8

PUNTATORI, VETTORI, STRUTTURE

Questo è un capitolo da prendere con le molle: i puntatori e le strutture sono degli strumenti di notevole utilità, ma richiedono una certa dose di attenzione e di familiarità.

La discussione potrebbe diventare lunga e contorta, ma non intendo dedicare troppo spazio all'argomento: provare e imparare dai propri errori è più utile che leggere pagine e pagine senza afferrarne il significato. Cercherò dunque di isolare pochi concetti che a mia esperienza sono essenziali, e di illustrarli con qualche esempio. Poi lascerò il resto alla tua immaginazione ed alla tua pazienza.

8.1 Puntatori e vettori

Abbiamo già avuto occasione, sia pur sporadicamente, di far uso dei puntatori. In effetti, più di una volta ho insistito molto nel precisare che un *dato* e *l'indirizzo* a cui si trova il dato non sono la stessa cosa. Ebbene, un puntatore è semplicemente una variabile che contiene un indirizzo.

Come primo esempio ricorda come si scrive una funzione che restituisca un valore tramite un argomento. Torna a rivedere la funzione `minmax` del paragrafo 3.3.6. Per l'argomento abbiamo usato una dichiarazione della forma

```
double *vmin,*vmax;
```

e ti ho spiegato che quell'asterisco significa che `vmin` e `vmax` contengono degli indirizzi.

Come secondo esempio rileggi il paragrafo 5.3.2, dove ti ho spiegato l'uso della funzione `malloc`. Ho usato le istruzioni

```
double *vet;
...
vet=malloc(...);
```

ed anche qui ti ho spiegato che `vet` è una variabile che contiene un indirizzo.

Ebbene, `vmin` e `vmax` e `vet` sono dei *puntatori*; il loro valore è l'indirizzo di un dato o di un vettore.

In altre parole, i puntatori non hanno molto di diverso dagli altri tipi di dati che hai visto fin qui: quello che li caratterizza è l'uso che ne puoi fare, ed in particolare il tipo di operazioni che puoi compiere su di essi.

8.1.1 Qualche richiamo

Qui è essenziale padroneggiare il significato e l'uso degli operatori unari `*` e `&`. In realtà già lo conosci, ma lasciami richiamare le informazioni essenziali.

(i) Se `x` è un dato (di qualunque tipo, incluso il tipo puntatore), allora `&x` è l'indirizzo di memoria a cui si trova `x`;

(ii) Se `adrx` è un puntatore, `*adrx` è il dato che si trova all'indirizzo `adrx`.

(iii) Per definire un indirizzo si usa la dichiarazione
$\langle tipo \rangle$ `*`$\langle nome \rangle$`;`
il $\langle tipo \rangle$ si riferisce al tipo di dato in questione; l'asterisco significa che il $\langle nome \rangle$ non fa riferimento al dato, ma al suo indirizzo.

Tieni ben presenti queste tre regole, e capirai meglio tutto il resto. In particolare ricorda la terza regola, per quanto ti sembri strana: imparerai tra breve che al compilatore C non basta sapere che una variabile è un puntatore; vuole sapere anche a *cosa* punta, e di questa informazione fa ampio uso.

8.1.2 L'aritmetica degli indirizzi

L'aritmetica degli indirizzi sta alla base della gestione di tabelle, o arrays, o vettori, o matrici. Procediamo, al solito, con un esempio. Supponiamo che tu voglia trasferire il contenuto di un vettore `vet1` al vettore `vet2`, senza cambiare nulla. Naturalmente dovrai conoscere la lunghezza del vettore, che supporrò essere `n`. Facile, dirai, e scriverai:

```
int j;
for(j=0; j<n; j++) vet2[j]=vet1[j];
```

Giusto, e ti ricorderai anche, come ti ho spiegato a suo tempo, che la scrittura `vet1[j]` significa: prendi l'indirizzo iniziale del vettore `vet1`, incrementalo di `j` volte la lunghezza di ciascun dato di `vet1`, e otterrai l'indirizzo del dato su cui devi agire.

Ebbene, nel linguaggio C l'indirizzo di `vet1[j]` si indica semplicemente con `vet1+j`. Così potrai scrivere, indifferentemente, `vet1[j]` o `*(vet1+j)`.

Lasciami esporre tutto questo in modo generale. Se `adr` è un puntatore ad un tipo assegnato di dato, l'espressione `adr+j`, dove `j` è un dato di tipo intero, significa: l'indirizzo `adr` incrementato di `j` volte la lunghezza del dato. Qui vedi perché il compilatore C sia tanto interessato a sapere a cosa effettivamente si riferisca un puntatore: se il dato è di tipo `char` il puntatore `adr` verrà incrementato di `1`; se il dato è `double` il puntatore `adr` verrà incrementato di `sizeof(double)`, qualunque cosa essa sia sulla tua macchina, e così via. Anche le espressioni `++adr`, `adr++`, `--adr` o `adr--` hanno significato: si tratta semplicemente di incrementare o decrementare di 1 un puntatore — nulla di sostanzialmente diverso da ciò che abbiamo visto per gli interi, salvo il fatto che il risultato dipende dal tipo di dato indirizzato dal puntatore.

