il valore restituito è -1. Se al posto della stringa multibyte si fornisce il puntatore nullo, la funzione restituisce il valore uno o zero, a seconda che sia prevista o meno una codifica multibyte con una gestione dello stato (<i>shift state</i>).</p>

Funzione	Descrizione
<pre data-bbox="108 568 896 739">int mbtowc (wchar_t *restrict <i>pwc</i>,             const char *restrict <i>s</i>,             size_t <i>n</i>);</pre>	<p data-bbox="970 159 1487 1120">           Converte il carattere multibyte contenuto nella stringa <i>s</i>, per un massimo di <i>n</i> byte, nel carattere esteso a cui punta <i>pwc</i>, restituendo la quantità di byte utilizzati dalla stringa di origine, oppure <math>-1</math> se si presentano errori. Se al posto della stringa <i>s</i> si mette il puntatore nullo, si ottiene un valore pari a uno o zero, a seconda che sia prevista o meno una codifica multibyte con una gestione dello stato (<i>shift state</i>).         </p>

Funzione	Descrizione
<pre data-bbox="108 560 831 598">int wctomb (char *s, wchar_t wc;</pre>	<p data-bbox="970 159 1487 1003">           Converte il carattere esteso <i>wc</i> in una sequenza multibyte che va a essere contenuta nella stringa <i>s</i>, restituendo la quantità di byte prodotti, oppure <math>-1</math> se si presentano errori. Se al posto della stringa <i>s</i> si mette il puntatore nullo, si ottiene un valore pari a uno o zero, a seconda che sia prevista o meno una codifica multibyte con una gestione dello stato (<i>shift state</i>).         </p>

Funzione	Descrizione
<pre data-bbox="108 682 735 911"> size_t mbstowcs     (wchar_t *restrict <i>pwcs</i>,      const char *restrict <i>s</i>,      size_t <i>n</i>); </pre>	<p data-bbox="970 159 1487 1406">           Converte la stringa multi-byte <i>s</i> nella stringa estesa <i>pwcs</i>, producendo al massimo <i>n</i> caratteri estesi nella destinazione (incluso il carattere nullo di terminazione). La funzione restituisce la quantità di caratteri estesi copiati nella destinazione, escludendo il carattere nullo di terminazione, oppure l'equivalente di <math>-1</math> in caso di errori. Se al posto della destinazione viene messo il puntatore nullo, l'operazione viene simulata ignorando il valore di <i>n</i> e senza memorizzare il risultato; pertanto è utile per contare lo spazio necessario nella destinazione.         </p>

Funzione	Descrizione
<pre data-bbox="108 682 730 911"> size_t wcstombs (char *restrict <i>s</i>, wchar_t *restrict <i>pwcs</i>, size_t <i>n</i>); </pre>	<p data-bbox="970 159 1487 1406">         Converte la stringa estesa <i>pwcs</i> nella stringa multibyte <i>s</i>, producendo al massimo <i>n</i> byte nella destinazione (incluso il carattere nullo di terminazione). La funzione restituisce la quantità di byte copiati nella destinazione, escludendo il carattere nullo di terminazione, oppure l'equivalente di -1 in caso di errori. Se al posto della destinazione viene messo il puntatore nullo, l'operazione viene simulata ignorando il valore di <i>n</i> e senza memorizzare il risultato; pertanto è utile per contare lo spazio necessario nella destinazione.       </p>

File «string.h»



Funzione	Descrizione
<pre>void *memcpy     (void *restrict <i>dst</i>,      const void *restrict <i>org</i>,      size_t <i>n</i>);</pre>	<p>Copia <i>n</i> caratteri a partire dall'indirizzo indicato da <i>org</i>, per riprodurli a partire dall'indirizzo <i>dst</i>, alla condizione che i due insiemi non risultino sovrapposti. La funzione restituisce l'indirizzo <i>dst</i>.</p>
<pre>void *memmove     (void *<i>dst</i>, const void *<i>org</i>,      size_t <i>n</i>);</pre>	<p>Copia <i>n</i> caratteri a partire dall'indirizzo indicato da <i>org</i>, per riprodurli a partire dall'indirizzo <i>dst</i>, senza il vincolo che gli insiemi siano disgiunti. La funzione restituisce l'indirizzo <i>dst</i>.</p>
<pre>char *strcpy     (char *restrict <i>dst</i>,      const char *restrict <i>org</i>);</pre>	<p>Copia la stringa <i>org</i> nell'array a cui punta <i>dst</i>, includendo anche il carattere nullo di conclusione delle stringhe, alla condizione che le due stringhe non si sovrappongano. La funzione restituisce <i>dst</i>.</p>

Funzione	Descrizione
<pre>char *strncpy     (char *restrict <i>dst</i>,      const char *restrict <i>org</i>,      size_t <i>n</i>);</pre>	<p>Copia <i>n</i> caratteri della stringa <i>org</i> nell'array a cui punta <i>dst</i>, contando tra i caratteri copiati anche il carattere nullo di conclusione delle stringhe, alla condizione che le due stringhe non si sovrappongano. Se la stringa di origine è più corta di <i>n</i>, i caratteri mancanti sono rimpiazzati dal carattere nullo di conclusione delle stringhe. La funzione restituisce <i>dst</i>.</p>
<pre>char *strcat     (char *restrict <i>dst</i>,      const char *restrict <i>org</i>);</pre>	<p>Copia la stringa <i>org</i> a partire dalla fine della stringa <i>dst</i> (sovrascrivendo il carattere nullo preesistente), alla condizione che le due stringhe non siano sovrapposte. La funzione restituisce <i>dst</i>.</p>



Funzione	Descrizione
<pre>char *strncat     (char *restrict <i>dst</i>,      const char *restrict <i>org</i>,      size_t <i>n</i>);</pre>	<p>Copia al massimo <i>n</i> caratteri della stringa <i>org</i> a partire dalla fine della stringa <i>dst</i> (sovrascrivendo il carattere nullo preesistente), aggiungendo alla fine il carattere nullo di terminazione, il tutto alla condizione che le due stringhe non siano sovrapposte. La funzione restituisce <i>dst</i>.</p>
<pre>int memcmp     (const void *<i>s1</i>,      const void *<i>s2</i>, size_t <i>n</i>);</pre>	<p>Confronta i primi <i>n</i> caratteri delle aree di memoria a cui puntano <i>s1</i> e <i>s2</i>, restituendo: un valore pari a zero se le due sequenze si equivalgono; un valore maggiore di zero se la sequenza di <i>s1</i> è maggiore di <i>s2</i>; un valore minore di zero se la sequenza di <i>s1</i> è minore di <i>s2</i>.</p>
<pre>int strcmp (const char *<i>s1</i>,             const char *<i>s2</i>);</pre>	<p>Confronta due stringhe restituendo: un valore pari a zero se sono uguali; un valore maggiore di zero se la stringa <i>s1</i> è maggiore di <i>s2</i>; un valore minore di zero se la stringa <i>s1</i> è minore di <i>s2</i>.</p>

Funzione	Descrizione
<pre data-bbox="113 431 756 539">int strcoll (const char *s1,              const char *s2);</pre>	<p data-bbox="970 159 1485 778">La funzione <i>strcoll()</i> è analoga a <i>strcmp()</i>, con la differenza che la comparazione avviene sulla base della configurazione locale (la categoria <b>LC_COLLATE</b>). Nel caso della configurazione locale <b>C</b> la funzione si comporta esattamente come <i>strcmp()</i>.</p>
<pre data-bbox="113 911 847 1022">int strncmp (const char *s1,              char *s2, size_t n);</pre>	<p data-bbox="970 793 1485 1107">La funzione <i>strncmp()</i> si comporta in modo analogo a <i>strcmp()</i>, con la differenza che la comparazione si arresta al massimo dopo <b>n</b> caratteri.</p>

Funzione	Descrizione
<pre> size_t strxfrm     (char *restrict <i>dst</i>,      const char *restrict <i>org</i>,      size_t <i>n</i>); </pre>	<p>La funzione <i>strxfrm()</i> trasforma la stringa <i>org</i> sovrascrivendo la stringa <i>dst</i> in modo relativo alla configurazione locale. In pratica, la stringa trasformata che si ottiene può essere comparata con un'altra stringa trasformata nello stesso modo attraverso la funzione <i>strcmp()</i> ottenendo lo stesso esito che si avrebbe confrontando le stringhe originali con la funzione <i>strcoll()</i>.</p>
<pre> void *memchr (const void *<i>s</i>,               int <i>c</i>, size_t <i>n</i>); </pre>	<p>Cerca un carattere a partire da una certa posizione in memoria, scandendo al massimo una quantità determinata di caratteri, restituendo il puntatore al carattere trovato. Se nell'ambito specificato non trova il carattere, restituisce il puntatore nullo.</p>

Funzione	Descrizione
<pre>char *strchr (const char *s,               int c);</pre>	<p>Cerca un carattere all'interno di una stringa, restituendo il puntatore al carattere trovato, oppure il puntatore nullo se la ricerca fallisce. Nella scansione viene preso in considerazione anche il carattere nullo di terminazione della stringa.</p>
<pre>char *strrchr (const char *s,                int c);</pre>	<p>Cerca un carattere all'interno di una stringa, restituendo il puntatore all'ultimo carattere corrispondente trovato, oppure il puntatore nullo se la ricerca fallisce. Nella scansione viene preso in considerazione anche il carattere nullo di terminazione della stringa.</p>
<pre>size_t strspn (const char *s,                 const char *accept);</pre>	<p>Calcola la lunghezza massima iniziale della stringa <i>s</i>, composta esclusivamente da caratteri contenuti nella stringa <i>accept</i>, restituendo tale valore.</p>

Funzione	Descrizione
<pre>size_t strcspn (const char *s,                 const char *reject);</pre>	<p>Calcola la lunghezza massima iniziale della stringa <i>s</i>, composta esclusivamente da caratteri differenti da quelli contenuti nella stringa <i>reject</i>.</p>
<pre>char *strpbrk (const char *s,                const char *accept);</pre>	<p>Scandisce la stringa <i>s</i> alla ricerca del primo carattere che risulti contenuto nella stringa <i>accept</i>, restituendo il puntatore al carattere trovato, oppure, in mancanza di alcuna corrispondenza, il puntatore nullo.</p>
<pre>char *strstr (const char *string,               const char *substring);</pre>	<p>Cerca la stringa <i>substring</i> nella stringa <i>string</i> restituendo il puntatore alla prima corrispondenza trovata (nella stringa <i>string</i>). Se la corrispondenza non c'è, la funzione restituisce il puntatore nullo.</p>

