```
1 int k[2], *pi;
2 k[0] = 137;
3 pi = k;
4 *pi++ = 100;      /* incrementa il puntatore */
5 k[0] = 137;
6 pi = k;
7 (*pi)++;          /* incrementa la variabile */
```

Listato 6.7 Priorità degli operatori applicati a un puntatore.

Se ad esempio l'indirizzo dell'elemento `k[0]` vale `0xbffff7a0`, la variabile `pi` alla riga 3 assume questo valore. Alla riga 4 `k[0]` vale 100 mentre il puntatore `pi` è incrementato di una posizione (quattro byte) e vale ora `0xbffff7a4`. Infine alla riga 7, la variabile a cui `pi` punta, cioè `k[0]`, è incrementata da 137 a 138; il puntatore `pi` continua invece a valere `0xbffff7a0`.

6.4 Considerazioni di efficienza

Accedere agli elementi di un array mediante i puntatori può sembrare meno "naturale" che farlo con i consueti indici. Una buona ragione per usare i puntatori nella navigazione degli elementi di un array consiste nella maggiore efficienza di calcolo. Nel codice

```
double data[10] = {0}, a;
data[3] = 3.14;
a = data[3];
```

ogni volta che facciamo riferimento a un elemento dell'array `data` attraverso un suo indice il compilatore deve calcolare la posizione in memoria di `data[3]`. Per fare questo deve leggere il valore dell'indirizzo nella variabile `data`, calcolare il corretto incremento con una moltiplicazione (3×8 byte) e infine aggiungere l'incremento a `data`. In questo caso, se usiamo i puntatori non c'è nessuna differenza nel numero delle operazioni:

```
double data[10] = {0}, a;
*(data + 3) = 3.14;
a = *(data + 3);
```

Le cose cambiano se abbiamo un ciclo che percorre sequenzialmente tutti gli elementi di un array; in questo caso usare gli indici è meno efficiente che usare i puntatori. Per mostrare questo fatto consideriamo un array di cento elementi ed esaminiamolo in un ciclo alla ricerca di elementi con valore negativo. Il Listato 6.8 contiene il codice con indirizzamento degli elementi dell'array.

Nell'esecuzione di questo ciclo, oltre alle operazioni nel `for`, è necessario calcolare l'indirizzo di `data[i]` per ciascuna esecuzione del controllo `if (data[i] < 0.)`; stiamo quindi eseguendo 100 addizioni (`data + i`) e 100 moltiplicazioni (per ogni addizione `i`$\times$8 byte).

Estratto dal libro **PROGRAMMAZIONE SCIENTIFICA** di
L. M. Barone, E. Marinari, G. Organtini e F. Ricci-Tersenghi
I diritti di riproduzione e memorizzazione elettronica degli estratti appartengono a Pearson Education Italia S.r.l. e sono riservati per tutti i paesi. La stampa di essi è concessa gratuitamente solo a titolo personale. Ogni altro uso è vietato.

```
1 double data[100];
2 int i;
3 ... codice omesso ...
4 for (i = 0; i < 100; i++) {
5   if (data[i] < 0.) {
6     printf("Trovato valore negativo %f \n", data[i]);
7   }
8 }
```

Listato 6.8 Accedere a un array per indici.

Vediamo invece il Listato 6.9. In questo esempio l'aritmetica del ciclo resta invariata ma in ogni operazione di controllo `if (*pd < 0.)` non c'è alcun calcolo da fare per accedere all'elemento corrente dell'array `*pd`. Abbiamo anche introdotto una variabile di appoggio `lastpd` per evitare di calcolare 100 volte la somma `data + 100` nella condizione di uscita del ciclo `for` alla riga 4.

```
1 double data[100], *pd, *lastpd;
2 ... codice omesso ...
3 lastpd = data + 100;
4 for (pd = data; pd < lastpd; pd++) {
5   if (*pd < 0.) {
6     printf("Trovato valore negativo %f \n", *pd);
7   }
8 }
```

Listato 6.9 Accedere a un array con i puntatori.

La conclusione è che conviene svolgere operazioni sequenziali su array, in particolare su array lunghi, usando i puntatori all'array piuttosto che gli indici espliciti dell'array stesso. Paghiamo però un prezzo per questa maggiore efficienza: il codice nel Listato 6.9 è meno chiaro e, in generale, l'uso dei puntatori si presta di piú a errori di programmazione.

