Parte vi

Corso basilare di programmazione

«

Introduzione

Il corso contenuto in questa parte riguarda i concetti elementari della programmazione, al livello minimo di astrazione possibile, utilizzando il linguaggio C per la messa in pratica degli algoritmi. Il corso è «basilare», ma gli argomenti trattati non sono così semplici come il termine potrebbe fare supporre.

Gli argomenti del corso sono già trattati in altri capitoli dell'opera, ma qui, in più, si inseriscono degli esercizi corretti.[1]

Per svolgere il corso correttamente è indispensabile fare tutti gli esercizi, verificando le soluzioni. Se il corso è guidato da un tutore, è bene presentarsi sempre alle lezioni avendo già studiato gli argomenti che devono essere trattati e avendo fatto gli esercizi indicati.

Programma didattico

Il corso, se assistito da un tutore, prevede l'impiego di circa 45 ore, di cui, almeno otto da dedicare alle verifiche (due ore di verifica per modulo, più due ore aggiuntive per una verifica di recupero complessiva).

Modulo 1

- **sistemi di numerazione**
  - decimale
  - binario
  - ottale
  - esadecimale
  - conversioni numeriche intere
  - conversioni numeriche intere tra binario, ottale e esadecimale
- **operazioni aritmetiche elementari in binario**
  - complemento a uno
  - complemento a due
  - somma binaria
  - sottrazione binaria
  - rappresentazione dei numeri interi con segno
- **operazioni elementari all'interno della CPU**
  - aumento e riduzione delle cifre binarie di un numero intero senza segno
  - aumento e riduzione delle cifre binarie di un numero intero con segno
  - somme con i numeri interi con segno
  - somme e sottrazioni con i numeri interi senza segno
  - scorrimento logico (senza segno)
  - scorrimento aritmetico (con segno)
  - rotazione
  - AND
  - OR
  - XOR
  - NOT
- **organizzazione della memoria**
  - pila dei dati o *stack* (cenni)
  - chiamata di funzioni e passaggio degli argomenti attraverso la pila (cenni)
  - variabili scalari
  - array
  - stringhe
  - puntatori
  - ordine dei byte

Modulo 2

- **primo approccio al linguaggio C**
  - commenti, istruzioni, raggruppamenti
  - compilazione

– emissione di messaggi testuali
– sospensione dell'esecuzione del programma in attesa della pressione di [*Invio*]
– costruzione del primo programma che emette un messaggio e attende la pressione di [*Invio*] per terminare

• **tipi principali del linguaggio C**
– tipi scalari primitivi: char, short int, int, long int, float, double
– tipi scalari primitivi: distinzione tra presenza e assenza del segno
– costanti letterali
– dichiarazione di variabili scalari
– il tipo void

• **operatori ed espressioni del linguaggio C**
– operatori aritmetici
– operatori di confronto
– operatori logici
– operatori binari
– cast (conversione di tipo)
– espressioni multiple

• **strutture di controllo di flusso del linguaggio C**
– if
– switch
– while
– for

• **funzioni del linguaggio C**
– funzione '`main (void)`'
– prototipo
– descrizione della funzione
– valore restituito dalla funzione
– valore restituito dal programma

Modulo 3

• **puntatori in C**
– espressioni a cui si assegnano dei valori (*lvalue*)
– dichiarazione di una variabile puntatore
– dereferenziazione di un puntatore
– *big endian*, *little endian* e puntatori
– puntatori come parametri di una funzione

• **array in C**
– dichiarazione di un array
– selezione di un elemento all'interno di un array all'interno delle espressioni
– array a più dimensioni
– uso del ciclo '`for`' per la scansione di un array
– relazione tra array e puntatori
– dereferenziazione di un puntatore come se fosse un array
– array come parametri di una funzione
– aritmetica dei puntatori
– stringhe

• **puntatori di puntatori**
– dichiarazione e dereferenziazione
– puntatori a più dimensioni, ovvero: array di puntatori
– parametri della funzione ***main()***

Strumenti per la compilazione

Per potersi esercitare nell'uso del linguaggio C, è possibile avvalersi di un servizio *pastebin* completo, come *http://codepad.org* e *http://ideone.com*. A questi servizi ci si deve iscrivere, in modo da poter salvare i propri esercizi.

Se si dispone di un elaboratore completo, si può utilizzare un compilatore vero e proprio. I sistemi GNU e derivati, dispongono di norma

del compilatore GNU C, ma in generale ogni sistema Unix dovrebbe consentire di compilare un programma utilizzando semplicemente il comando '`cc`', a cui si fa riferimento inizialmente nel capitolo del corso che introduce alla compilazione stessa.

Per compilare un programma C in un sistema operativo come MS-Windows, occorre uno strumento apposito. Nel caso di MS-Windows si suggerisce l'uso di Dev-C++ che è molto facile da installare e da usare, pur non offrendo il classico '`cc`' da riga di comando. Nelle figure successive viene mostrato, intuitivamente, il procedimento per creare un file, compilarlo ed eseguirlo.

Figura u5.1. Aspetto di Dev-C++ dopo l'avvio.

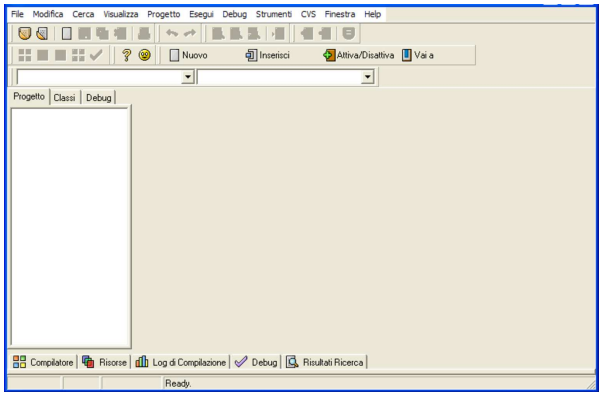


Figura u5.2. Creazione di un file sorgente nuovo.

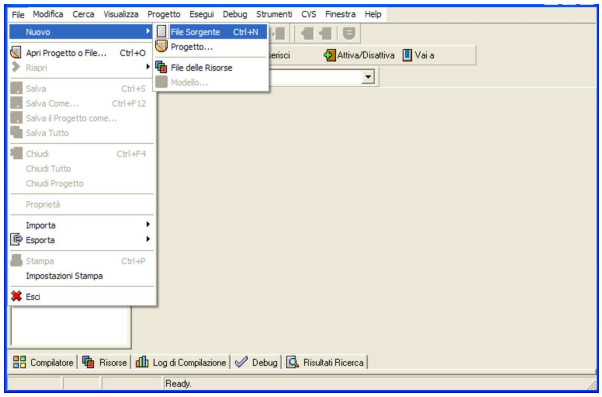


Figura u5.3. Un file che mostra un messaggio, attende la pressione di [*Invio*] e termina di funzionare.

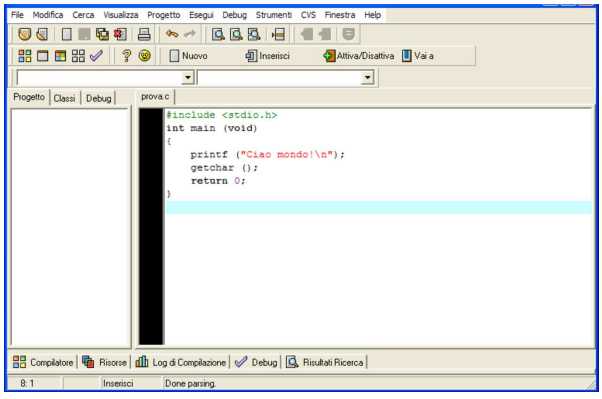


Figura u5.4. Compilazione.

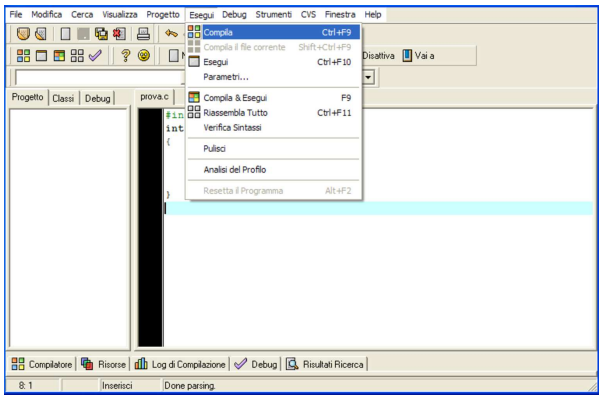


Figura u5.5. Esecuzione.

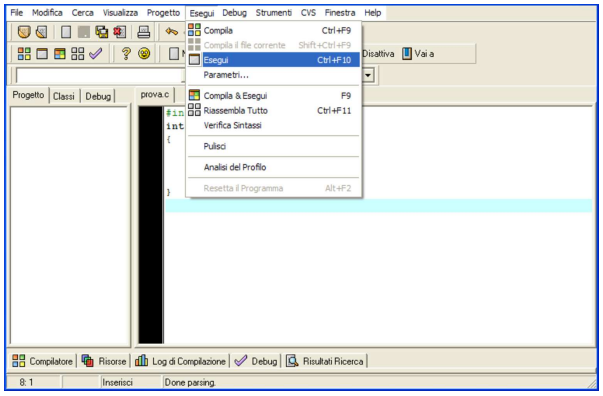
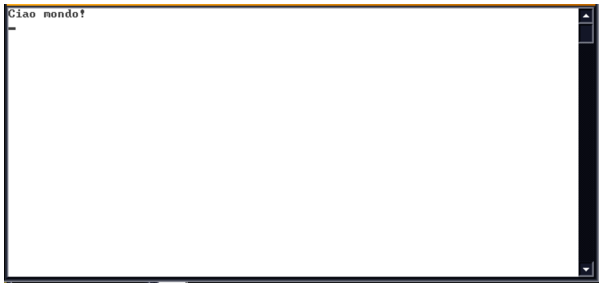


Figura u5.6. Finestra testuale da dove si vede l'emissione del messaggio del programma. Basta premere [*Invio*] per fare terminare il funzionamento del programma e lasciare così che la finestra si chiuda.



Riferimenti:

- *Codepad*, *http://codepad.org*
- *Ideone.com*, *http://ideone.com*
- BloodshedSoftware, *Dev-C++*, *http://www.bloodshed.net/devcpp.html*, *http://www.bloodshed.net/dev/*, *http://sourceforge.net/projects/dev-cpp/*

[1] Va tenuta sempre in considerazione la possibilità che alcune soluzioni o correzioni non siano esatte, pertanto, in caso di dubbio, va consultato un docente o comunque una persona competente.

Capitolo 80
Dai sistemi di numerazione all'organizzazione della memoria

80.1 Sistemi di numerazione

I sistemi di numerazione più comuni sono di tipo posizionale, definiti in tal modo perché la posizione in cui appaiono le cifre ha significato. I sistemi di numerazione posizionali si distinguono per la ***base di numerazione***.

80.1.1 Sistema decimale

Il sistema di numerazione decimale è tale perché utilizza dieci simboli, pertanto è un sistema ***in base dieci***. Trattandosi di un sistema di numerazione posizionale, le cifre numeriche, da «0» a «9», vanno considerate secondo la collocazione relativa tra di loro.

A titolo di esempio si può prendere il numero 745 che, eventualmente, va rappresentato in modo preciso come 745_{10}: secondo l'esperienza comune si comprende che si tratta di settecento, più quaranta, più cinque, ovvero, settecentoquarantacinque. Si arriva a questo valore sapendo che la prima cifra a destra rappresenta delle unità (cinque unità), la seconda cifra a partire da destra rappresenta delle decine (quattro decine), la terza cifra a partire da destra rappresenta delle centinaia (sette centinaia).

Figura 80.1. Esempio di scomposizione di un numero in base dieci.

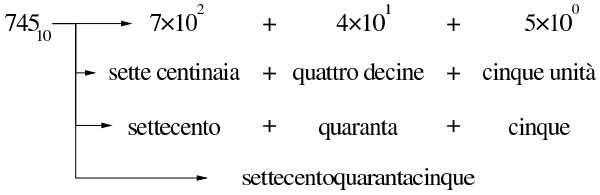
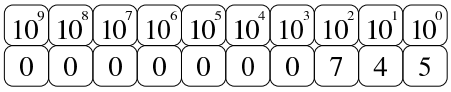


Figura 80.2. Scomposizione di un numero in base dieci.



80.1.2 Sistema binario

Il sistema di numerazione binario (in base due), utilizza due simboli: «0» e «1».

Figura 80.3. Esempio di scomposizione di un numero in base due.

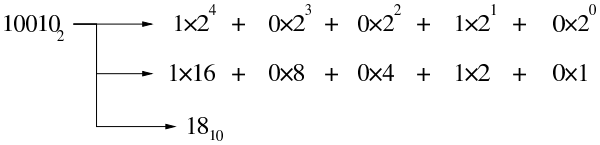
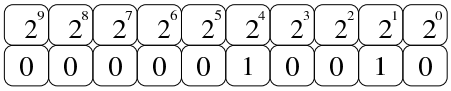
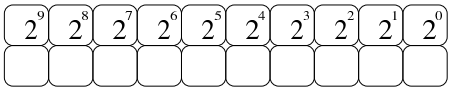


Figura 80.4. Scomposizione di un numero in base due.

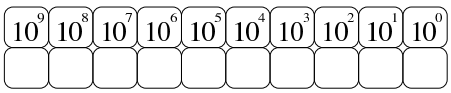


80.1.2.1 Esercizio

Si traduca il valore 11110011_2 in base dieci, con l'aiuto dello schema successivo, completandolo con una matita o con una penna, eventualmente con l'uso di una calcolatrice comune:

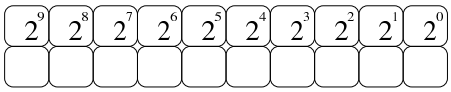


Pertanto, il risultato in base dieci è:

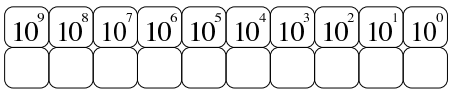


80.1.2.2 Esercizio

Si traduca il valore 01100110_2 in base dieci, con l'aiuto dello schema successivo, completandolo con una matita o con una penna, eventualmente con l'uso di una calcolatrice comune:



Pertanto, il risultato in base dieci è:



80.1.3 Sistema ottale

Il sistema di numerazione ottale (in base otto), utilizza otto simboli: da «0» a «7».

Figura 80.9. Esempio di scomposizione di un numero in base otto.

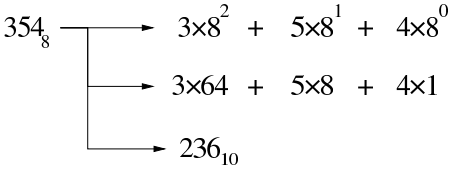
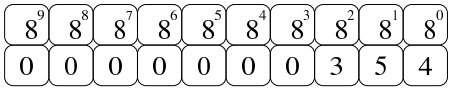
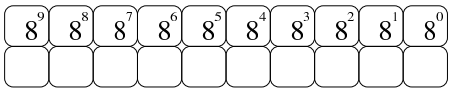


Figura 80.10. Scomposizione di un numero in base otto.

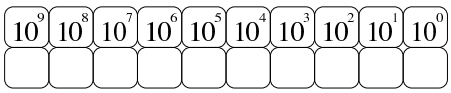


80.1.3.1 Esercizio

Si traduca il valore 1357_8 in base dieci, con l'aiuto dello schema successivo, completandolo con una matita o con una penna, eventualmente con l'uso di una calcolatrice comune:

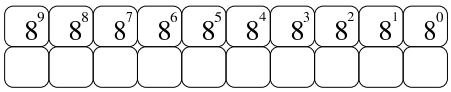


Pertanto, il risultato in base dieci è:

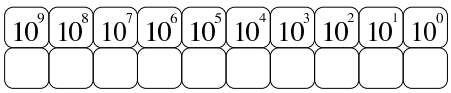


80.1.3.2 Esercizio

Si traduca il valore 7531_8 in base dieci, con l'aiuto dello schema successivo, completandolo con una matita o con una penna, eventualmente con l'uso di una calcolatrice comune:



Pertanto, il risultato in base dieci è:



80.1.4 Sistema esadecimale

Il sistema di numerazione esadecimale (in base sedici), utilizza sedici simboli: le cifre numeriche da «0» a «9» e le lettere (maiuscole) dalla «A» alla «F».

Figura 80.15. Esempio di scomposizione di un numero in base sedici.

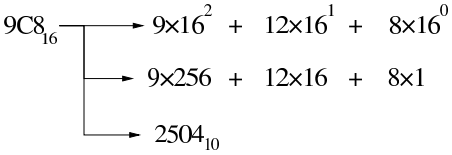


Figura 80.16. Scomposizione di un numero in base sedici.

16^9	16^8	16^7	16^6	16^5	16^4	16^3	16^2	16^1	16^0
0	0	0	0	0	0	0	9	C	8

80.1.4.1 Esercizio

Si traduca il valore $15AC_{16}$ in base dieci, con l'aiuto dello schema successivo, completandolo con una matita o con una penna, eventualmente con l'uso di una calcolatrice comune:

16^9	16^8	16^7	16^6	16^5	16^4	16^3	16^2	16^1	16^0

Pertanto, il risultato in base dieci è:

10^9	10^8	10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0

80.1.4.2 Esercizio

Si traduca il valore $CF58_{16}$ in base dieci, con l'aiuto dello schema successivo, completandolo con una matita o con una penna, eventualmente con l'uso di una calcolatrice comune:

16^9	16^8	16^7	16^6	16^5	16^4	16^3	16^2	16^1	16^0

Pertanto, il risultato in base dieci è:

10^9	10^8	10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0

80.2 Conversioni numeriche di valori interi

Un numero intero espresso in base dieci, viene interpretato sommando il valore di ogni singola cifra moltiplicando per 10^n (***n*** rappresenta la cifra ***n***-esima, a partire da zero). Per esempio, 12345 si può esprimere come $5\times10^0 + 4\times10^1 + 3\times10^2 + 2\times10^3 + 1\times10^4$. Nello stesso modo, si può scomporre un numero per esprimerlo in base dieci dividendo ripetutamente il numero per la base, recuperando ogni volta il resto della divisione. Per esempio, 12345 (che ovviamente è già espresso in base dieci), si scompone nel modo seguente: 12345/10=1234 con il resto di cinque; 1234/10=123 con il resto di quattro; 123/10=12 con il resto di tre; 12/10=1 con il resto di due; 1/10=0 con il resto di uno (quando si ottiene un quoziente nullo, la conversione è terminata). Ecco che la sequenza dei resti dà il numero espresso in base dieci: 12345.

Riquadro 80.21. Il resto della divisione.

Per riuscire a convertire un numero intero da una base di numerazione a un'altra, occorre sapere calcolare il resto della divisione.
Si immagini di avere un sacchetto di nove palline uguali, da dividere equamente fra quattro amici. Per calcolare quante palline spettano a ognuno, si esegue la divisione seguente:
9/4 = 2,25
Il risultato intero della divisione è due, pertanto ognuno dei quattro amici può avere due palline e il resto della divisione è costituito dalle palline che non possono essere suddivise. Come si comprende facilmente, il resto è di una pallina:
9 − (2×4) = 1

80.2.1 Numerazione ottale

La numerazione ottale, ovvero in base otto, si avvale di otto cifre per rappresentare i valori: da zero a sette. La tecnica di conversione di un numero ottale in un numero decimale è la stessa mostrata a titolo esemplificativo per il sistema decimale, con la differenza che la base di numerazione è otto. Per esempio, per interpretare il numero ottale 12345_8, si procede come segue: $5\times8^0 + 4\times8^1 + 3\times8^2 + 2\times8^3 + 1\times8^4$. Pertanto, lo stesso numero si potrebbe rappresentare in base dieci come 5349. Al contrario, per convertire il numero 5349 (qui espresso in base 10), si può procedere nel modo seguente: 5349/8=668 con il resto di cinque; 668/8=83 con il resto di quattro; 83/8=10 con il resto di tre; 10/8=1 con il resto di due; 1/8=0 con il resto di uno. Ecco che così si riottiene il numero ottale 12345_8.

Figura 80.22. Conversione in base otto.

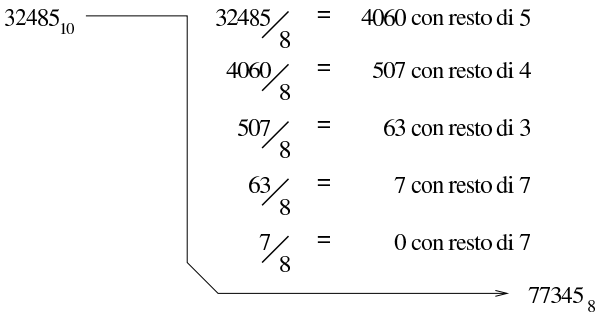
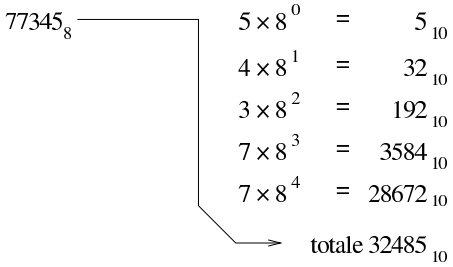


Figura 80.23. Calcolo del valore corrispondente di un numero espresso in base otto.



80.2.1.1 Esercizio

Si traduca il valore 1234_{10} in base otto, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:

8^9	8^8	8^7	8^6	8^5	8^4	8^3	8^2	8^1	8^0

80.2.1.2 Esercizio

Si traduca il valore 4321_{10} in base otto, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:

8^9	8^8	8^7	8^6	8^5	8^4	8^3	8^2	8^1	8^0

80.2.2 Numerazione esadecimale

La numerazione esadecimale, ovvero in base sedici, funziona in modo analogo a quella ottale, con la differenza che si avvale di 16 cifre per rappresentare i valori, per cui si usano le cifre numeriche da zero a nove, più le lettere da «A» a «F» per i valori successivi. In pratica, la lettera «A» nelle unità corrisponde al numero 10 e la lettera «F» nelle unità corrisponde al numero 15.

La tecnica di conversione è la stessa già vista per il sistema ottale, tenendo conto della difficoltà ulteriore introdotta dalle lette-

re aggiuntive. Per esempio, per interpretare il numero esadecimale $19ADF_{16}$, si procede come segue: $15\times16^0 + 13\times16^1 + 10\times16^2 + 9\times16^3 + 1\times16^4$. Pertanto, lo stesso numero si potrebbe rappresentare in base dieci come 105183. Al contrario, per convertire il numero 105183 (qui espresso in base 10), si può procedere nel modo seguente: 105183/16=6573 con il resto di 15, ovvero F_{16}; 6573/16=410 con il resto di 13, ovvero D_{16}; 410/16=25 con il resto di 10, ovvero A_{16}; 25/16=1 con il resto di nove; 1/16=0 con il resto di uno. Ecco che così si riottiene il numero esadecimale $19ADF_{16}$.

Figura 80.26. Conversione in base sedici.

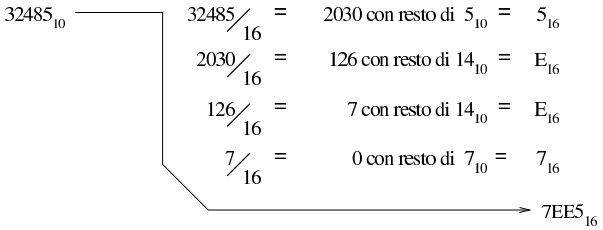
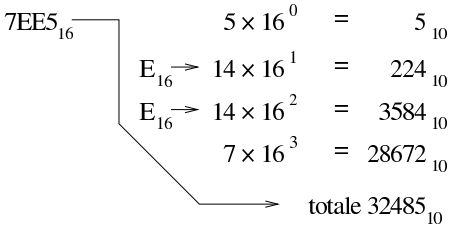
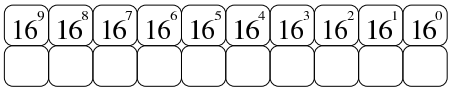


Figura 80.27. Calcolo del valore corrispondente di un numero espresso in base sedici.