Funzione	Descrizione
<pre>char *strtok     (char *restrict <i>string</i>,      const char *restrict <i>delim</i>);</pre>	<p>Serve a suddividere una stringa in unità, definite <i>token</i>, specificando un elenco di caratteri da intendere come delimitatori, in una seconda stringa. La funzione va usata in fasi successive, fornendo solo inizialmente la stringa da suddividere che continua poi a essere utilizzata se al suo posto viene fornito il puntatore nullo. La funzione restituisce, di volta in volta, il puntatore alla sottostringa contenente l'unità individuata, oppure il puntatore nullo, se non può trovarla.</p>
<pre>void *memset (void *<i>s</i>, int <i>c</i>,               size_t <i>n</i>);</pre>	<p>Inizializza una certa area di memoria, a partire dall'indirizzo <i>s</i>, con la ripetizione del carattere <i>c</i>, tradotto in un carattere senza segno, copiandolo per <i>n</i> volte. La funzione restituisce <i>s</i>.</p>
<pre>char *strerror (int <i>errnum</i>);</pre>	<p>Trasforma un numero nella descrizione del tipo di errore corrispondente.</p>

Funzione	Descrizione
<pre>size_t strlen (const char *s);</pre>	<p>Calcola la lunghezza di una stringa, escludendo dal conteggio il carattere nullo di terminazione.</p>

## File «time.h»

Funzione	Descrizione
<pre>clock_t clock (void);</pre>	<p>Restituisce il tempo di CPU espresso in unità <code>'clock_t'</code>, utilizzato dal processo elaborativo a partire dall'avvio del programma. Se la funzione non è in grado di dare questa indicazione, allora restituisce il valore <code>-1</code>, o più precisamente <code>'(clock_t) (-1)'</code>.</p>
<pre>time_t time (time_t *timer);</pre>	<p>Determina il tempo attuale secondo il calendario del sistema operativo, restituendolo nella forma del tipo <code>'time_t'</code>. Se il puntatore di tipo <code>'time_t *'</code> è valido, la stessa informazione che viene restituita viene anche memorizzata nell'indirizzo indicato da tale puntatore.</p>

Funzione	Descrizione
<pre>double difftime (time_t <i>time1</i>,                  time_t <i>time0</i>);</pre>	Calcola la differenza tra due date, espresse in forma ' <b>time_t</b> ', restituendo l'intervallo in secondi.



Funzione	Descrizione
<pre>time_t mktime (struct tm *<i>timeptr</i>);</pre>	<p>Riceve come argomento il puntatore a una variabile strutturata di tipo '<b>struct tm</b>', contenente le informazioni sull'ora locale, e determina il valore di quella data secondo la rappresentazione interna, di tipo '<b>time_t</b>'. La funzione non tiene conto del giorno della settimana e del giorno dell'anno; inoltre, ammette anche valori al di fuori degli intervalli stabiliti per i vari membri della struttura; infine, considera un valore negativo per il membro '<i>timeptr</i>-&gt;<b>tm_isdst</b>' come la richiesta di determinare se sia o meno in vigore l'ora estiva per la data indicata.</p> <p>Se la funzione non è in grado di restituire un valore rappresentabile nel tipo '<b>time_t</b>', o comunque se non può eseguire il suo compito, restituisce il valore -1, o più precisamente '<b>(time_t) (-1)</b>'. Se invece tutto procede regolarmente, la funzione provvede anche a correggere i valori dei vari membri della struttura e a ricalcolare il giorno della settimana e dell'anno.</p>

Funzione	Descrizione
<pre>struct tm *gmtime     (const time_t *<i>timer</i>);</pre>	<p>Converte una data espressa nella forma del tipo <code>'time_t'</code>, in una data suddivisa nella struttura <code>'tm'</code>, relativa al tempo universale coordinato (UTC).</p>
<pre>struct tm *localtime     (const time_t *<i>timer</i>);</pre>	<p>Converte una data espressa nella forma del tipo <code>'time_t'</code>, in una data suddivisa nella struttura <code>'tm'</code>, relativa all'ora locale.</p>
<pre>char *asctime     (const struct tm *<i>timeptr</i>);</pre>	<p>Converte un'informazione data-orario, espressa nella forma di una struttura <code>'struct tm'</code>, in una stringa che esprime l'ora locale, usando però una rappresentazione fissa in lingua inglese.</p>
<pre>char *ctime (const time_t *<i>timer</i>);</pre>	<p>Converte un'informazione data-orario, espressa nella forma del tipo <code>'time_t'</code>, in una stringa che esprime l'ora locale, usando però una rappresentazione fissa in lingua inglese.</p>

Funzione	Descrizione
<pre>size_t strftime (char *restrict s, size_t <i>maxsize</i>, const char *restrict <i>format</i>, const struct tm *restrict <i>timeptr</i>);</pre>	<p>Interpreta il contenuto di una struttura di tipo <code>'struct tm'</code> e lo traduce in un testo, secondo una stringa di composizione libera. Il comportamento è affine a quello di <code>printf()</code>, dove l'input è costituito dalla struttura contenente le informazioni data-orario.</p>

Specifi- catore di conversione	Corrispondenza
%C	<i>century</i> Il secolo, ottenuto dividendo l'anno per 100 e ignorando i decimali.
%y %Y	<i>year</i> L'anno: nel primo caso si mostrano solo le ultime due cifre, mentre nel secondo si mostrano tutte.
%b   %h %B	Rispettivamente, il nome abbreviato e il nome per esteso del mese.
%m	<i>month</i> Il numero del mese, da 01 a 12.

Specificatore di conversione	Corrispondenza
%d %e	<i>day</i> Il giorno del mese, in forma numerica, da 1 a 31, utilizzando sempre due cifre: nel primo caso si aggiunge eventualmente uno zero; nel secondo si aggiunge eventualmente uno spazio.
%a %A	Rispettivamente, il nome abbreviato e il nome per esteso del giorno della settimana.
%H %L	<i>hour</i> L'ora, espressa rispettivamente a 24 ore e a 12 ore.
%p	La sigla da usare, secondo la configurazione locale, per specificare che si tratta di un'ora antimeridiana o pomeridiana. Nella convenzione inglese si ottengono, per esempio, le sigle «AM» e «PM».
%r	L'ora espressa a 12 ore, completa dell'indicazione se trattasi di ora antimeridiana o pomeridiana, secondo le convenzioni locali.
%R	L'ora e i minuti, equivalente a '%H:%M'.
%M	<i>minute</i> I minuti, da 00 a 59.
%S	<i>second</i> I secondi, espresso con valori da 00 a 60.
%T	<i>time</i> L'ora, i minuti e i secondi, equivalente a '%H:%M:%S'.

Specificatore di conversione	Corrispondenza
%z %Z	<i>time zone</i> La rappresentazione del fuso orario, nel primo caso come distanza dal tempo coordinato universale (UTC), mentre nel secondo si usa una rappresentazione conforme alla configurazione locale.
%j	<i>julian</i> Il giorno dell'anno, usando sempre tre cifre numeriche: da 001 a 366.
%g %G	L'anno a cui appartiene la settimana secondo lo standard ISO 8601: nel primo caso si mostrano solo le ultime due cifre, mentre nel secondo si ha l'anno per esteso. Secondo lo standard ISO 8601 la settimana inizia con lunedì e la prima settimana dell'anno è quella che include il 4 gennaio.
%V	Il numero della settimana secondo lo standard ISO 8601. I valori vanno da 01 a 53. Secondo lo standard ISO 8601 la settimana inizia con lunedì e la prima settimana dell'anno è quella che include il 4 gennaio.
%u %W	Il giorno della settimana, espresso in forma numerica, dove, rispettivamente, si conta da 1 a 7, oppure da 0 a 6. Zero e sette rappresentano la domenica; uno è il lunedì.
%U %W	Il numero della settimana, contando, rispettivamente, dalla prima domenica o dal primo lunedì di gennaio. Si ottengono cifre da 00 a 53.
%x	La data, rappresentata secondo le convenzioni locali.
%X	L'ora, rappresentata secondo le convenzioni locali.

Specificatore di conversione	Corrispondenza
%c	La data e l'ora, rappresentate secondo le convenzioni locali.
%D	<i>date</i> La data, rappresentata come '%m/%d/%Y'.
%F	La data, rappresentata come '%Y-%m-%d'.
%n	Viene rimpiazzato dal codice di interruzione di riga.
%t	Viene rimpiazzato da una tabulazione orizzontale.
%%	Viene rimpiazzato da un carattere di percentuale.

## File «stdio.h» per la gestione dei file e degli errori

<<

Macro-variabile	Significato mnemonico	Descrizione
__IOFBF	<i>input output fully buffered</i>	Indica simbolicamente la richiesta di utilizzo di una memoria tampone a blocchi.
__IOLBF	<i>input output line buffered</i>	Indica simbolicamente la richiesta di utilizzo di una memoria tampone gestita a righe di testo.
__IONBF	<i>input output with no buffering</i>	Indica simbolicamente la richiesta di non utilizzare alcuna memoria tampone.

Macro-variabile	Significato mnemonico	Descrizione
BUFSIZE	<i>buffer size</i>	Rappresenta la dimensione predefinita della memoria tampone.
EOF	<i>end of file</i>	È un numero intero di tipo <b>'int'</b> , negativo, che rappresenta il raggiungimento della fine del file. È in pratica ciò che si ottiene leggendo oltre la fine del file.
FOPEN_MAX	<i>file open max</i>	Il numero di file che un processo elaborativo può aprire simultaneamente, in base alle limitazioni poste dal sistema operativo.
FILENAME_MAX		La dimensione di un array di elementi <b>'char'</b> , tale da essere abbastanza grande da contenere il nome del file più lungo (incluse le eventuali sequenze multibyte) che il sistema consenta di gestire.
L_tmpnam	<i>temporary name</i>	La dimensione di un array di elementi <b>'char'</b> , tale da essere abbastanza grande da contenere il nome di un file temporaneo generato dalla funzione <i>tmpnam()</i> .
SEEK_CUR	<i>seek current</i>	Indica di eseguire un posizionamento a partire dalla posizione corrente del file.

Macro-variabile	Significato mnemonico	Descrizione
SEEK_END		Indica di eseguire un posizionamento a partire dalla fine di un file.
SEEK_SET		Indica di eseguire un posizionamento a partire dall'inizio di un file.
TMP_MAX		Rappresenta la quantità massima di nomi di file differenti che possono essere generati dalla funzione <i>tmpnam()</i> .