È possibile anche confrontare dei puntatori: se `adr1` e `adr2` sono puntatori l'espressione `adr1 < adr2` darà come risultato *vero* se l'indirizzo `adr1` precede `adr2`, o *falso* in caso contrario. Analogo significato hanno le espressioni `adr1 <= adr2`, `adr1 == adr2`, `adr1 >= adr2`, `adr1 > adr2` o `adr1 != adr2`. Con qualche cautela, è possibile anche calcolare la differenza tra indirizzi: `adr1 - adr2` restituisce un intero che rappresenta l'incremento da assegnare ad `adr2` per ottenere l'indirizzo `adr1`. La cautela è giustificata perché nel calcolare la differenza è essenziale il fatto che `adr1` ed `adr2` puntino allo stesso tipo di dato, e che la distanza tra gli indirizzi sia un multiplo della lunghezza del dato. [1] Non sono ammesse invece espressioni del tipo `adr1 + adr2` o `adr1 | adr2`, o altre espressioni arimetiche o logiche a cui non è evidentemente possibile attribuire un significato. [2]

Quello che to ho detto non è poi tanto astratto. Ti faccio un esempio apparentemente inutile ed uno di una certa utilità.

Il primo esempio consiste nel riscrivere le istruzioni che fanno la copia di un vettore usando gli indirizzi. Oltre a quello che ti ho mostrato poco fa

[1] Probabilmente ti verrà spontanea una domanda: ma che differenza c'è tra un puntatore ed un intero? Come rappresentazione di memoria, praticamente nessuna: un puntatore è un numero binario contenuto in una cella di una certa lunghezza. Quale sia questa lunghezza, è faccenda che dipende dall'architettura del processore; tipicamente si tratterà di 32 o 64 bit, e comunque la potrai conoscere facendoti stampare il valore di `sizeof(void *)`. Questo fatto ha dato origine, tra i programmatori, all'abitudine di trattare puntatori ed interi come fossero la stessa cosa. Di fatto la differenza è rilevante proprio perché un puntatore è caratterizzato anche dal tipo di dato che indirizza, e questo ha conseguenze non banali sull'aritmetica. I compilatori più recenti tendono a mettere in evidenza questa differenza segnalando regolarmente ogni conversione da puntatore ad intero o viceversa, logicamente non corretta.

[2] Pensa ad un puntatore come ad un segnale che indica una posizione; lo potrai traslare, e questo è il significato della somma di un puntatore con un intero, ma non potrai assegnare un significato decente alla somma o al prodotto di due segnali. Un caso analogo – se tieni ben conto del corretto significato delle espressioni che scrivi – si presenta per i caratteri nella rappresentazione ASCII. L'espressione `'A'+040` ha significato se la pensi come la posizione del carattere `'A'` traslata in avanti di `040` nella rappresentazione ASCII; se ricordi che `040` sta per la rappresentazione ottale di un numero intero vedrai subito che questa è proprio la conversione da maiuscolo in minuscolo. Non ha molto senso invece scrivere `'A'+'B'`, a meno che tu non usi `'B'` come un modo un po' bizzarro per scrivere il valore intero `0102`. Tanto meno ha senso scrivere `'A'+3.141592` o `'A'*'B'`.

