Capitolo 82

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?

Codice	Questione

`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:

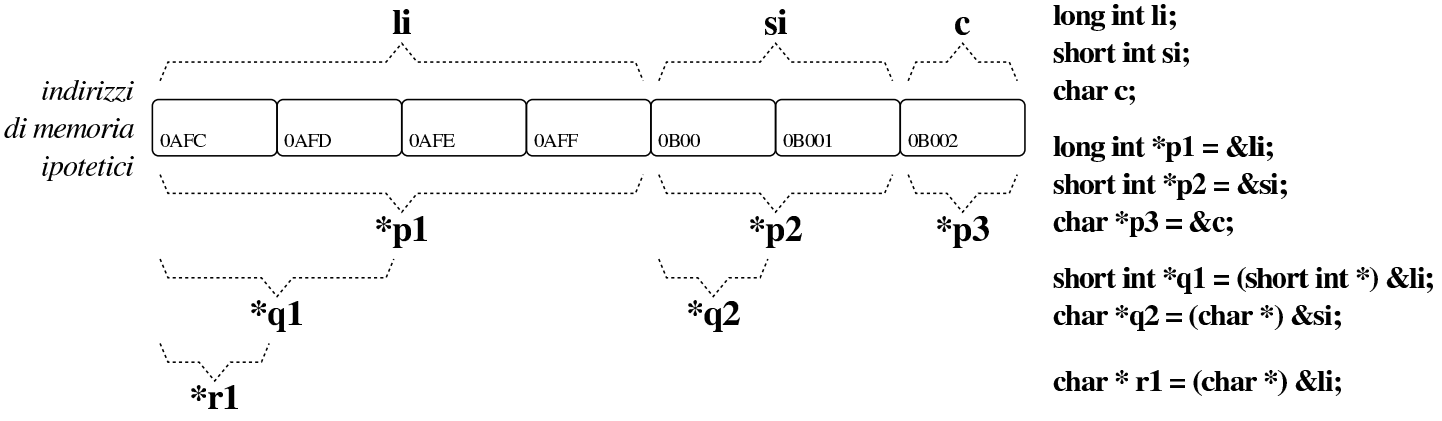
```
char *pc;
int  *pi;
…
pi = pc;   // I due puntatori si riferiscono a dati di tipo
           // differente!
…
```

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.



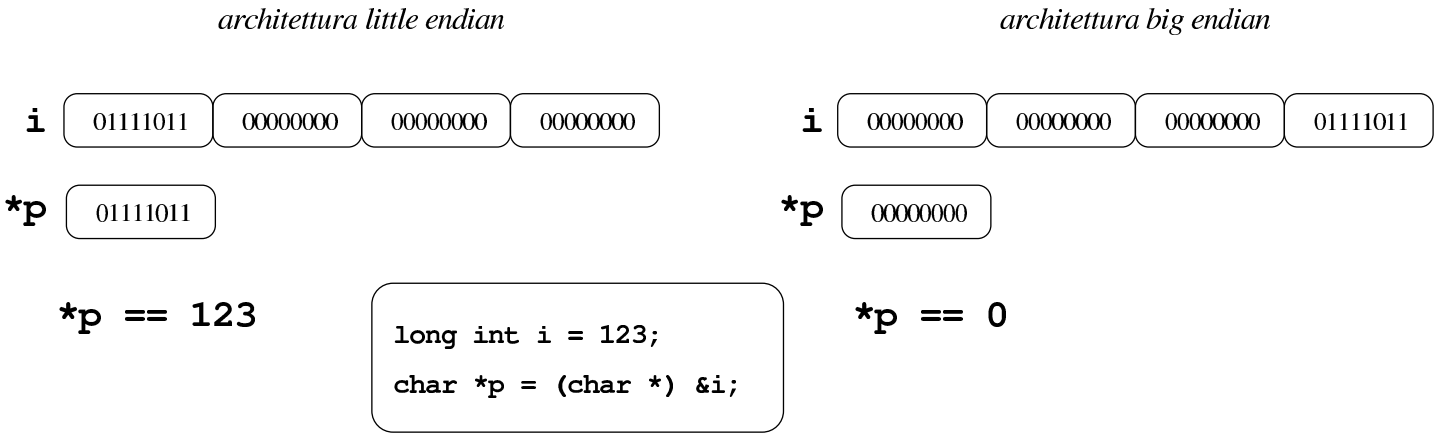
L'esempio seguente rappresenta un programma completo che ha lo scopo di determinare se l'architettura dell'elaboratore è di tipo *big endian* o di tipo *little endian*. Per capirlo si dichiara una variabile di tipo '**`long int`**' che si intende debba essere di rango superiore rispetto al tipo '**`char`**', 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 varia-

bile 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* .