Modalità di accesso ai file	Mnemonico	Descrizione
r	<i>read</i>	Accesso in sola lettura di un file di testo.
w	<i>write</i>	Accesso a un file di testo in scrittura, che implica la creazione del file all'apertura, ovvero il suo troncamento a zero, se esiste già.
a	<i>append</i>	Accesso a un file di testo in aggiunta, che implica la creazione del file all'apertura, ovvero la sua estensione se esiste già.
rb wb ab	<i>binary</i>	Accesso in lettura, scrittura o aggiunta, ma di tipo binario.



Modalità di accesso ai file	Mnemonico	Descrizione
r+ w+ a+	<i>update</i>	Accesso a un file di testo in lettura, scrittura o aggiunta, assieme alla modalità di aggiornamento. In pratica, con la lettura è consentita anche la scrittura; con la scrittura e l'aggiunta è consentita anche la rilettera.
rb+   r+b wb+   w+b ab+   a+b		Accesso a un file binario in lettura, scrittura o aggiunta, assieme alla modalità di aggiornamento. In pratica, con la lettura è consentita anche la scrittura; con la scrittura e l'aggiunta è consentita anche la rilettera. Si può osservare che il segno '+' può essere messo indifferentemente in mezzo o alla fine.

Funzione	Descrizione
<code>int remove (const char *<i>filename</i>);</code>	Cancella il file il cui nome viene fornito come argomento. Il nome del file va espresso secondo le convenzioni del sistema operativo. Restituisce zero se l'operazione ha successo, altrimenti un valore differente.

Funzione	Descrizione
<pre>int rename (const char *<i>old</i>,            const char *<i>new</i>);</pre>	<p>Cambia il nome del file indicato come primo argomento, in modo che assuma quello del secondo argomento. Se l'operazione avviene con successo restituisce zero, altrimenti produce un valore differente.</p>
<pre>FILE *tmpfile (void);</pre>	<p>Crea e apre un file temporaneo binario in aggiornamento ('wb+'), restituendone il puntatore. Se il file temporaneo non può essere creato, la funzione restituisce il puntatore nullo.</p>
<pre>char *tmpnam (char *<i>s</i>);</pre>	<p>Genera il nome di un file che può essere usato come file temporaneo. La funzione richiede come argomento un array di almeno 'L_tmpnam' caratteri, da usare per scriverci il nome e per restituirne il puntatore. Se alla funzione viene passato il puntatore nullo, allora questa usa un'area di memoria statica che viene sovrascritta a ogni chiamata successiva della funzione stessa. Se la funzione non può eseguire il suo lavoro, restituisce il puntatore nullo.</p>

Funzione	Descrizione
<pre>FILE *fopen     (const char *restrict <i>filename</i>,      const char *restrict <i>io_mode</i>);</pre>	<p>Apre il file indicato come primo argomento, secondo la modalità espressa dalla stringa che costituisce il secondo argomento, restituendo il puntatore che ne rappresenta il flusso aperto. Se l'operazione fallisce la funzione restituisce il puntatore nullo e aggiorna il valore della variabile globale <i>errno</i>.</p>
<pre>FILE *freopen     (const char *restrict <i>filename</i>,      const char *restrict <i>io_mode</i>,      FILE *restrict <i>stream</i>);</pre>	<p>Apre il file indicato come primo argomento, secondo la modalità espressa dalla stringa che costituisce il secondo argomento, utilizzando il flusso di file individuato dal puntatore che costituisce l'ultimo argomento, restituendo lo stesso puntatore. Se il puntatore indicato come ultimo argomento riguarda un flusso di file ancora aperto, questo viene chiuso e quindi riaperto. Se l'operazione fallisce la funzione restituisce il puntatore nullo e aggiorna il valore della variabile globale <i>errno</i>. Lo scopo di questa funzione è quello di ridirigere i flussi di file, associando file differenti.</p>

Funzione	Descrizione
<pre>int fclose (FILE *<i>stream</i>) ;</pre>	<p>Chiude il flusso di file individuato dal puntatore che costituisce l'argomento della funzione. La funzione restituisce il valore zero se l'operazione ha successo, altrimenti produce il valore corrispondente alla macrovariabile <b><i>EOF</i></b>. Va sottolineato che un flusso già chiuso non deve essere chiuso nuovamente, perché in tal caso l'effetto che se ne produce è imprecisato.</p>

Funzione	Descrizione
<pre data-bbox="108 793 874 1022">int setvbuf (FILE *restrict <i>stream</i>,              char *restrict <i>buffer</i>,              int <i>buf_mode</i>,              size_t <i>size</i>);</pre>	<p data-bbox="970 149 1485 840">Attribuisce una memoria tampone a un file che è appena stato aperto e per il quale non è ancora stato eseguito alcun accesso. Il primo argomento della funzione è il puntatore al flusso relativo e il secondo è il puntatore all'inizio dell'array di caratteri da usare come memoria tampone. Se al posto del riferimento alla memoria tampone si indica un puntatore nullo, si intende che la funzione debba allocare automaticamente lo spazio necessario; se invece l'array viene fornito, è evidente che deve rimanere disponibile per tutto il tempo in cui il flusso rimane aperto.</p> <p data-bbox="970 854 1485 1453">Il terzo argomento atteso dalla funzione è un numero che esprime la modalità di funzionamento della memoria tampone. Questo numero viene fornito attraverso l'indicazione di una tra le macro-variabili <i>_IOFBF</i>, <i>_IOLBF</i> e <i>_IONBF</i>. Il quarto argomento indica la dimensione dell'array da usare come memoria tampone: se l'array viene fornito effettivamente, si tratta della dimensione che può essere utilizzata; altrimenti è la dimensione richiesta per l'allocazione automatica.</p> <p data-bbox="970 1467 1485 1637">La funzione restituisce zero se l'operazione richiesta è eseguita con successo; diversamente restituisce un valore differente.</p>

Funzione	Descrizione
<pre>void setbuf (FILE *restrict <i>stream</i> ,              char *restrict <i>buffer</i>) ;</pre>	<p>Si tratta di una versione semplificata di ‘<b>setvbuf()</b>’ che non restituisce alcun valore, che prevede implicitamente una modalità di gestione completa della memoria tampone (‘<b>_IOFBF</b>’), che richiede implicitamente un array di ‘<b>BUFSIZ</b>’ elementi. Anche in questo caso, se l’argomento corrispondente al parametro <i>buffer</i> è un puntatore nullo, l’allocazione avviene in modo automatico.</p>
<pre>int fflush (FILE *<i>stream</i>) ;</pre>	<p>Scarica la memoria tampone del flusso di file indicato, procedendo così alla memorizzazione dei dati rimasti in sospeso. Restituisce zero se l’operazione viene completata con successo, altrimenti restituisce il valore corrispondente alla macro-variabile <b>EOF</b>.</p>
<pre>int fgetc (FILE *<i>stream</i>) ; int getc (FILE *<i>stream</i>) ;</pre>	<p>Legge un carattere (senza segno) dal flusso di file indicato come argomento e ne restituisce il valore numerico (positivo). Se l’operazione fallisce, la funzione restituisce il valore corrispondente alla macro-variabile <b>EOF</b>. Tradizionalmente, <i>fgetc()</i> è sempre una funzione, mentre <i>getc()</i> potrebbe essere una macroistruzione che valuta anche più volte l’espressione che costituisce l’argomento.</p>

Funzione	Descrizione
<pre>int ungetc (int <i>c</i>, FILE *<i>stream</i>);</pre>	<p>Rimanda indietro il carattere <i>c</i> nel flusso di file <i>stream</i>; in altri termini dovrebbe annullare l'effetto dell'ultima chiamata a una funzione <i>fgetc()</i> o <i>getc()</i>.</p>
<pre>int fputc (int <i>c</i>, FILE *<i>stream</i>);</pre> <pre>int putc (int <i>c</i>, FILE *<i>stream</i>);</pre>	<p>Scrive un carattere, rappresentato dal primo argomento, nel flusso di file indicato come secondo argomento, restituendo lo stesso valore del carattere scritto, se l'operazione si conclude con successo, oppure il valore corrispondente a 'EOF' se l'operazione fallisce. Tradizionalmente, <i>fputc()</i> è sempre una funzione, mentre <i>putc()</i> potrebbe essere una macroistruzione che valuta anche più volte le espressioni che costituiscono gli argomenti.</p>
<pre>int putchar (int <i>c</i>);</pre>	<p>Scrive un carattere, rappresentato dall'argomento, nello standard output, restituendo lo stesso valore del carattere scritto, se l'operazione si conclude con successo, oppure il valore corrispondente a 'EOF' se l'operazione fallisce. Tradizionalmente si tratta di una macroistruzione che valuta anche più volte l'espressione che costituisce l'argomento.</p>

Funzione	Descrizione
<pre data-bbox="108 609 895 778">char *fgets (char *restrict <i>s</i>,              int <i>n</i>,              FILE *restrict <i>stream</i>);</pre>	<p data-bbox="970 145 1485 744">Legge al massimo <math>n-1</math> caratteri (elementi 'char') attraverso il flusso di file <i>stream</i>, copiandoli in memoria a partire dall'indirizzo <i>s</i> e aggiungendo alla fine il carattere nullo di terminazione delle stringhe. La lettura si esaurisce prima di <math>n-1</math> caratteri se viene incontrato il codice di interruzione di riga, il quale viene rappresentato nella stringa a cui punta <i>s</i>, ovvero se si raggiunge la fine del file. In ogni caso, la stringa <i>s</i> viene terminata correttamente con il carattere nullo.</p> <p data-bbox="970 758 1485 1212">La funzione restituisce la stringa <i>s</i> se la lettura avviene con successo, ovvero se ha prodotto almeno un carattere; altrimenti, il contenuto dell'array a cui punta <i>s</i> non viene modificato e la funzione restituisce il puntatore nullo. Se si creano errori imprevisti, la funzione potrebbe restituire il puntatore nullo, ma senza garantire che l'array <i>s</i> sia rimasto intatto.</p>