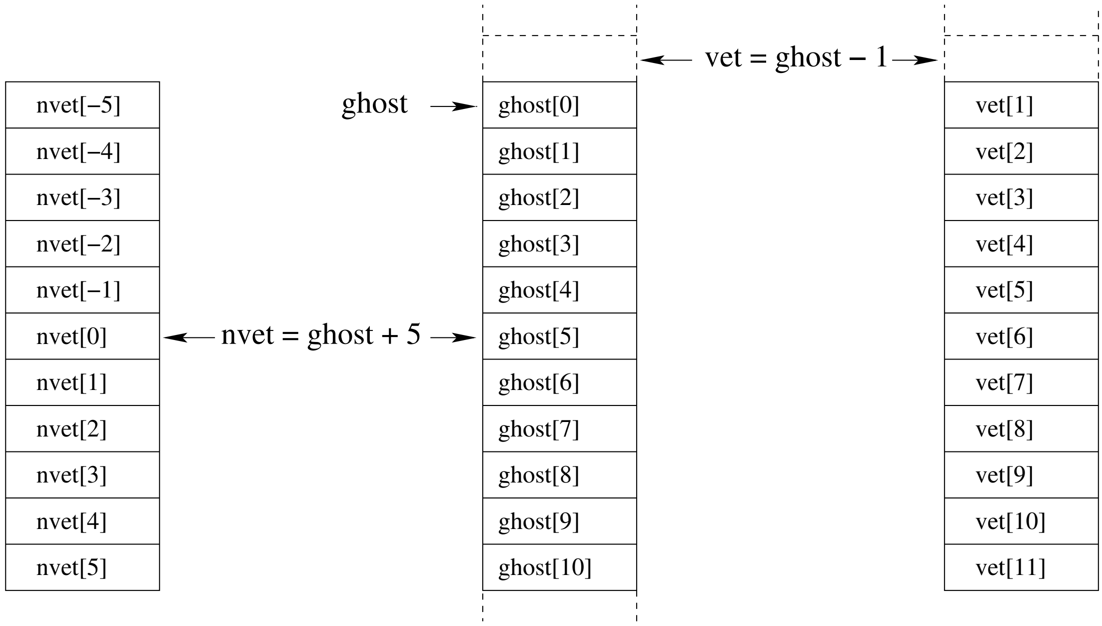


Figura 8.1. Uso dell'aritmetica degli indirizzi per traslare gli indici dei vettori. Se il vettore `ghost` ha 11 elementi, il vettore `nvet` può utilizzare indici tra -5 e 5, mentre il vettore `vet` ha indici tra 1 e 11.

ci sono almeno altri due modi per ottenere lo stesso risultato. Il primo modo è in pratica una riscrittura del precedente:

```
int j;
for(j=0; j<n; j++) *(vet2+j) = *(vet1+j);
```

Il secondo modo è più misterioso, ma evita l'uso di un indice. Eccoti il frammento di codice (suppongo che `vet1` e `vet2` siano di tipo `double`):

```
double *adr1,*adr2,*adrmax;
adrmax=(adr1=vet1)+n; adr2=vet2;
while(adr1<adrmax) *adr2++ = *adr1++;
```

Per chi non è abituato all'uso degli indirizzi sembra una scrittura complicata, ma il risultato è lo stesso. Un solo commento: l'espressione `*adr2++` significa: usa l'indirizzo `adr2` per accedere al dato, poi incrementa l'indirizzo.

Il secondo esempio, quello di una certa utilità, risponde ad una domanda che spesso viene fatta dai programmatori – soprattutto quelli che conoscono già altri linguaggi: ma gli indici dei vettori devono iniziare per forza da zero? non posso proprio usare un indice negativo? La risposta è semplice: puoi far partire l'indice di un vettore da dove ti pare, pur di usare l'aritmetica degli indirizzi e di sfruttare appieno il meccanismo di indirizzamento degli elementi di un vettore. Il metodo è illustrato in figura 8.1. Eccoti il frammento di codice che corrisponde alla situazione della figura:

```
double ghost[11],*vet,*nvet;
vet=ghost-1;  nvet=ghost+5;
```

Se ho bisogno di un vettore `double` di 11 elementi, la dichiarazione `double ghost[11]` provvede ad allocarmi lo spazio necessario. Ma c'è la parte nascosta, che del resto ti ho già detto più volte e che ti ripeto: il nome `ghost` in realtà è solo un riferimento ad un indirizzo di memoria, ed io posso indirizzare qualunque cella di memoria scrivendo semplicemente `ghost[j]`. Se `j` esce dai limiti 0–10 il processore non ci trova nulla da ridire: incrementa `ghost` di `j` secondo le regole dell'aritmetica degli indirizzi ed accede al dato, almeno fin che non esco dai confini dello spazio assegnato al mio programma. Questo è proprio lo strumento che mi consente di far uso di indici negativi. L'assegnazione `vet=ghost-1` fa puntare `vet` all'elemento `ghost[-1]`. Se io usassi `vet[0]` provocherei un'invasione di memoria, con tutte le conseguenze. Ma se il mio programma si limitasse ad usare `vet[1],...,vet[11]` non farebbe altro che accedere all'area riservata sotto il nome `ghost`, e tutto funzionerebbe perfettamente.[3] Analogamente, la definizione `nvet=ghost+5` mi permette di usare tranquillamente gli elementi `nvet[-5],...,nvet[5]` senza paura di provocare invasioni di memoria.

Ti ho illustrato le regole del gioco usando dati di tipo `double`, ma il meccanismo dell'aritmetica degli indirizzi vale in generale: le implicazioni ti portano molto più lontano di quanto questo breve paragrafo lasci intravedere. Vedrai qualcosa in più nel resto del capitolo, ma sappi fin d'ora che si tratterà solo di una parte delle possibilità nascoste in questi meccanismi.

8.1.3 Le matrici

Le matrici altro non sono che tabelle a due o più indici, e apparentemente non ci sarebbe molto da aggiungere. In effetti, il modo più semplice consiste nel dichiarare una matrice con un'istruzione del tipo

```
double mat[10][20];
```

in tal caso, un elemento di matrice viene indirizzato semplicemente scrivendo `mat[j][k]`. Il tutto, naturalmente, stando ben attenti a non superare i limiti degli indici: se hai dichiarato la matrice `mat` come ho scritto qua sopra `j` deve stare tra 0 e 9, e `k` deve stare tra 0 e 19.