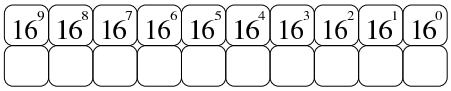
80.2.2.1 Esercizio

Si traduca il valore 44221_{10} in base sedici, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.2.2.2 Esercizio

Si traduca il valore 12244_{10} in base sedici, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.2.3 Numerazione binaria

La numerazione binaria, ovvero in base due, si avvale di sole due cifre per rappresentare i valori: zero e uno. Si tratta evidentemente di un esempio limite di rappresentazione di valori, dal momento che utilizza il minor numero di cifre. Questo fatto semplifica in pratica la conversione.

Seguendo la logica degli esempi già mostrati, si analizza brevemente la conversione del numero binario 1100_2: $0\times2^0 + 0\times2^1 + 1\times2^2 + 1\times2^3$. Pertanto, lo stesso numero si potrebbe rappresentare come 12 secondo il sistema standard. Al contrario, per convertire il numero 12, si può procedere nel modo seguente: 12/2=6 con il resto di zero; 6/2=3 con il resto di zero; 3/2=1 con il resto di uno; 1/2=0 con il resto di uno. Ecco che così si riottiene il numero binario 1100_2.

Figura 80.30. Conversione in base due.

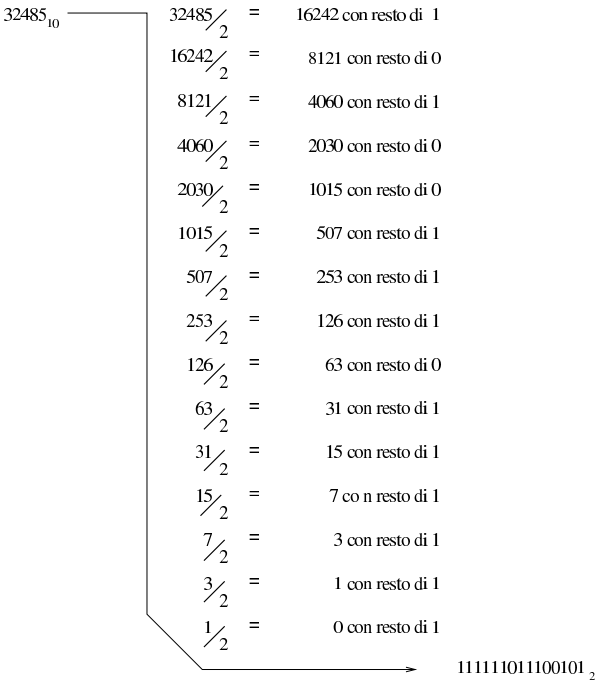
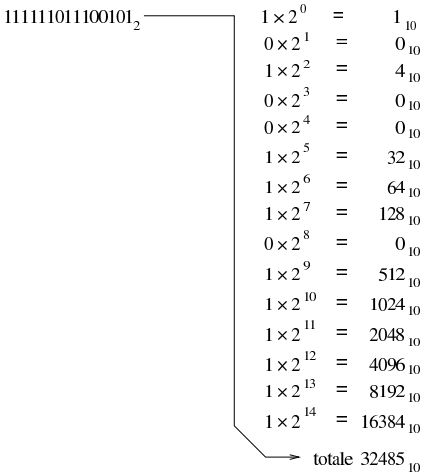
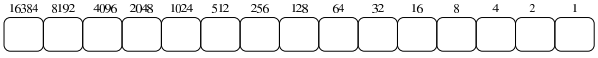


Figura 80.31. Calcolo del valore corrispondente di un numero espresso in base due.

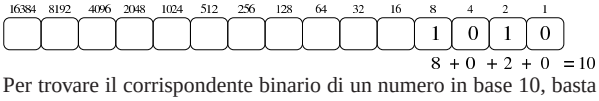


Si può convertire un numero in binario, in modo più semplice, se si costruisce una tabellina simile a quella seguente:

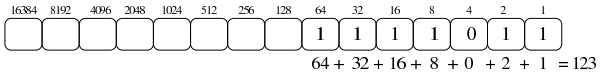


I valori indicati sopra ogni casellina sono la sequenza delle potenze di due: 2^0, 2^1, 2^2,... 2^n.

Se si vuole convertire un numero binario in base dieci, basta disporre le sue cifre dentro le caselline, allineato a destra, moltiplicando ogni singola cifra per il valore che gli appare sopra, sommando poi ciò che si ottiene. Per esempio:

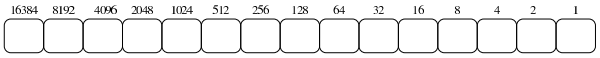


Per trovare il corrispondente binario di un numero in base 10, basta sottrarre sempre il valore più grande possibile. Supponendo di voler convertire il numero 123 in binario, si possono sottrarre i valori: 64, 32, 16, 8, 2 e 1:



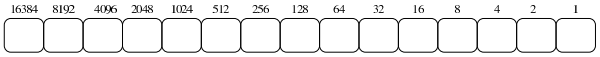
80.2.3.1 Esercizio

Si traduca il valore 1234_{10} in base due, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.2.3.2 Esercizio

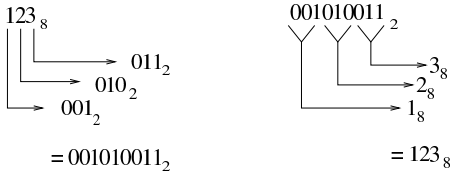
Si traduca il valore 4321_{10} in base due, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.2.4 Conversione tra ottale, esadecimale e binario

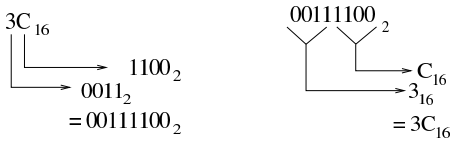
I sistemi di numerazione ottale ed esadecimale hanno la proprietà di convertirsi in modo facile in binario e viceversa. Infatti, una cifra ottale richiede esattamente tre cifre binarie per la sua rappresentazione, mentre una cifra esadecimale richiede quattro cifre binarie per la sua rappresentazione. Per esempio, il numero ottale 123_8 si converte facilmente in 001010011_2; inoltre, il numero esadecimale $3C_{16}$ si converte facilmente in 00111100_2.

Figura 80.37. Conversione tra la numerazione ottale e numerazione binaria.



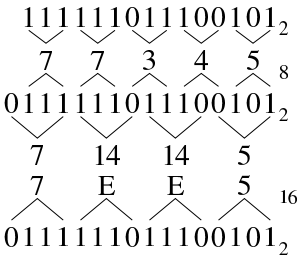
In pratica, è sufficiente convertire ogni cifra ottale o esadecimale nel valore corrispondente in binario. Quindi, sempre nel caso di 123_8, si ottengono 001_2, 010_2 e 011_2, che basta attaccare come già è stato mostrato. Nello stesso modo si procede nel caso di $3C_{16}$, che forma rispettivamente 0011_2 e 1100_2.

Figura 80.38. Conversione tra la numerazione esadecimale e numerazione binaria.



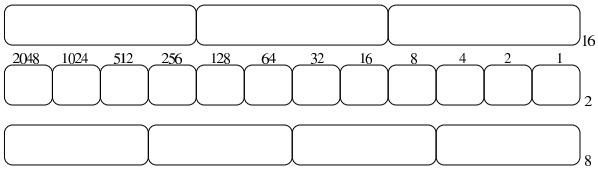
È evidente che risulta facilitata ugualmente la conversione da binario a ottale o da binario a esadecimale.

Figura 80.39. Riassunto della conversione tra binario-ottale e binario-esadecimale.



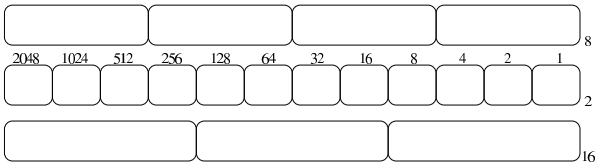
80.2.4.1 Esercizio

Si traduca il valore ABC_{16} in base due e quindi in base otto, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.2.4.2 Esercizio

Si traduca il valore 7655_8 in base due e quindi in base sedici, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.3 Conversioni numeriche di valori non interi

La conversione di valori non interi in basi di numerazione differenti, richiede un procedimento più complesso, dove si convertono, separatamente, la parte intera e la parte restante.

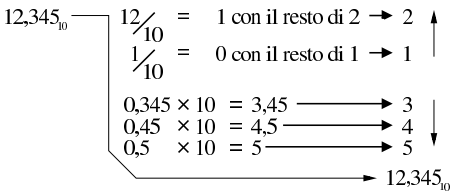
Il procedimento di scomposizione di un numero che contenga delle cifre dopo la parte intera, si svolge in modo simile a quello di un numero intero, con la differenza che le cifre dopo la parte intera vanno moltiplicate per la base elevata a una potenza negativa. Per esempio, il numero $12{,}345_{10}$ si può esprimere come $1\times10^1 + 2\times10^0 + 3\times10^{-1} + 4\times10^{-2} + 5\times10^{-3}$.

80.3.1 Conversione da base 10 ad altre basi

Come accennato nella premessa del capitolo, la conversione di un numero in un'altra base procede in due fasi: una per la parte intera, l'altra per la parte restante, unendo poi i due valori trovati. Per comprendere il meccanismo conviene simulare una conversione dalla base 10 alla stessa base 10, con un esempio: 12,345.

Per la parte intera, si procede come al solito, dividendo per la base di numerazione del numero da trovare e raccogliendo i resti; per la parte rimanente, il procedimento richiede invece di moltiplicare il valore per la base di destinazione e raccogliere le cifre intere trovate. Si osservi la figura successiva che rappresenta il procedimento.

Figura 80.42. Conversione da base 10 a base 10.



Quello che si deve osservare dalla figura è che l'ordine delle cifre cambia nelle due fasi del calcolo. Nelle figure successive si vedono altri esempi di conversione nelle altre basi di numerazione comuni.

Figura 80.43. Conversione da base 10 a base 16.

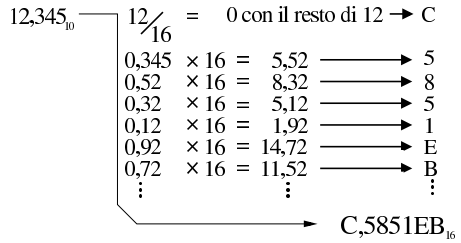


Figura 80.44. Conversione da base 10 a base 8.

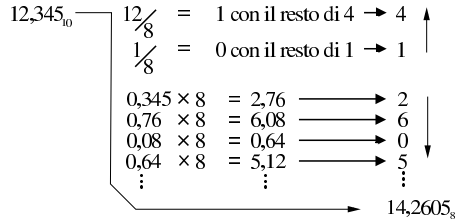
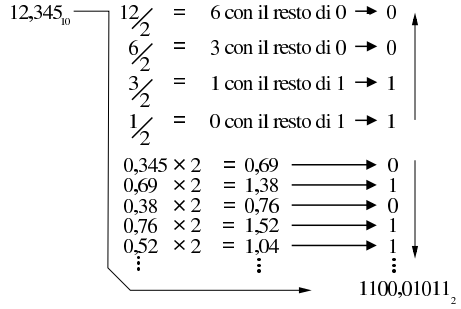
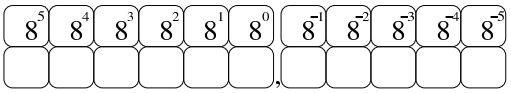


Figura 80.45. Conversione da base 10 a base 2.



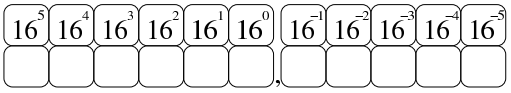
80.3.1.1 Esercizio

Si traduca il valore $43{,}21_{10}$ in base otto, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



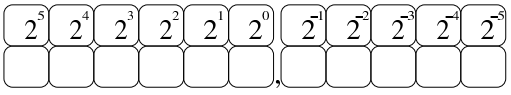
80.3.1.2 Esercizio

Si traduca il valore $765{,}4321_{10}$ in base sedici, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.3.1.3 Esercizio

Si traduca il valore $21{,}11_{10}$ in base due, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.3.2 Conversione a base 10 da altre basi

Per convertire un numero da una base di numerazione qualunque alla base 10, è necessario attribuire a ogni cifra il valore corrispondente, da sommare poi per ottenere il valore complessivo. Nelle figure successive si vedono gli esempi relativi alle basi di numerazione più comuni.

Figura 80.49. Conversione da base 16 a base 10.

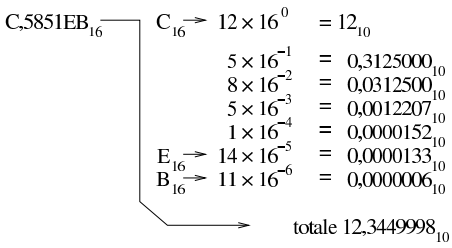


Figura 80.50. Conversione da base 8 a base 10.

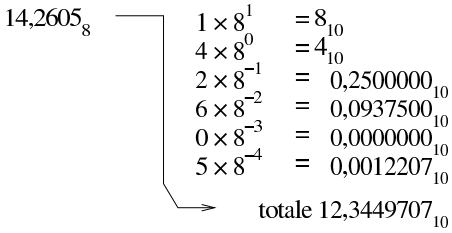
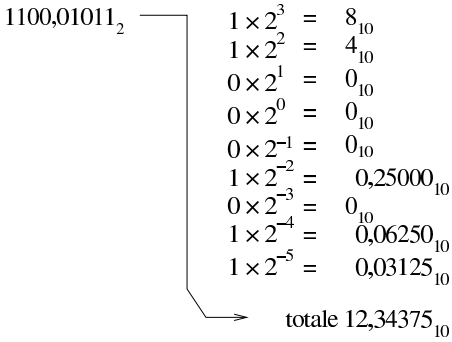
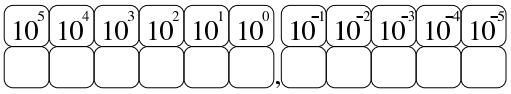


Figura 80.51. Conversione da base 2 a base 10.



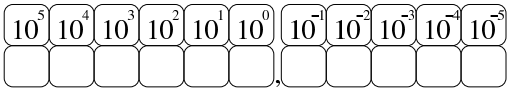
80.3.2.1 Esercizio

Si traduca il valore $765{,}432_8$ in base dieci, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



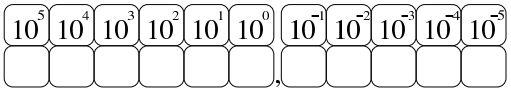
80.3.2.2 Esercizio

Si traduca il valore $AB{,}CD_{16}$ in base dieci, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.3.2.3 Esercizio

Si traduca il valore $101010{,}110011_2$ in base dieci, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:

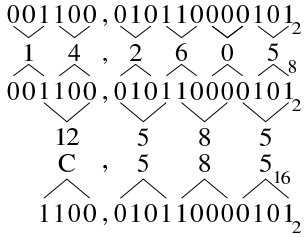


80.3.3 Conversione tra ottale, esadecimale e binario

Per quanto riguarda la conversione tra sistemi di numerazione ottale, esadecimale e binario, vale lo stesso principio dei numeri interi,

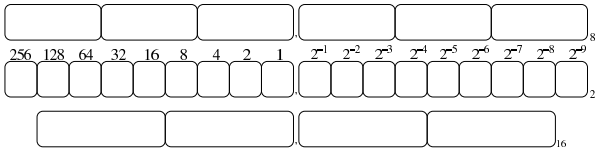
con la differenza che occorre rispettare la separazione della parte intera da quella decimale. L'esempio della figura successiva dovrebbe essere abbastanza chiaro.

Figura 80.55. Conversione tra binario-ottale e binario-esadecimale.



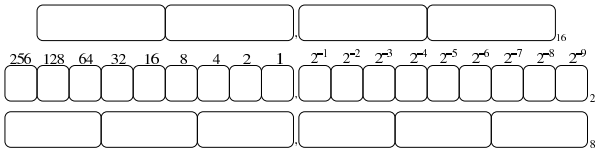
80.3.3.1 Esercizio

Si traduca il valore $76{,}55_8$ in base due e quindi in base sedici, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.3.3.2 Esercizio

Si traduca il valore $A7{,}C1_{16}$ in base due e quindi in base otto, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.4 Operazioni elementari e sistema di rappresentazione binaria dei valori

È importante conoscere alcuni concetti legati ai calcoli più semplici, applicati al sistema binario; soprattutto il modo in cui si utilizza il complemento a due. Infatti, la memoria di un elaboratore consente di annotare esclusivamente delle cifre binarie, in uno spazio di dimensione prestabilita e fissa; pertanto, attraverso il complemento a due si ha la possibilità di gestire in modo «semplice» la rappresentazione dei numeri interi negativi.

80.4.1 Complemento alla base di numerazione

Dato un numero $\boldsymbol{n}$, espresso in base $\boldsymbol{b}$, con $\boldsymbol{k}$ cifre, il ***complemento alla base*** è costituito da $\boldsymbol{b^k - n}$.

Per esempio, il complemento alla base del numero 00123456789 (espresso in base dieci utilizzando 11 cifre) è 99876543211:

$$\begin{array}{r} 100000000000_{10} - \\ 00123456789_{10} = \\ \hline 99876543211_{10} \end{array}$$

Dall'esempio si deve osservare che la quantità di cifre utilizzata è determinante nel calcolo del complemento, infatti, il complemento alla base dello stesso numero, usando però solo nove cifre (123456789) è invece 876543211:

$$\begin{array}{r} 1000000000_{10} - \\ 123456789_{10} = \\ \hline 876543211_{10} \end{array}$$

In modo analogo si procede con i valori aventi una base diversa; per esempio, il complemento alla base del numero binario 00110011_2, composto da otto cifre, è pari a 11001101_2:

$$\begin{array}{r} 100000000_2 - \\ 00110011_2 = \\ \hline 11001101_2 \end{array}$$

Il calcolo del complemento alla base, nel sistema binario, avviene in modo molto semplice, se si trasforma in questo modo:

$$\begin{array}{r} 11111111_2 - \\ 00110011_2 = \\ \hline 11001100_2 + \\ 1_2 = \\ \hline 11001101_2 \end{array}$$

In pratica, si prende un numero composto da una quantità di cifre a uno, pari alla stessa quantità di cifre del numero di partenza; quindi si esegue la sottrazione, poi si aggiunge il valore uno al risultato finale. Si osservi però cosa accade con una situazione leggermente differente, per il calcolo del complemento alla base di 0011001100_2:

$$\begin{array}{r} 1111111111_2 - \\ 0011001100_2 = \\ \hline 1100110011_2 + \\ 1_2 = \\ \hline 1100110100_2 \end{array}$$

Per eseguire una sottrazione, si può calcolare il complemento alla base del sottraendo (il valore da sottrarre), sommandolo poi al valore di partenza, trascurando il riporto eventuale. Per esempio, volendo sottrarre da 1757 il valore 758, si può calcolare il complemento alla base di 0758 (usando la stessa quantità di cifre dell'altro valore), per poi sommarla. Il complemento alla base di 0758 è 9242:

$$\begin{array}{r} 10000_{10} - \\ 0758_{10} = \\ \hline 9242_{10} \end{array}$$

Invece di eseguire la sottrazione, si somma il valore ottenuto a quello di partenza, ignorando il riporto:

$$\begin{array}{r} 1757_{10} + \\ 9242_{10} = \\ \hline 10999_{10} - \\ 10000_{10} = \\ \hline 999_{10} \end{array}$$

Infatti: 1757−758=999.

80.4.1.1 Esercizio

Si determini il complemento alla base del valore 0000123456_{10} (a dieci cifre), compilando lo schema successivo:

80.4.1.2 Esercizio

Si determini il complemento alla base del valore 9999123456_{10} (a dieci cifre), compilando lo schema successivo:

80.4.2 Complemento a uno e complemento a due

Quando si fa riferimento a numeri in base due, il complemento alla base è più noto come «complemento a due» (che evidentemente è la stessa cosa). D'altro canto, il complemento a uno è ciò che è già stato descritto con l'esempio seguente, dove si ottiene a partire dal numero 0011001100_2:

$$\begin{array}{r} 1111111111_2 - \\ 0011001100_2 = \\ \hline 1100110011_2 \end{array}$$

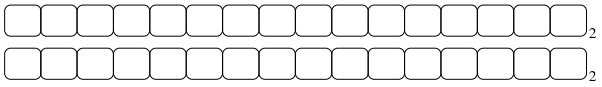
Si comprende intuitivamente che il complemento a uno si ottiene semplicemente invertendo le cifre binarie:

$$0011001100_2 \downarrow 1100110011_2$$

Pertanto, il complemento a due di un numero binario si ottiene facilmente invertendo le cifre del numero di partenza e aggiungendo una unità al risultato.

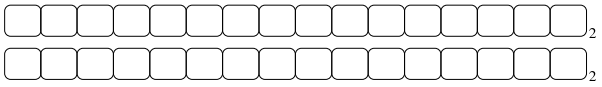
80.4.2.1 Esercizio

Si determini il complemento a uno e il complemento a due del valore 0011001001000101_2, compilando gli schemi successivi:



80.4.2.2 Esercizio

Si determini il complemento a uno e il complemento a due del valore 1111001100010101_2, compilando gli schemi successivi:



80.4.3 Addizione binaria

L'addizione binaria avviene in modo analogo a quella del sistema decimale, con la differenza che si utilizzano soltanto due cifre numeriche: 0 e 1. Pertanto, si possono presentare solo i casi seguenti:

$0_2 + 0_2 = 0_2$
$0_2 + 1_2 = 1_2$
$1_2 + 0_2 = 1_2$
$1_2 + 1_2 = 10_2$ ovvero: zero con riporto di uno

Segue l'esempio di una somma tra due numeri in base due:

$$\begin{array}{rr} 10011001_2 + & (153_{10}) \\ 00110011_2 = & (51_{10}) \\ \hline 11001100_2 & (204_{10}) \end{array}$$

80.4.4 Sottrazione binaria

La sottrazione binaria può essere eseguita nello stesso modo di quella che si utilizza nel sistema decimale. Come avviene nel sistema decimale, quando una cifra del minuendo (il numero di partenza) è minore della cifra corrispondente nel sottraendo (il numero da sottrarre), si prende a prestito una unità dalla cifra precedente (a sinistra), che così si somma al minuendo con il valore della base di numerazione. L'esempio seguente mostra una sottrazione con due numeri binari:

$$\begin{array}{rr} 10011001_2 - & (153_{10}) \\ 00110011_2 = & (51_{10}) \\ \hline 01100110_2 & (102_{10}) \end{array}$$

Generalmente, la sottrazione binaria viene eseguita sommando il complemento alla base del sottraendo. Il complemento alla base di 00110011_2 con otto cifre è 11001101_2:

$$\begin{array}{r} 100000000_2 - \\ 00110011_2 = \\ \hline 11001101_2 \end{array}$$

Pertanto, la sottrazione originale diventa una somma, dove si trascura il riporto:

$$\begin{array}{rr} 10011001_2 + & (153_{10}) \\ 11001101_2 = & \\ \hline 101100110_2 - & \\ 100000000_2 = & \\ \hline 01100110_2 & (102_{10}) \end{array}$$

80.4.5 Moltiplicazione binaria

La moltiplicazione binaria si esegue in modo analogo a quella per il sistema decimale, con il vantaggio che è sufficiente sommare il moltiplicando, facendolo scorrere verso sinistra, in base al valore del moltiplicatore. Naturalmente, lo spostamento di un valore binario verso sinistra di ***n*** posizioni, corrisponde a moltiplicarlo per 2^n. Si osservi l'esempio seguente dove si moltiplica 10011001_2 per 1011_2:

$$\begin{array}{rr} 10011001_2 \times & (153_{10}) \\ 1011_2 = & (11_{10}) \\ \hline 10011001_2 + & \\ 10011001_2 + & \\ 00000000_2 + & \\ 10011001_2 = & \\ \hline 11010010011_2 & (1683_{10}) \end{array}$$

80.4.6 Divisione binaria

La divisione binaria si esegue in modo analogo al procedimento per i valori in base dieci. Si osservi l'esempio seguente, dove si divide il numero 10110_2 (22_{10}) per 100_2 (4_{10}):

$$\begin{array}{l} 10110_2 : 100_2 = 101{,}1_2 \\ \hline 100_2 \\ \hline 0110_2 \\ 000_2 \\ \hline 110_2 \\ 100_2 \\ \hline 10_2 \\ 100_2 \\ \hline 0_2 \end{array}$$

In questo caso il risultato è 101_2 (5_{10}), con il resto di 10_2 (2_{10}); ovvero $101{,}1_2$ ($5{,}5_{10}$).