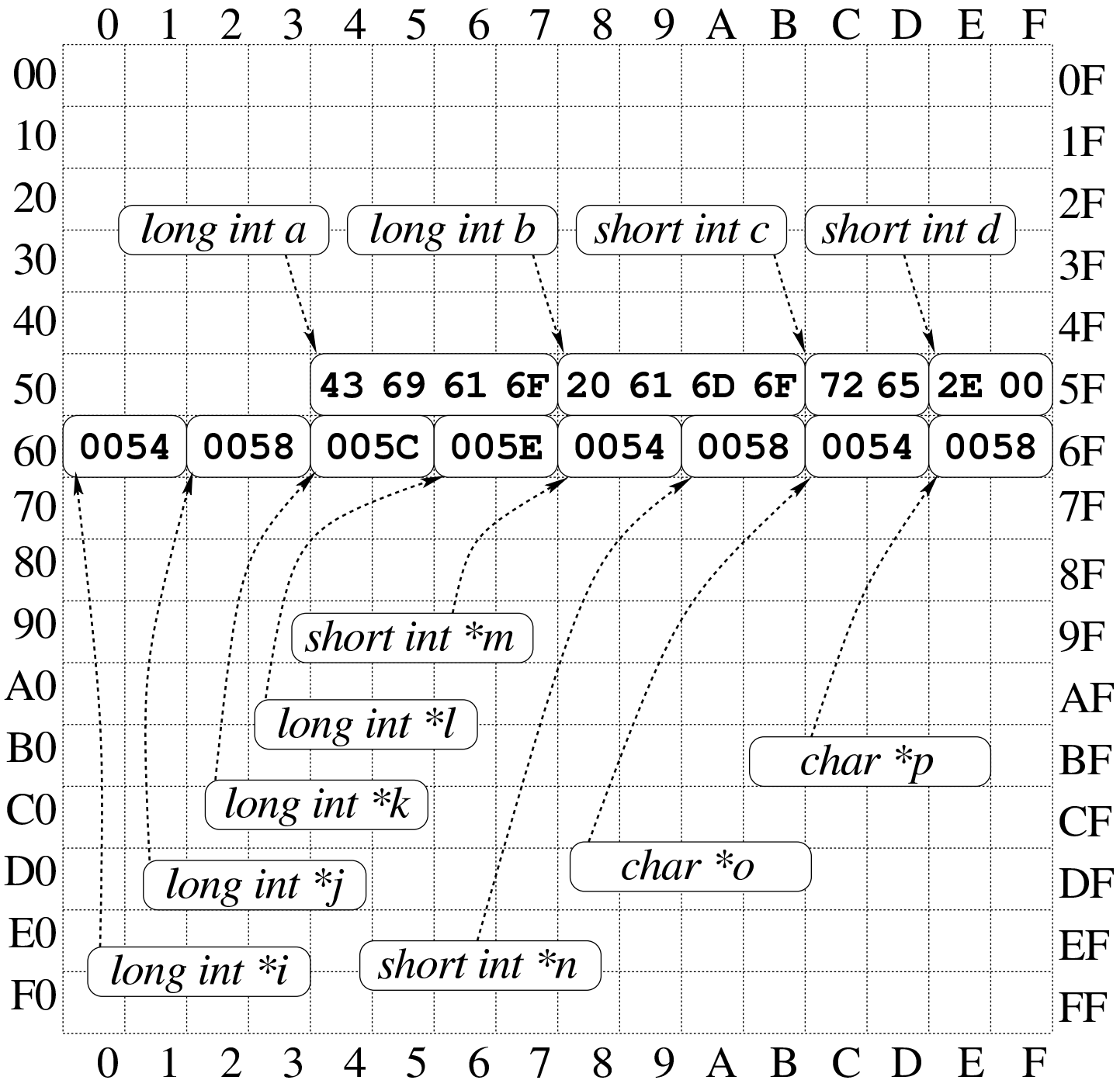
```
#include <stdio.h>
int main (void)
{
    long int i = 123;
    char *p = (char *) &i;
    if (*p == 123)
      {
        printf ("little endian\n");
      }
    else
      {
        printf ("big endian\n");
      }
    getchar ();
    return 0;
}
```