I problemi nascono quando si passa una matrice come argomento ad una funzione – e questo prima o poi capita a tutti – oppure quando si considera l'aritmetica degli indirizzi.

Per capire come funziona la faccenda è necessario tener ben presente come viene memorizzata una matrice. Questo è illustrato dalla figura 8.2 — dove ho fatto riferimento al caso 5×3 per ovvie considerazioni di spazio. La disposizione dei dati in memoria viene effettuata facendo variare più rapidamente il secondo indice, quello solitamente usato per indicare la colonna. Gli indici, naturalmente, partono da 0, in accordo con le convenzioni abituali del

[3] Questo esempio è interessante per i programmatori FORTRAN, che convenzionalmente usano indici che partono da 1. Basta traslare l'indirizzo iniziale, e si torna allo schema abituale.

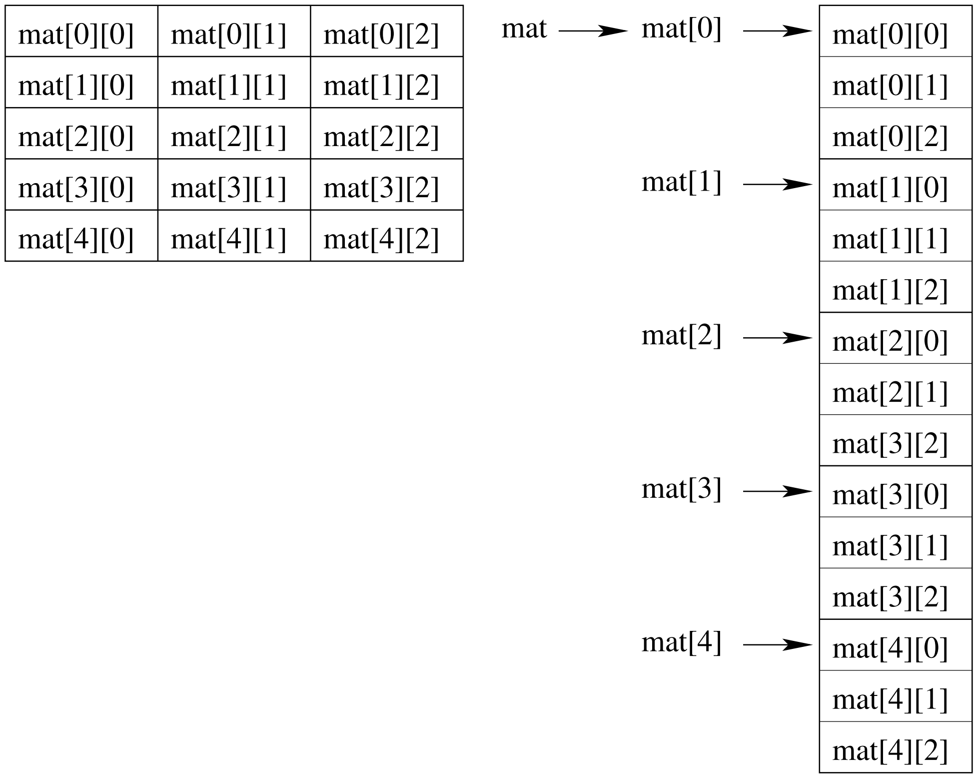


Figura 8.2. La rappresentazione delle matrici. La memorizzazione avviene per righe: prima il vettore costituito dalla prima riga, poi quello della seconda, &c.

linguaggio C e in disaccordo con la consueta notazione matematica in cui gli indici partono da 1.

Lascia che ti esponga le regole in modo generale, in modo che ti sia anche evidente come si possa intervenire con l'aritmetica degli indirizzi. Supponiamo di aver dichiarato una matrice come

⟨*tipo*⟩ `mat[n][m];`

allora:

(i) la matrice viene rappresentata come un vettore di `n` elementi, numerati dal primo indice o indice di riga, ciascuno dei quali a sua volta è un vettore di `m` elementi numerati dal secondo indice, o indice di colonna;

(ii) `mat[j]` identifica l'indirizzo iniziale della `j`–esima riga, e quindi l'indirizzo di un vettore di `m` elementi;

(iii) `mat` identifica l'indirizzo iniziale della matrice, che coincide con l'indirizzo della prima riga;

(iv) la notazione `mat[j][k]` significa: prendi l'indirizzo `mat`, incrementalo di `j` volte la lunghezza della riga (ovvero di `j*m` volte la lunghezza del singolo dato della matrice), poi incrementalo ancora di `k` volte la lunghezza del dato; quello che ottiene è l'indirizzo del dato a cui devi accedere.