Intuitivamente si comprende che: si prende il divisore, senza zeri anteriori, lo si fa scorrere a sinistra in modo da trovarsi allineato inizialmente con il dividendo; se la sottrazione può avere luogo, si scrive la cifra 1_2 nel risultato; si continua con gli scorrimenti e le sottrazioni; al termine, il valore residuo è il resto della divisione intera.

80.4.7 Rappresentazione binaria di numeri interi senza segno

La rappresentazione di un valore intero senza segno coincide normalmente con il valore binario contenuto nella variabile gestita dall'elaboratore. 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})

80.4.8 Rappresentazione binaria di numeri interi con segno

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. Attraverso questo metodo, per cambiare di segno a un valore è sufficiente calcolarne il complemento a due.

Per esempio, se si prende un valore positivo rappresentato in otto cifre binarie come 00010100_2, pari a $+20_{10}$, il complemento a due è: 11101100_2, pari a -20_{10} secondo questa convenzione. Per trasformare il valore negativo nel valore positivo corrispondente, basta calcolare nuovamente il complemento a due: da 11101100_2 si ottiene ancora 00010100_2 che è il valore positivo originario.

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 80.80. Confronto tra due valori interi con segno.

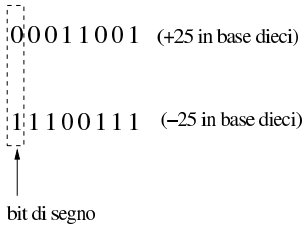
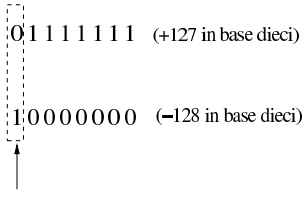


Figura 80.81. Valori massimi rappresentabili con soli otto bit.



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

80.4.8.1 Esercizio

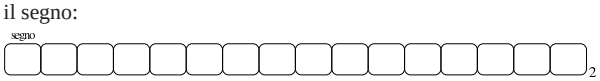
Come si rappresenta il numero $+103_{10}$, in una variabile binaria, a sedici cifre con segno?

80.4.8.2 Esercizio

Come si rappresenta il numero -103_{10}, in una variabile binaria, a sedici cifre con segno?

80.4.8.3 Esercizio

Data una variabile a sedici cifre che rappresenta un numero con segno, contenente il valore 1111111111110001_2, si calcoli il complemento a due e poi il valore corrispondente in base dieci, specificando il segno:



80.4.8.4 Esercizio

Data una variabile a sedici cifre che rappresenta un numero con segno, contenente il valore 0000000000110001_2, si calcoli il valore corrispondente in base dieci:

Si calcoli quindi il complemento a due:

Supponendo di interpretare il valore binario ottenuto dal complemento a due, come se si trattasse di un'informazione priva di segno, si calcoli nuovamente il valore corrispondente in base dieci:

80.4.8.5 Esercizio

Data una variabile a dodici cifre binarie che rappresenta un numero con segno, leggendo il suo contenuto come se fosse una variabile priva di segno, si potrebbe determinare quel segno originale in base al valore che si ottiene. Si scrivano gli intervalli che riguardano valori positivi e valori negativi:

Intervallo che rappresenta valori positivi	Intervallo che rappresenta valori negativi

80.4.8.6 Esercizio

Data una variabile a sedici cifre binarie che rappresenta un numero con segno, leggendo il suo contenuto come se fosse una variabile priva di segno, si potrebbe determinare quel segno originale in base al valore che si ottiene. Si scrivano gli intervalli che riguardano valori positivi e valori negativi:

Intervallo che rappresenta valori positivi	Intervallo che rappresenta valori negativi

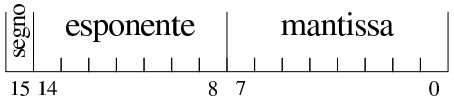
80.4.9 Cenni alla rappresentazione binaria di numeri in virgola mobile

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 molti 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.[1]

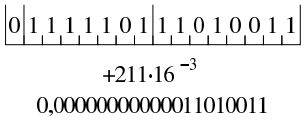
Figura 80.91. Ipotesi di una variabile a 16 bit per rappresentare dei numeri a virgola mobile.



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,***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.[2]

Figura 80.92. 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.



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.

80.5 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).

80.5.1 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 80.93. Aumento e riduzione delle cifre di un numero intero senza segno.

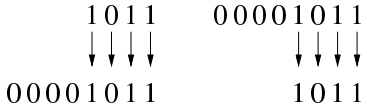


Figura 80.94. Aumento e riduzione delle cifre di un numero intero positivo.

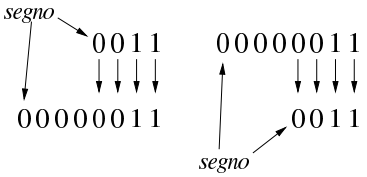
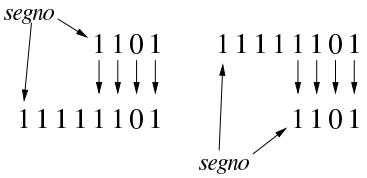


Figura 80.95. Aumento e riduzione delle cifre di un numero intero negativo.



80.5.1.1 Esercizio

Una variabile a otto cifre binarie, conviene un valore con segno, pari a 11100011_2. Questo valore viene copiato in una variabile a sedici cifre con segno. Indicare il valore che deve apparire nella variabile di destinazione.

80.5.1.2 Esercizio

Una variabile a sedici cifre binarie, contiene un valore con segno, pari a 0000111110001111_2. Questo valore viene copiato in una variabile a otto cifre con segno. Il risultato della copia è valido? Perché?

Il valore originale della variabile a sedici cifre con segno è pari a:

Il valore contenuto nella variabile a otto cifre con segno, potrebbe essere pari a:

80.5.1.3 Esercizio

Una variabile a otto cifre binarie, contiene un valore con segno, pari a 11100011_2. Questo valore viene copiato in una variabile a sedici cifre senza segno, ignorando il fatto che la variabile originale abbia un segno. Indicare il valore che appare nella variabile di destinazione dopo la copia.

Se successivamente si volesse considerare la variabile a sedici cifre usata per la destinazione della copia, come se fosse una variabile con segno, il valore che vi si potrebbe leggere al suo interno risulterebbe uguale a quello della variabile di origine?

80.5.2 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 80.100. Somma di due valori positivi che genera un risultato valido.

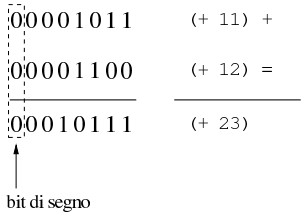


Figura 80.101. Somma di due valori positivi, dove il risultato apparentemente negativo indica la presenza di un traboccamento.

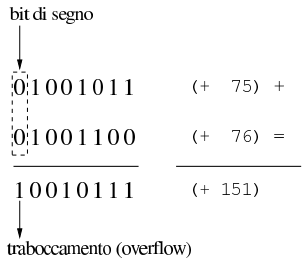


Figura 80.102. Somma di un valore positivo e di un valore negativo: il risultato è sempre valido.

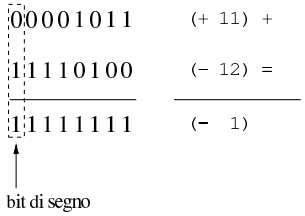


Figura 80.103. Somma di un valore positivo e di un valore negativo: in tal caso il risultato è sempre valido e se si manifesta un riporto, come in questo caso, va ignorato semplicemente.

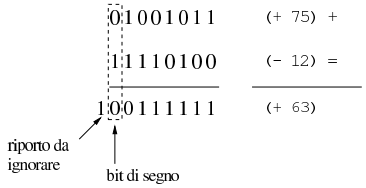


Figura 80.104. Somma di due valori negativi che produce un segno coerente e un riporto da ignorare.

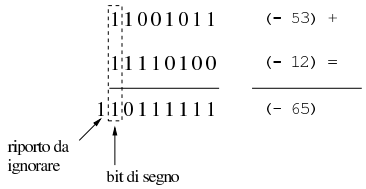
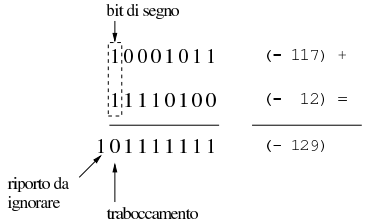


Figura 80.105. Somma di due valori negativi che genera un traboccamento, evidenziato da un risultato con un segno incoerente.



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.

80.5.2.1 Esercizio

Si esegua la somma tra due valori binari a otto cifre con segno, indicando anche il riporto eventuale: $01010101_2 + 01111110_2$.

Il risultato della somma è valido?

80.5.2.2 Esercizio

Si esegua la somma tra due valori binari a otto cifre con segno, indicando anche il riporto eventuale: $11010101_2 + 01111110_2$.

Il risultato della somma è valido?

80.5.2.3 Esercizio

Si esegua la somma tra due valori binari a otto cifre con segno, indicando anche il riporto eventuale: $11010101_2 + 10000001_2$.

Il risultato della somma è valido?

80.5.3 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 11111111_2. A questo punto si sommano algebricamente i due valori e si ottiene 00000000_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 11111111_2, senza riporto, ma questo valore che va inteso senza segno è evidentemente errato.

Figura 80.109. 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.

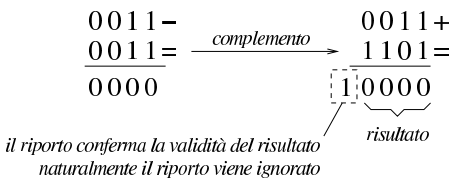
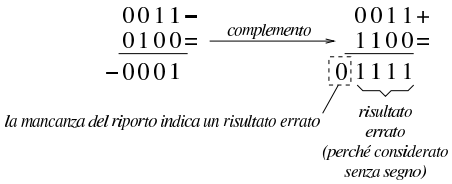
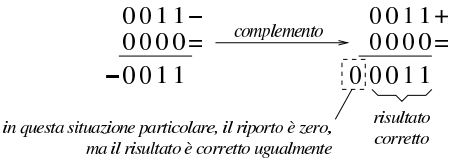


Figura 80.110. Sottrazione tra due numeri interi senza segno, dove il sottraendo ha un valore assoluto maggiore di quello del minuendo: l'assenza di un riporto indica un risultato errato della sottrazione.



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



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 80.112. 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.

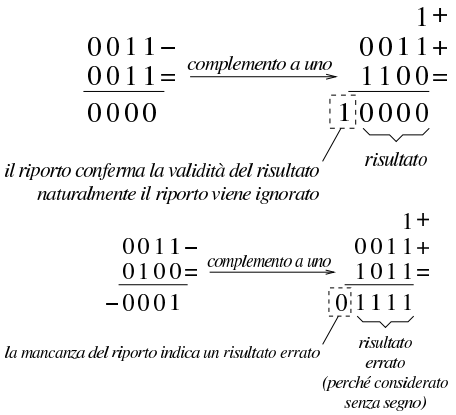
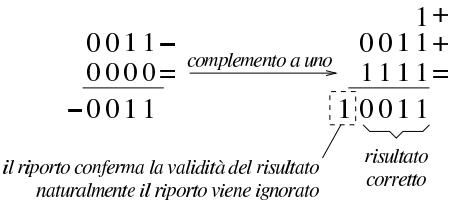


Figura 80.114. Sottrazione con sottraendo a zero: calcolando il complemento a due attraverso il complemento a uno, si ottiene un riporto coerente.



80.5.3.1 Esercizio

«

Si esegua la somma tra due valori binari a otto cifre senza segno, indicando anche il riporto eventuale: $11010101_2 + 01110110_2$.

Il risultato della somma è valido?

80.5.3.2 Esercizio

«

Si esegua la somma tra due valori binari a otto cifre senza segno, indicando anche il riporto eventuale: $11010101_2 + 11110110_2$.

Il risultato della somma è valido?

80.5.3.3 Esercizio

«

Si esegua la sottrazione tra due valori binari a otto cifre senza segno, indicando anche il riporto eventuale: $11010101_2 - 11110110_2$.

Il risultato della somma è valido?

80.5.3.4 Esercizio

«

Si esegua la sottrazione tra due valori binari a otto cifre senza segno, indicando anche il riporto eventuale: $11010101_2 - 00001111_2$.

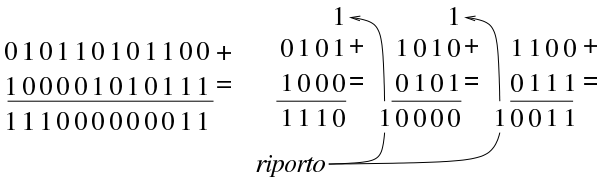
Il risultato della sottrazione è valido?

80.5.4 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 80.119. Somma per fasi successive, tenendo conto del 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 una unità al gruppo successivo.

Figura 80.120. Sottrazione per fasi successive, tenendo conto del prestito delle cifre.

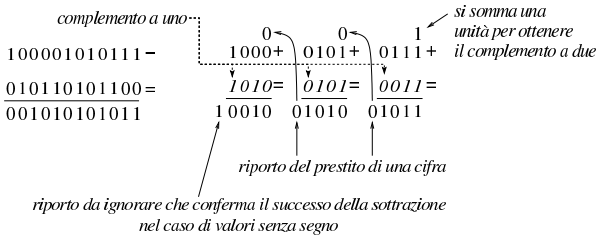
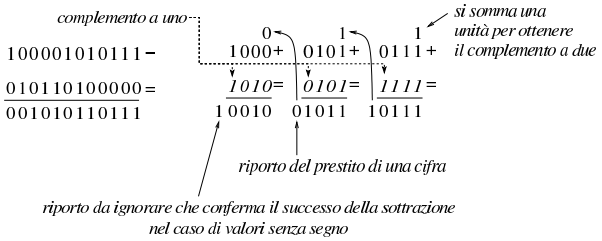


Figura 80.121. Verifica del procedimento anche in presenza di un sottraendo a zero.



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 essere 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.

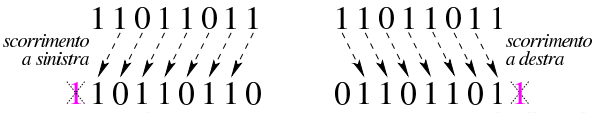
80.6 Scorrimenti, rotazioni, operazioni logiche

Le operazioni più semplici che si possono compiere con un microprocessore sono quelle che riguardano la logica booleana e lo scorrimento dei bit. Proprio per la loro semplicità è importante conoscere alcune applicazioni interessanti di questi procedimenti elaborativi.

80.6.1 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 80.122. 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.

80.6.1.1 Esercizio

Si esegua lo scorrimento logico a sinistra (di una sola cifra) del valore 11010101_2.

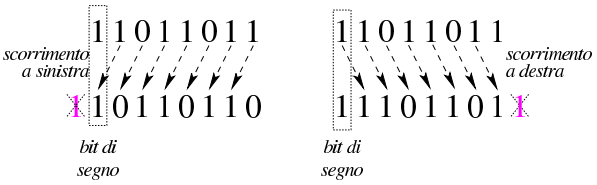
80.6.1.2 Esercizio

Si esegua lo scorrimento logico a destra (di una sola cifra) del valore 11010101_2.

80.6.2 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 segno, 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 80.125. Scorrimento aritmetico a sinistra e a destra, di un valore negativo.



80.6.2.1 Esercizio

Si esegua lo scorrimento aritmetico a sinistra (di una sola cifra) del valore con segno 01010101_2.

Il risultato dello scorrimento è valido?

80.6.2.2 Esercizio

Si esegua lo scorrimento aritmetico a destra (di una sola cifra) del valore con segno 01010101_2.

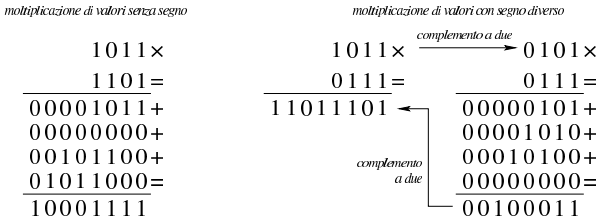
80.6.2.3 Esercizio

Si esegua lo scorrimento aritmetico a destra (di una sola cifra) del valore con segno 11010101_2.

80.6.3 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.

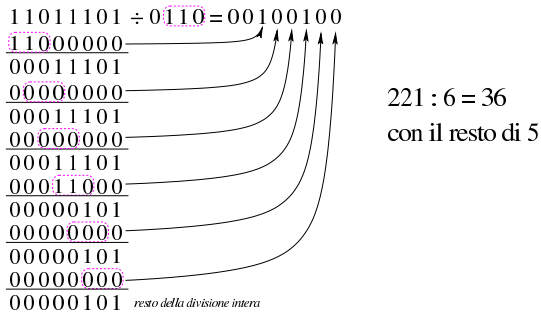
Figura 80.129. Moltiplicazione.



80.6.4 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 80.130. Divisione: i valori sono intesi senza segno.



80.6.5 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 80.131. Rotazione normale.

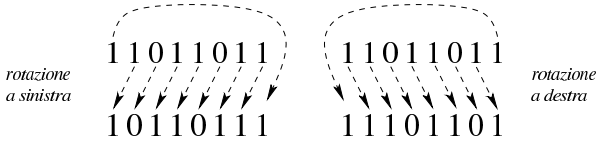
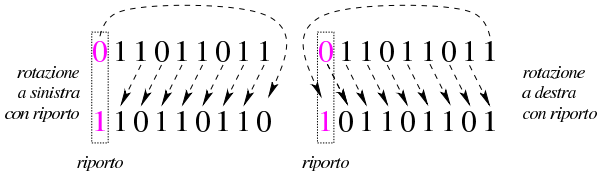


Figura 80.132. Rotazione con riporto.



80.6.6 Operatori logici

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

Figura 80.133. AND e OR.

11011011 *a*	11011011 *a*
01101101 *b*	01101101 *b*
01001001 *a AND b*	11111111 *a OR b*

Figura 80.134. XOR e NOT.

11011011 *a*	11011011 *a*
01101101 *b*	
10110110 *a XOR b*	00100100 *NOT a*

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 assemblatore 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 80.135. XOR per azzerare i valori.

11011011	*a*
11011011	*a*
00000000	*a XOR a*

80.6.6.1 Esercizio

Eseguire l'operazione seguente, considerando i valori privi di segno: 0010010101011111_2 AND 0110001111000011_2.

80.6.6.2 Esercizio

Eseguire l'operazione seguente, considerando i valori privi di segno: 0010010101011111_2 OR 0110001111000011_2.

80.6.6.3 Esercizio

Eseguire l'operazione seguente, considerando i valori privi di segno: 0010010101011111_2 XOR 0110001111000011_2.

80.6.6.4 Esercizio

Eseguire l'operazione seguente, considerando i valori privi di segno: NOT 0010010101011111_2.

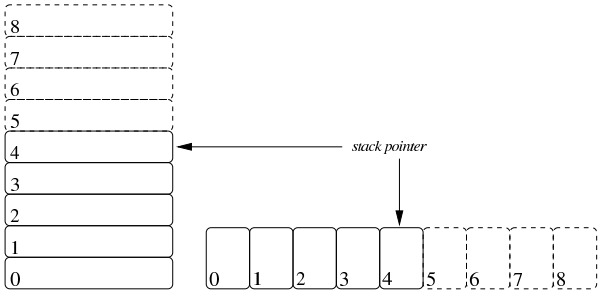
80.7 Organizzazione della memoria

Nello studio del linguaggio C è importante avere un'idea di come venga gestita la memoria di un elaboratore, molto vicina a ciò che si percepirebbe usando direttamente il linguaggio della CPU.

80.7.1 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 80.140. Esempio di 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.

80.7.2 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 descritto 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 che consenta di comprendere il meccanismo, si può osservare l'esempio seguente, schematizzato attraverso una pseudocodifica: la funzione '**SOMMA**' prevede l'uso di due parametri (ovvero due argomenti nella chiamata) e di una variabile «locale». Per chiamare la funzione, occorre mettere i valori dei parametri nella pila; successivamente, si dichiara la stessa variabile locale nella pila. Si consideri che il programma inizia e finisce nella funzione '**MAIN**', all'interno della quale si fa la chiamata della funzione '**SOMMA**':

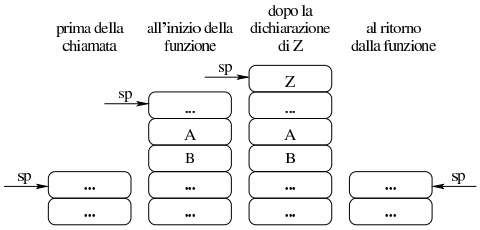
```
SOMMA (X, Y)
    LOCAL Z INTEGER
    Z := X + Y
    RETURN Z
END SOMMA
```



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

Nel disegno successivo, si schematizza ciò che accade nella pila (nel vettore che rappresenta la pila dei dati), dove si vede che inizialmente c'è una situazione indefinita, con l'indice «sp» (*stack pointer*) in una certa posizione. Quando viene eseguita la chiamata della funzione, automaticamente si incrementa la pila inserendo gli argomenti della chiamata (qui si mettono in ordine inverso, come si fa nel linguaggio C), mettendo in cima anche altre informazioni che nello schema non vengono chiarite (nel disegno appare un elemento con dei puntini di sospensione).

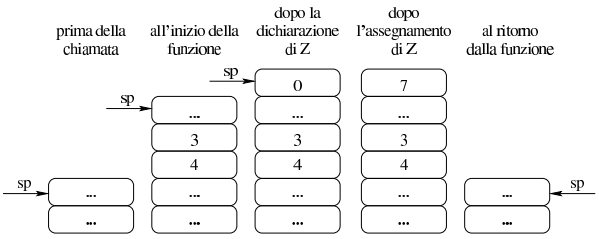
Figura 80.142. Situazione della pila nelle varie fasi della chiamata della funzione '**SOMMA**'.



La variabile locale «Z» viene allocata in cima alla pila, incrementando ulteriormente l'indice «sp». Al termine, la funzione trasmette in qualche modo il proprio risultato (tale modalità non viene chiarita qui e dipende dalle convenzioni di chiamata) e la pila viene riportata alla sua condizione iniziale.

Dal momento che l'esempio di programma contiene dei valori particolari, il disegno di ciò che succede alla pila dei dati può essere reso più preciso, mettendo ciò che contengono effettivamente le varie celle della pila.