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	

Variabile o puntatore dereferenziato	Contenuto
c	
d	
****i***	$4369616F_{16}$
****j***	
****k***	$72652E00_{16}$
****l***	
****m***	
****n***	
****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]);
```

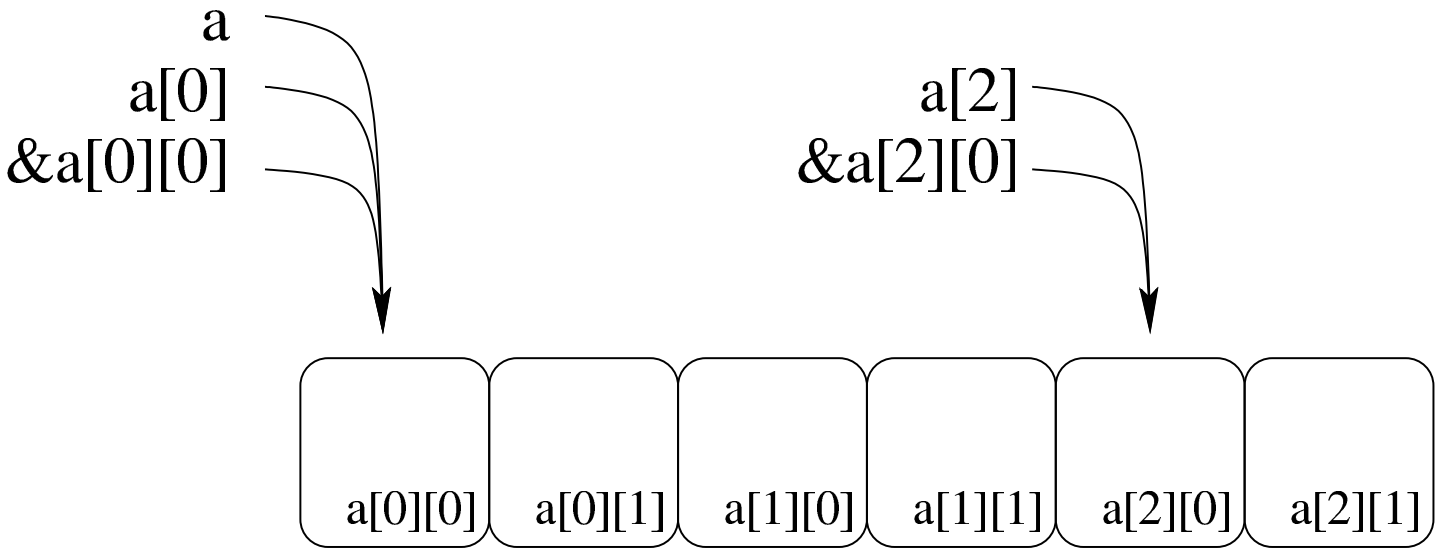
```
  getchar ();
  return 0;
}
```

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;`	

Codice problematico	Spiegazione

`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 fun-

zione 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'esem-

pio 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.

Codice di escape	Descrizione
`\\`	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?

Codice	Richiesta

…
`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?

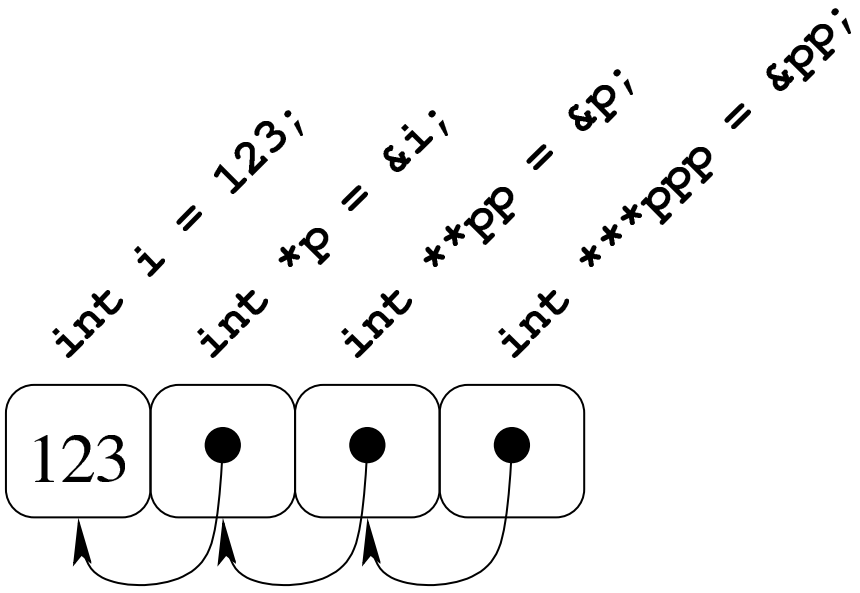
…
`char *i = "ciao"`
…
`i++;`
	Al termine, cosa rappresenta ***i***?

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* .

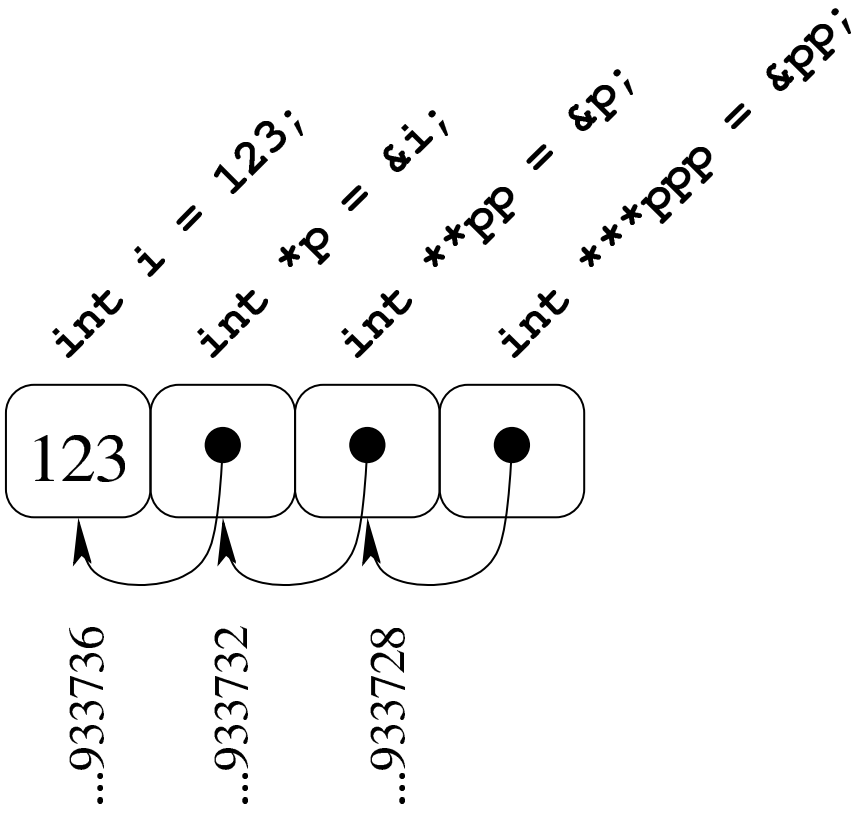
```
#include <stdio.h>
int main (void)
{
  int i = 123;
  int *p = &i;       // Puntatore al tipo "int".
  int **pp = &p;     // Puntatore di puntatore al tipo "int".
  int ***ppp = &pp; // Puntatore di puntatore di puntatore
                     // al tipo "int".