Anche qui, naturalmente, il processore non si preoccupa di verificare che `j` e `k` siano entro i limiti dichiarati: applica l'algoritmo che ti ho detto, e tenta di accedere alla memoria. Le invasioni di memoria sono problemi tuoi.

Rileggi con attenzione le regole che ti ho enunciato, perché hanno conseguenze che a prima vista sono ben nascoste. Pensa all'aritmetica degli indirizzi. Forse ti verrà spontaneo pensare che `mat` e `mat[0]` siano la stessa cosa. Nulla di più sbagliato: contengono di fatto lo stesso indirizzo, ma `mat` è un puntatore ad una matrice a due indici, mentre `mat[0]` è un puntatore ad un vettore con un solo indice. Analogamente, l'indirizzo `mat[j]+k` coincide con l'indirizzo del dato `mat[j][k]`, ma `mat[j]+k` identifica l'indirizzo, mentre `mat[j][k]` identifica il dato. Per essere ancora più precisi (e complicati), posso indifferentemente e liberamente indirizzare il dato `mat[j][k]` con le notazioni `*(*(mat+j)+k)` o `*(mat[j]+k)`. Se la cosa ti sembra strana, prova a scrivere un programmino elementare che riempia una matrice $n \times m$, e poi stampa il contenuto usando questo frammento di programma:

```
for(j=0;j<n;j++) {
  for(k=0;k<m;k++) {
  printf("%d,%d: mat[j][k] = %d; ",j,k,mat[j][k]);
  printf("*(mat[j]+k) = %d; ",*(mat[j]+k));
  printf("*(*(mat+j)+k) = %d ;\n",*(*(mat+j)+k));
  }
}
```

potrai facilmente verificare che gli elementi stampati sulla stessa riga sono proprio gli stessi.

Se tutto questo ti sembra un gioco puramente formale ai limiti del sadismo informatico ti dirò che hai la tua buona parte di ragioni: in effetti nella maggior parte dei casi non c'e bisogno di ricorrere a queste complicazioni. Ma a volte servono. Ad esempio, sono molto utili per gestire una situazione abbastanza frequente. Per spiegarti quale devo però premettere un paragrafo.

8.1.4 I vettori di puntatori

Ma si possono anche costruire vettori di puntatori? Certamente, e spesso questo risulta decisamente utile. Il fatto è che come ti ho già spiegato, e dovresti convincertene, i puntatori sono tipi di dati che si differenziano dagli altri solo per l'uso che ne puoi fare. Quindi una dichiarazione del tipo

```
double *adrvet[100];
```

è perfettamente lecita. Significa semplicemente: voglio allocare una tabella di 100 elementi, ciascuno dei quali è un puntatore ad un dato di tipo `double`. Analogamente è lecito scrivere la dichiarazione

```
double **adrvet;
```

significa che `adrvet` punta ad un puntatore, che a sua volta punta ad un dato `double`. Naturalmente, ogni puntatore ha tutti i diritti di contenere

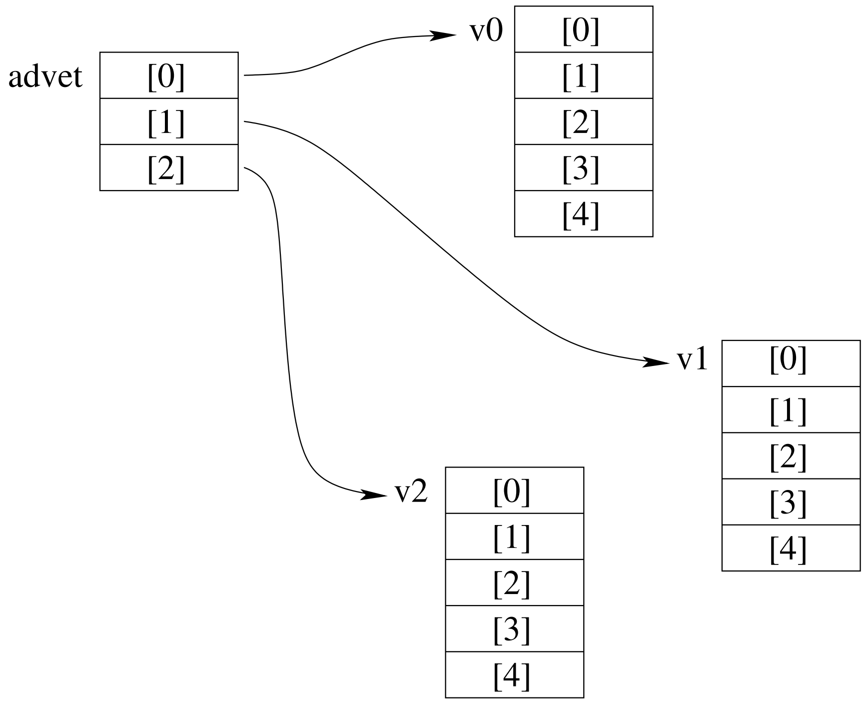


Figura 8.3. Costruzione di una matrice facendo uso di un vettore di puntatori. Il vantaggio è che i vettori `v0`, `v1` e `v2` non devono essere necessariamente allocati in un zona contigua di memoria.

l'indirizzo iniziale di un vettore: nessuno lo proibisce.