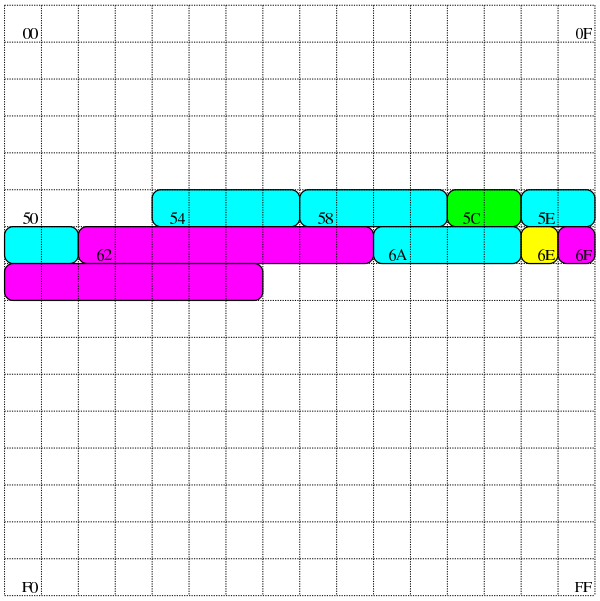
Figura 80.143. Situazione della pila nelle varie fasi della chiamata della funzione '**SOMMA**', osservando i contenuti delle varie celle.



80.7.3 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 nelle prime sezioni del capitolo. In questo modo, le variabili in memoria si raggiungono attraverso un indirizzo che individua il primo byte che le compone ed è compito del programma il sapere di quanti byte sono composte complessivamente.

Figura 80.144. 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}.

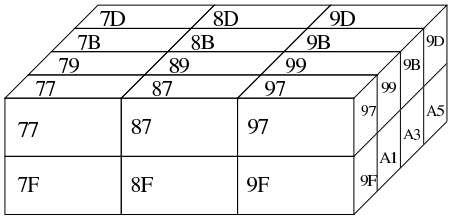


Nel disegno in cui si ipotizza una memoria complessiva di 256 byte, sono state evidenziate alcune aree di memoria:

Indirizzo	Dimensione	Indirizzo	Dimensione
54_{16}	4 byte	58_{16}	4 byte
$5C_{16}$	2 byte	$5E_{16}$	4 byte
62_{16}	8 byte	$6A_{16}$	4 byte
$6E_{16}$	1 byte	$6F_{16}$	8 byte

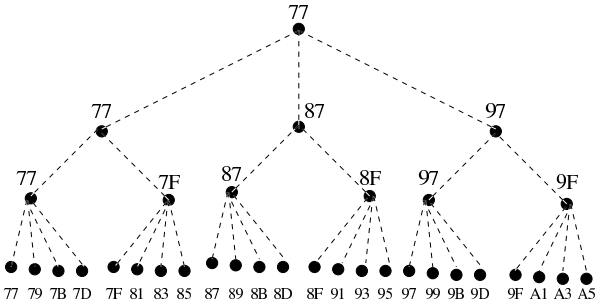
Con una gestione di questo tipo della memoria, la rappresentazione degli array richiede un po' di impegno da parte del programmatore. Nella figura successiva si rappresenta una matrice a tre dimensioni; per ora si ignorino i codici numerici associati alle celle visibili.

Figura 80.146. 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.



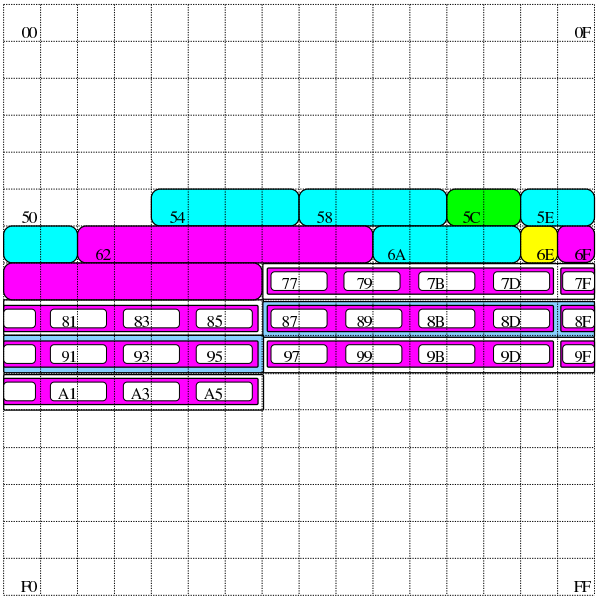
Dal momento che la rappresentazione tridimensionale rischia di creare confusione, quando si devono rappresentare matrici che hanno più di due dimensioni, è più conveniente pensare a strutture ad albero. Nella figura successiva viene tradotta in forma di albero la matrice rappresentata precedentemente.

Figura 80.147. 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 80.148. 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. Considerando 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 effet-

tivamente 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.

80.7.3.1 Esercizio

«

Una certa variabile occupa quattro unità di memoria, a partire dall'indirizzo $2F_{16}$. Qual è l'indirizzo dell'ultima unità di memoria occupata dalla variabile?

Indirizzo iniziale	Indirizzo dell'ultima unità di memoria della variabile
$2F_{16}$	

80.7.3.2 Esercizio

«

In memoria viene rappresentato un array di sette elementi da quattro unità di memoria ciascuno. Se l'indirizzo iniziale di questo array è 17_{16}, qual è l'indirizzo dell'ultima cella di memoria usata da questo array?

Indirizzo iniziale	Indirizzo dell'ultima unità di memoria dell'array
17_{16}	

80.7.3.3 Esercizio

«

In memoria viene rappresentato un array di elementi da quattro unità di memoria ciascuno. Se l'indirizzo iniziale di questo array è 17_{16}, qual è l'indirizzo iniziale del secondo elemento dell'array?

Indirizzo iniziale	Indirizzo del secondo elemento dell'array
17_{16}	

80.7.3.4 Esercizio

«

In memoria viene rappresentato un array di ***n*** elementi da quattro unità di memoria ciascuno. Se l'indirizzo iniziale di questo array è 17_{16}, a quale elemento punta l'indirizzo $2B_{16}$?

Indirizzo iniziale	Indirizzo dato	Elemento dell'array
17_{16}	$2B_{16}$	

80.7.3.5 Esercizio

«

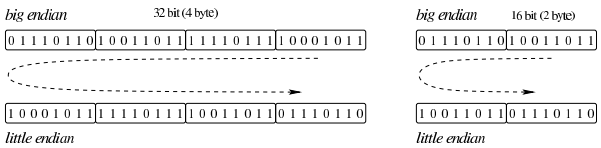
In memoria viene rappresentato un array di ***n*** elementi da quattro unità di memoria ciascuno. Se l'indirizzo iniziale di questo array è 17_{16}, l'indirizzo 22_{16} potrebbe puntare all'inizio di un certo elemento di questo?

80.7.4 Ordine dei byte

«

Come già descritto in questo capitolo, 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 80.156. 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.

80.7.4.1 Esercizio

«

In memoria viene rappresentata una variabile di 2 byte di lunghezza, a partire dall'indirizzo 21_{16}, contenente il valore 1111110011000000_2. Se la CPU accede alla memoria secondo la modalità *big endian*, che valore si legge all'indirizzo 21_{16} se si pretende di trovare una variabile da un solo byte?

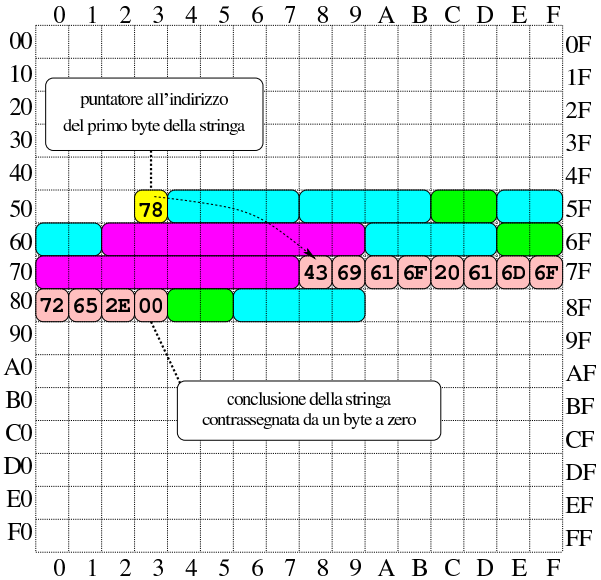
Cosa si legge, invece, se la CPU accede alla memoria secondo la modalità *little endian* (invertita)?

80.7.5 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 80.159. 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}.

80.7.5.1 Esercizio

In memoria viene rappresentata la stringa «Ciao a tutti». Sapendo che ogni carattere utilizza un solo byte e che la stringa è terminata regolarmente con il codice nullo di terminazione (00_{16}), quanti byte occupa la stringa in memoria?

80.7.5.2 Esercizio

In memoria viene rappresentata la stringa «Ciao a tutti» (come nell'esercizio precedente). Sapendo che la stringa inizia all'indirizzo $3F_{16}$, a quale indirizzo si trova la lettera «u» di «tutti»?

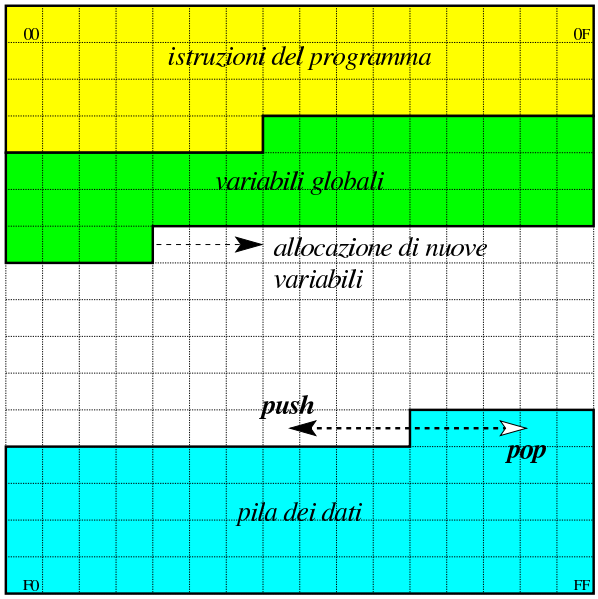
80.7.5.3 Esercizio

Se la memoria dell'elaboratore consente di raggiungere indirizzi da 0000_{16} a $FFFF_{16}$, quanto deve essere grande una variabile scalare che si utilizza come puntatore? Si indichi la quantità di cifre binarie.

80.7.6 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 80.160. Esempio di disposizione delle componenti di un programma in esecuzione in memoria, secondo il modello Unix.



80.8 Riferimenti

- 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

- Tony R. Kuphaldt, *Lessons In Electric Circuits, Digital*, http://www.faqs.org/docs/electric/ , http://www.faqs.org/docs/electric/Digital/index.html
- Wikipedia, *Sistema numerico binario*, http://it.wikipedia.org/wiki/Sistema_numerico_binario
- Wikipedia, *IEEE 754*, http://it.wikipedia.org/wiki/IEEE_754
- 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/

80.9 Soluzioni agli esercizi proposti

Esercizio	Soluzione
80.1.2.1	$11110011_2 = 243_{10}$.
80.1.2.2	$01100110_2 = 102_{10}$.
80.1.3.1	$1357_8 = 751_{10}$.
80.1.3.2	$7531_8 = 3929_{10}$.
80.1.4.1	$15AC_{16} = 5548_{10}$.
80.1.4.2	$CF58_{16} = 53080_{10}$.
80.2.1.1	$1234_{10} = 2322_8$.
80.2.1.2	$4321_{10} = 10341_8$.
80.2.2.1	$44221_{10} = ACBD_{16}$.
80.2.2.2	$12244_{10} = 2FD4_{16}$.
80.2.3.1	$1234_{10} = 10011010010_2$.
80.2.3.2	$4321_{10} = 1000011100001_2$.
80.2.4.1	$ABC_{16} = 101010111100_2 = 5274_8$.
80.2.4.2	$7655_8 = 111110101101_2 = FAD_{16}$.
80.3.1.1	$43{,}21_{10} = 53{,}15341_8$.
80.3.1.2	$765{,}4321_{10} = 2FD{,}6E9E1_{16}$.
80.3.1.3	$21{,}11_{10} = 10101{,}00011_2$.
80.3.2.1	$765{,}432_8 = 501{,}55078_{10}$.
80.3.2.2	$AB{,}CD_{16} = 171{,}80078_{10}$.
80.3.2.3	$101010{,}110011_2 = 42{,}79687_{10}$.
80.3.3.1	$76{,}55_8 = 00111110{,}10110100_2 = 3E{,}B4_{16}$.
80.3.3.2	$A7{,}C1_{16} = 010100111{,}110000010_2 = 247{,}602_8$.
80.4.1.1	complemento alla base di $0000123456_{10} = 9999876544_{10}$.
80.4.1.2	complemento alla base di $9999123456_{10} = 0000876544_{10}$.
80.4.2.1	complemento a uno di $0011001001000101_2 = 1100110110111010_2$; complemento a due = 1100110110111011_2.
80.4.2.2	complemento a uno di $1111001100010101_2 = 0000110011101010_2$; complemento a due = 0000110011101011_2.
80.4.8.1	$+103_{10} = 0000000001100111_2$.
80.4.8.2	$-103_{10} = 1111111110011001_2$.
80.4.8.3	$1111111111110001_2 = -15_{10}$; complemento a due = 0000000000001111_2.
80.4.8.4	$0000000000110001_2 = +49_{10}$; complemento a due = 1111111111001111_2; se 1111111111001111_2 fosse inteso senza segno sarebbe uguale a 65487_{10}.
80.4.8.5	da 0_{10} a 2047_{10} indica valori positivi; da 2048_{10} a 4095_{10} indica valori negativi.

Esercizio	Soluzione
80.4.8.6	da 0_{10} a 32767_{10} indica valori positivi; da 32768_{10} a 65535_{10} indica valori negativi.
80.5.1.1	11100011_2 con segno si traduce, a sedici cifre in 1111111111100011_2.
80.5.1.2	0000111110001111_2 con segno equivale a $+3983_{10}$, mentre 10001111_2 con segno equivale a -113_{10}; se poi si volesse supporre che la riduzione di cifre mantenga il segno, si avrebbe 00001111_2 che equivale a $+15_{10}$. Pertanto, in questo caso, la riduzione di cifre non può essere valida.
80.5.1.3	11100011_2 con segno equivale a -29_{10}; copiando questo valore in una variabile senza segno, a sedici cifre, si ottiene 0000000011100011_2, pari a 227_{10}. Se, successivamente, si interpreta il nuovo valore con segno, si ottiene $+227_{10}$, che non corrisponde in alcun modo al valore originale.
80.5.2.1	01010101_2 con segno + 01111110_2 con segno = 11010011_2 con riporto di zero. Il risultato non è valido perché, pur sommando due valori positivi, il segno è diventato negativo.
80.5.2.2	11010101_2 con segno + 01111110_2 con segno = 01010011_2 con riporto di uno. Il risultato della somma tra un numero positivo e un numero negativo è sempre valido.
80.5.2.3	11010101_2 con segno + 10000001_2 con segno = 01010110_2 con riporto di uno. Il risultato non è valido perché si sommano due numeri negativi, ma il risultato è positivo.
80.5.3.1	11010101_2 + 10000001_2 = 01001011_2 con riporto di uno. Il risultato non è valido perché c'è un riporto.
80.5.3.2	11010101_2 + 11110110_2 = 11001011_2 con riporto di uno. Il risultato non è valido perché c'è un riporto.
80.5.3.3	La sottrazione 11010101_2 − 11110110_2 va trasformata nella somma 11010101_2 + 00001010_2 = 11011111_2 senza riporto. Il risultato non è valido perché manca il riporto (d'altra parte si sta sottraendo un valore più grande del minuendo, pertanto il risultato senza segno non può essere valido
80.5.3.4	La sottrazione 11010101_2 − 00001111_2 va trasformata nella somma 11010101_2 + 11110001_2 = 11000110_2 con riporto di uno. Il risultato è valido perché si ha un riporto
80.6.1.1	Lo scorrimento logico a sinistra di 11010101_2, di una sola cifra, è pari a 10101010_2.
80.6.1.2	Lo scorrimento logico a destra di 11010101_2, di una sola cifra, è pari a 01101010_2.
80.6.2.1	Lo scorrimento aritmetico a sinistra di 01010101_2 (con segno), di una sola cifra, è pari a 10101010_2, ma si ottiene un cambiamento di segno e il risultato non è valido.
80.6.2.2	Lo scorrimento aritmetico a destra di 01010101_2 (con segno), di una sola cifra, è pari a 00101010_2. Il risultato è valido, in quanto è stato possibile preservare il segno e il valore ottenuto è pari alla divisione per due di quello originale.
80.6.2.3	Lo scorrimento aritmetico a destra di 11010101_2 (con segno), di una sola cifra, è pari a 11101010_2. Il risultato è valido, in quanto è stato possibile preservare il segno e il valore ottenuto è pari alla divisione per due di quello originale.
80.6.6.1	0010010101011111_2 AND 0110001111000011_2 = 0010000101000011_2.
80.6.6.2	0010010101011111_2 OR 0110001111000011_2 = 0110011111011111_2.
80.6.6.3	0010010101011111_2 XOR 0110001111000011_2 = 0100011010011100_2.
80.6.6.4	NOT 0010010101011111_2 = 1101101010100000_2.
80.7.3.1	L'ultima unità di memoria usata dalla variabile scalare si trova all'indirizzo 32_{16}.
80.7.3.2	L'array è lungo 28 unità di memoria e termina all'indirizzo 32_{16} incluso.
80.7.3.3	L'indirizzo del secondo elemento dell'array è $1B_{16}$.
80.7.3.4	L'indirizzo $2B_{16}$ punta al sesto elemento dell'array.
80.7.3.5	L'indirizzo 22_{16} individua una cella di memoria del terzo elemento dell'array, ma non trattandosi dell'inizio di tale elemento, non è utile come indice dello stesso.

Esercizio	Soluzione
80.7.4.1	In modalità *big endian*, la variabile che contiene 1111110011000000_2, se viene letta come se fosse costituita da un solo byte, darebbe 11111100_2, ovvero la porzione più significativa della stessa. Invece, in modalità *little endian*, ciò che si leggerebbe sarebbe la porzione meno significativa: 11000000_2.
80.7.5.1	La stringa «Ciao a tutti», terminata regolarmente, occupa 13 byte.
80.7.5.2	Sapendo che la stringa «Ciao a tutti» inizia all'indirizzo $3F_{16}$, la lettera «u» si trova all'indirizzo 47_{16}.
80.7.5.3	La variabile che consenta di rappresentare puntatori per indirizzi da 0000_{16} a $FFFF_{16}$, deve essere almeno da 16 bit (sedici cifre binarie).

[1] 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***.

[2] Si osservi che lo standard IEEE 754 utilizza una «mantissa normalizzata» che indica la frazione di valore tra uno e due: «1,***mantissa***.

Capitolo 81

Nozioni minime sul linguaggio C

81.1 Primo approccio al linguaggio C

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. In alternativa, disponendo solo di un sistema MS-Windows, potrebbe essere utile il pacchetto DevCPP che ha la caratteristica di essere molto semplice da installare.

81.1.1 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 libreria che viene inclusa più frequentemente è quella necessaria alla gestione dei flussi di standard input, standard output e standard error; si dichiara il suo utilizzo 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 ('**{ }**').[1]

```
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;
- in teoria i nomi potrebbero iniziare anche con il trattino basso, ma è sconsigliabile farlo, se non ci sono motivi validi per questo;[2]
- 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[])**'.

81.1.2 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.

Listato 81.3. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/vYaJyc7X* , *http://ideone.com/mxSUL* .

```
/*
 *      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");

    /* Attende la pressione di un tasto, quindi termina. */
    getchar ();
    return 0;
}
```

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**'.

81.1.2.1 Esercizio

Si modifichi l'esempio di programma mostrato, in modo da usare solo commenti del tipo '**//**'. Si può completare a penna il listato successivo

Listato 81.4. Per eseguire l'esercizio attraverso un servizio *pastebin*: *http://codepad.org/Pqit6Nna* , *http://ideone.com/0h1QC* .

```
        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");

        Attende la pressione di un tasto, quindi termina.

    getchar ();
    return 0;
}
```

81.1.2.2 Esercizio

Si modifichi l'esempio di programma mostrato, in modo da emettere il testo seguente, come si può vedere:

```
Il mio primo programma
scritto in linguaggio C.
```

Si completi per questo lo schema seguente.

Listato 81.6. Per eseguire l'esercizio attraverso un servizio *pastebin*: *http://codepad.org/5Rl7VtD7* , *http://ideone.com/WxWGX* .

```
#include <stdio.h>
int main (void)
{




    getchar ();
    return 0;
}
```

81.1.3 Compilazione

Per compilare un programma scritto in C, nell'ambito di un sistema operativo tradizionale, 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]
```

81.1.3.1 Esercizio

Quale comando si deve dare per compilare il file '`prova.c`' e ottenere il file eseguibile '`programma`'?

```
$                                                    [Invio]
```

81.1.4 Emissione dati attraverso «printf()»

L'esempio di programma presentato sopra si avvale della funzione ***printf()***[3] 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 '**%d**', '**%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 '**%d**':

```
    printf ("Totale fatturato: %d\n", 12345);
```

Lo specificatore di conversione '**%d**' 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
```

81.1.4.1 Esercizio

Si vuole visualizzare il testo seguente:

```
Imponibile: 1000, IVA: 200.
```

Sulla base delle conoscenze acquisite, si completi l'istruzione seguente:

```
printf ("                                ", 1000, 200);
```

81.2 Variabili e tipi del linguaggio C

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 costituito 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*).

81.2.1 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. 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.[4]

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.

81.2.1.1 Esercizio

«

Secondo la logica del linguaggio C, se un byte è formato da 8 bit, ci può essere una variabile scalare da 12 bit? Perché?

81.2.2 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 successiva.

Tabella 81.14. Elenco dei tipi comuni di dati primitivi elementari in C.

Tipo	Descrizione
`char`	Carattere (generalmente di 8 bit).
`int`	Intero normale.
`float`	Virgola mobile a precisione singola.
`double`	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**'[5] e '**unsigned**'.[6] I primi tre si riferiscono al rango, mentre gli altri modificano il modo di valutare il contenuto di alcune variabili. La tabella successiva riassume i vari tipi primitivi con le combinazioni ammissibili dei qualificatori.

Tabella 81.16. Elenco dei tipi comuni di dati primitivi in C assieme ai qualificatori usuali.

Tipo	Abbreviazione	Descrizione
`char`		Tipo '**char**' per il quale non conta sapere se il segno viene considerato o meno.
`signed char`		Tipo '**char**' usato numericamente con segno.
`unsigned char`		Tipo '**char**' usato numericamente senza segno.
`short int` `signed short int`	`short` `signed short`	Intero più breve di '**int**', con segno.
`unsigned short int`	`unsigned short`	Tipo '**short**' senza segno.
`int` `signed int`		Intero normale, con segno.
`unsigned int`	`unsigned`	Tipo '**int**' senza segno.
`long int` `signed long int`	`long` `signed long`	Intero più lungo di '**int**', con segno.
`unsigned long int`	`unsigned long`	Tipo '**long**' senza segno.
`long long int` `signed long long int`	`long long` `signed long long`	Intero più lungo di '**long int**', con segno.
`unsigned long long int`	`unsigned long long`	Tipo '**long long**' 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 '**double**'.

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.[7]

Listato 81.18. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/92vD92wUlM* , *http://ideone.com/q5unh* .

```
#include <stdio.h>

int main (void)
{
  printf ("char          %d\n", (int) sizeof (char));
  printf ("short int     %d\n", (int) sizeof (short int));
  printf ("int           %d\n", (int) sizeof (int));
  printf ("long int      %d\n", (int) sizeof (long int));
  printf ("long long int %d\n", (int) sizeof (long long int));
  printf ("float         %d\n", (int) sizeof (float));
  printf ("double        %d\n", (int) sizeof (double));
  printf ("long double   %d\n", (int) sizeof (long double));
  getchar ();
  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.[8]

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, 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. Ma 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.