Funzione	Descrizione
<pre>int fputs (const char *restrict <i>s</i>,           FILE *restrict <i>stream</i>);</pre>	<p>Copia la stringa a cui punta <i>s</i> nel file rappresentato dal flusso di file <i>stream</i>. La copia della stringa avviene escludendo però il carattere nullo di terminazione. Va osservato che questa funzione, pur essendo contrapposta evidentemente a '<b>fgets()</b>', <b>non conclude la riga</b> del file, ovvero, non aggiunge il codice di interruzione di riga. Per ottenere la conclusione della riga di un file di testo, occorre inserire nella stringa, espressamente, il carattere '<b>\n</b>'.</p> <p>La funzione restituisce il valore rappresentato da '<b>EOF</b>' se l'operazione di scrittura produce un errore; altrimenti restituisce un valore positivo qualunque.</p>
<pre>char *gets (char *<i>s</i>);</pre>	<p>Legge una riga dallo standard input, copiandola in memoria a partire dall'indirizzo <i>s</i> e aggiungendo alla fine il carattere nullo di terminazione delle stringhe. Per il resto, il funzionamento è conforme a quello di <i>fgets()</i>.</p>
<pre>int puts (const char *<i>s</i>);</pre>	<p>Copia la stringa a cui punta <i>s</i> nello standard output, aggiungendo in coda il codice di interruzione di riga. Per il resto, il funzionamento è analogo a quello di <i>fputs()</i>.</p>

Funzione	Descrizione
<pre>size_t fread (void *restrict <i>ptr</i>,               size_t <i>size</i>,               size_t <i>nmemb</i>,               FILE *restrict <i>stream</i>);</pre>	<p>Legge dal flusso di file <i>stream</i>, <i>nmemb</i> blocchi da <i>size</i> byte, copiando questi dati in memoria a partire dall'indirizzo <i>ptr</i>. Restituisce la quantità di blocchi da '<i>size</i>' byte che sono stati copiati con successo; pertanto, se questo valore è inferiore a <i>nmemb</i>, si è verificato un problema.</p>
<pre>size_t fwrite (const void *restrict <i>ptr</i>,  size_t <i>size</i>,  size_t <i>nmemb</i>,  FILE *restrict <i>stream</i>);</pre>	<p>Scrive nel flusso di file <i>stream</i>, <i>nmemb</i> blocchi da <i>size</i> byte, leggendo questi dati dalla memoria a partire dall'indirizzo <i>ptr</i>. Restituisce la quantità di blocchi da '<i>size</i>' byte che sono stati copiati con successo; pertanto, se questo valore è inferiore a <i>nmemb</i>, si è verificato un problema.</p>

Funzione	Descrizione
<pre data-bbox="108 541 670 711">int fseek (FILE *<i>stream</i> ,            long int <i>offset</i> ,            int <i>whence</i> );</pre>	<p data-bbox="970 147 1485 1071">Sposta la posizione corrente relativa al flusso di file <i>stream</i> (associato preferibilmente a un file binario), nella nuova posizione determinata dai parametri <i>whence</i> e <i>offset</i>. Il parametro <i>whence</i> viene fornito attraverso una macro-variabile che può essere <b>SEEK_SET</b>, <b>SEEK_CUR</b> o <b>SEEK_END</b>, indicando rispettivamente l'inizio del file, la posizione corrente o la fine del file. Dalla posizione indicata dal parametro <i>whence</i> viene aggiunta, algebricamente, la quantità di byte indicata dal parametro <i>offset</i>. La funzione restituisce zero se può eseguire l'operazione, altrimenti dà un risultato diverso.</p>
<pre data-bbox="108 1359 758 1394">long int ftell (FILE *<i>stream</i> );</pre>	<p data-bbox="970 1087 1485 1667">Restituisce la posizione corrente del flusso di file indicato come argomento. Questo valore può essere usato con <i>fseek()</i>, al posto dello scostamento (il parametro <i>offset</i>), indicando come posizione di riferimento l'inizio del file, ovvero '<b>SEEK_SET</b>'. Se la funzione non riesce a fornire la posizione, restituisce il valore -1 (tradotto in '<b>long int</b>') e annota il fatto nella variabile <i>errno</i>.</p>

Funzione	Descrizione
<pre>void rewind (FILE *<i>stream</i>);</pre>	<p>Riposiziona il flusso di file all'inizio. In pratica è come utilizzare la funzione <i>fseek()</i> specificando uno scostamento pari a zero a partire da 'SEEK_SET', ignorando il valore restituito.</p>
<pre>int fgetpos (FILE *restrict <i>stream</i>,              fpos_t *restrict <i>pos</i>);</pre>	<p>Memorizza nella variabile a cui punta il parametro <i>pos</i> le informazioni sulla posizione corrente del file, assieme allo stato di interpretazione relativo alle sequenze multibyte. Restituisce zero se l'operazione è stata compiuta con successo, altrimenti dà un altro valore</p>
<pre>int fsetpos (FILE *<i>stream</i>,              const fpos_t *<i>pos</i>);</pre>	<p>Utilizza la variabile a cui punta <i>pos</i> per ripristinare la posizione memorizzata, assieme allo stato di avanzamento dell'interpretazione di una sequenza multibyte. Restituisce zero se l'operazione è stata compiuta con successo, altrimenti dà un altro valore e aggiorna la variabile <i>errno</i>.</p>
<pre>void clearerr (FILE *<i>stream</i>);</pre>	<p>Azzera gli indicatori di errore e di fine file per il flusso di file indicato come argomento, senza restituire alcunché.</p>
<pre>int feof (FILE *<i>stream</i>);</pre>	<p>Controlla lo stato dell'indicatore di fine file per il flusso di file indicato. Se questo non è attivo restituisce zero, altrimenti restituisce un valore diverso da zero.</p>

Funzione	Descrizione
<pre>int ferror (FILE *<i>stream</i>) ;</pre>	<p>Controlla lo stato dell'indicatore di errore per il flusso di file indicato. Se questo non è attivo restituisce zero, altrimenti restituisce un valore diverso da zero.</p>
<pre>void perror (const char *<i>s</i>) ;</pre>	<p>Prende in considerazione la variabile <i>errno</i> e cerca di tradurla in un messaggio testuale da emettere attraverso lo standard error (con tanto di terminazione della riga, in modo da riposizionare a capo il cursore). Se il parametro <i>s</i> corrisponde a una stringa non vuota, il testo di questa viene posto anteriormente al messaggio, separandolo con due punti e uno spazio (': '). Il contenuto del messaggio è lo stesso che si otterrebbe con la funzione <i>strerror()</i>, fornendo come argomento la variabile <i>errno</i>.</p>

## File «stdio.h» per la composizione dell'output



`% [ simbolo ] [ n_ampiezza ] [ . n_precision ] [ hh | h | l | ll | j | z | t | L ] tipo`

Simbolo	Tipo di argomento	Conversione applicata
<code>%...d</code>	<code>int</code>	Numero intero con segno da rappresentare in base dieci.
<code>%...i</code>		

Simbolo	Tipo di argomento	Conversione applicata
%...u	unsigned int	Numero intero senza segno da rappresentare in base dieci.
%...o	unsigned int	Numero intero senza segno da rappresentare in ottale (senza lo zero iniziale che viene usato spesso per caratterizzare un tale tipo di rappresentazione).
%...x %...X	unsigned int	Numero intero senza segno da rappresentare in esadecimale (senza il prefisso '0x' o '0X' che viene usato spesso per caratterizzare un tale tipo di rappresentazione).
%...c	int	Un carattere singolo, dopo la conversione in ' <b>unsigned char</b> '.
%...s	char *	Una stringa.
%...f	double	Un numero a virgola mobile, da rappresentare in notazione decimale fissa: <b>[ - ] iii . dddddd</b>
%...e %...E	double	Un numero a virgola mobile, da rappresentare in notazione esponenziale: <b>[ - ] i . dddddd e ± xx</b> <b>[ - ] i . dddddd E ± xx</b>

Simbolo	Tipo di argomento	Conversione applicata
%...g %...G	double	Un numero a virgola mobile, rappresentato in notazione decimale fissa o in notazione esponenziale, a seconda di quale si presti meglio in base ai vincoli posti da altri componenti dello specificatore di conversione.
%p	void *	Un puntatore generico rappresentato in qualche modo in forma grafica.
%n	int *	Questo specificatore non esegue alcuna conversione e si limita a memorizzare un valore intero (di tipo <code>int</code> ) nella variabile a cui punta l'argomento. Per la precisione, viene memorizzata la quantità di caratteri generati fino a quel punto dalla conversione.
%%		Questo specificatore si limita a produrre un carattere di percentuale ('%') che altrimenti non sarebbe rappresentabile.

Simbolo	Corrispondenza
%+... %#+... %+0 <i>ampiezza</i> ... %#+0 <i>ampiezza</i> ...	Il segno «+» fa sì che i numeri con segno lo mostrino anche se è positivo. Può combinarsi con lo zero e il cancelletto.

Simbolo	Corrispondenza
<p><math>\%0</math> <i>ampiezza...</i></p> <p><math>\%+0</math> <i>ampiezza...</i></p> <p><math>\\#0</math> <i>ampiezza...</i></p> <p><math>\\#+0</math> <i>ampiezza...</i></p>	<p>Lo zero fa sì che siano inseriti degli zeri a sinistra per allineare a destra il valore, nell'ambito dell'ampiezza specificata. Può combinarsi con il segno «+» e il cancelletto.</p>
<p><math>\%</math> <i>ampiezza...</i></p> <p><math>\%</math> <i> </i> <i>ampiezza...</i></p>	<p>In mancanza di uno zero iniziale, in presenza dell'indicazione dell'ampiezza, il valore viene allineato a destra usando degli spazi. È possibile esprimere esplicitamente l'intenzione di usare gli spazi mettendo proprio uno spazio, ma in generale non è richiesto. Se si mette lo spazio letteralmente, questo non è poi compatibile con lo zero, mentre le combinazioni con gli altri simboli sono ammissibili.</p>
<p><math>\%-</math> <i>ampiezza...</i></p> <p><math>\%+ -</math> <i>ampiezza...</i></p> <p><math>\\#-</math> <i>ampiezza...</i></p> <p><math>\\#+ -</math> <i>ampiezza...</i></p>	<p>Il segno meno, usato quando la conversione prevede l'uso di una quantità fissa di caratteri con un valore che appare di norma allineato a destra, fa sì che il risultato sia allineato a sinistra. Il segno meno si può combinare il segno «+» e il cancelletto.</p>



Simbolo	Corrispondenza
%#...	Il cancelletto richiede una modalità di rappresentazione alternativa, ammesso che questa sia prevista per il tipo di conversione specificato. È compatibile con gli altri simboli, ammesso che il suo utilizzo serva effettivamente per ottenere una rappresentazione alternativa.