6.5 Array multidimensionali e puntatori

Nel Paragrafo 5.3 abbiamo studiato gli array multidimensionali; vediamo come si può accedere agli elementi di tali array usando i puntatori. Ricordiamo che un array a due indici `a[4][7]` (di tipo `double` per fissare le idee) occupa in memoria 4×7 posizioni contigue e quindi, per certi versi, si potrebbe considerare come un array unidimensionale di lunghezza 28. L'ordine nella memoria degli elementi dell'array segue la regola per cui l'indice piú a sinistra di un array multidimensionale cresce piú lentamente; si può visualizzare questo comportamento pensando a come si muovono le cifre del contachilometri di un'automobile.

Estratto dal libro **PROGRAMMAZIONE SCIENTIFICA** di
L. M. Barone, E. Marinari, G. Organtini e F. Ricci-Tersenghi
I diritti di riproduzione e memorizzazione elettronica degli estratti appartengono a Pearson Education Italia S.r.l. e sono riservati per tutti i paesi. La stampa di essi è concessa gratuitamente solo a titolo personale. Ogni altro uso è vietato.

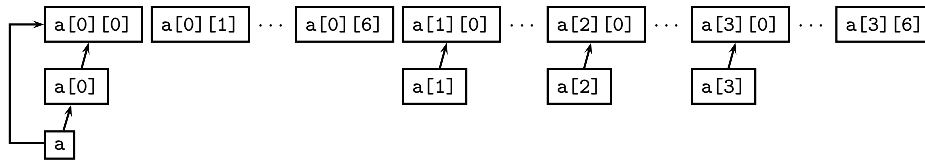

Figura 6.1 Un array bidimensionale schematizzato come array di array.

Quindi in memoria dopo l'elemento `a[0][0]` abbiamo `a[0][1]`, poi `a[0][2]` e cosí via. Tuttavia possiamo considerare la dichiarazione di un array bidimensionale dal punto di vista della priorità dell'operatore `[]`, che compare due volte ed è valutato come sempre da sinistra verso destra. Possiamo pensare all'array `a[4][7]` come a un array di 4 elementi `a[4]`, in cui ciascun elemento è un array di 7 elementi. Se adottiamo questo punto di vista appare chiaro che il nome `a` oppure `a[0]` è un puntatore al primo elemento dell'array bidimensionale, cioè al primo *sottoarray* di lunghezza 7; quindi `a + 1` o `a[1]` punta al secondo sottoarray di lunghezza 7 e cosí via. La Figura 6.1 mostra questo schema. L'espressione `*(a + 1)` è uguale al contenuto della locazione di memoria a cui punta `a + 1`; questa locazione contiene l'indirizzo del secondo sottoarray, quindi `*(a + 1)` *non è un elemento dell'array ma è ancora un puntatore.* E nello stesso modo è ancora un puntatore l'espressione `*(a + 1) + 1`, che è l'indirizzo del secondo elemento del secondo sottoarray. E infine l'espressione `*(*(a + 1))` dereferenzia il puntatore `*(a + 1)` restituendo il contenuto della locazione di memoria puntata: tale contenuto corrisponde al primo elemento del secondo sottoarray, che, in termini di indici, possiamo scrivere `a[1][0]`, e nello stesso modo l'espressione `*(*(a + 1) + 3)` è uguale ad `a[1][3]`.

Come facciamo quindi a percorrere un array multidimensionale con un puntatore? Dobbiamo rendere esplicito il punto di partenza, cioè usare `&a[0][0]`, oppure `*a` e non `a`, come indirizzo iniziale, e poi tenere conto della mappa di memoria, cioè del fatto che l'indice piú a destra varia piú velocemente, quindi il secondo piú a destra e cosí via. Il codice che segue spiega quale algoritmo occorre seguire. Possiamo scrivere

```
double a[4][7];
*(&a[0][0] + 1) = 3.14;     /* pone a[0][1] = 3.14 */
*(&a[0][0] + 7) = 6.28;     /* pone a[1][0] = 6.28 */
*(&a[0][0] + 7 + 1) = 2.73; /* pone a[1][1] = 2.73 */
```

oppure

```
double a[4][7];
*(*a + 1)   = 3.14;     /* pone a[0][1] = 3.14 */
*(*(a + 1)) = 6.28;     /* pone a[1][0] = 6.28 */
*(*(a + 1) + 1) = 2.73; /* pone a[1][1] = 2.73 */
```

In generale, avendo dichiarato un array multidimensionale `a[N1][N2]`, possiamo derefe-

Estratto dal libro **PROGRAMMAZIONE SCIENTIFICA** di
L. M. Barone, E. Marinari, G. Organtini e F. Ricci-Tersenghi
I diritti di riproduzione e memorizzazione elettronica degli estratti appartengono a Pearson Education Italia S.r.l. e sono riservati per tutti i paesi. La stampa di essi è concessa gratuitamente solo a titolo personale. Ogni altro uso è vietato.