  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:



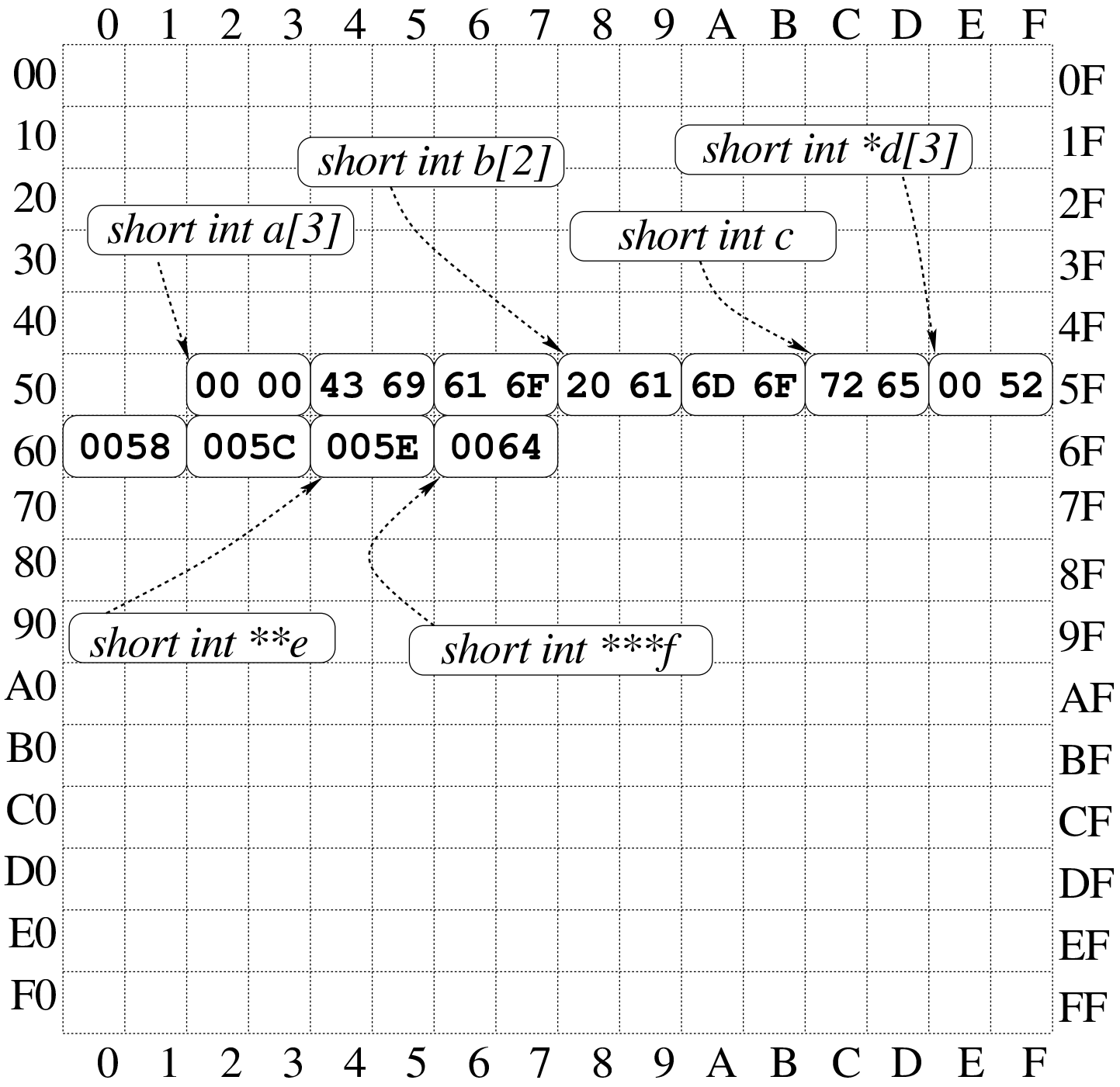

```
i == *p

i == **pp

i == ***ppp
```

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]	