Simbolo	Tipo	Simbolo	Tipo
%...hhd %...hhi	signed char	%...hhu %...hho %...hhx   %...hhX	unsigned char
%...hd %...hi	short int	%...hu %...ho %...hx   %...hX	unsigned short int
%...ld %...li	long int	%...lu %...lo %...lx   %...lX	unsigned long int
%...lc	wint_t	%...ls	wchar_t *

Simbolo	Tipo	Simbolo	Tipo
%...lld %...lli	long long int	%...llu %...llo %...llx   %...llX	unsigned long long int
%...jd %...ji	intmax_t	%...ju %...jo %...jx   %...jX	uintmax_t
%...zd %...zi	size_t	%...zu %...zo %...zx   %...zX	size_t
%...td %...ti	ptrdiff_t	%...tu %...to %...tx   %...tX	ptrdiff_t
%...Le   %...LE %...Lf   %...LF %...Lg   %...LG	long double		

Funzione	Descrizione
<pre>int sprintf (char *restrict s, const char *restrict <i>format</i>, ...);</pre>	<p>Produce il risultato della composizione memorizzando a partire dal puntatore costituito dal primo parametro (<i>s</i>) e aggiungendo il carattere nullo di terminazione.</p>
<pre>int snprintf (char *restrict s, size_t <i>n</i>, const char *restrict <i>format</i>, ...);</pre>	<p>Produce al massimo <i>n</i>−1 caratteri, aggiungendo sempre il carattere nullo di terminazione.</p>
<pre>int fprintf (FILE *restrict <i>stream</i>, const char *restrict <i>format</i>, ...);</pre>	<p>Scrive il risultato della composizione attraverso il flusso di file <i>stream</i>.</p>
<pre>int printf (const char *restrict <i>format</i>, ...);</pre>	<p>Scrive il risultato della composizione attraverso lo standard output.</p>
<pre>int vsprintf (char *restrict s, const char *restrict <i>format</i>, va_list <i>arg</i>);</pre>	<p>Come la funzione <i>sprintf()</i>, ricevendo gli argomenti variabili attraverso un puntatore al loro inizio.</p>

Funzione	Descrizione
<pre>int vsnprintf (char *restrict s, size_t n, const char *restrict format, va_list arg);</pre>	Come la funzione <i>snprintf()</i> , ricevendo gli argomenti variabili attraverso un puntatore al loro inizio.
<pre>int vfprintf (FILE *restrict stream, const char *restrict format, va_list arg);</pre>	Come la funzione <i>fprintf()</i> , ricevendo gli argomenti variabili attraverso un puntatore al loro inizio.
<pre>int vprintf (const char *restrict format, va_list arg);</pre>	Come la funzione <i>printf()</i> , ricevendo gli argomenti variabili attraverso un puntatore al loro inizio.

## File «stdio.h» per l'interpretazione dell'input

«

```
% [*] [n_ampiezza] [hh|h|l|ll|j|z|t|L] tipo
```

Simbolo	Tipo di argomento	Conversione applicata
%...d	int *	Numero intero con segno rappresentato in base dieci.
%...i	int *	Numero intero con segno rappresentare in base dieci o in base otto, avendo come prefisso uno zero, oppure in base sedici, avendo come prefisso '0x' o '0X'.
%...u	unsigned int *	Numero intero senza segno rappresentato in base dieci.

Simbolo	Tipo di argomento	Conversione applicata
%...o	unsigned int *	Numero intero senza segno rappresentato in ottale (con o senza lo zero iniziale).
%...x	unsigned int *	Numero intero senza segno rappresentato in esadecimale (con o senza il prefisso '0x' o '0X').
%...c	char *	Interpreta un solo carattere, o più caratteri se si specifica l'ampiezza. Nella lettura contano anche gli spazi o qualunque altro carattere e non viene aggiunto il carattere nullo di terminazione.
%...s	char *	Interpreta una sequenza di caratteri che non siano spazi, aggiungendo alla fine il carattere nullo di terminazione.
%...a %...e %...f %...g	double *	Un numero a virgola mobile rappresentato in notazione decimale fissa o in notazione esponenziale: $\begin{bmatrix} - \end{bmatrix} iii . dddddd$ $\begin{bmatrix} - \end{bmatrix} i . dddddd e \pm xx$ $\begin{bmatrix} - \end{bmatrix} i . dddddd E \pm xx$
%p	void *	Interpreta il valore di un puntatore che sia rappresentato nello stesso modo in cui farebbe la funzione <code>'printf("%p", puntatore)'</code> .

Simbolo	Tipo di argomento	Conversione applicata
<code>%n</code>	<code>int *</code>	Questo specificatore non esegue alcuna conversione e si limita a memorizzare la quantità di caratteri ( <b>'char'</b> ) letti fino a quel punto.
<code>%... [...]</code>	<code>char *</code>	Interpreta una stringa non vuota contenente solo i caratteri elencati tra parentesi quadre, aggiungendo alla fine il carattere nullo di terminazione. Se tra i caratteri si cerca anche la parentesi quadra chiusa, questa va messa all'inizio dell'elenco: <code>'%... [...]'</code> .
<code>%... [^...]</code>	<code>char *</code>	Interpreta una stringa non vuota contenente solo caratteri diversi da quelli elencati tra parentesi quadre, aggiungendo alla fine il carattere nullo di terminazione. Se tra i caratteri da escludere si vuole indicare anche la parentesi quadra chiusa, questa va messa all'inizio dell'elenco: <code>'%... [^] ...'</code> .
<code>%%</code>		Interpreta un carattere di percentuale tra i dati in ingresso, ma senza memorizzare alcunché.

Simbolo	Tipo	Simbolo	Tipo
%...hhd %...hhi	signed char *	%...hhu %...hho %...hhx %...hhn	unsigned char *
%...hd %...hi	short int *	%...hu %...ho %...hx %...hn	unsigned short int *
%...ld %...li	long int *	%...lu %...lo %...lx %...ln	unsigned long int *
		%...lc %...ls %...lc %...l [...]	wchar_t *

Simbolo	Tipo	Simbolo	Tipo
%...lld %...lli	long long int *	%...llu %...llo %...llx %...lln	unsigned long long int *
%...jd %...ji	intmax_t *	%...ju %...jo %...jx %...jn	uintmax_t *
%...zd %...zi	size_t *	%...zu %...zo %...zx %...zn	size_t *
%...td %...ti	ptrdiff_t *	%...tu %...to %...tx %...tn	ptrdiff_t *



Simbolo	Tipo	Simbolo	Tipo
%...Le			
%...Lf	long double *		
%...Lg			

Funzione	Descrizione
<pre>int fscanf (FILE *restrict <i>stream</i>, const char *restrict <i>format</i>, ...);</pre>	<p>Scandisce l'input proveniente dal flusso_di_file che costituisce il primo parametro (<i>stream</i>), restituendo la quantità di valori assegnati alle variabili rispettive, oppure il valore corrispondente alla macro-variabile <b>EOF</b> nel caso si verifichi un errore prima di qualunque conversione.</p>
<pre>int sscanf (const char *restrict <i>s</i>, const char *restrict <i>format</i>, ...);</pre>	<p>Scandisce il contenuto della stringa indicata come primo parametro (<i>s</i>), restituendo la quantità di valori assegnati alle variabili rispettive, oppure il valore corrispondente alla macro-variabile <b>EOF</b> nel caso si verifichi un errore prima di qualunque conversione.</p>

Funzione	Descrizione
<pre>int scanf     (const char *restrict <i>format</i>,      ...);</pre>	<p>Scandisce lo standard input, restituendo la quantità di valori assegnati alle variabili rispettive, oppure il valore corrispondente alla macro-variabile <b>EOF</b> nel caso si verifichi un errore prima di qualunque conversione.</p>
<pre>int vfscanf     (FILE *restrict <i>stream</i>,      const char *restrict <i>format</i>,      va_list <i>arg</i>);</pre>	<p>Come <i> fscanf()</i>, con la differenza che gli argomenti variabili sono sostituiti da un puntatore al loro inizio.</p>
<pre>int vsscanf     (const char *restrict <i>s</i>,      const char *restrict <i>format</i>,      va_list <i>arg</i>);</pre>	<p>Come <i> sscanf()</i>, con la differenza che gli argomenti variabili sono sostituiti da un puntatore al loro inizio.</p>
<pre>int vscanf     (const char *restrict <i>format</i>,      va_list <i>arg</i>);</pre>	<p>Come <i> scanf()</i>, con la differenza che gli argomenti variabili sono sostituiti da un puntatore al loro inizio.</p>

## File «assert.h»

Macroistruzione	Descrizione
<pre>void assert (<i>tipo_scalare espressione</i>);</pre>	Nell'uso di questa macroistruzione, in pratica si scrive solo l'espressione tra parentesi, senza indicare espressamente il tipo scalare. Se l'espressione si traduce in un valore pari a zero, l'asserzione fallisce e viene mostrato un messaggio di errore, con le informazioni necessarie per risalire alla posizione nel file sorgente.

## File «stddef.h»

Macroistruzione	Descrizione
<pre>size_t offsetof (<i>type, member</i>);</pre>	Restituisce lo scostamento che separa un membro di una struttura dall'inizio della stessa.

## File «locale.h»

<<

Funzione	Descrizione
<pre>char *setlocale (int <i>category</i>, NULL);</pre>	<p>Restituisce un puntatore alla stringa che descrive la configurazione locale corrente, riferita alla categoria specificata. Se non è in grado di fornire l'informazione, fornisce un puntatore nullo.</p>
<pre>char *setlocale (int <i>category</i>,                  const char *<i>locale</i>);</pre>	<p>Imposta la configurazione locale, secondo il contenuto della stringa che costituisce il secondo parametro. La funzione restituisce un puntatore che descrive la stessa configurazione, oppure, se l'operazione fallisce, restituisce il puntatore nullo.</p>

Macro-variabile	Descrizione
LC_ALL	Individua simultaneamente tutte le categorie relative alla configurazione locale.
LC_COLLATE	Categoria che definisce l'ordine alfabetico dei caratteri tipografici.
LC_CTYPE	Categoria che definisce il modo di raggruppare i caratteri tipografici per tipologia.
LC_MONETARY	Categoria che definisce le convenzioni legate alla rappresentazione dei valori che esprimono importi in valuta.