Forse nel tuo cervello sta insinuandosi un dubbio: ma questa non è la stessa faccenda delle matrici? Risposta: quasi, ma non esattamente. In effetti, se costruissi un vettore di n puntatori ciascuno dei quali punta a sua volta ad un vettore di m elementi (ad esempio di tipo `double`) avrei costruito qualcosa di molto simile ad una matrice $n \times m$. Ma non prendere la faccenda alla leggera: la dichiarazione

```
double mat[3][5]
```

significa: voglio allocare memoria per una matrice rettangolare composta da 3 righe, ciascuna delle quali è un vettore di 5 elementi. Tutto il resto viene di conseguenza, ed a costo di essere noioso te lo ripeto: `mat` identifica la matrice, ed è di fatto un puntatore alla prima riga; `mat[j]` identifica la `j`–esima riga; `mat[j][k]` identifica il `k`–esimo elemento della `j`–esima riga. Invece la dichiarazione

```
double *advet[3]
```

alloca la memoria per 3 puntatori a dati di tipo `double`; la memoria per i dati non viene allocata, e neppure vengono definiti i puntatori. Per costruire una struttura di dati consistente dovrei scrivere, ad esempio

```
double v0[5],v1[5],v2[5];
double *advet[3];
advet[0]=v0; advet[1]=v1; advet[2]=v2;
```

Spiegazione: la prima riga richiede l'allocazione di 3 vettori di lunghezza 5; la seconda richiede l'allocazione di 3 puntatori; la terza riga definisce i puntatori mettendoci gli indirizzi iniziali dei 3 vettori. La situazione è illustrata in figura 8.3: non ti dovrebbe essere difficle convincerti che il tutto è perfettamente consistente.

La domanda spontanea è: ma adesso come indirizzo un singolo dato? Risposta: con una delle notazioni `*(*(advet+j)+k)`, `*(advet[j]+k)` oppure `advet[j][k]`. Forse stai strabuzzando gli occhi, e pensando che ci deve essere un errore. Invece ci vedi benissimo e non c'è nessun errore: la notazione è esattamente la stessa che ho usato sopra per le matrici. Prova ad esempio a riflettere sul significato di `*(*(advet+j)+k)`: `advet` è l'indirizzo iniziale di un vettore; trasla questo indirizzo di `j` volte la lunghezza di un elemento (in questo caso un puntatore), e ottieni l'indirizzo `advet+j`; prendi il dato `*(advet+j)`, che è a sua volta un puntatore, e sommagli `k` volte la lunghezza del dato a cui punta questo indirizzo (in questo caso si tratta di un double), e ottieni l'indirizzo `*(advet+j)+k` di un double; ora agisci sul dato che si trova a quest'ultimo indirizzo. Più difficile da dire che da fare.

Ma se per accedere ad un singolo dato posso usare le stesse notazioni che ho usato per le matrici dove sta la differenza? Semplice: nelle dichiarazioni che allocano la memoria. Riguardale bene, e ti renderai conto che sono ben diverse!

Sadomasochismo informatico, starai pensando. Ma continua a leggere, e forse ti renderai conto che in fondo non è inutile. La faccenda rilevante sta proprio nella possibilità di definire una matrice senza ricorrere allo schema rigido dell'allocazione sequenziale per righe.

8.1.5 Un errore frequente e subdolo

Il passaggio di una matrice come argomento di una funzione richiede particolare attenzione. In linea di principio la faccenda è semplice. Eccoti i frammenti di programma che ti servono. Potrai scrivere ad esempio la funzione come

```
int ciccio(mat,n,m)
    int n,m;
    double mat[n][m];
{ ... }
```

Il significato è abbastanza ovvio: il parametro `mat` è l'indirizzo iniziale di una matrice di `n` righe e `m` colonne. All'interno della funzione potrò liberamente indirizzare un elemento della matrice con la consueta notazione `mat[j][k]`. Tutto facile, fin qui. Osserva che non hai alcun bisogno di dichiarare le dimensioni usando delle costanti: la memoria è già stata allocata dal modulo chiamante, e qui devi solo definire la struttura della matrice: `n` righe ciascuna delle quali contiene `m` elementi.

Ma veniamo al frammento del modulo chiamante. Userò l'istruzione `#define` per fissare le dimensioni della matrice.

```
#define NRIGHE ...
#define NCOLONNE ...
int main()
{
    ...
  double mat[NRIGHE][NCOLONNE];
    ...
  j=ciccio(mat,NRIGHE,NCOLONNE);
    ...
}
```

La dichiarazione `double mat[...]` alloca la memoria per una matrice di `NRIGHE` righe e `NCOLONNE` colonne; la chiamata alla funzione specifica sia l'indirizzo della matrice che la sua struttura. Tutto perfetto.