81.2.2.1 Esercizio

Dovendo rappresentare numeri interi da 0 a 99999, può bastare una variabile scalare di tipo '**unsigned char**', sapendo che il tipo '**char**' utilizza 8 bit?

81.2.2.2 Esercizio

Qual è l'intervallo di valori che si possono rappresentare con una variabile di tipo '**unsigned char**', sapendo che il tipo '**char**' utilizza 8 bit?

Valore minimo	Valore massimo

81.2.2.3 Esercizio

Qual è l'intervallo di valori che si possono rappresentare con una variabile di tipo '**signed short int**', sapendo che il tipo '**short int**' utilizza 16 bit e che i valori negativi si esprimono attraverso il complemento a due?

Valore minimo	Valore massimo

81.2.2.4 Esercizio

Dovendo rappresentare il valore 12,34, è possibile usare una variabile di tipo '**int**'? Se non fosse possibile, quale tipo si potrebbe usare?

81.2.3 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: '**0*n***', dove ***n*** contiene esclusivamente cifre da zero a sette, viene inteso come un numero in base otto; '**0x*n***' o '**0X*n***', 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 81.22. 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 ('**unsigned**').
`L`	Intero più grande della dimensione normale ('**long**').
`LL`	Intero molto più grande della dimensione normale ('**long long**').
`UL`	Intero senza segno, più grande della dimensione normale ('**unsigned long**').
`ULL`	Intero senza segno, molto più grande della dimensione normale ('**unsigned long long**').

Tabella 81.23. Suffissi per le costanti che esprimono valori con virgola.

Suffisso	Descrizione
assente	Tipo '**double**'.
`F`	Tipo '**float**'.
`L`	Tipo '**long double**'.

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.

81.2.3.1 Esercizio

Indicare il valore, in base dieci, rappresentato dalle costanti che appaiono nella tabella successiva:

Costante	Valore corrispondente in base dieci
`12`	
`012`	
`0x12`	

81.2.3.2 Esercizio

Indicare i tipi delle costanti elencate nella tabella successiva:

Costante	Tipo corrispondente
`12`	
`12U`	
`12L`	
`1.2`	
`1.2L`	

81.2.4 Caratteri privi di rappresentazione grafica

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 '**\x*hh***', 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 successiva riporta i vari tipi di rappresentazione delle costanti carattere attraverso codici di escape.

Tabella 81.26. Elenco dei modi di rappresentazione delle costanti carattere attraverso codici di escape.

Codice	ASCII	Altra codifica
`\ooo`	Notazione ottale in base alla codifica.	idem
`\xhh`	Notazione esadecimale in base alla codifica.	idem
`\\`	Una singola barra obliqua inversa ('\').	idem
`\'`	Un apice singolo destro.	idem
`\"`	Un apice doppio.	idem
`\?`	Un punto interrogativo (per impedire che venga inteso come parte di una sequenza triplice, o *trigraph*).	idem
`\0`	Il codice <*NUL*>.	Il carattere nullo (con tutti i bit a zero).
`\a`	Il codice <*BEL*> (*bell*).	Il codice che, rappresentato sullo schermo o sulla stampante, produce un segnale acustico (*alert*).
`\b`	Il codice <*BS*> (*backspace*).	Il codice che fa arretrare il cursore di una posizione nella riga (*backspace*).
`\f`	Il codice <*FF*> (*form feed*).	Il codice che fa avanzare il cursore all'inizio della prossima pagina logica (*form feed*).
`\n`	Il codice <*LF*> (*line feed*).	Il codice che fa avanzare il cursore all'inizio della prossima riga logica (*new line*).
`\r`	Il codice <*CR*> (*carriage return*).	Il codice che porta il cursore all'inizio della riga attuale (*carriage return*).
`\t`	Una tabulazione orizzontale (<*HT*>).	Il codice che porta il cursore all'inizio della prossima tabulazione orizzontale (*horizontal tab*).
`\v`	Una tabulazione verticale (<*VT*>).	Il codice che porta il cursore all'inizio della prossima tabulazione verticale (*vertical tab*).

A parte i casi di '**\ooo**' e '**\x*hh***', 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.

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.

81.2.5 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 variabile '**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 81.28. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/5EZCPetn* , *http://ideone.com/KuRkv* .

```
#include <stdio.h>

int main (void)
{
    char c;
    for (c = '0'; c <= 'Z'; c++)
      {
        printf ("%c", c);
      }
    printf ("\n");
    getchar ();
    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:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

81.2.5.1 Esercizio

Indicare che valore si ottiene dalle espressioni elencate nella tabella successiva. Il primo caso appare risolto, come esempio:

Espressione	Costante carattere equivalente
`'3'+1`	`'4'`
`'3'-2`	
`'5'+4`	

81.2.6 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.

81.2.7 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 inizializzala contestualmente alla sua dichiarazione. L'esempio seguente dichiara la costante simbolica ***pi*** con il valore del π:

```
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 assemblatore, 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`**'.

81.2.7.1 Esercizio

Indicare le istruzioni di dichiarazione delle variabili descritte nella tabella successiva. I primi due casi appaiono risolti, come esempio:

Descrizione	Dichiarazione corrispondente
Variabile «a» in qualità di carattere senza segno.	`unsigned char x;`
Variabile «b» in qualità di carattere senza segno, inizializzata al valore 21.	`unsigned char x = 21;`
Variabile «d» in qualità di intero normale (con segno).	
Variabile «e» in qualità di intero più grande del solito, senza segno, inizializzata al valore 2111.	
Variabile «f» inizializzata al valore 21,11.	
Costante simbolica «g» inizializzata al valore 21,11.	

81.2.8 Il tipo indefinito: «void»

Lo standard del linguaggio C definisce un tipo particolare di valore, individuato dalla parola chiave '**`void`**'. Si tratta di un valore indefinito che a seconda del contesto può rappresentare il nulla o qualcosa da ignorare esplicitamente. A ogni modo, volendo ipotizzare una variabile di tipo '**`void`**', questa occuperebbe zero byte.

81.3 Operatori ed espressioni del linguaggio C

L'operatore è qualcosa che esegue un qualche tipo di funzione, su uno o più operandi, restituendo un valore.[9] 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 81.35. 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
`(a)` `[a]` `a->b` `a.b`	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.
`!a` `~a` `++a` `--a` `+a` `-a` `*a` `&a` `(tipo)` `sizeof a`	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.
`a*b` `a/b` `a%b`	Moltiplicazione, divisione e resto della divisione intera.
`a+b` `a-b`	Somma e sottrazione.
`a<<b` `a>>b`	Scorrimento binario.
`a<b` `a<=b` `a>b` `a=>b`	Confronto.

Operatori	Annotazioni
`a==b  a!=b`	Confronto.
`a&b`	AND bit per bit.
`a^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^=a  b|=a`
`b<<=a  b>>=a`
	Operatori di assegnamento.

`a, b`	Sequenza di espressioni (espressione multipla).

81.3.1 Tipo del risultato di un'espressione

Un'espressione è un qualche cosa composto da operandi e da operatori, che nel complesso si traduce in un qualche risultato. Per esempio, '`5+6`' è un'espressione aritmetica che si traduce nel numero 11. Così come le variabili, le costanti simboliche e le costanti letterali, hanno un tipo, con il quale si definisce in che modo vengono rappresentate in memoria, anche il risultato delle espressioni ha un tipo, in quanto tale risultato deve poi essere rappresentabile in memoria in qualche modo.

La regola che definisce di che tipo è il risultato di un'espressione è piuttosto articolata, ma in generale è sufficiente rendersi conto che si tratta della scelta più logica in base al contesto. Per esempio, l'espressione già vista, '`5+6`', essendo la somma di due interi con segno, dovrebbe dare come risultato un intero con segno. Nello stesso modo, un'espressione del tipo '`5.1-6.3`', essendo costituita da operandi in virgola mobile (precisamente '`double`'), dà il risultato -1,2, rappresentato sempre in virgola mobile (sempre '`double`'). Va osservato che la regola di principio vale anche per le divisioni, per cui '`11/2`' dà 5, di tipo intero ('`int`'), perché per avere un risultato in virgola mobile occorrerebbe invece scrivere '`11.0/2.0`'.

Si osservi che se in un'espressione si mescolano operandi interi assieme a operandi in virgola mobile, il risultato dell'espressione dovrebbe essere di tipo a virgola mobile. Per esempio, '`5+6.3`' dà il valore 11,3, in virgola mobile ('`double`'). Inoltre, se gli operandi hanno tra loro un rango differente, dovrebbe prevalere il rango maggiore.

81.3.1.1 Esercizio

Indicare il tipo che si dovrebbe ottenere dalla valutazione delle espressioni proposte. Il primo caso appare risolto, come esempio:

Espressione	Tipo che dovrebbe avere il risultato dell'espressione
`4+3.5`	`double`
`4+4`	
`4/3`	
`4.0/3`	
`4L*3`	

81.3.2 Operatori aritmetici

Gli operatori che intervengono su valori numerici sono elencati nella tabella successiva. Per dare un significato alle descrizioni della tabella, occorre tenere presenta una caratteristica importante del linguaggio, per la quale, la maggior parte delle espressioni restituisce un valore. Per esempio, '`b = a = 1`' fa sì che la variabile ***a*** ottenga il valore 1 e che, successivamente, la variabile ***b*** ottenga il valore di ***a***. In questo senso, al problema dell'ordine di precedenza dei vari operatori si aggiunge anche l'ordine in cui le espressioni restituiscono un valore. 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 81.37. Elenco degli operatori aritmetici e di quelli di assegnamento relativi a valori numerici.

Operatore e operandi	Descrizione
`++op`	Incrementa di un'unità l'operando prima che venga restituito il suo valore.
`op++`	Incrementa di un'unità l'operando dopo averne restituito il suo valore.
`--op`	Decrementa di un'unità l'operando prima che venga restituito il suo valore.
`op--`	Decrementa di un'unità l'operando dopo averne restituito il suo valore.
`+op`	Non ha alcun effetto.
`-op`	Inverte il segno dell'operando (prima di restituirne il valore).
`op1 + op2`	Somma i due operandi.
`op1 - op2`	Sottrae dal primo il secondo operando.
`op1 * op2`	Moltiplica i due operandi.
`op1 / op2`	Divide il primo operando per il secondo.
`op1 % op2`	Calcola il resto della divisione tra il primo e il secondo operando, i quali devono essere costituiti da valori interi.
`var = valore`	Assegna alla variabile il valore alla destra.
`op1 += op2`	`op1 = (op1 + op2)`
`op1 -= op2`	`op1 = (op1 - op2)`
`op1 *= op2`	`op1 = (op1 * op2)`
`op1 /= op2`	`op1 = (op1 / op2)`
`op1 %= op2`	`op1 = (op1 % op2)`

81.3.2.1 Esercizio

Osservando i pezzi di codice indicati, si scriva il valore contenuto nelle variabili a cui si assegna un valore, attraverso l'elaborazione di un'espressione. Il primo caso appare risolto, come esempio:

Codice	Valore contenuto nelle variabili dopo l'esecuzione del codice mostrato

`int a = 3;`
`int b;`
`b = a++;`
	a contiene 4;
	b contiene 3.

`int a = 3;`
`int b;`
`b = --a;`	

`int a = 3;`
`int b = 2;`
`b = a + b;`	

Codice	Valore contenuto nelle variabili dopo l'esecuzione del codice mostrato

`int a = 7;`
`int b = 2;`
`b = a % b;`	

`int a = 7;`
`int b;`
`b = (a = a * 2);`	

`int a = 3;`
`int b = 2;`
`b += a;`	

`int a = 7;`
`int b = 2;`
`b %= a;`	

`int a = 7;`
`int b;`
`b = (a *= 2);`	

81.3.2.2 Esercizio

Scrivere diversi programmi per verificare l'esercizio precedente. Viene proposto il primo, da usare come modello per gli altri.

Listato 81.39. Per svolgere l'esercitazione attraverso un servizio *pastebin*: *http://codepad.org/ZHmO0ycC* , *http://ideone.com/Rv6o1* .

```
#include <stdio.h>
int main (void)
{
    int a = 3;
    int b;
    b = a++;
    printf ("a contiene %d;\n", a);
    printf ("b contiene %d.\n", b);
    getchar ();
    return 0;
}
```

81.3.3 Operatori di confronto

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 successiva.

Il linguaggio C non ha una rappresentazione specifica per i valori booleani *Vero* e *Falso*,[10] 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 81.40. Elenco degli operatori di confronto. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
op1 `==` ***op2***	1 (*Vero*) se gli operandi si equivalgono.
op1 `!=` ***op2***	1 (*Vero*) se gli operandi sono differenti.
op1 `<` ***op2***	1 (*Vero*) se il primo operando è minore del secondo.
op1 `>` ***op2***	1 (*Vero*) se il primo operando è maggiore del secondo.
op1 `<=` ***op2***	1 (*Vero*) se il primo operando è minore o uguale al secondo.

Operatore e operandi	Descrizione
op1 `>=` ***op2***	1 (*Vero*) se il primo operando è maggiore o uguale al secondo.

81.3.3.1 Esercizio

Osservando i pezzi di codice indicati, si scriva il valore contenuto nella variabile ***c***. Il primo caso appare risolto, come esempio:

Codice	Valore contenuto nella variabile ***c*** l'esecuzione del codice mostrato

`int a = 5;`
`signed char b = -4;`
`int c = a > b;`
	c contiene 1.

`int a = 4 + 1;`
`signed char b = 5;`
`int c = a == b;`	

`int a = 4 + 1;`
`signed char b = 5;`
`int c = a != b;`	

`unsigned int a = 4 + 3;`
`signed char b = -5;`
`int c = a >= b;`	

`unsigned int a = 4 + 3;`
`signed char b = -5;`
`int c = a <= b;`	

81.3.3.2 Esercizio

Scrivere diversi programmi per verificare l'esercizio precedente. Viene proposto il primo, da usare come modello per gli altri.

Listato 81.42. Per svolgere l'esercitazione attraverso un servizio *pastebin*: *http://codepad.org/a3E5IGIT* , *http://ideone.com/0zob2* .

```
#include <stdio.h>
int main (void)
{
    int a = 5;
    signed char b = -4;
    int c = a > b;
    printf ("(%d > %d) produce %d\n", a, b, c);
    getchar ();
    return 0;
}
```

81.3.4 Operatori logici

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 successiva.

Tabella 81.43. Elenco degli operatori logici. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
`!` ***op***	Inverte il risultato logico dell'operando.
op1 `&&` ***op2***	Se il risultato del primo operando è *Falso* non valuta il secondo.
op1 `||` ***op2***	Se il risultato del primo operando è *Vero* 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.

81.3.4.1 Esercizio

Osservando i pezzi di codice indicati, si scriva il valore contenuto nella variabile ***c***. Il primo caso appare risolto, come esempio:

Codice	Valore contenuto nella variabile ***c*** dopo l'esecuzione del codice mostrato

`int a = 5;`
`signed char b = -4;`
`int c = ! (a > b);`
	c contiene 0.

`int a = 4;`
`signed char b = (3 < 5);`
`int c = a && b;`	

`int a = 4;`
`signed char b = (3 < 5);`
`int c = a || b;`	

`unsigned int a = 4 + 3;`
`signed char b = -5;`
`int c = (b > 0) || (a > b);`	

81.3.5 Operatori binari

Il linguaggio C consente di eseguire alcune operazioni binarie, sui **valori interi**, come spesso è possibile fare con un linguaggio assemblatore, anche se non è possibile interrogare degli indicatori (*flag*) che informino sull'esito delle azioni eseguite. Sono disponibili le operazioni elencate nella tabella successiva.

Tabella 81.45. 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 `^` ***op2***	XOR bit per bit (OR esclusivo).
op1 `<<` ***op2***	Scorrimento a sinistra di ***op2*** bit. A destra vengono aggiunti bit a zero
op1 `>>` ***op2***	Scorrimento a destra di ***op2*** bit. Il valore dei bit aggiunti a sinistra potrebbe tenere conto del segno.
`~`***op1***	Complemento a uno.
op1 `&=` ***op2***	***op1*** `=` `(`***op1*** `&` ***op2***`)`
op1 `|=` ***op2***	***op1*** `=` `(`***op1*** `|` ***op2***`)`
op1 `^=` ***op2***	***op1*** `=` `(`***op1*** `^` ***op2***`)`
op1 `<<=` ***op2***	***op1*** `=` `(`***op1*** `<<` ***op2***`)`
op1 `>>=` ***op2***	***op1*** `=` `(`***op1*** `>>` ***op2***`)`
op1 `~=` ***op2***	***op1*** `=` `~`***op2***

A seconda del compilatore e della piattaforma, lo scorrimento a destra potrebbe essere di tipo aritmetico, ovvero potrebbe tenere conto del segno. Pertanto, non potendo fare sempre affidamento su questa ipotesi, è prudente far sì che i valori di cui si fa lo scorrimento a destra siano sempre senza segno, o comunque positivi.

Per aiutare a comprendere il meccanismo 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`

00101010_2 (42_{10}) AND
00110011_2 (51_{10}) =
00100010_2 (34_{10})

`c = a | b`

00101010_2 (42_{10}) OR
00110011_2 (51_{10}) =
00111011_2 (59_{10})

`c = a ^ b`

00101010_2 (42_{10}) XOR
00110011_2 (51_{10}) =
00011001_2 (25_{10})

Lo scorrimento, invece, viene mostrato sempre solo per una singola unità: ***a*** contenente sempre il valore 42; ***b*** contenente il valore 1.

`c = a << b`

00101010_2 (42_{10}) <<
00000001_2 (1_{10}) =
01010100_2 (84_{10})

`c = a >> b`

00101010_2 (42_{10}) >>
00000001_2 (1_{10}) =
00010101_2 (21_{10})

`c = ~a`

00101010_2 (42_{10})
11010101_2 (213_{10})

81.3.5.1 Esercizio

Osservando i pezzi di codice indicati, si scriva il valore contenuto nella variabile ***c***. L'architettura a cui ci si riferisce prevede l'uso del complemento a due per la rappresentazione dei numeri negativi e lo scorrimento a destra è di tipo aritmetico (in quanto preserva il segno). I primi casi appaiono risolti, come esempio:

Codice	Valore contenuto nella variabile ***c*** dopo l'esecuzione del codice mostrato
`int a = -20;` `int c = a >> 1;`	***c*** contiene -10.

`int a = 5;`
`int b = 12;`
`int c = a & b;`	

`int a = 5;`
`int b = 12;`
`int c = a | b;`	

`int a = 5;`
`int b = 12;`
`int c = a ^ b;`	

`int a = 5;` `int c = a << 1;`	
`int a = 21;` `int c = a >> 1;`	

81.3.5.2 Esercizio

Scrivere diversi programmi per verificare l'esercizio precedente. Viene proposto un esempio, riferito a un caso che non appare nell'esercizio precedente, con cui si ottiene il complemento a uno.

Listato 81.49. Per svolgere l'esercitazione attraverso un servizio *pastebin*: *http://codepad.org/CqxVMIHG* , *http://ideone.com/iIEL0* .

```
#include <stdio.h>
int main (void)
{
    int a = 21;
    int c = ~a;
    printf ("c contiene %d\n", c);
    getchar ();
    return 0;
}
```

81.3.6 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`**'.

81.3.6.1 Esercizio

Indicare il tipo che si dovrebbe ottenere dalla valutazione delle espressioni proposte e il risultato effettivo. Il primo caso appare risolto, come esempio:

Espressione	Tipo che dovrebbe avere il risultato dell'espressione
`4+((double) 3)`	`(double) 7`
`(int) (4.4+4.9)`	
`(double) 4/3`	
`((double) 4)/3`	
`4 * ((long int) 3)`	

81.3.7 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, 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*** che ***y*** contengono alla fine il numero 10, perché le precedenze sono tali che è come se fosse scritto: '**`y = (x = 10)`**'.

81.3.7.1 Esercizio

Osservando i pezzi di codice indicati, si scriva il valore contenuto nella variabile ***c***. Il primo caso appare risolto, come esempio:

Codice	Valore contenuto nella variabile ***c*** dopo l'esecuzione del codice mostrato

`int a = -20;`
`int b = 10;`
`int c = (a *= 2, b += 10, c = a + b);`
	c contiene -20.

`int a = -20;`
`int b = 10;`
`int c = (b = 2, b = a, c = a + b);`	

`int a = -20;`
`int b = 10;`
`int c = (b = 2, b = a++, c = a + b);`	

`int a = -20;`
`int b = 10;`
`int c = (b = 2, b = ++a, c = a + b);`	

81.3.7.2 Esercizio

Scrivere diversi programmi per verificare l'esercizio precedente. Viene proposto un esempio, riferito al caso iniziale risolto.

Listato 81.56. Per svolgere l'esercitazione attraverso un servizio *pastebin*: *http://codepad.org/v4Aj19Ae19* , *http://ideone.com/ZTA1L* .

```
#include <stdio.h>
int main (void)
{
    int a = -20;
    int b = 10;
    int c = (a *= 2, b += 10, c = a + b);
    printf ("c contiene %d\n", c);
    getchar ();
    return 0;
}
```

81.4 Strutture di controllo di flusso del linguaggio C

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*.

81.4.1 Struttura condizionale: «if»

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 completi, dove è possibile variare il valore assegnato inizialmente alla variabile ***importo*** per verificare il comportamento delle istruzioni.

Listato 81.57. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/BbrdEx7f* , *http://ideone.com/qZ30j* .

```
#include <stdio.h>
int main (void)
{
    int importo;
    importo = 10001;
    if (importo > 10000) printf ("L'offerta è vantaggiosa\n");
    getchar ();
    return 0;
}
```

Listato 81.58. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/5OQZsFk1* , *http://ideone.com/9s9DH* .

```
#include <stdio.h>
int main (void)
{
    int importo;
    int memorizza;
    importo = 10001;
    if (importo > 10000)
      {
        memorizza = importo;
        printf ("L'offerta è vantaggiosa\n");
      }
    else
      {
        printf ("Lascia perdere\n");
      }
    getchar ();
    return 0;
}
```

L'esempio successivo, in particolare, mostra un modo grazioso per allineare le sottocondizioni, senza eccedere negli annidamenti.

Listato 81.59. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/99OA99Zbff* , *http://ideone.com/aQKgZ* .

```
#include <stdio.h>
int main (void)
{
    int importo;
    int memorizza;
    importo = 10001;
    if (importo > 10000)
      {
        memorizza = importo;
        printf ("L'offerta è vantaggiosa\n");
      }
    else if (importo > 5000)
      {
        memorizza = importo;
        printf ("L'offerta è accettabile\n");
      }
    else
      {
        printf ("Lascia perdere\n");
      }
    getchar ();
    return 0;
}
```

81.4.1.1 Esercizio

Partendo dalla struttura successiva, si scriva un programma che, in base al valore della variabile ***x***, mostri dei messaggi differenti: se ***x*** è inferiore a 1000 oppure è maggiore di 10000, si viene avvisati che il valore non è valido; se invece ***x*** è valido, se questo è maggiore di 5000, si viene avvisati che «il livello è alto», se invece fosse inferiore si viene avvisati che «il livello è basso»; infine, se il valore è pari a 5000, si viene avvisati che il livello è ottimale.

Listato 81.60. Per svolgere l'esercitazione si può usare eventualmente un servizio *pastebin*: *http://codepad.org/0vfX5Un9* , *http://ideone.com/gVhow* .

```
#include <stdio.h>
int main (void)
{
    int x;
    x = 5000;

    if ((x < 1000) || (x > 10000))
      {
        printf ("Il valore di x non è valido!\n");
      }
    else if …
      {
        …
        …
        …
      }
    getchar ();
    return 0;
}
```

81.4.1.2 Esercizio

Si osservi il programma successivo è si indichi cosa viene visualizzato alla sua esecuzione, spiegando il perché.

```
#include <stdio.h>
int main (void)
{
    int x;
    x = -1;

    if (x)
      {
        printf ("Sono felice :-)\n");
      }
    else
      {
        printf ("Sono triste :-(\n");
      }
    getchar ();
    return 0;
}
```

81.4.2 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 interna alla 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.