Macro-variabile	Descrizione
LC_NUMERIC	Categoria che definisce il modo in cui vanno rappresentati i valori numerici, soprattutto per quanto riguarda la separazione tra parte intera e parte decimale.
LC_TIME	Categoria che definisce il modo corretto di esprimere le informazioni data-orario.
LC_MESSAGES	POSIX: Categoria che definisce il formato dei messaggi emessi per informazioni generali e di quelli diagnostici.

## File «regex.h»

Funzione	Descrizione
<pre>int regcomp     (regex_t *restrict <i>re</i>,      const char *restrict <i>regex</i>,      int <i>cflags</i>);</pre>	<p>Compila l'espressione regolare descritta dalla stringa <i>regex</i>, componendo il contenuto della variabile strutturata <i>re</i>, tenendo conto delle opzioni <i>cflags</i>.</p>
<pre>void regfree     (regex_t *<i>re</i>);</pre>	<p>Libera la memoria associata all'espressione regolare compilata nella variabile <i>re</i>.</p>



Funzione	Descrizione
<pre>int regexec     (const regex_t *restrict <i>re</i>,      const char *restrict <i>s</i>,      size_t <i>n</i>,      regmatch_t <i>m</i>[restrict],      int <i>eflags</i>);</pre>	<p>Compara l'espressione regolare <i>re</i> con la stringa <i>s</i>, tenendo conto delle opzioni <i>eflags</i>, immettendo le sottostringhe estratte nell'array di stringhe <i>m</i>, sapendo che questo può contenere al massimo <i>n</i> stringhe.</p>
<pre>size_t regerror     (int <i>e</i>,      const regex_t *restrict <i>re</i>,      char *restrict <i>t</i>,      size_t <i>n</i>);</pre>	<p>Sulla base del numero di errore <i>e</i> e dell'espressione regolare <i>re</i>, produce un messaggio di errore nella stringa <i>t</i> che non può essere più lunga di <i>n</i> caratteri.</p>

Tipo	Nome	Membri noti del tipo ' <b>regex_t</b> '
size_t	re_sub	Quantità di sottoespressioni tra parentesi tonde.

Tipo	Nome	Membri noti di una variabile di tipo ' <b>regmatch_t</b> '
regoff_t	rm_so	Scostamento in byte, dall'inizio della stringa, corrispondente all'inizio della sottostringa individuata.
regoff_t	rm_eo	Scostamento in byte, dall'inizio della stringa, corrispondente al carattere <b>successivo</b> alla sottostringa individuata.

Macro-variabile	Da usare come opzioni della funzione <i>regcomp()</i> per la compilazione di un'espressione regolare.
REG_EXTENDED	L'espressione regolare fornita è di tipo ERE (estesa). Se non si usa questa opzione, si intende che l'espressione sia di tipo BRE.
REG_ICASE	L'espressione regolare fornita va valutata senza distinguere tra lettere maiuscole o minuscole.
REG_NOSUB	Per la compilazione dell'espressione regolare non si intende tenere conto della corrispondenza eventuale di sottostringhe; in altri termini, non si vogliono considerare le parentesi '\(' e '\)', oppure '(' e ')' (a seconda che si tratti di ERE o BRE). In tal caso, l'espressione regolare serve per il confronto, ma non per estrapolare porzioni del risultato ottenuto.
REG_NEWLINE	In condizioni normali, il codice <i>new-line</i> contenuto nella stringa con cui l'espressione regolare deve essere confrontata, viene trattato come gli altri caratteri. Con l'opzione ' <b>REG_NEWLINE</b> ', invece, l'operatore '^' individua l'inizio di un testo che segue un codice <i>new-line</i> , mentre l'operatore '\$' individua la fine di un testo che precede un codice <i>new-line</i> .

Macro-variabile  Significato	Da usare come opzioni della funzione <i>regexexec()</i> per la comparazione di un'espressione regolare già compilata con una stringa.
REG_NOTBOL	In condizioni normali, il carattere '^' trova corrispondenza con l'inizio di una stringa. Con questa opzione, si inibisce tale corrispondenza ( <i>not begin of line</i> ).
REG_NOTEOL	In condizioni normali, il carattere '\$' trova corrispondenza con la fine di una stringa. Con questa opzione, si inibisce tale corrispondenza ( <i>not end of line</i> ).

Macro-variabile	Tipo di errore restituito dalla funzione <i>regcomp()</i> o da <i>regexexec()</i> .
REG_BADBR	Il contenuto di '{...}' (nel caso di BRE) o di '{...}' (nel caso di ERE), risulta non valido: potrebbe non trattarsi di un numero, oppure potrebbe esserci un numero troppo grande, oppure potrebbero esserci più di due numeri, oppure il primo potrebbe essere più grande del secondo. Infatti, il contenuto di tale raggruppamento deve essere un numero singolo, oppure due numeri separati da una virgola, dove il primo deve essere inferiore al secondo.
REG_BADPAT	Espressione regolare non valida (errore di sintassi).



Macro-variabile	Tipo di errore restituito dalla funzione <i>regcomp()</i> o da <i>regexexec()</i> .
REG_BADRPT	Un operatore di ripetizione, del tipo ‘?’ , ‘*’ o ‘+’, non è preceduto a un’espressione regolare, ovvero si trova in una posizione sbagliata.
REG_ECOLLATE	Elemento di collazione ( <i>collating element</i> ) non valido, nell’ambito della configurazione locale attuale.
REG_ECTYPE	Riferimento a una classe di caratteri non valida.
REG_EESCAPE	L’espressione regolare termina con ‘\’ e ciò non è ammissibile.
REG_ESUBREG	Una sequenza ‘\n’, dove <i>n</i> è un numero, è errata.
REG_EBRACE	Le parentesi graffe che descrivono la ripetizione di qualcosa non bilanciano. Può trattarsi di sequenze del tipo ‘\{...\}’ per le espressioni BRE o del tipo ‘{...}’ per le espressioni ERE.
REG_EBRACK	Parentesi quadre non bilanciate (parentesi aperta e non chiusa, o viceversa).
REG_EPAREN	Le parentesi tonde che descrivono delle sottoespressioni non bilanciano. Può trattarsi di sequenze del tipo ‘\(...\)’ per le espressioni BRE o del tipo ‘(...)’ per le espressioni ERE.
REG_ERANGE	Un’estremità di un intervallo di valori non è valido.
REG_ESPACE	Il procedimento di interpretazione dell’espressione regolare porta all’esaurimento della memoria disponibile.

Macro-variabile	Tipo di errore restituito dalla funzione <i>regcomp()</i> o da <i>regexexec()</i> .
REG_NOMATCH	Non c'è corrispondenza tra espressione regolare e stringa.

## File «sys/stat.h»

«

Funzione	Descrizione
<pre>int chmod (const char *<i>path</i>,            mode_t <i>mode</i>);</pre>	Cambia i permessi di un file, individuato dal suo percorso nel file system, rappresentati da una variabile di tipo ' <b>mode_t</b> ' (il tipo di file non può essere cambiato).
<pre>int fchmod (int <i>fdn</i>,            mode_t <i>mode</i>);</pre>	Cambia i permessi di un file aperto, individuato da un descrittore, rappresentati da una variabile di tipo ' <b>mode_t</b> ' (il tipo di file non può essere cambiato).
<pre>int fstat (int <i>fdn</i>,           struct stat *<i>buffer</i>);</pre>	Aggiorna i membri della struttura a cui punta <i>buffer</i> , con le informazioni relative al file aperto con descrittore <i>fdn</i> .

Funzione	Descrizione
<pre>int lstat     (const char *restrict <i>path</i>,      struct stat *restrict <i>buffer</i>);</pre>	<p>Aggiorna i membri della struttura a cui punta <i>buffer</i>, con le informazioni relative al file individuato dal percorso <i>path</i>. Ma a differenza di <i>stat()</i>, se il file richiesto è un collegamento simbolico, si ottengono le informazioni del file che rappresenta il collegamento stesso, ignorando ciò a cui questo punterebbe.</p>
<pre>int mkdir (const char *<i>path</i>,            mode_t <i>mode</i>);</pre>	<p>Crea una directory, specificata attraverso un percorso del file system, con i permessi indicati nel parametro <i>mode</i>, dove non si può specificare il tipo di file. I permessi richiesti vengono filtrati dalla maschera dei permessi.</p>
<pre>int mkfifo (const char *<i>path</i>,             mode_t <i>mode</i>);</pre>	<p>Crea un file FIFO, individuato dal suo percorso nel file system, utilizzando i permessi richiesti, subordinatamente al filtro della maschera dei permessi.</p>

Funzione	Descrizione
<pre>int mknod (const char *<i>path</i>,            mode_t mode, dev_t dev);</pre>	<p>Crea virtualmente un file di qualunque tipo, specificando il percorso nel file system, il tipo di file e i permessi (in questo caso il parametro <i>mode</i> serve principalmente per specificare il tipo di file) e il numero di dispositivo complessivo, nel caso particolare di un file di dispositivo. In condizioni normali, non dovrebbe essere possibile la creazione di una directory, ma se anche fosse possibile, sarebbe sconsigliabile l'uso di questa funzione per tale scopo.</p>
<pre>int stat (const char *restrict <i>path</i>,  struct stat *restrict <i>buffer</i>);</pre>	<p>Aggiorna i membri della struttura a cui punta <i>buffer</i>, con le informazioni relative al file individuato dal percorso <i>path</i>.</p>
<pre>mode_t umask (mode_t <i>mask</i>);</pre>	<p>Definisce la maschera dei permessi per il processo in corso, restituendo il valore precedente della maschera.</p>

Macroistruzione	Descrizione
S_ISBLK ( <i>mode</i> )	Restituisce <i>Vero</i> se il parametro <i>mode</i> rappresenta un file di dispositivo a blocchi.
S_ISCHR ( <i>mode</i> )	Restituisce <i>Vero</i> se il parametro <i>mode</i> rappresenta un file di dispositivo a caratteri.
S_ISFIFO ( <i>mode</i> )	Restituisce <i>Vero</i> se il parametro <i>mode</i> rappresenta un file FIFO.
S_ISREG ( <i>mode</i> )	Restituisce <i>Vero</i> se il parametro <i>mode</i> rappresenta un file puro e semplice.
S_ISDIR ( <i>mode</i> )	Restituisce <i>Vero</i> se il parametro <i>mode</i> rappresenta una directory.
S_ISLNK ( <i>mode</i> )	Restituisce <i>Vero</i> se il parametro <i>mode</i> rappresenta un collegamento simbolico.
S_ISSOCK ( <i>mode</i> )	Restituisce <i>Vero</i> se il parametro <i>mode</i> rappresenta un socket di dominio Unix.