Ma a volte i programmatori non troppo esperti ragionano così. Supponiamo che io debba scrivere un programma in cui la dimensione della matrice possa variare. Lo schema qui sopra non sembra adattarsi molto bene, perché il dimensionamento è rigido. Come fare? E pensano ad una soluzione brillante, suggerita dalla figura 8.4: definiscono la matrice assegnandole il numero massimo previsto di righe e di colonne, e poi utilizzano una sottomatrice facendo variare gli indici in un insieme più ristretto di quello definito inizialmente. Un po' oscura, come spiegazione. Entriamo in dettaglio. Ecco il main:

```
#define NRIGHE 10
#define NCOLONNE 10
int main()
{                             /* Esempio da non seguire */
  double mat[NRIGHE][NCOLONNE];
  int n=3,m=5;
    ...
  j=ciccio(mat,n,m);
    ...
}
```

Brillante, non è vero? Se `matrice` può contenere 10×10 elementi ne potrà ben contenere 3×5: lo spazio inutilizzato resterà vuoto. Peccato che quella stupida macchina si ostini a restituire risultati assurdi. Perché?

Se hai già immaginato quale sia l'errore salta pure al paragrafo successivo; altrimenti prova a pensarci, e poi continua a leggere. Eccoti la spiegazione dell'arcano. In forza della dichiarazione

`double mat[NRIGHE][NCOLONNE]`

che ha trovato nel `main` il compilatore ha predisposto la memoria allocando un vettore di 100 dati; la prima riga della matrice inizia con l'elemento 0

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]	[...][...]			[0][9]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]	[...][...]			[1][9]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]	[...][...]			[2][9]
[3][0]	[3][1]	[3][2]	[3][3]	[3][4]	[3][5]	[...][...]			[3][9]
[...][...]	[...][...]	[...][...]	[...][...]	[...][...]	[...][...]	[...][...]			[...][...]
[...][...]	[...][...]							[...][...]	[...][...]
[9][0]	[...][...]							[...][...]	[9][9]

Figura 8.4. Ad illustrazione di un errore comune nel passaggio di matrici a funzioni. La sottomatrice 3×5 occupa una porzione della matrice 10×10 – quella con sfondo bianco in figura – utilizzando di fatto gli stessi indici. Ma nella rappresentazione utilizzata per allocare i dati in memoria le righe vengono disposte sequenzialmente, e quindi agli stessi indici non corrisponde la stessa locazione di memoria.

del vettore che corrisponde all'elemento `[0][0]` della matrice; la seconda riga inizia con l'elemento 10 del vettore che corrisponde all'elemento `[1][0]` della matrice, &c, fino all'ultima riga che corrisponde all'elemento `[9][0]` della matrice ed inizia con l'elemento 90 del vettore. Poi il programmatore ha usato solo 3 righe di 5 elementi, indirizzandoli con un indice di riga `j` compreso tra 0 e 2 ed un indice di colonna `k` compreso tra 0 e 4. Nulla di male, apparentemente, ma alla fine risultano utili solo i primi cinque elementi della prima riga, i primi 5 della seconda ed i primi 5 della terza, cioè la parte a sfondo bianco della figura 8.4.

Passiamo alla funzione. Questa ha ricevuto come parametri un indirizzo `mat` e due interi `n` e `m` che specificano il numero di righe e colonne della matrice. Poichè `n` vale 3 e `m` vale 5 la funzione assume che la matrice sia composta da 15 elementi organizzati in 3 righe di 5 elementi ciascuna. Dunque, pensando ad un vettore di 15 elementi che parte dall'indirizzo `mat`, l'elemento `[0][0]` della matrice si trova al posto `[0]` del vettore; l'elemento `[1][0]` della matrice si trova al posto `[5]` del vettore, l'elemento `[2][0]` della matrice si trova al posto `[10]` del vettore. Il che non corrisponde certamente a quanto è stato fatto nel `main`: i dati sono tutti disallineati.

Conclusione: quello che ti ho descritto è un errore da evitare nel modo più assoluto. Commento: ma allora la struttura delle matrici è molto rigida! Beh, lasciami osservare che la si può rendere molto flessibile, e proprio giocando sui puntatori.

8.1.6 Come aggirare l'errore, con qualche trucco

Ci sono almeno due soluzioni al problema che ti ho posto. La prima consiste nel costringere il main a far uso di una struttura compatibile con quella utilizzata dalla funzione; la seconda consiste nel modificare la funzione in modo che si aspetti non l'indirizzo di una matrice, ma un vettore di puntatori.