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

82.15 Puntatori a più dimensioni

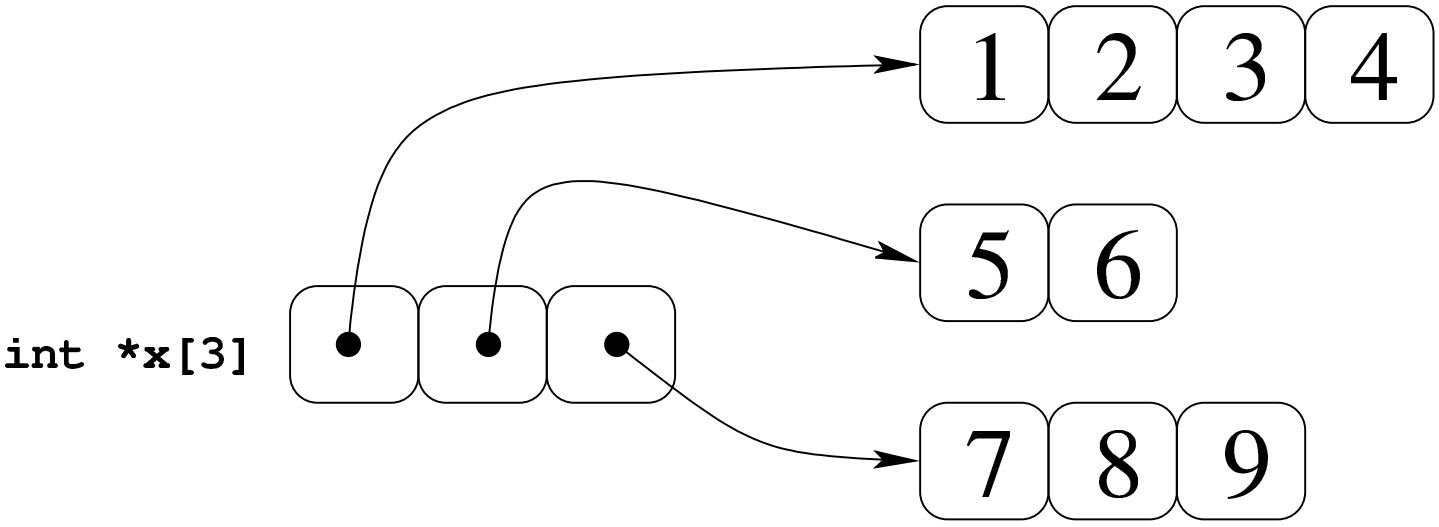
Un array di puntatori consente di realizzare delle strutture di dati ad albero, non più uniformi come invece devono essere gli array a più dimensioni consueti. L'esempio seguente mostra la dichiarazione di tre array di interi, con una quantità di elementi disomogenea, e la successiva dichiarazione di 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;