Tipo	Nome	Membri noti del tipo derivato ' <b>struct stat</b> '
dev_t	st_dev	Numero di dispositivo dell'unità contenente il file.
ino_t	st_ino	Numero di inode del file.
mode_t	st_mode	Tipo di file e permessi di accesso relativi.
nlink_t	st_nlink	Collegamenti riferiti al file.
uid_t	st_uid	Numero UID dell'utente proprietario del file.

Tipo	Nome	Membri noti del tipo derivato ' <b>struct stat</b> '
gid_t	st_gid	Numero GID del gruppo proprietario del file.
dev_t	st_rdev	Numero di dispositivo rappresentato, nel caso si tratti effettivamente di un file di dispositivo.
off_t	st_size	Dimensione del file.
time_t	st_atime	Data e orario dell'ultimo accesso al file.
time_t	st_mtime	Data e orario dell'ultima modifica apportata al contenuto del file.
time_t	st_ctime	Data e orario dell'ultima modifica di inode (data di creazione dell'inode).
blksize_t	st_blksize	Dimensione del blocco per le operazioni di input-output.
blkcnt_t	st_blocks	Dimensione del file espressa in blocchi.

<b>Macro-variabile</b>	Da usare per la definizione di un tipo di file, in variabili di tipo ' <b>mode_t</b> '.
S_IFMT	Maschera che raccoglie tutti i bit che individuano il tipo di file.
S_IFBLK	File di dispositivo a blocchi.
S_IFCHR	File di dispositivo a caratteri.
S_IFIFO	File FIFO.
S_IFREG	File puro e semplice.
S_IFDIR	Directory.

<b>Macro-variabile</b>	Da usare per la definizione di un tipo di file, in variabili di tipo <b>'mode_t'</b> .
S_IFLNK	Collegamento simbolico.
S_IFSOCK	Socket di dominio Unix.

<b>Macro-variabile</b>	Da usare per la definizione dei permessi di un file, in variabili di tipo <b>'mode_t'</b> .
S_ISUID	SUID.
S_ISGID	SGID.
S_ISVTX	Sticky.
S_IRWXU	Lettura, scrittura ed esecuzione per l'utente proprietario.
S_IRUSR	Lettura per l'utente proprietario.
S_IWUSR	Scrittura per l'utente proprietario.
S_IXUSR	Esecuzione per l'utente proprietario.
S_IRWXG	Lettura, scrittura ed esecuzione per il gruppo.
S_IRGRP	Lettura per il gruppo.
S_IWGRP	Scrittura per il gruppo.
S_IXGRP	Esecuzione per il gruppo.
S_IRWXO	Lettura, scrittura ed esecuzione per gli altri utenti.

Macro-variabile	Da usare per la definizione dei permessi di un file, in variabili di tipo <code>'mode_t'</code> .
S_IROTH	Lettura per gli altri utenti.
S_IWOTH	Scrittura per gli altri utenti.
S_IXOTH	Esecuzione per gli altri utenti.



# Gettext

Gettext<sup>1</sup> è un sistema che aiuta nella traduzione dei messaggi dei programmi e al loro mantenimento, il quale però non fa parte della libreria standard dei linguaggi di programmazione. Ci possono essere molti modi per realizzare un programma multilingua, ma Gettext rappresenta probabilmente il metodo più semplice in pratica, consentendo la traduzione successiva senza interferire con un eseguibile già pronto, purché predisposto per questo.

## 71.1 Principio di funzionamento

La logica di Gettext è molto semplice: il programma incorpora solo i messaggi in inglese; all'esterno si associano dei file, uno per ogni linguaggio disponibile, con le traduzioni corrispondenti. Non è necessario «codificare» i messaggi in qualche modo, perché la corrispondenza avviene in modo letterale, in base al testo originale.

```
msgid "%s: cannot create the temporary file %s\n"  
msgstr "%s: non è possibile creare il file temporaneo %s\n"
```

L'esempio, che mostra un estratto ipotetico di un file PO di Gettext (*Portable object*), serve a comprendere il concetto: la stringa preceduta dalla parola chiave **msgid** (*message identity*) è quella di riferimento, che viene rimpiazzata automaticamente da quella sottostante, preceduta dalla parola chiave **msgstr**.

Le stringhe e le traduzioni di Gettext sono costanti, nel senso che **%s** viene preso come tale, mentre è il programma che lo sostituisce opportunamente. In questo senso, bisogna considerare che Gettext è nato per il linguaggio C, per essere usato in stringhe che siano argomento di funzioni come *printf()* e *sprintf()*.

## 71.2 Fasi di preparazione



La predisposizione di un programma per Gettext potrebbe essere fatta in modo più o meno automatico, attraverso strumenti specifici, oppure si può procedere in modo più semplice, anche se più oneroso dal punto di vista del tempo impiegato. Qui si intende mostrare questo modo più semplice per permettere al lettore di comprendere il concetto. La documentazione di Gettext è di per sé molto dettagliata.

Per prima cosa, il sorgente C deve essere predisposto attraverso l'inclusione di alcuni file di intestazione, quindi le stringhe vengono inglobate dalla funzione *gettext()*. Quello che segue è il classico programma che visualizza un messaggio ed esce; si suppone che si tratti del file 'ciao.c':

```
#include <stdio.h>
int main ()
{
    printf ("Hello world\n");
}
```

Nel listato successivo si vede come deve essere trasformato; va osservata l'inclusione del file di intestazione 'libintl.h':

```
#include <stdio.h>
#include <libintl.h>
#include <locale.h>

#define PACKAGE "ciao"
#define LOCALEDIR "/var/tmp"

int main ()
{
    setlocale (LC_ALL, "");
    bindtextdomain (PACKAGE, LOCALEDIR);
    textdomain (PACKAGE);

    printf (gettext ("Hello world\n"));
}
```

Le funzioni ‘**bindtextdomain**’ e ‘**textdomain**’ utilizzano come argomenti delle macro (costanti manifeste), in modo da generalizzare il funzionamento e rendere esterna la definizione di queste componenti. A parte questi particolari, si nota che *printf()* non ha più come argomento la costante di prima, ma la funzione *gettext()*.

Il programma può essere compilato, anche se non è ancora disponibile una traduzione:

```
$ cc -o ciao ciao.c [Invio]
```

La fase successiva richiede la creazione di un file PO, attraverso l’aiuto del programma ‘**xgettext**’:

```
$ xgettext ciao.c [Invio]
```

Quello che si ottiene nella directory corrente è il file ‘`messages.po`’, contenente esattamente il testo seguente:

```
# SOME DESCRIPTIVE TITLE.
```

```
# Copyright (C) YEAR Free Software Foundation, Inc.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"POT-Creation-Date: 2000-05-15 23:05+0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: ENCODING\n"

#: ciao.c:18
msgid "Hello world\n"
msgstr ""
```

Questo file deve essere modificato, in particolare per ciò che riguarda le prime direttive, oltre che per aggiungere la traduzione della frase che viene visualizzata dal programma. Per esempio, così:

```
# Ciaomondo PO file.
# Copyright (C) 2000 Pinco Pallino
# Pinco Pallino <ppinco@dinkel.brot.dg>, 2000.
#
msgid ""
msgstr ""
"Project-Id-Version: ciao-0.1\n"
"POT-Creation-Date: 2000-05-15 23:05+0200\n"
"PO-Revision-Date: 2000-05-15 22:52+0200\n"
"Last-Translator: Pinco Pallino <ppinco@dinkel.brot.dg>\n"
"Language-Team: Italian <it@li.org>\n"
"MIME-Version: 1.0\n"
```

```
"Content-Type: text/plain; charset=iso-8859-1\n"
"Content-Transfer-Encoding: 8bit\n"

#: ciao.c:18
msgid "Hello world\n"
msgstr "Ciao mondo\n"
```

Si osservi che è stato necessario togliere la riga contenente il commento speciale **#, fuzzy**.

Il file viene salvato con un nome appropriato; per esempio `ciao.po`. Quindi si passa alla sua compilazione, per ottenere il file `ciao.mo`:

```
$ msgfmt -vvvv -o ciao.mo ciao.po [Invio]
```

Dal momento che il file in questione contiene la traduzione in italiano del programma, deve essere collocato all'interno della gerarchia `it/LC_MESSAGES/`, a sua volta a partire dalla directory dichiarata con la funzione ***bindtextdomain()***, cioè `/var/tmp/it/LC_MESSAGES/` secondo quanto definito nel sorgente C.

A questo punto, dopo la collocazione appropriata del file compilato della traduzione, se la configurazione locale è corretta, lanciando l'eseguibile `ciao` si dovrebbe vedere il messaggio tradotto. Eventualmente si veda quanto descritto nella sezione [16.11](#) per quanto riguarda la configurazione della localizzazione.

## 71.3 Abbinamento a un «pacchetto»

«

Perché Gettext sappia qual è il file che contiene i messaggi tradotti, nell'ambito della configurazione locale, fa riferimento a un nome che viene definito «pacchetto», che di solito si sceglie opportunamente simile a quello del programma per il quale si fa la traduzione:

```
textdomain ("pippo");
```

L'esempio mostra l'istruzione da usare in un programma C per stabilire il nome del pacchetto secondo Gettext. Questo nome stabilisce che Gettext debba cercare il file *'sigla\_locale/LC\_MESSAGES/pippo.mo'*. Gettext determina il nome della prima parte del percorso, corrispondente a ciò che qui è stato mostrato con la metavariable *'sigla\_locale'*, analizzando alcune variabili di ambiente; precisamente segue questo ordine:

- **'LANGUAGE'**
- **'LC\_ALL'**
- altre variabili **'LC\_\*'**
- **'LANG'**

Dal valore contenuto in queste variabili si estrae la prima parte: quella che arriva fino al primo punto, se c'è. In pratica, se per ipotesi la variabile **'LANG'** contiene il valore **'it\_IT.ISO-8859-1'**, per Gettext è importante solo **'it\_IT'**. Tuttavia, anche questa informazione tende a essere eccessiva, dal momento che contiene, oltre al linguaggio, anche l'area nazionale. In pratica, alla fine contano solo le prime due lettere, che esprimono il linguaggio in base allo standard ISO 639 (tabella 13.4).

Gettext analizza il contenuto delle variabili di ambiente perché con la funzione *setlocale()* è stata azzerata internamente la definizione ‘**LC\_ALL**’. Usando la funzione *setlocale()* si potrebbe imporre un certo linguaggio, indipendentemente dalle variabili di ambiente relative.