Listato 81.62. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/UOMoRmPm* , *http://ideone.com/Z9PqE* .

```
#include <stdio.h>
int main (void)
{
    int mese;
    mese = 11;

    switch (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;
      }
    getchar ();
    return 0;
}
```

Come si vede, dopo l'istruzione con cui si emette il nome del mese attraverso lo standard output, viene richiesto di uscire dalla struttura, attraverso l'istruzione '**`break`**', perché altrimenti si passerebbe all'esecuzione delle istruzioni del caso successivo, se presente. Sulla base di questo principio, un gruppo di casi può essere raggruppato assieme, quando si vuole che ognuno di questi esegua lo stesso insieme di istruzioni.

Listato 81.63. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/p3uFTLyn* , *http://ideone.com/glcnI* .

```
#include <stdio.h>
int main (void)
{
    int anno;
    int mese;
    int giorni;
    anno = 2013;
    mese = 2;

    switch (mese)
      {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            giorni = 31;
            break;
        case 4:
        case 6:
        case 9:
        case 11:
            giorni = 30;
            break;
        case 2:
            if (((anno % 4 == 0) && !(anno % 100 == 0))
                || (anno % 400 == 0))
              {
                giorni = 29;
              }
            else
              {
                giorni = 28;
              }
            break;
      }
    printf ("Il mese %d dell'anno %d ha %d giorni.\n",
            mese, anno, giorni);
    getchar ();
    return 0;
}
```


È anche possibile dichiarare un caso predefinito che si verifichi quando nessuno degli altri si avvera.

Listato 81.64. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/8TIUpduT* , *http://ideone.com/3YhHc* .

```
#include <stdio.h>
int main (void)
{
    int mese;
    mese = 13;

    switch (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;
        default: printf ("mese non corretto\n"); break;
      }
    getchar ();
    return 0;
}
```

81.4.2.1 Esercizio

In un esempio già mostrato, appare la porzione di codice seguente. Si spieghi nel dettaglio come viene calcolata la quantità di giorni di febbraio:

```
        case 2:
            if (((anno % 4 == 0) && !(anno % 100 == 0))
                || (anno % 400 == 0))
              {
                giorni = 29;
              }
            else
              {
                giorni = 28;
              }
            break;
```

81.4.3 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».

Listato 81.66. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/ZKrSA3IF* , *http://ideone.com/68bD684* .

```
#include <stdio.h>
int main (void)
{
    int i = 0;

    while (i < 10)
      {
```

```
        i++;
        printf ("x");
      }
    printf ("\n");
    getchar ();
    return 0;
}
```

Ma si osservi anche la variante seguente, con cui si ottiene un codice più semplice in linguaggio macchina:

Listato 81.67. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/RVF64ri64O* , *http://ideone.com/EFVta* .

```
#include <stdio.h>
int main (void)
{
    int i = 10;

    while (i)
      {
        i--;
        printf ("x");
      }
    printf ("\n");
    getchar ();
    return 0;
}
```

Nel blocco di istruzioni di un ciclo '**while**', ne possono apparire alcune particolari, che rappresentano dei salti incondizionati nell'ambito del ciclo:

- '**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.

Listato 81.68. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/Eyewc3QS* , *http://ideone.com/M0Mwz* .

```
#include <stdio.h>
int main (void)
{
    int i = 0;

    while (1)
      {
        if (i >= 10)
          {
            break;
          }
        i++;
        printf ("x");
      }
    printf ("\n");
    getchar ();
    return 0;
}
```

81.4.3.1 Esercizio

Sulla base delle conoscenze acquisite, si scriva un programma che calcola il fattoriale di un numero senza segno, contenuto nella variabile ***x***. Il fattoriale di ***x*** si ottiene con una serie di moltiplicazioni successive: $x \cdot (x-1) \cdot (x-2) \cdot \ldots \cdot 1$.

81.4.3.2 Esercizio

Sulla base delle conoscenze acquisite, si scriva un programma che verifica se un numero senza segno, contenuto nella variabile ***x***, è un numero primo.

81.4.4 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 in precedenza per visualizzare una sequenza di dieci «x», 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.

81.4.4.1 Esercizio

Modificare il programma che verifica se un numero è primo, usando un ciclo '**do**…**while**'.

81.4.5 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**'.

Listato 81.70. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/4Rw1BygV* , *http://ideone.com/wckol* .

```
#include <stdio.h>
int main (void)
{
    int i;

    for (i = 0; i < 10; i++)
      {
        printf ("x");
      }
    printf ("\n");
    getchar ();
    return 0;
}
```

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.

Listato 81.71. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/Tz85p3aO* , *http://ideone.com/Nqq5d* .

```
#include <stdio.h>
int main (void)
{
    int i;

    for (i = 0; i < 10; printf ("x"), i++)
      {
        ;
      }
    printf ("\n");
    getchar ();
    return 0;
}
```

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**'.

Listato 81.72. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/oM2mmrei* , *http://ideone.com/JUF2V* .

```
#include <stdio.h>
int main (void)
{
    int i = 0;
    for (;;)
      {
        if (i >= 10)
          {
            break;
          }
        printf ("x");
        i++;
      }
    getchar ();
    return 0;
}
```


81.4.5.1 Esercizio

Modificare il programma che calcola il fattoriale di un numero, usando un ciclo '**for**'.

81.4.5.2 Esercizio

Modificare il programma che verifica se un numero è primo, usando un ciclo '**for**'.

81.5 Funzioni del linguaggio C

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 che vengono importate nel codice attraverso l'istruzione '**#include**' del precompilatore. In pratica, in questo modo si importa la parte di codice necessaria alla dichiarazione e descrizione di queste funzioni. 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.

81.5.1 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 ();`
  Si tratta della dichiarazione di una funzione che fa qualcosa senza bisogno di ricevere alcun parametro e senza restituire alcun valore (*void*).
- `void elenca (void);`
  Esattamente come nell'esempio precedente, solo che è indicato in modo esplicito il fatto che la funzione non riceve argomenti (il tipo '**void**' è stato messo all'interno delle parentesi), come prescrive lo standard.

81.5.1.1 Esercizio

Scrivere i prototipi delle funzioni descritte nello schema successivo:

Nome della funzione	Tipo di valore restituito	Parametri
`alfa`	non restituisce alcunché	***x*** di tipo intero senza segno ***y*** di tipo carattere ***z*** di tipo a virgola mobile normale
`beta`	intero normale senza segno	non ci sono parametri
`gamma`	numero a virgola mobile di tipo normale	***x*** di tipo intero con segno ***y*** di tipo carattere senza segno

81.5.2 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[11] 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.[12] 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».

81.5.2.1 Esercizio

Completare i programmi successivi con la dichiarazione dei prototipi e con la descrizione delle funzioni necessarie.

Listato 81.80. Per svolgere l'esercizio attraverso un servizio *pastebin*: *http://codepad.org/04dX04L2kd* , *http://ideone.com/y7APt* .

```
#include <stdio.h>
//
// Mettere qui il prototipo della funzione «fattoriale».
//
// Mettere qui la descrizione della funzione «fattoriale».
//
int main (void)
{
    unsigned int x = 7;
    unsigned int f;
    f = fattoriale (x);
    printf ("Il fattoriale di %d è pari a %d.\n", x, f);
    getchar ();
    return 0;
}
```

Listato 81.81. Per svolgere l'esercizio attraverso un servizio *pastebin*: *http://codepad.org/g8Og2JQ1* , *http://ideone.com/aTWpX* .

```
#include <stdio.h>
//
// Mettere qui il prototipo della funzione «primo».
//
// Mettere qui la descrizione della funzione «primo».
//
int main (void)
{
    unsigned int x = 11;
    if (primo (x))
      {
        printf ("%d è un numero primo.\n", x);
      }
    else
      {
        printf ("%d non è un numero primo.\n", x);
      }
    getchar ();
    return 0;
}
```

Listato 81.82. Per svolgere l'esercizio attraverso un servizio *pastebin*: *http://codepad.org/Sof9C5lT* , *http://ideone.com/J5X1A* .

```
#include <stdio.h>
//
// Mettere qui il prototipo della funzione «interesse».
//
// Mettere qui la descrizione della funzione «interesse».
//
// L'interesse si ottiene come capitale * tasso * tempo.
//
int main (void)
{
    double    capitale = 10000; // Euro
    double       tasso = 0.03;  // pari al 3 %
    unsigned int tempo = 3      // anni
    double   interessi;
    interessi = interesse (capitale, tasso, tempo);
    printf ("Un capitale di %f Euro ", capitale);
    printf ("investito al tasso del %f%% ", tasso * 100);
    printf ("Per %d anni, dà interessi per %f Euro.\n",
            tempo, interessi);
    getchar ();
    return 0;
}
```

81.5.3 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 anda-

re in conflitto con l'evoluzione del linguaggio. A questo proposito va osservata una regola molto semplice: non si possono usare nomi «esterni» che inizino con il trattino basso ('_'); in tutti gli altri casi, invece, 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.

81.5.4 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    montante = (capitale * tasso) / 100;
…
printf ("%s: il capitale %f, ", "Ciao", capitale);
printf ("investito al tasso %f%% ", tasso);
printf ("ha prodotto un montante pari a %d.\n", montante);
…
```

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**'; '**%d**' 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 montante pari a 1005.»

La funzione ***scanf()*** è un po' più difficile da comprendere: la stringa che definisce il procedimento di interpretazione e conversione deve 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 ("%d", &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 ("%d%f", &capitale, &tasso);
```

81.5.5 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***.[13] 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);
      }
    …
}
```

81.5.5.1 Esercizio

Modificare uno degli esercizi già fatti, dove si verifica se un numero è primo, allo scopo di far concludere il programma con '**`EXIT_SUCCESS`**' se il numero è primo effettivamente; in caso contrario il programma deve terminare con il valore corrispondente a '**`EXIT_FAILURE`**'.

In un sistema operativo in cui si possa utilizzare una shell POSIX, per verificare il valore restituito dal programma appena terminato è possibile usare il comando seguente:

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

Si ricorda che la conclusione con successo di un programma si traduce normalmente nel valore zero.

81.6 Riferimenti

- Brian W. Kernighan, Dennis M. Ritchie, *Il linguaggio C: principi di programmazione e manuale di riferimento*, Pearson, ISBN 88-7192-200-X, *http://cm.bell-labs.com/cm/cs/cbook/*
- 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*

81.7 Soluzioni agli esercizi proposti

Esercizio	Soluzione

81.1.2.1	`//      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");`
	`    //  Attende la pressione di un tasto, quindi termina.`
	`    getchar ();`
	`    return 0;`
	`}`

81.1.2.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`    printf ("Il mio primo programma\n");`
	`    printf ("scritto in linguaggio C.\n");`
	`    getchar ();`
	`    return 0;`
	`}`

81.1.3.1	`$ cc -o programma prova.c` [*Invio*] Ma se si dispone del compilatore GNU C, è meglio usare l'opzione '**`-Wall`**': `$ cc -Wall -o programma prova.c` [*Invio*]
81.1.4.1	`printf ("Imponibile: %d, IVA: %d.\n", 1000, 200);`

Esercizio	Soluzione
81.2.1.1	Con un byte da 8 bit non dovrebbe esserci la possibilità di avere una variabile scalare da 12 bit, perché di norma il byte è esattamente un sottomultiplo del rango delle variabili scalari disponibili.
81.2.2.1	Una variabile scalare di tipo '**`unsigned char`**' (da 8 bit) può rappresentare valori da 0 a 255, pertanto non è possibile assegnare a una tale variabile valori fino a 99999.
81.2.2.2	Una variabile scalare di tipo '**`unsigned char`**' (da 8 bit) può rappresentare valori da 0 a 255.
81.2.2.3	Una variabile scalare di tipo '**`signed short int`**' (da 16 bit) può rappresentare valori da -32768 a +32767.
81.2.2.4	Dovendo rappresentare il valore 12,34, si devono usare variabili in virgola mobile. Possono andare bene tutti i tipi: '**`float`**', '**`double`**' e '**`long double`**'.
81.2.3.1	La costante letterale '**`12`**' corrisponde a 12_{10}; la costante '**`012`**' rappresenta il numero 12_8, ovvero 10_{10}; la costante '**`0x12`**' indica il numero 12_{16}, ovvero 18_{10}.
81.2.3.2	La costante letterale '**`12`**' è di tipo '**`int`**'; la costante '**`12U`**' è di tipo '**`unsigned int`**'; la costante '**`12L`**' è di tipo '**`long int`**'; la costante '**`1.2`**' è di tipo '**`double`**'; la costante '**`1.2`**' è di tipo '**`long double`**'.
81.2.5.1	L'espressione ''**`3'-2`**' corrisponde in pratica alla costante carattere ''**`1'`**'; l'espressione ''**`5'+4`**' corrisponde in pratica alla costante carattere ''**`9'`**'.

81.2.7.1	`int d;`
	`unsigned long int e = 2111;`
	`float f = 21.11;`
	`const float g = 21.11;`

81.3.1.1	L'espressione '**`4+4`**' dovrebbe dare un risultato di tipo '**`int`**'; l'espressione '**`4/3`**' dovrebbe dare un risultato di tipo '**`int`**'; l'espressione '**`4.0/3`**' dovrebbe essere di tipo '**`double`**'; l'espressione '**`4L*3`**' dovrebbe essere di tipo '**`long int`**'.

81.3.2.1	`int a = 3;`
	`int b;`
	`b = --a;`
	a contiene 2 e ***b*** contiene 2.

81.3.2.1	`int a = 3;`
	`int b = 2;`
	`b = a + b;`
	a contiene 3 e ***b*** contiene 5.

81.3.2.1	`int a = 7;`
	`int b = 2;`
	`b = a % b;`
	a contiene 7 e ***b*** contiene 1.

81.3.2.1	`int a = 7;`
	`int b;`
	`b = (a = a * 2);`
	a contiene 14 e ***b*** contiene 14.

81.3.2.1	`int a = 3;`
	`int b = 2;`
	`b += a;`
	a contiene 3 e ***b*** contiene 5.

81.3.2.1	`int a = 7;`
	`int b = 2;`
	`b %= a;`
	a contiene 7 e ***b*** contiene 2.

81.3.2.1	`int a = 7;`
	`int b;`
	`b = (a *= 2);`
	a contiene 14 e ***b*** contiene 14.

81.3.2.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`    int a = 3;`
	`    int b;`
	`    b = --a;`
	`    printf ("a contiene %d;\n", a);`
	`    printf ("b contiene %d.\n", b);`
	`    getchar ();`
	`    return 0;`
	`}`

81.3.2.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`    int a = 3;`
	`    int b = 2;`
	`    b = a + b;`
	`    printf ("a contiene %d;\n", a);`
	`    printf ("b contiene %d.\n", b);`
	`    getchar ();`
	`    return 0;`
	`}`

81.3.2.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`    int a = 7;`
	`    int b = 2;`
	`    b = a % b;`
	`    printf ("a contiene %d;\n", a);`
	`    printf ("b contiene %d.\n", b);`
	`    getchar ();`
	`    return 0;`
	`}`

Esercizio	Soluzione

81.3.2.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int a = 7;`
	`  int b;`
	`  b = (a = a * 2);`
	`  printf ("a contiene %d;\n", a);`
	`  printf ("b contiene %d.\n", b);`
	`  getchar ();`
	`  return 0;`
	`}`

81.3.2.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int a = 3;`
	`  int b = 2;`
	`  b += a;`
	`  printf ("a contiene %d;\n", a);`
	`  printf ("b contiene %d.\n", b);`
	`  getchar ();`
	`  return 0;`
	`}`

81.3.2.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int a = 7;`
	`  int b = 2;`
	`  b %= a;`
	`  printf ("a contiene %d;\n", a);`
	`  printf ("b contiene %d.\n", b);`
	`  getchar ();`
	`  return 0;`
	`}`

81.3.2.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int a = 7;`
	`  int b;`
	`  b = (a *= 2);`
	`  printf ("a contiene %d;\n", a);`
	`  printf ("b contiene %d.\n", b);`
	`  getchar ();`
	`  return 0;`
	`}`

81.3.3.1	`int a = 4 + 1;`
	`signed char b = 5;`
	`int c = a == b;`
	c contiene 1.

81.3.3.1	`int a = 4 + 1;`
	`signed char b = 5;`
	`int c = a != b;`
	c contiene 0.

81.3.3.1	`unsigned int a = 4 + 3;`
	`signed char b = -5;`
	`int c = a >= b;`
	c contiene 1.

81.3.3.1	`unsigned int a = 4 + 3;`
	`signed char b = -5;`
	`int c = a <= b;`
	c contiene 0.

81.3.3.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int a = 4 + 1;`
	`  signed char b = 5;`
	`  int c = a == b;`
	`  printf ("(%d == %d) produce %d\n", a, b, c);`
	`  getchar ();`
	`  return 0;`
	`}`

81.3.3.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int a = 4 + 1;`
	`  signed char b = 5;`
	`  int c = a != b;`
	`  printf ("(%d != %d) produce %d\n", a, b, c);`
	`  getchar ();`
	`  return 0;`
	`}`

81.3.3.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  unsigned int a = 4 + 3;`
	`  signed char b = -5;`
	`  int c = a >= b;`
	`  printf ("(%d >= %d) produce %d\n", a, b, c);`
	`  getchar ();`
	`  return 0;`
	`}`

Esercizio	Soluzione

81.3.3.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  unsigned int a = 4 + 3;`
	`  signed char b = -5;`
	`  int c = a <= b;`
	`  printf ("(%d <= %d) produce %d\n", a, b, c);`
	`  getchar ();`
	`  return 0;`
	`}`

81.3.4.1	`int a = 4;`
	`signed char b = (3 < 5);`
	`int c = a && b;`
	c contiene 1.

81.3.4.1	`int a = 4;`
	`signed char b = (3 < 5);`
	`int c = a || b;`
	c contiene 1.

81.3.4.1	`unsigned int a = 4 + 3;`
	`signed char b = -5;`
	`int c = (b > 0) || (a > b);`
	c contiene 1.

81.3.5.1	`int a = 5;`
	`int b = 12;`
	`int c = a & b;`
	c contiene 4.

81.3.5.1	`int a = 5;`
	`int b = 12;`
	`int c = a | b;`
	c contiene 13.

81.3.5.1	`int a = 5;`
	`int b = 12;`
	`int c = a ^ b;`
	c contiene 9.

81.3.5.1	`int a = 5;`
	`int c = a << 1;`
	c contiene 10.

81.3.5.1	`int a = 21;`
	`int c = a >> 1;`
	c contiene 10.

81.3.5.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int a = 5;`
	`  int b = 12;`
	`  int c = a & b;`
	`  printf ("c contiene %d\n", c);`
	`  getchar ();`
	`  return 0;`
	`}`

81.3.5.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int a = 5;`
	`  int b = 12;`
	`  int c = a | b;`
	`  printf ("c contiene %d\n", c);`
	`  getchar ();`
	`  return 0;`
	`}`

81.3.5.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int a = 5;`
	`  int b = 12;`
	`  int c = a ^ b;`
	`  printf ("c contiene %d\n", c);`
	`  getchar ();`
	`  return 0;`
	`}`

81.3.5.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int a = 5;`
	`  int c = a << 1;`
	`  printf ("c contiene %d\n", c);`
	`  getchar ();`
	`  return 0;`
	`}`

81.3.5.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int a = 21;`
	`  int c = a >> 1;`
	`  printf ("c contiene %d\n", c);`
	`  getchar ();`
	`  return 0;`
	`}`

81.3.6.1	L'espressione '**`(int) (4.4+4.9)`**' è equivalente a '**`(int) 9`**'; l'espressione '**`(double) 4/3`**' è equivalente a '**`(double) 1`**'; l'espressione '**`((double) 4)/3`**' è equivalente a '**`((double) 1.33333`**'.

81.3.7.1	`int a = -20;`
	`int b = 10;`
	`int c = (b = 2, b = a, c = a + b);`
	c contiene -40.

81.3.7.1	`int a = -20;`
	`int b = 10;`
	`int c = (b = 2, b = a++, c = a + b);`
	c contiene -39.

Esercizio	Soluzione

81.3.7.1	`int a = -20;`
	`int b = 10;`
	`int c = (b = 2, b = ++a, c = a + b);`
	c contiene -38.

81.3.7.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int a = -20;`
	`  int b = 10;`
	`  int c = (b = 2, b = a, c = a + b);`
	`  printf ("c contiene %d\n", c);`
	`  getchar ();`
	`  return 0;`
	`}`

81.3.7.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int a = -20;`
	`  int b = 10;`
	`  int c = (b = 2, b = a++, c = a + b);`
	`  printf ("c contiene %d\n", c);`
	`  getchar ();`
	`  return 0;`
	`}`

81.3.7.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int a = -20;`
	`  int b = 10;`
	`  int c = (b = 2, b = ++a, c = a + b);`
	`  printf ("c contiene %d\n", c);`
	`  getchar ();`
	`  return 0;`
	`}`

81.4.1.1	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int x;`
	`  x = 5000;`
	
	`  if ((x < 1000) || (x > 10000))`
	`  {`
	`  printf ("Il valore di x non è valido!\n");`
	`  }`
	`  else if (x > 5000)`
	`  {`
	`  printf ("Il livello di x è alto: %d\n", x);`
	`  }`
	`  else if (x < 5000)`
	`  {`
	`  printf ("Il livello di x è basso: %d\n", x);`
	`  }`
	`  else`
	`  {`
	`  printf ("Il livello di x è ottimale: %d\n", x);`
	`  }`
	`  getchar ();`
	`  return 0;`
	`}`

81.4.1.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int x;`
	`  x = -1;`
	
	`  if (x)`
	`  {`
	`  printf ("Sono felice :-)\n");`
	`  }`
	`  else`
	`  {`
	`  printf ("Sono triste :-(\n");`
	`  }`
	`  getchar ();`
	`  return 0;`
	`}`
	Il programma visualizza la scritta «Sono felice :-)», perché un qualunque valore numerico diverso da zero viene inteso pari a *Vero*.

81.4.2.1	`  case 2:`
	`  if (((anno % 4 == 0) && !(anno % 100 == 0))`
	`  || (anno % 400 == 0))`
	`  {`
	`  giorni = 29;`
	`  }`
	`  else`
	`  {`
	`  giorni = 28;`
	`  }`
	`  break;`
	Se l'anno è divisibile per quattro (pertanto la divisione per quattro non dà resto) e se l'anno non è divisibile per 100 (quindi non si tratta di un secolo), oppure se l'anno è divisibile per 400, il mese di febbraio ha 29, mentre ne ha 28 negli altri casi. In pratica, di norma gli anni bisestili sono quelli il cui anno è divisibile per quattro, ma questa regola non si applica se l'anno è l'inizio di un secolo, ma ogni quattro secoli si fa eccezione (pertanto, anche se di norma l'anno che inizia un secolo non è bisestile, il secolo che si ha ogni 400 anni è invece, nuovamente, bisestile).

Esercizio	Soluzione

81.4.3.1	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  unsigned int x = 4;`
	`  unsigned int f = x;`
	`  unsigned int i = (x - 1);`
	`  while (i > 0)`
	`  {`
	`  f = f * i;`
	`  i--;`
	`  }`
	`  printf ("%d! è pari a %d\n", x, f);`
	`  getchar ();`
	`  return 0;`
	`}`

81.4.3.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int x = 11;`
	`  int i = 2;`
	`  if (x <= 1)`
	`  {`
	`  printf ("%d non è un numero primo.\n", x);`
	`  }`
	`  else`
	`  {`
	`  while (i < x)`
	`  {`
	`  if ((x % i) == 0)`
	`  {`
	`  printf ("%d è divisibile per %d.\n", x, i);`
	`  break;`
	`  }`
	`  i++;`
	`  }`
	`  if (i >= x)`
	`  {`
	`  printf ("%d è un numero primo.\n", x);`
	`  }`
	`  }`
	`  getchar ();`
	`  return 0;`
	`}`

81.4.4.1	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int x = 11;`
	`  int i = 2;`
	`  if (x <= 1)`
	`  {`
	`  printf ("%d non è un numero primo.\n", x);`
	`  }`
	`  else`
	`  {`
	`  do`
	`  {`
	`  if ((x % i) == 0)`
	`  {`
	`  printf ("%d è divisibile per %d.\n", x, i);`
	`  break;`
	`  }`
	`  i++;`
	`  }`
	`  while (i < x);`
	`  if (i >= x)`
	`  {`
	`  printf ("%d è un numero primo.\n", x);`
	`  }`
	`  }`
	`  getchar ();`
	`  return 0;`
	`}`

81.4.5.1	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  unsigned int x = 4;`
	`  unsigned int f = x;`
	`  unsigned int i;`
	`  for (i = (x - 1); i > 0; i--)`
	`  {`
	`  f = f * i;`
	`  }`
	`  printf ("%d! è pari a %d\n", x, f);`
	`  getchar ();`
	`  return 0;`
	`}`

Esercizio	Soluzione

81.4.5.2	`#include <stdio.h>`
	`int main (void)`
	`{`
	`  int x = 11;`
	`  int i;`
	`  if (x <= 1)`
	`  {`
	`  printf ("%d non è un numero primo.\n", x);`
	`  }`
	`  else`
	`  {`
	`  for (i = 2; i < x; i++)`
	`  {`
	`  if ((x % i) == 0)`
	`  {`
	`  printf ("%d è divisibile per %d.\n", x, i);`
	`  break;`
	`  }`
	`  }`
	`  if (i >= x)`
	`  {`
	`  printf ("%d è un numero primo.\n", x);`
	`  }`
	`  }`
	`  getchar ();`
	`  return 0;`
	`}`

81.5.1.1	`void alfa (unsigned int x, char y, double z);`
	
	`unsigned int beta (void);`
	
	`double gamma (int x, signed char y);`

81.5.2.1	`#include <stdio.h>`
	`unsigned int fattoriale (unsigned int x);`
	`unsigned int`
	`fattoriale (unsigned int x)`
	`{`
	`  unsigned int f = x;`
	`  unsigned int i;`
	`  for (i = (x - 1); i > 0; i--)`
	`  {`
	`  f = f * i;`
	`  }`
	`  return i;`
	`}`
	`int main (void)`
	`{`
	`  unsigned int x = 7;`
	`  unsigned int f;`
	`  f = fattoriale (x);`
	`  printf ("Il fattoriale di %d è pari a %d.\n", x, f);`
	`  getchar ();`
	`  return 0;`
	`}`

81.5.2.1	`#include <stdio.h>`
	`unsigned int primo (unsigned int x);`
	`unsigned int`
	`primo (unsigned int x)`
	`{`
	`  unsigned int i;`
	`  if (x <= 1)`
	`  {`
	`  return 0;`
	`  }`
	`  for (i = 2; i < x; i++)`
	`  {`
	`  if ((x % i) == 0)`
	`  {`
	`  return 0;`
	`  }`
	`  }`
	`  if (i >= x)`
	`  {`
	`  return 1;`
	`  }`
	`  else`
	`  {`
	`  return 0;`
	`  }`
	`}`
	`int main (void)`
	`{`
	`  unsigned int x = 11;`
	`  if (primo (x))`
	`  {`
	`  printf ("%d è un numero primo.\n", x);`
	`  }`
	`  else`
	`  {`
	`  printf ("%d non è un numero primo.\n", x);`
	`  }`
	`  getchar ();`
	`  return 0;`
	`}`

Esercizio	Soluzione

81.5.2.1	`#include <stdio.h>`
	`double interesse (double c, double i, unsigned int t);`
	`double`
	`interesse (double c, double i, unsigned int t)`
	`{`
	`  return (c * i * t);`
	`}`
	`int main (void)`
	`{`
	`  double  capitale = 10000; // Euro`
	`  double  tasso = 0.03;  // pari al 3 %`
	`  unsigned int tempo = 3;  // anni`
	`  double  interessi;`
	`  interessi = interesse (capitale, tasso, tempo);`
	`  printf ("Un capitale di %f Euro ", capitale);`
	`  printf ("investito al tasso del %f%% ", tasso * 100);`
	`  printf ("Per %d anni, dà interessi per %f Euro.\n", tempo, interessi);`
	`  getchar ();`
	`  return 0;`
	`}`

81.5.5.1	`#include <stdio.h>`
	`#include <stdlib.h>`
	`unsigned int primo (unsigned int x);`
	`unsigned int`
	`primo (unsigned int x)`
	`{`
	`  unsigned int i;`
	`  if (x <= 1)`
	`  {`
	`  return 0;`
	`  }`
	`  for (i = 2; i < x; i++)`
	`  {`
	`  if ((x % i) == 0)`
	`  {`
	`  return 0;`
	`  }`
	`  }`
	`  if (i >= x)`
	`  {`
	`  return 1;`
	`  }`
	`  else`
	`  {`
	`  return 0;`
	`  }`
	`}`
	`int main (void)`
	`{`
	`  unsigned int x = 11;`
	`  if (primo (x))`
	`  {`
	`  return (EXIT_SUCCESS);`
	`  }`
	`  else`
	`  {`
	`  return (EXIT_FAILURE);`
	`  }`
	`}`

[1] È bene osservare che un'istruzione composta, ovvero un raggruppamento di istruzioni tra parentesi graffe, non è concluso dal punto e virgola finale.

[2] In particolare, i nomi che iniziano con due trattini bassi ('__'), oppure con un trattino basso seguito da una lettera maiuscola ('*_X*') sono riservati.

[3] Il linguaggio C, puro e semplice, non comprende alcuna funzione, benché esistano comunque molte funzioni più o meno standardizzate, come nel caso di ***printf()***.

[4] 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.

[5] 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.

[6] La distinzione tra valori con segno o senza segno, riguarda solo i numeri interi, perché quelli in virgola mobile sono sempre espressi con segno.

[7] 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.

[8] 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.

[9] Gli operandi di '`?` `:`' sono tre.

[10] 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*.

[11] Per la precisione, i parametri di una funzione corrispondono alla dichiarazione di variabili di tipo automatico.

[12] 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.

[13] In pratica, ***EXIT_SUCCESS*** equivale a zero, mentre ***EXIT_FAILURE*** equivale a uno.

Puntatori, array e stringhe in C

Nel linguaggio C, per poter utilizzare gli array si gestiscono dei puntatori alle zone di memoria contenenti tali strutture.

82.1 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*).

Il concetto di *lvalue* serve a chiarire che un'espressione può rappresentare una «variabile», ovvero una certa posizione in memoria, pur senza averle dato un nome.

82.1.1 Esercizio

Nelle espressioni seguenti, indicare quali sono i componenti che costituiscono un *lvalue*:

Espressione	*lvalue*
`x = 4, y = 3 * 2`	***x*** e ***y***
`y = 3 * x`	
`z += 3 * x`	
`j = i++ * 5`	

82.2 Puntatori

Una variabile, di qualunque tipo sia, rappresenta normalmente un valore posto da qualche parte nella memoria del sistema. 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;
…
// L'indirizzo di «i» viene assegnato al puntatore «p».
p = &i;
```