}
```

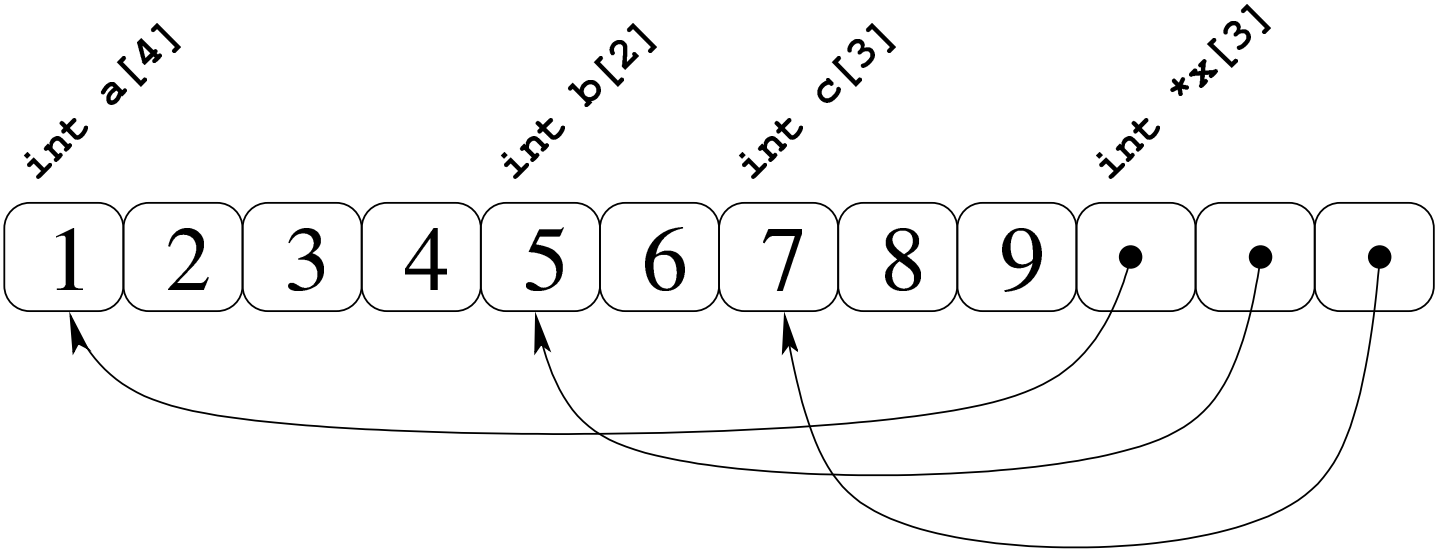
La figura successiva dovrebbe facilitare la comprensione del senso dell'array di puntatori. Come si può osservare, per accedere agli elementi degli array a cui puntano quelli di ***x*** è necessario dereferenziare gli elementi. Pertanto, '***x*`[0]`' corrisponde al contenuto del primo elemento del primo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

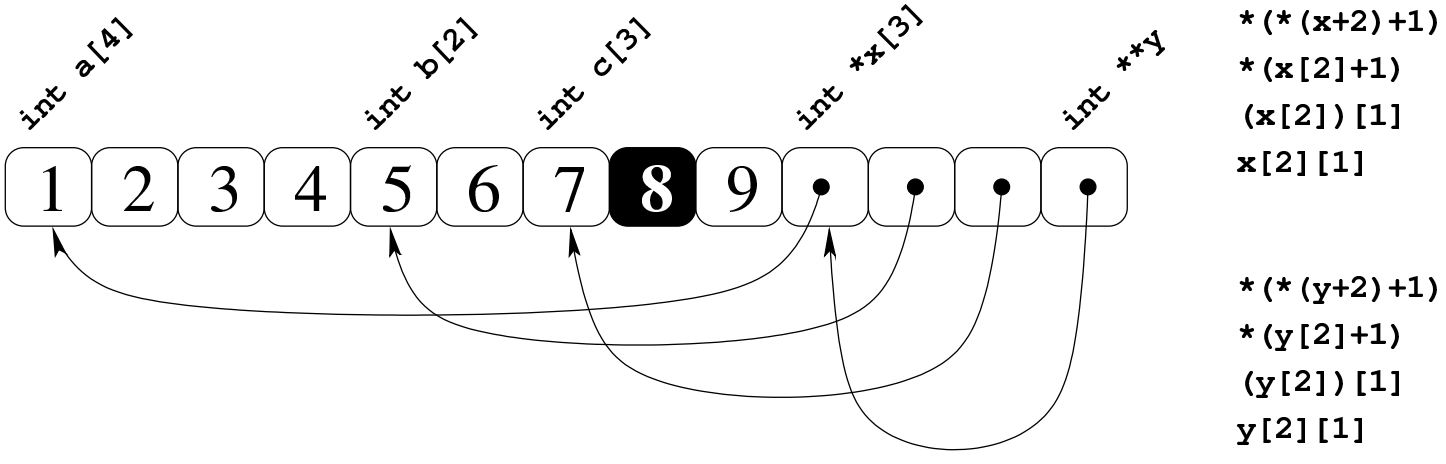
```
#include <stdio.h>