Tornando all’esempio iniziale, si tratta del file ‘`it/LC_MESSAGES/pippo.mo`’, che in condizioni normali, Gettext cerca a partire dalla directory ‘`/usr/share/locale/`’ (o eventualmente da un’altra posizione in base al modo in cui è stato compilato). Tuttavia è possibile richiedere espressamente una collocazione differente attraverso un’istruzione già vista da collocare nel programma interessato:

```
bindtextdomain ("pippo", "/var/tmp");
```

In tal caso, se si scrive questo in un programma, Gettext va a cercare precisamente il file ‘`/var/tmp/it/LC_MESSAGES/pippo.mo`’.

## 71.4 Creazione e mantenimento dei file PO

È già stato mostrato in breve come si crea un file PO attraverso il programma ‘**xgettext**’. È il caso di osservare che ‘**xgettext**’ può ricevere l’indicazione di più file sorgenti che fanno capo allo stesso dominio di traduzione:

```
xgettext [opzioni] file_sorgente...
```

In particolare, tra le opzioni può essere interessante segnalare ‘**--default-domain=dominio**’, che serve a ‘**xgettext**’ per conoscere il dominio a cui si fa riferimento, creando così il file ‘*dominio.po*’, invece del solito ‘`messages.po`’.

```
$ xgettext --default-domain=ciao *.c [Invio]
```

L'esempio mostra come ottenere il file 'ciao.po' a partire da tutti i file che terminano con l'estensione '.c'.

Quando si aggiornano i sorgenti di un programma già tradotto, si pone il problema di aggiornare nello stesso modo i file PO precedenti. Per fare questo si deve ricreare il file PO iniziale non tradotto, nel modo appena visto, quindi si usa il programma '**msgmerge**':

```
msgmerge [opzioni] file_po_originale file_po_successivo > file_po_aggiornato
```

In pratica, '**msgmerge**' fonde assieme due file PO, preservando le traduzioni del primo file riferite a messaggi che si trovano ancora nel secondo. Per esempio, se si dispone già del file 'vecchio.po' con le traduzioni, mentre con '**xgettext**' è appena stato generato un file PO non tradotto per lo stesso programma, che qui viene chiamato 'non\_tradotto.po', si può ottenere un nuovo file PO con le traduzioni vecchie ancora valide e con i messaggi nuovi da tradurre:

```
$ msgmerge vecchio.po non_tradotto.po > nuovo.po [Invio]
```

Naturalmente, questa operazione si fa nel momento in cui ci si accinge ad aggiornare materialmente la traduzione del programma, altrimenti questo lavoro non avrebbe senso, dal momento che un file PO contenente messaggi non tradotti non può essere compilato.

```
msgfmt [opzioni] file_po
```

Il programma '**msgfmt**' è quello che si occupa di compilare i file PO



ottenendo i file MO, adatti alla propria piattaforma. È praticamente indispensabile utilizzare l'opzione '`--output-file=file_mo`' ('`-o`'), per indicare il nome del file da creare. Inoltre, è opportuno utilizzare più di una volta l'opzione '`--verbose`' ('`-v`') per avere una visione chiara del procedimento, ovvero dei motivi per i quali alle volte il file non viene compilato.

```
$ msgfmt -vvvv --output-file=prova.mo prova.po [Invio]
```

L'esempio mostra l'utilizzo tipico di questo programma, dove in particolare viene richiesto un livello di dettaglio delle informazioni generate molto elevato (quattro volte '`-v`').

### 71.4.1 Commenti «fuzzy»

Ogni volta che qualche indicazione all'interno di un file PO è incerta, in quanto predefinita o determinata automaticamente in modo non sicuro, viene aggiunto un commento speciale contenente la parola '**fuzzy**'. In presenza di commenti del genere si richiede un intervento manuale, dopo il quale deve essere rimossa tale parola, altrimenti '**msgfmt**' si rifiuta di completare la compilazione dei file PO.

## 71.5 Gettext con i programmi Perl

Esiste la possibilità di utilizzare Gettext anche nei programmi Perl. Per questo è necessario includere nel programma Perl il riferimento a un modulo esterno: Perl-gettext. Il tutto si svolge in maniera molto simile a un programma C, inserendo inizialmente le istruzioni seguenti:

```
use POSIX;
use Locale::gettext;
setlocale (LC_ALL, "");
textdomain ("dominio_gettext");
[bindtextdomain ("dominio_gettext", "directory");]
```

Per esempio, se si tratta del programma «Pippo», il dominio per Gettext potrebbe essere convenientemente «pippo», arrivando al risultato seguente:

```
use POSIX;
use Locale::gettext;
setlocale (LC_ALL, "");
textdomain ("pippo");
bindtextdomain ("pippo", "/opt/pippo/locale");
```

Potrebbe essere che la funzione *bindtextdomain()* non si comporti come previsto; in tal caso sarebbe meglio evitarne l'uso.

Per il resto, tutto funziona come per i sorgenti scritti in C:

```
print STDOUT (gettext ("Hello world\n"));
```

Tuttavia, Perl non è identico al C, per cui occorre osservare alcune situazioni specifiche. In particolare, non è possibile inserire in un argomento della funzione *gettext()* una variabile di Perl che deve essere espansa, perché questa espansione avverrebbe prima che *gettext()* possa ricevere tale argomento. Pertanto, l'esempio seguente non può essere tradotto:

```
# Esempio errato.
print STDOUT (gettext ("Il file $file contiene "
                       "caratteri non validi\n"));
```

Il modo giusto di agire è quello di sostituire *print()* con *printf()*, come nell'esempio seguente:

```
# Esempio corretto.
printf STDOUT (gettext ("Il file %s contiene "
                        "caratteri non validi\n"),
              $file);
```

Infatti, il parametro '**s**' viene sostituito alla fine da '**printf**', per cui inizialmente la stringa non viene modificata.

Un altro problema da considerare sono i messaggi lunghi, che richiedono più righe. In Perl si potrebbe fare una cosa del genere:

```
printf STDOUT
(gettext
 ( "Usage: %s --input-type=TYPE INPUT_FILE REPORT_FILE\n"
 . "      %s --help\n"
 . "      %s --version\n"
 . "\n"
 . "Check for HTTP and FTP URI inside a text.\n"
 . "\n"
 . "Options:\n"
 . "--help          display this help and exit.\n"
 . "--version       display version information \n"
 . "                  and exit.\n"
 . "--input-type=TYPE define the input type:\n"
 . "  standard        input is a simple text file;\n"
 . "  html, sgml      input is a typical SGML file;\n"
 . "  texi, texinfo   input is a Texinfo source \n"
 . "                  file.\n"
 . "\n"
 . "Arguments:\n"
 . "\n"
 . "INPUT_FILE       the input file.\n"
 . "\n"
 . "REPORT_FILE      a file that is generated with \n"
```

```
. "                                the reported errors.\n"),
$program_name, $program_name, $program_name);
```

Ma questo non viene riconosciuto da **'xgettext'** che riesce a prelevare solo la prima riga:

```
#: urichk:55
#, c-format
msgid "Usage: %s --input-type=TYPE INPUT_FILE REPORT_FILE\n"
msgstr ""
```

In queste situazioni eccezionali, occorre intervenire a mano nel file PO; sia la prima volta che si crea il file, sia tutte le volte successive in cui lo si aggiorna.

### 71.5.1 Adattare il sorgente Perl per facilitare l'estrazione dei messaggi da tradurre

«

Se si prendono delle piccole precauzioni nella scrittura delle stringhe con Perl, è possibile filtrare il sorgente successivamente, per passarlo a **'xgettext'** in modo che questo possa interpretarlo correttamente (come se fosse un programma C). In pratica, occorre fare in modo che le stringhe siano unite, senza ottenerle attraverso un concatenamento. L'esempio già mostrato va modificato nel modo seguente:

```
printf STDOUT
    (gettext
      ("\
Usage: %s --input-type=TYPE INPUT_FILE REPORT_FILE\
      %s --help\
      %s --version\
\
Check for HTTP and FTP URI inside a text.\
```

```

\
Options:\
--help                display this help and exit.\
--version             display version information and exit.\
--input-type=TYPE    define the input type:\
    standard          input is a simple text file;\
    html, sgml        input is a typical SGML file;\
    texi, texinfo     input is a Texinfo source file.\
\
Arguments:\
\
INPUT_FILE            the input file.\
\
REPORT_FILE           a file that is generated with the \
                      reported errors."),
                      $program_name, $program_name, $program_name);

```

A questo punto, si può realizzare un piccolo programma Perl che inserisce il codice ‘\n’ alla fine delle righe (sostituendolo alla barra obliqua inversa) e sostituisce il codice ‘\@’ con ‘@’:

```

#!/usr/bin/perl
$line = "";
while ($line = <STDIN>)
{
    $line =~ s/\\$/\\n\\;/;
    $line =~ s/\\@/\\/g;
    print STDOUT ($line);
}

```

Come si vede, il programma Perl che si ottiene legge dallo standard input ed emette il risultato della sua trasformazione attraverso lo standard output.

## 71.5.2 Alleviare gli inconvenienti di un modulo in più

&lt;&lt;

Scrivere un programma Perl che faccia uso di `Gettext`, significa costringere a installare il modulo `Perl-gettext`. Purtroppo, una delle cose che complicano di più l'utilizzo di programmi Perl sono i moduli aggiuntivi necessari che devono essere installati perfettamente come previsto.

Questo potrebbe sembrare un problema secondario; invece non lo è affatto. A questo punto, se si vuole consentire al proprio programma Perl di funzionare anche in un ambiente non tanto amichevole, si deve prevedere una via di uscita:

```
#!/usr/bin/perl
#
...

use POSIX;
use Locale::gettext;
setlocale (LC_ALL, "");
textdomain ("pippo");

#sub gettext
#{
#   return $_[0];
#}

...
```

Come si vede nell'esempio, appare la dichiarazione di una funzione commentata, il cui scopo sarebbe quello di sostituirsi alla funzione `gettext()` del modulo `'Locale::gettext'`. Se non si dispone di `Perl-gettext` basta commentare la prima parte e togliere i commenti

dalla seconda: ovviamente i messaggi rimangono così nella lingua di partenza.

```
#!/usr/bin/perl
#
...

#use POSIX;
#use Locale::gettext;
#setlocale (LC_ALL, "");
#textdomain ("pippo");

sub gettext
{
    return $_[0];
}

...
```

<sup>1</sup> **Gettext** GNU GPL