82.3 Dichiarazione di una variabile puntatore

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**'.

```
int *p;
```

Sia chiaro che la variabile dichiarata in questo modo ha il nome ***p*** ed è di tipo '**int ***', ovvero puntatore al tipo intero normale. Pertanto, l'asterisco, benché lo si rappresenti attaccato al nome della variabile, qui fa parte della dichiarazione del tipo.

Normalmente, il puntatore è costituito da un numero che rappresenta un indirizzo di memoria. Il fatto di precisare il tipo di variabile a cui si riferisce il puntatore, consente di sapere per quanti byte si estende l'informazione in questione.

82.3.1 Esercizio

Nella tabella successiva sono riportate delle istruzioni, a fianco delle quali si fanno delle domande. Si risponda a tali domande.

Codice	Questione
`int a = 20;` `b = &a;`	Quale dovrebbe essere il tipo della variabile ***b***?
	Come si dichiara la variabile ***x***, in qualità di puntatore al tipo '**long long int**'?
`long int *z;`	Cosa può contenere la variabile ***z***?

82.4 Dereferenziazione

Così come esiste l'operatore di indirizzamento, costituito dalla e-commerciale ('**&**'), con il quale si ottiene il puntatore corrispondente a una variabile, è disponibile un operatore di «dereferenziazione», con cui è possibile raggiungere la zona di memoria a cui si riferisce un puntatore, come se si trattasse di una variabile comune. L'operatore di dereferenziazione è l'asterisco ('*****').

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.

Nell'esempio seguente, l'area di memoria a cui si riferisce il puntatore ***p*** viene sovrascritta con il valore 123:

```
int *p;
…
*p = 123;
```

Nell'esempio seguente, l'area di memoria a cui si riferisce il puntatore ***p***, corrispondente in pratica alla variabile ***v***, viene sovrascritta con il valore 456:

```
int v;
int *p;
…
p = &v;
*p = 456;
```

Nell'esempio appena apparso, si osserva alla fine che è possibile fare riferimento alla stessa area di memoria, sia attraverso la variabile ***v***, sia attraverso il puntatore dereferenziato ****p***.

L'esempio seguente serve a chiarire un po' meglio il ruolo delle variabili puntatore:

```
int v = 10;
int *p;
int *p2;
…
p = &v;
…
p2 = p;
…
*p2 = 20;
```

Alla fine, la variabile ***v*** e i puntatori dereferenziati ****p*** e ****p2*** contengono tutti lo stesso valore; ovvero, i puntatori ***p*** e ***p2*** individuano entrambi l'area di memoria corrispondente alla variabile ***v***, la quale si trova a contenere il valore 20.

Si osservi che l'asterisco è un operatore che, evidentemente, ha la precedenza rispetto a quelli di assegnamento. Eventualmente si possono usare le parentesi per togliere ambiguità al codice:

```
(*p2) = 20;
```

82.4.1 Esercizio

Nella tabella successiva sono riportate delle istruzioni, a fianco delle quali si fanno delle domande. Si risponda a tali domande.

Codice	Questione

`long int *i;`
`long int  j;`
`…`
`i = j;`
	L'ultima istruzione è errata: quale potrebbe essere la soluzione giusta?

`int *i;`
`int  j = 10;`
`…`
`i = &j;`
`(*i)++;`
	Cosa contiene alla fine la variabile ***j***?

`long int *i;`
`int  j;`
`…`
`*i = j;`
	L'ultima istruzione contiene un problema: come lo si può correggere?

82.5 «Little endian» e «big endian»

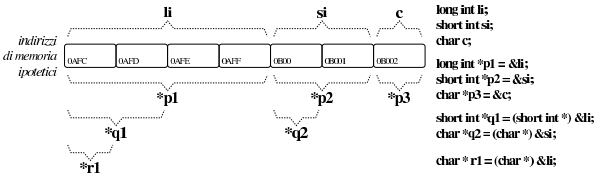
Il tipo di dati a cui un puntatore si rivolge, fa parte integrante dell'informazione rappresentata dal puntatore stesso. Ciò è importante perché quando si dereferenzia un puntatore occorre sapere quanto è grande l'area di memoria a cui si deve accedere a partire dal puntatore. Per questa ragione, quando si assegna a una variabile puntatore un altro puntatore, questo deve essere compatibile, nel senso che deve 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.

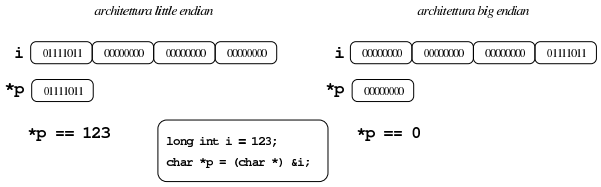


L'esempio seguente rappresenta un programma completo che ha lo scopo di determinare se l'architettura dell'elaboratore è di tipo *big endian* o di tipo *little endian*. Per capirlo si dichiara una variabile di tipo '**`long int`**' che si intende debba essere di rango superiore rispetto al tipo '**`char`**', 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 82.13. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/IRCiWUyg* , *http://ideone.com/aFfAG* .

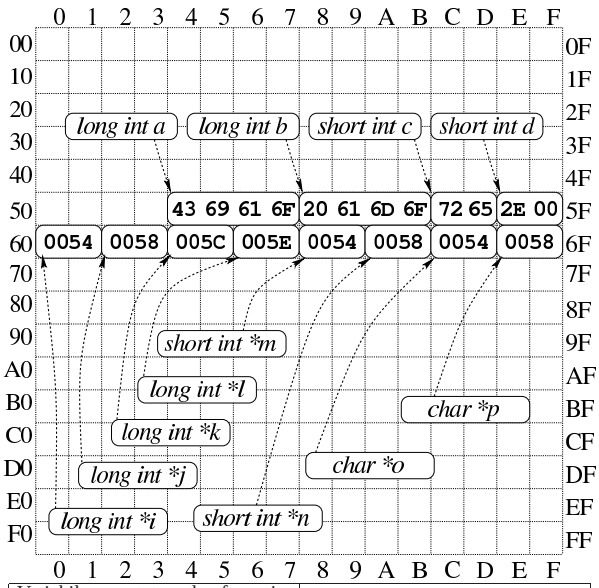
```
#include <stdio.h>
int main (void)
{
    long int i = 123;
    char *p = (char *) &i;
    if (*p == 123)
      {
        printf ("little endian\n");
      }
    else
      {
        printf ("big endian\n");
      }
    getchar ();
    return 0;
}
```


Figura 82.14. Schematizzazione dell'operato del programma di esempio, per determinare l'ordine dei byte usato nella propria architettura.



82.5.1 Esercizio

La figura successiva mostra una mappa ipotetica di memoria, con indirizzi che vanno da 00_{16} a FF_{16}, in cui sono evidenziate delle variabili scalari comuni e delle variabili puntatore. Ogni cella di memoria corrisponde a un byte e si presume che l'architettura del microprocessore preveda un accesso in modalità *big endian* (quello più semplice dal punto di vista umano). Si vuole conoscere il contenuto delle variabili scalari normali e quello rappresentato dai puntatori dereferenziati, con l'aiuto di alcuni suggerimenti.



Variabile o puntatore dereferenziato	Contenuto
a	$4369616F_{16}$
b	
c	
d	
****i***	$4369616F_{16}$
****j***	
****k***	$72652E00_{16}$
****l***	
****m***	
****n***	

Variabile o puntatore dereferenziato	Contenuto
***o**	
***p**	

82.6 Chiamata di funzione con puntatori

Il linguaggio C utilizza il passaggio degli argomenti alle funzioni per valore, per cui, anche se gli argomenti sono indicati in qualità di variabili, le modifiche ai valori rispettivi apportati nel codice delle funzioni non si riflettono sul contenuto delle variabili originali; per farlo, occorre usare invece argomenti costituiti da 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 il proprio parametro.

Listato 82.17. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/eEWeJvo2* , *http://ideone.com/bKTQx* .

```
#include <stdio.h>
void funzione (int *x)
{
    (*x)++;
}
int main (void)
{
    int y = 10;
    funzione (&y);
    printf ("y = %i\n", y);
    getchar ();
    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 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, ovvero 11.

82.6.1 Esercizio

Si prenda in considerazione il programma successivo e si scriva il valore contenuto nelle tre variabili ***i***, ***j*** e ***k***, così come rappresentato dalla funzione ***printf()***.

```
#include <stdio.h>
int f (int *x, int y)
{
    return ((*x)++ + y);
}
int main (void)
{
    int i = 1;
    int j = 2;
    int k;
    k = f (&i, j);
    printf ("i=%i, j=%i, k=%i\n", i, j, k);
    getchar ();
    return 0;
}
```


82.6.2 Esercizio

Si modifichi il programma dell'esercizio precedente, creando nella funzione ***main()*** la variabile ***l***, in qualità di puntatore a un intero, assegnando a questa variabile il puntatore dell'area di memoria rappresentata da ***i***, usando poi la variabile ***l*** nella chiamata della funzione ***f()***.

82.7 Array

Nel linguaggio C, l'array è una sequenza ordinata di elementi dello stesso tipo nella rappresentazione ideale di memoria di cui si dispone. 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.

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 manifesti alcun errore, arrivando però a dei risultati imprevedibili.

82.8 Array a una dimensione

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.

```
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, purché siano compatibili:

```
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!
```

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.

82.8.1 Esercizio

Si completi la tabella successiva con il codice necessario a creare gli array richiesti.

Richiesta	Codice
Si vuole creare l'array ***a[]*** di 11 elementi di tipo intero senza segno.	
Si vuole creare l'array ***b[]*** di 3 elementi di tipo intero normale, contenente i valori 2, 7 e 123.	
Si vuole creare l'array ***c[]*** di 7 elementi di tipo intero normale, contenente inizialmente i valori 2, 7 e 123.	

82.8.2 Esercizio

Completare il codice successivo, in cui si dichiara un array e lo si popola successivamente con i primi valori numerici interi, a partire da uno.

```
…
int a[5];
int i;
…
for (i =         ; i         ; i          )
  {
    a[i] =               ;
  }
…
```

Dopo il ciclo '**for**', si vuole che l'array contenga la sequenza dei numeri: 1, 2, 3,... 5.

82.8.3 Esercizio

Si vuole ottenere lo stesso risultato dell'esercizio precedente, ma in questo caso viene posto un vincolo nel codice, in cui si vede che l'indice ***i*** viene decrementato nel ciclo '**for**'.

```
…
int a[5];
int i;
…
for (i =         ; i         ; i--)
  {
    a[i] =               ;
  }
…
```

Che valore ha la variabile ***i***, al termine del ciclo '**for**'?

82.9 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 82.32. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/d60HA60Fgn* , *http://ideone.com/4VFM9* .

```
#include <stdio.h>

int main (void)
{
  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");
    }

  getchar ();
  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.
…
```

82.9.1 Esercizio

Si completi la tabella successiva con il codice necessario a creare gli array richiesti.

Richiesta	Codice
Si vuole creare l'array ***a[]*** di 11×7 elementi di tipo intero senza segno.	
Si vuole creare l'array ***b[]*** di 3×2 elementi di tipo intero normale, contenente i valori {2, 7}, {5, 11} e {100, 123}.	
Si vuole creare l'array ***c[]*** di 7×2 elementi di tipo intero normale, contenente i valori {2, 7} e {5, 11}.	

82.9.2 Esercizio

Completare il codice successivo, in cui si dichiara un array e lo si popola successivamente con i primi valori numerici interi, a partire da uno.

```
…
int a[5][7];
int i;
int j;
…
for (i =         ; i         ; i          )
  {
    for (j =         ; j         ; j          )
      {
        a[i][j] =               ;
      }
  }
…
```

Dopo il ciclo '**for**', si vuole che l'array contenga la sequenza dei numeri: 1, 2, 3,... 35, cominciando dall'elemento ***a[0][0]***, per finire con l'elemento ***a[4][6]***.

82.9.3 Esercizio

Si vuole ottenere lo stesso risultato dell'esercizio precedente, ma in questo caso viene posto un vincolo nel codice, in cui si vede che gli indici ***i*** e ***j*** vengono decrementati nel ciclo '**for**' rispettivo.


```
…
int a[5][7];
int i;
int j;
…
for (i =         ; i         ; i--)
  {
    for (j =         ; j         ; j--)
      {
        a[i][j] =               ;
      }
  }
…
```

82.10 Natura dell'array

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 copiare il contenuto, 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 variabile ***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 82.41. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/MPcyb9yQ* , *http://ideone.com/j7lVY* .

```
#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]);

  getchar ();
  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 82.44. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/LTyTlzk1* , *http://ideone.com/ndTqs* .

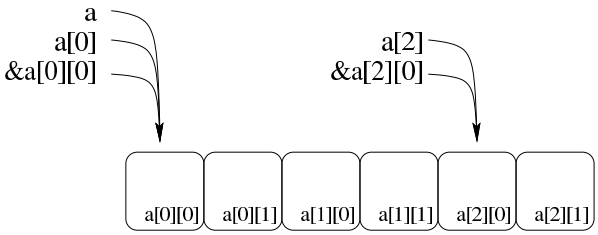
```
#include <stdio.h>

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]);

  getchar ();
  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.

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.

82.10.1 Esercizio

Il codice che appare nella tabella successiva, contiene dei problemi. Si spieghi perché.

Codice problematico	Spiegazione

`signed int a[7];`
`unsigned int b[8];`
…
`a = b;`	

`int a[7][5];`
`long int *b;`
…
`b = a;`	

`int a[7][5];`
`int *b;`
…
`b = &a[3];`	

82.10.2 Esercizio

Da un array viene estrapolato il puntatore, del suo inizio o di una posizione interna, e con quello si fanno delle modifiche al contenuto. Indicare dove avvengono le modifiche.

Codice	Richiesta

`int a[7][5];`
`int *p;`
…
`p = (int *) a;`
`p[7] = 123;`
	L'ultima istruzione evidenziata, modifica un elemento dell'array ***a[][]***; quale?

`int a[7][5];`
`int *p;`
…
`p = (int *) &a[1];`
`*p = 123;`
	L'ultima istruzione evidenziata, modifica un elemento dell'array ***a[][]***; quale?

`int a[7][5];`
`int *p;`
…
`p = (int *) a;`
`p[35] = 123;`
	L'ultima istruzione evidenziata, modifica il contenuto dell'array ***a[][]***? Cosa fa invece?

82.11 Array e funzioni

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 82.50. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/GmqgyheC* , *http://ideone.com/59ix59q* .

```
#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]);

  getchar ();
  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.

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.

82.12 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 adiacente 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.
```

82.12.1 Esercizio

Da un array viene estrapolato il puntatore, del suo inizio o di una posizione interna, e con quello si fanno delle cose. Rispondere alle domande a fianco del codice mostrato.

Codice	Richiesta

`int a[7][5];`
`int *p;`
`…`
`p = (int *) a;`
`p += 7;`
`*p = 123;`
	Attraverso la variabile puntatore ***p*** viene modificato un elemento dell'array ***a[][]***; quale?

`int a[7][5];`
`int *p;`
`…`
`p = (int *) &a[1];`
`*(p+7) = 123;`
	Attraverso la variabile puntatore ***p*** viene modificato un elemento dell'array ***a[][]***; quale? Che differenza c'è rispetto al caso precedente?

`int a[7][5];`
`int *p;`
`int i;`
`…`
`p = (int *) a;`
`for (i = 0 ; i < 35 ; i++, p++)`
`  {`
`  *p = i;`
`  }`
	Cosa succede al contenuto dell'array ***a[][]***?
	Al termine del ciclo '**for**', a cosa punta la variabile puntatore ***p***?

82.13 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 semplicemente 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 82.65. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/9Id0fIdf* , *http://ideone.com/CCkFd* .

```
#include <stdio.h>

void elabora (char *z)
{
  printf (z);
}

int main (void)
{
  elabora ("ciao\n");
  getchar ();
  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 82.66. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/ksRqufBV* , *http://ideone.com/jmtac* .