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.



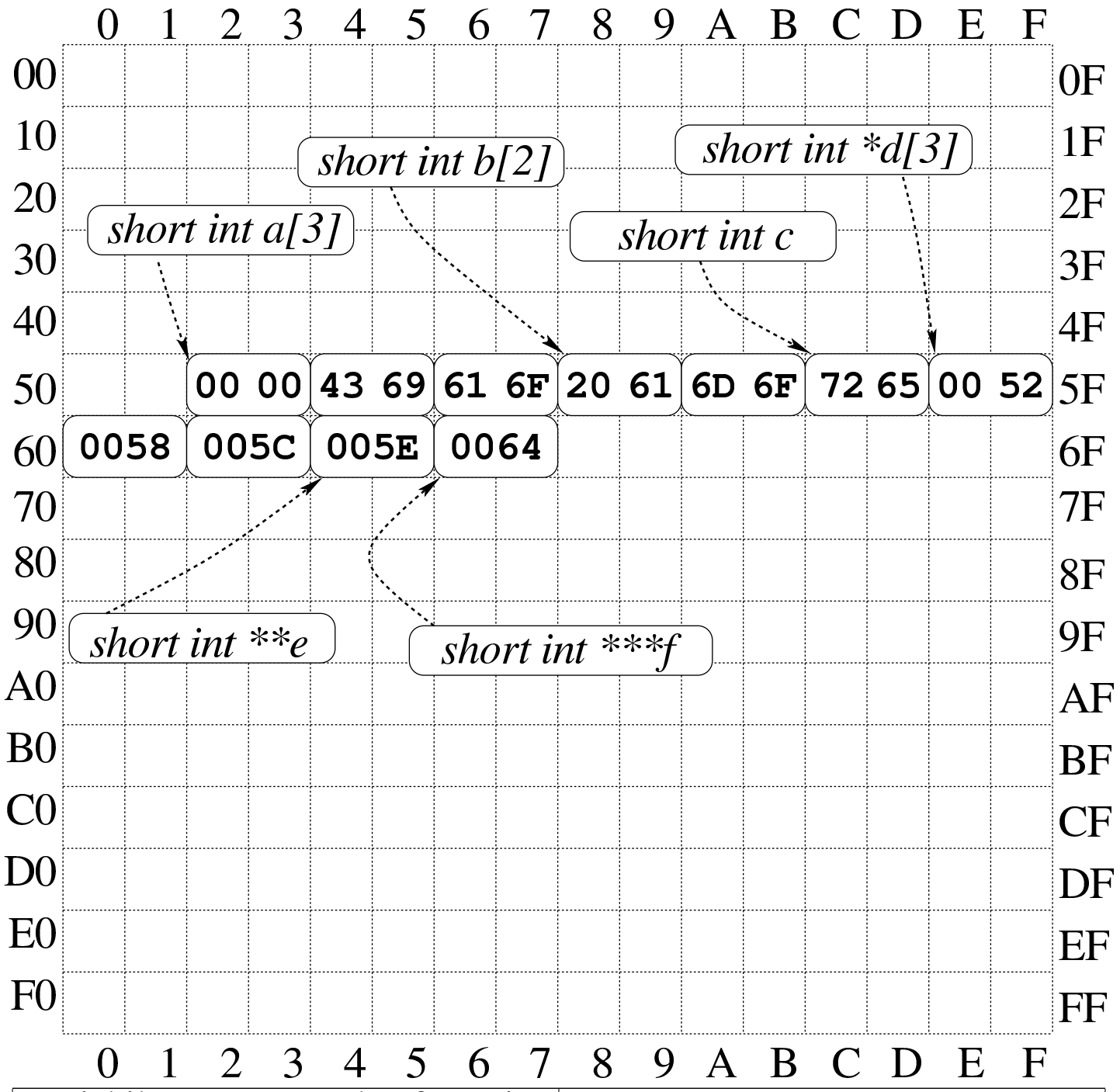
Seguendo lo stesso ragionamento si possono gestire strutture ad albero più complesse, con più livelli di puntatori, ma qui non vengono proposti esempi di questo tipo.

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]	

Variabile o puntatore dereferenziato	Contenuto
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]	

Variabile o puntatore dereferenziato	Contenuto
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]

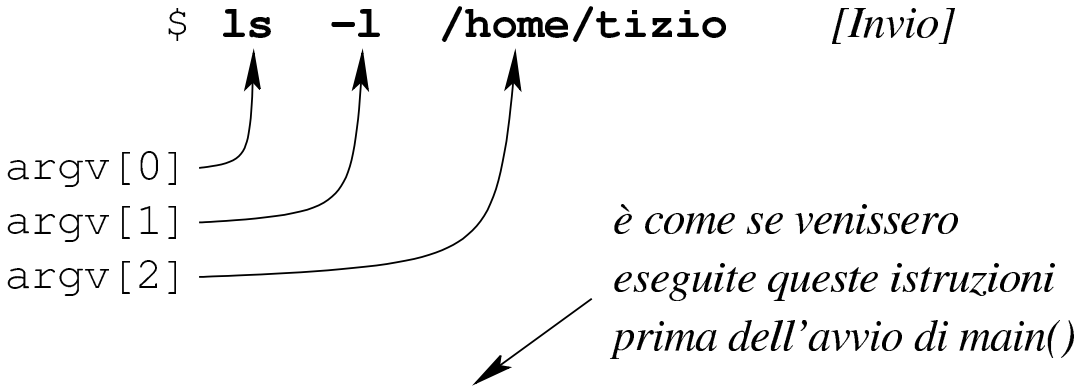
L'esempio seguente mostra in che modo gestire tale array, con la semplice riemissione degli argomenti attraverso lo standard output.

```
#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.




```
int argc = 3;
char *argv[argc];
argv[0] = "/bin/ls";
argv[1] = "-l";
argv[2] = "/home/tizio";
main (argc, argv);
```

Chi è abituato a utilizzare linguaggi di programmazione più evoluti del C, può trovare strano che non si possa scrivere '**`main (int argc, char argv[][])`**' e usare di conseguenza l'array. Il motivo per cui ciò non è possibile dipende dal fatto che gli array a più dimensioni sono ottenuti attraverso sottoinsiemi uniformi del tipo dichiarato, così, in questo caso le stringhe dovrebbero essere della stessa dimensione, ma evidentemente ciò non corrisponde alla realtà. Inoltre, la dichiarazione della funzione dovrebbe contenere le dimensioni dell'array che non possono essere note. Pertanto, un array formato da stringhe diseguali, può essere ottenuto solo come array di puntatori al tipo '**`char`**'.

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`'.

<table>
<tr><th>Esercizio</th><th>Soluzione</th></tr>
<tr><td>82.5.1</td><td>***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***.</td></tr>
<tr><td>82.6.1</td><td>***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).</td></tr>
<tr><td>82.6.2</td><td><pre>
#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;
}
</pre></td></tr>
</table>

Esercizio	Soluzione

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;`
	`  }`
	`  }`

Esercizio	Soluzione

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.

Esercizio	Soluzione
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***.

Esercizio	Soluzione
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.