```
#include <stdio.h>

void elabora (char z[])
{
  printf (z);
}

int main (void)
{
  elabora ("ciao\n");
  getchar ();
  return 0;
}
```

Tabella 82.67. 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 ('\').
\'	Un apice singolo destro.
\"	Un apice doppio.
\?	Un punto interrogativo. Si usa in quanto le sequenze *trigraph* sono formate da un prefisso di due punti interrogativi.
\0	Il codice <*NUL*>.
\a	Il codice <*BEL*> (*bell*).
\b	Il codice <*BS*> (*backspace*).
\f	Il codice <*FF*> (*formfeed*).
\n	Il codice <*LF*> (*linefeed*).
\r	Il codice <*CR*> (*carriage return*).
\t	Una tabulazione orizzontale (<*HT*>).
\v	Una tabulazione verticale (<*VT*>).

82.13.1 Esercizio

Cosa contengono gli array rappresentati nella tabella successiva? Sono stringhe?

Codice
`int a[] = {'a', 'm', 'o', 'r', 'e', '\n', '\0'};`
`int b[] = {'a', 'm', 'o', 'r', 'e', '\0'};`
`int c[] = {'a', 'm', 'o', 'r', 'e', '\n'};`
`int d[] = {'a', 'm', 'o', 'r', 'e'};`
`char e[] = {'a', 'm', 'o', 'r', 'e', '\n', '\0'};`
`char f[] = {'a', 'm', 'o', 'r', 'e', '\0'};`
`char g[] = {'a', 'm', 'o', 'r', 'e', '\n'};`
`char h[] = {'a', 'm', 'o', 'r', 'e'};`
`char i[] = {'a', 'm', 'o', 'r', 'e', '\0', 'm', 'i', 'o'};`

82.13.2 Esercizio

Rispondere alle domande a fianco del codice contenuto nella tabella successiva.

Codice	Richiesta
… `char a[15];` …	Di cosa si tratta? Può essere una stringa?
… `char b[15] = "ciao";` …	Di cosa si tratta? Può essere una stringa?
… `char c[15] = "ciao";` … **`c = "amore";`**	È lecito l'assegnamento evidenziato? Perché?

…
`char d[15] = "ciao";`
`char *e = d;`
…
	È lecito l'assegnamento evidenziato? Perché?

…
`char *f;`
…
`f = "ciao";`
	Dopo l'assegnamento, cos'è ***f***?

…
`char *g = "ciao"`
…
`g = "amore";`
	Dopo l'assegnamento, si può ancora fare riferimento alla stringa contenente la parola «ciao»? Che fine fa la memoria che la contiene?

…
`char *h = "ciao"`
…
`*h = 'C';`
	A cosa serve l'assegnamento finale? È possibile attuarlo?

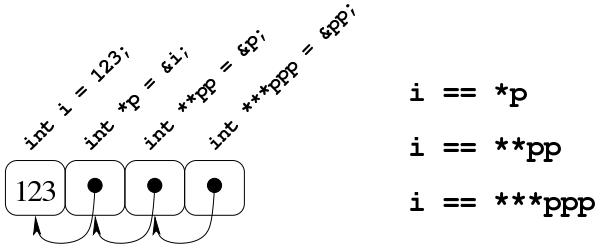
…
`char *i = "ciao"`
…
`i++;`
	Al termine, cosa rappresenta ***i***?

82.14 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.

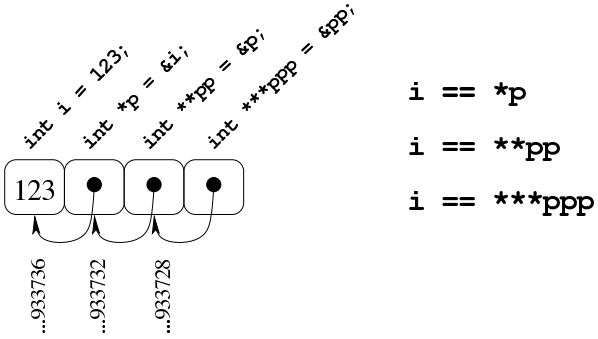
Listato 82.72. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/6BXKTQeS* , *http://ideone.com/1FLV9* .

```
#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);

  getchar ();
  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
```

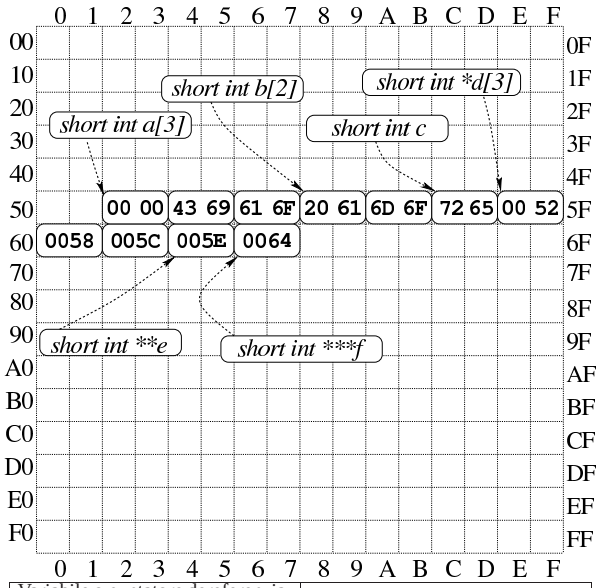
Pertanto si può ricostruire la disposizione in memoria delle variabili:



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.

82.14.1 Esercizio

La figura successiva mostra una mappa ipotetica di memoria, con indirizzi che vanno da 00_{16} a FF_{16}, in cui sono evidenziate delle variabili di vario tipo, inclusi i puntatori. Ogni cella di memoria corrisponde a un byte e si presume che l'architettura del microprocessore preveda un accesso in modalità *big endian* (quello più semplice dal punto di vista umano). Si vuole conoscere il risultato della dereferenziazione dei puntatori, secondo quanto richiesto espressamente nella tabella che segue la figura.



Variabile o puntatore dereferenziato	Contenuto
a[1]	4369_{16}
b[0]	
c	
d[0]	0052_{16}
****d[0]***	
****e***	0052_{16}
*****e***	
****f***	$005E_{16}$
*****f***	
******f***	

82.15 Puntatori a più dimensioni

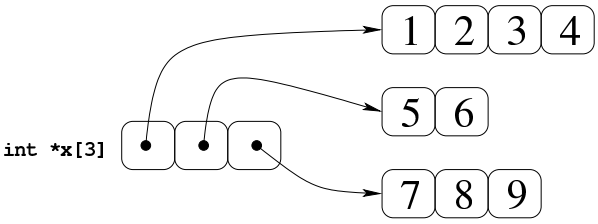
Un array di puntatori consente di realizzare delle strutture di dati ad albero, non più uniformi come invece devono essere gli array a più dimensioni consueti. L'esempio seguente mostra la dichiarazione di tre array di interi, con una quantità di elementi disomogenea, e la successiva dichiarazione di 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 82.77. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/0hJZbbZ5* , *http://ideone.com/WelMI* .

```
#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));

  getchar ();
  return 0;
}
```

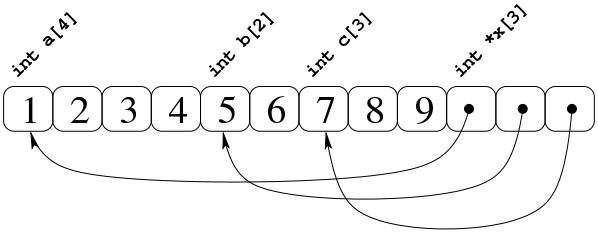

La figura successiva dovrebbe facilitare la comprensione del senso dell'array di puntatori. Come si può osservare, per accedere agli elementi degli array a cui puntano quelli di ***x*** è necessario dereferenziare gli elementi. Pertanto, '**`*x[0]`**' corrisponde al contenuto del primo elemento del primo 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 82.78. Schematizzazione semplificata del significato dell'array di puntatori definito nell'esempio.



Si potrebbe obbiettare 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 82.79. 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'esempio 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 82.80. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/D002Bp02rL* , *http://ideone.com/ozEKK* .

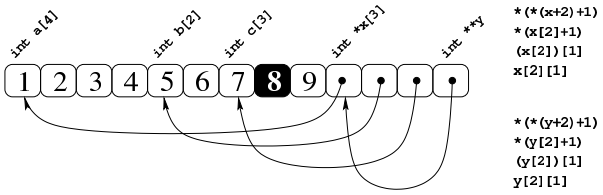
```
#include <stdio.h>

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]);

  getchar ();
```

```
  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 dereferenziando 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 82.81. Tanti modi alternativi per raggiungere lo stesso elemento.



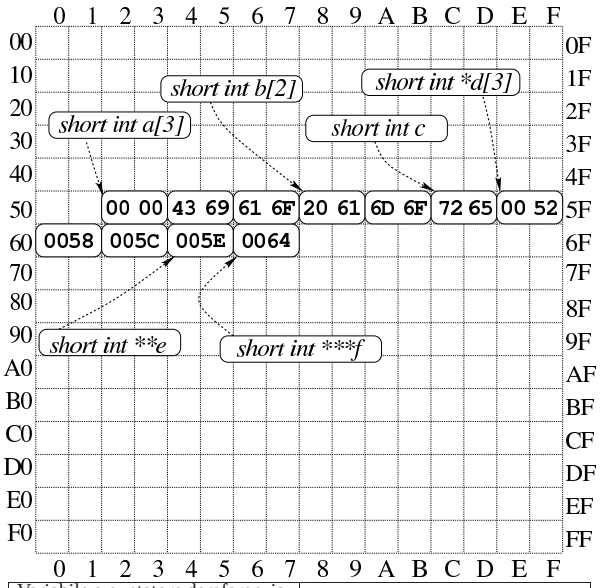
Seguendo lo stesso ragionamento si possono gestire strutture ad albero più complesse, con più livelli di puntatori, ma qui non vengono proposti esempi di questo tipo.

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 «normale» a matrice, oppure secondo uno schema ad albero attraverso dei puntatori: la differenza sta solo nella presenza o meno di elementi costituiti da puntatori.

82.15.1 Esercizio

«

La figura successiva mostra una mappa ipotetica di memoria, con indirizzi che vanno da 00_{16} a FF_{16}, in cui sono evidenziate delle variabili di vario tipo, inclusi i puntatori. Ogni cella di memoria corrisponde a un byte e si presume che l'architettura del microprocessore preveda un accesso in modalità *big endian* (quello più semplice dal punto di vista umano). Si vuole conoscere il risultato della dereferenziazione dei puntatori, secondo quanto richiesto espressamente nella tabella che segue la figura.



Variabile o puntatore dereferenziato	Contenuto
a[1]	4369_{16}
b[0]	
c	
d[0]	0052_{16}
d[0][0]	
d[0][1]	4369_{16}
d[1][0]	
d[1][1]	
d[2][0]	
e[0]	
e[0][0]	
e[0][1]	4369_{16}
e[1][0]	
e[1][1]	
e[2][0]	
f[0]	
f[0][0]	
f[0][0][0]	
f[0][0][1]	4369_{16}
f[0][1][0]	
f[0][1][1]	
f[0][2][0]	

82.16 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.[1]

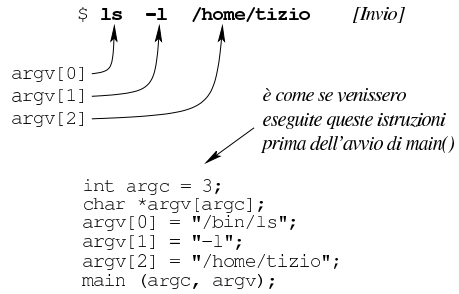
L'esempio seguente mostra in che modo gestire tale array, con la semplice riemissione degli argomenti attraverso lo standard output.

```
#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 82.87. Schematizzazione di ciò che accade alla chiamata della funzione ***main()***, con un esempio.



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**'.

82.17 Puntatori a variabili distrutte

L'esempio seguente potrebbe funzionare, ma contiene un errore di principio.

Listato 82.88. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/vO5J8vzi* , *http://ideone.com/30i0s* .

```
#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.

82.18 Soluzioni agli esercizi proposti

Esercizio	Soluzione
82.1.1	L'espressione multipla '**x = 4, y = 3 * 2**' ha come *lvalue* le variabili ***x*** e ***y***. L'espressione '**y = 3 * x**' ha come *lvalue* la variabile ***y***. L'espressione '**z += 3 * x**' ha come *lvalue* la variabile ***z***. L'espressione '**j=i++ * 5**' ha come *lvalue* le variabili ***j*** e ***i*** (la seconda viene incrementata di una unità dopo aver assegnato il prodotto di ***i*** per 5 alla variabile ***j***).
82.3.1	Per contenere il puntatore alla variabile ***a***, la quale è di tipo '**int**', la variabile ***b*** deve essere di tipo '**int ***'. La variabile ***x***, per essere un puntatore al tipo '**long long int**' si dichiara di tipo '**long long int ***'. La variabile ***z***, essendo un puntatore al tipo '**long int**', può contenere il valore che esprime un indirizzo di memoria, all'interno del quale ci si attende di trovare un dato che si estende quanto richiederebbe un intero di tipo '**long int**'.
82.4.1	1) L'assegnamento corretto potrebbe essere '**i = &j**' oppure '***i = j**', ma non si può sapere quale dei due fosse l'intenzione del programmatore. 2) La variabile ***j*** contiene alla fine il valore 11. 3) L'assegnamento richiederebbe un cast, perché il puntatore dereferenziato ****i*** è equivalente a una variabile di tipo '**long**', mentre ciò che gli viene assegnato è di tipo '**int**': '***i = (long) j**'.
82.5.1	***a*** contiene $4369616F_{16}$. ***b*** contiene $20616D6F_{16}$. ***c*** contiene 7265_{16}. ***d*** contiene $2E00_{16}$. ****i*** contiene $4369616F_{16}$. ****j*** contiene $20616D6F_{16}$. ****k*** contiene $72652E00_{16}$, perché ***k*** punta alla variabile ***c*** estendendosi fino a tutto il contenuto di ***d***. ****l*** contiene $2E000054_{16}$, perché ***l*** punta alla variabile ***d*** estendendosi fino a tutto il contenuto di ***j***. ****m*** contiene 4369_{16}, perché ***m*** punta alla variabile ***a*** estendendosi però solo fino alla sua metà. ****n*** contiene 2061_{16}, perché ***n*** punta alla variabile ***b*** estendendosi però solo fino alla sua metà. ****o*** contiene 43_{16}, perché ***o*** punta al primo byte della variabile ***a***. ****p*** contiene 20_{16}, perché ***p*** punta al primo byte della variabile ***b***.
82.6.1	***i***=2, ***j***=2, ***k***=3 Nella funzione ***f()***, il contenuto dell'area di memoria a cui punta ****x***, corrispondente a ***j***, viene incrementato di una unità dopo che si è svolta la somma; pertanto, il valore restituito dalla funzione è tre (uno+due).

Esercizio	Soluzione

82.6.2	`#include <stdio.h>`
	`int f (int *x, int y)`
	`{`
	`  return ((*x)++ + y);`
	`}`
	`int main (void)`
	`{`
	`  int i = 1;`
	`  int j = 2;`
	`  int k;`
	`  int *l;`
	`  l = &i;`
	`  k = f (l, j);`
	`  printf ("i=%i, j=%i, k=%i\n", i, j, k);`
	`  getchar ();`
	`  return 0;`
	`}`

82.8.1	`unsigned int a[11];`
	`int b[] = { 2, 7, 123 };`
	`int c[7] = { 2, 7, 123 };`

82.8.2	`int a[5];`
	`int i;`
	`…`
	`for (i = 0 ; i < 5 ; i++)`
	`  {`
	`  a[i] = i + 1;`
	`  }`

82.8.3	`int a[5];`
	`int i;`
	`…`
	`for (i = 4 ; i >= 0 ; i--)`
	`  {`
	`  a[i] = i + 1;`
	`  }`
	Al termine, la variabile ***i*** ha il valore -1.

82.9.1	`unsigned int a[11][7];`
	`int b[3][2] = {{2, 7}, {5, 11}, {100, 123}};`
	`int c[7][2] = {{2, 7}, {5, 11}};`

82.9.2	`int a[5][7];`
	`int i;`
	`int j;`
	`…`
	`for (i = 0 ; i < 5 ; i++)`
	`  {`
	`  for (j = 0 ; j < 6 ; j++)`
	`  {`
	`  a[i][j] = (i * 7) + j + 1;`
	`  }`
	`  }`

82.9.3	`int a[5][7];`
	`int i;`
	`int j;`
	`…`
	`for (i = 4 ; i >= 0 ; i--)`
	`  {`
	`  for (j = 7 ; j >= 0 ; j--)`
	`  {`
	`  a[i][j] = (i * 7) + j + 1;`
	`  }`
	`  }`

82.10.1	1) La variabile che rappresenta un array è in sola lettura, perciò non le si può assegnare alcunché. 2) La variabile puntatore ***b*** riguarda il tipo '`long int`', mentre l'array ***a*** si compone di elementi di tipo '`int`', pertanto i puntatori non possono essere dello stesso tipo; tuttavia, anche se non ci fosse questo problema, c'è da osservare che l'array ***a[][]*** è a due dimensioni, restituendo, in questo caso, il puntatore a un'area di memoria lunga cinque volte un intero normale, rendendo comunque incompatibile l'assegnamento alla variabile ***b***; pertanto, si richiede un cast. 3) Il puntatore che si ottiene da '`&a[3]`' si riferisce a un array di cinque elementi di tipo '`int`', pertanto è incompatibile con ***p*** e si richiederebbe eventualmente un cast, oppure si potrebbe togliere l'operatore '`&`', rendendo in questo caso compatibili i puntatori.
82.10.2	1) Viene modificato l'elemento ***a[1][1]***. 2) Viene modificato l'elemento ***a[1][0]***. 3) L'area di memoria a cui si riferisce ***p[35]*** è immediatamente successiva allo spazio occupato dall'array ***a[][]***; infatti, essendo questo composto da 35 elementi, ***p[35]*** si riferisce a un 36-esimo elemento non esistente.
82.12.1	1) Viene modificato l'elemento ***a[1][1]***. 2) Viene modificato l'elemento ***a[1][1]***. In questo caso, il puntatore corrispondente al contenuto della variabile ***p*** non viene modificato, continuando a riferirsi all'inizio dell'array ***a[][]***. 3) Le celle dell'array ***a[][]*** vengono inizializzate con un valore intero da uno a 34. Al termine, il puntatore contenuto nella variabile ***p*** si riferisce all'area di memoria immediatamente successiva all'array ***a[][]***.

Esercizio	Soluzione
82.13.1	***a[]*** è un array di interi, di sette elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore», oltre al codice <*LF*> e allo zero finale. ***b[]*** è un array di interi, di sei elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore», oltre allo zero finale. ***c[]*** è un array di interi, di sei elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore», oltre al codice <*LF*> finale. ***d[]*** è un array di interi, di cinque elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore» e nulla altro. ***e[]*** è un array di caratteri, di sette elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore», oltre al codice <*LF*> e allo zero finale: si tratta di una stringa che se visualizzata porta anche a capo il cursore al termine. ***f[]*** è un array di caratteri, di sette elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore», oltre allo zero finale: si tratta di una stringa che se visualizzata non porta a capo il cursore al termine. ***g[]*** è un array di caratteri, di sette elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore», oltre allo al codice <*LF*> finale: non si tratta di una stringa, perché manca lo zero finale. ***g[]*** è un array di caratteri, di sei elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore» e nulla altro: non si tratta di una stringa, perché manca lo zero finale. ***i[]*** è un array di caratteri, di nove elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore», uno zero e poi le lettere della parola «mio»: può valere come stringa, ma in tal caso si ignora il testo successivo allo zero.
82.13.2	***a[]*** è un array di caratteri che nel corso del programma potrebbe anche contenere una stringa. ***b[]*** è un array di caratteri contenente inizialmente una stringa. Non è possibile assegnare qualcosa direttamente a ***c***, perché si tratta di un puntatore in sola lettura; per cambiare il contenuto dell'array ***c[]*** bisogna invece intervenire cella per cella. ***e*** è un puntatore che può ricevere l'indirizzo iniziale di un array di caratteri; pertanto l'assegnamento è valido ed ***e*** diventa un modo alternativo per fare riferimento alla stringa contenuto nell'array ***d[]***. Dopo l'assegnamento, ***f*** è un puntatore a un carattere che contiene il valore corrispondente alla lettera «c»; tuttavia può essere usato in qualità di stringa, contenente la parola «ciao». Dopo l'assegnamento, ***g*** punta all'inizio di una stringa che rappresenta la parola «amore»; per converso, la stringa che rappresentava la parola «ciao» continua a occupare spazio in memoria, ma risulta irraggiungibile. Con l'assegnamento di ****h***, si vorrebbe sostituire l'iniziale della parola «ciao» con una maiuscola; tuttavia, ciò non è ammissibile, perché l'area di memoria che contiene inizialmente la stringa «ciao» dovrebbe essere in sola lettura. Con l'incremento di ***i***, questo puntatore rappresenta una stringa, contenente però solo la parola «iao».
82.14.1	***a[1]*** contiene 4369_{16}. ***b[0]*** contiene 2061_{16}. ***c*** contiene 7265_{16}. ***d[0]*** contiene 0052_{16}. ****d[0]*** contiene 0000_{16} e corrisponde a ***a[0]***. ****e*** contiene 0052_{16} e corrisponde a ***d[0]***. *****e*** contiene 0000_{16} e corrisponde a ***a[0]***, così come a ****d[0]***. ****f*** contiene $005E_{16}$ e corrisponde a ***e***. *****f*** contiene 0052_{16} e corrisponde a ***d[0]***, così come a ****e***. ******f*** contiene 0000_{16} e corrisponde a ***a[0]***, così come a ****d[0]*** e a *****e***.
82.15.1	***a[1]*** contiene 4369_{16}. ***b[0]*** contiene 2061_{16}. ***c*** contiene 7265_{16}. ***d[0]*** contiene 0052_{16}. ***d[0][0]***, ovvero ****d[0]***, contiene 0000_{16} e corrisponde a ***a[0]***. ***d[0][1]*** contiene 4369_{16} e corrisponde a ***a[1]***. ***d[1][0]*** contiene 2061_{16} e corrisponde a ***b[0]***. ***d[1][1]*** contiene $6D6F_{16}$ e corrisponde a ***b[1]***. ***d[2][0]*** contiene 7265_{16} e corrisponde a ***c***. ***e[0]*** contiene 0052_{16} e corrisponde a ***d[0]***. ***e[0][0]***, ovvero ****e[0]***, contiene 0000_{16} e corrisponde a ***a[0]***, così come a ***d[0][0]***. ***e[0][1]*** contiene 4369_{16} e corrisponde a ***a[1]***, così come a ***d[0][1]***. ***e[1][0]*** contiene 2061_{16} e corrisponde a ***b[0]***, così come a ***d[1][0]***. ***e[1][1]*** contiene $6D6F_{16}$ e corrisponde a ***b[1]***, così come a ***d[1][1]***. ***e[2][0]*** contiene 7265_{16} e corrisponde a ***c***, così come a ***d[2][0]***. ***f[0]*** contiene $005E_{16}$ e corrisponde a ***e***. ***f[0][0]*** contiene 0052_{16} e corrisponde a ***d[0]*** e a ***e[0]***. ***f[0][0][0]***, ovvero ******f***, contiene 0000_{16} e corrisponde a ***a[0]***, così come a ***d[0][0]*** e a ***e[0][0]***. ***f[0][0][1]*** contiene 4369_{16} e corrisponde a ***a[1]***, così come a ***d[0][1]*** e a ***e[0][1]***. ***f[0][1][0]*** contiene 2061_{16} e corrisponde a ***b[0]***, così come a ***d[1][0]*** e a ***e[1][0]***. ***f[0][1][1]*** contiene $6D6F_{16}$ e corrisponde a ***b[1]***, così come a ***d[1][1]*** e a ***e[1][1]***. ***f[0][2][0]*** contiene 7265_{16} e corrisponde a ***c***, così come a ***d[2][0]*** e a ***e[2][0]***.

[1] 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.

Indice analitico del volume