Per la prima soluzione scriverai il `main` così:

```
#define NRIGHE 10
#define NMAX 100
int main()
{
  double vet[NMAX],*mat[NRIGHE];
  int j,n=3,m=5;
  for(mat[0]=vet,j=1; j<n; j++) mat[j]=mat[j-1]+m;
    ...
  mat[j][k] = ...;
    ...
  j=ciccio(vet,n,m);
    ...
}
```

Eccoti la spiegazione. Alloco un vettore `vet` che riservi spazio per il numero massimo di elementi della matrice, ed alloco anche un vettore `mat` con un numero di puntatori pari al numero massimo di righe. Poi definisco i puntatori in modo che venga simulata una struttura di `n` righe contigue, ciascuna di `m` elementi: questo lo faccio definendo i primi `n` elementi del vettore dei puntatori in modo che puntino agli elementi 0, m, $2m, \ldots$ del vettore. Osserva bene che questa è proprio la disposizione degli elementi di una matrice $n \times m$. Quando devo riempire la matrice uso la notazione `mat[j][k]`. Infine chiamo la funzione passandole l'indirizzo `vet` del vettore. Il trucco consiste nello sfruttare l'aritmetica degli indirizzi per forzare la disposizione corretta dei dati nella matrice.

La seconda soluzione è questa. Modifico l'ultima versione del `main` semplicemente sostituendo la chiamata alla funzione con

```
j=ciccio(mat,n,m);
```

Nota che quello che viene passato alla funzione è l'indirizzo del vettore di puntatori, non quello dei dati. Dunque occorre modificare la funzione così:

```
int ciccio(mat,n,m)
    int n,m;
    double *mat[];
{ ... }
```

I puntini tra le parentesi graffe stanno ad indicare che il corpo della funzione non cambia per nulla. La dichiarazione `double *mat[]` significa proprio: `mat` è un vettori di puntatori a dati di tipo `double`. Quanti siano i puntatori, e quanto lunghi i vettori a cui puntano è del tutto irrilevante: i parametri `n` e `m` che dimensionano la matrice servono solo all'interno della funzione, ma non hanno alcuna importanza per il compilatore. Naturalmente, potresti ben scrivere `double **mat`, che significa di fatto la stessa cosa.

Quale delle due soluzioni è preferibile? A te la scelta; ma se io dovessi prendere una decisione preferirei la seconda, e ti spiego perché. Uno dei problemi tipici è che talvolta una matrice ha molti elementi nulli. In tal caso allocare spazio per l'intera matrice comporta uno spreco di memoria che per matrici di grandi dimensioni può essere significativo. Ad esempio, supponi di aver a che fare con una matrice triangolare: tutti gli elementi sopra la diagonale sono nulli. In altre parole, la metà dello spazio allocato è sprecata. Qui la via d'uscita è semplice ed elegante. Te la illustro con un esempio.

Supponi di voler costruire ed utilizzare il triangolo di Tartaglia (che i francesi si ostinano ad attribuire a Pascal):

$$
\begin{array}{ccccccccccccc}
 & & & & & & 1 & & & & & & \\
 & & & & & 1 & & 1 & & & & & \\
 & & & & 1 & & 2 & & 1 & & & & \\
 & & & 1 & & 3 & & 3 & & 1 & & & \\
 & & 1 & & 4 & & 6 & & 4 & & 1 & & \\
 & 1 & & 5 & & 10 & & 10 & & 5 & & 1 & \\
1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1 \\
 & \dots & & \dots & & \dots & & \dots & & \dots & & \dots &
\end{array}
$$

Suppongo che tu sappia come lo si costruisce, e sappia anche che si tratta semplicemente della tavola dei coefficienti binomiali.[4] Sarebbe molto bello poter indirizzare ciascun elemento di questa tabella scrivendo semplicemente `Coef_bin[j][k]` – naturalmente stando ben attenti a che `k` non superi `j` – ma se si usa una rappresentazione a matrice si spreca metà dello spazio. Ebbene, una via d'uscita è illustrata in figura 8.5. Basta usare un vettore di puntatori definiti in modo conveniente. Guardare la figura è più semplice che leggere una lunga spiegazione, e mette immediatamente in evidenza il fatto cruciale: la disposizione dei puntatori ottimizza l'uso della memoria, perché memorizza tutti gli elementi della tabella senza lasciare spazi vuoti.

Qui sotto trovi una funzione che dato n costruisce le prime $n+1$ righe del triangolo di Tartaglia, ossia tutti i coefficienti binomiali $\binom{j}{k}$ con $0 \le j \le n$ e $0 \le k \le j$. Eccoti una descrizione più dettagliata della funzione:

(i) accetta in ingresso un numero `n` che si assume intero positivo e non troppo grande (puoi arrivare al massimo a 33, poi il calcolo va in overflow);

[4] Gli orli sinistro destro sono tutti 1; gli altri elementi sono la somma dei due immediatamente superiori.

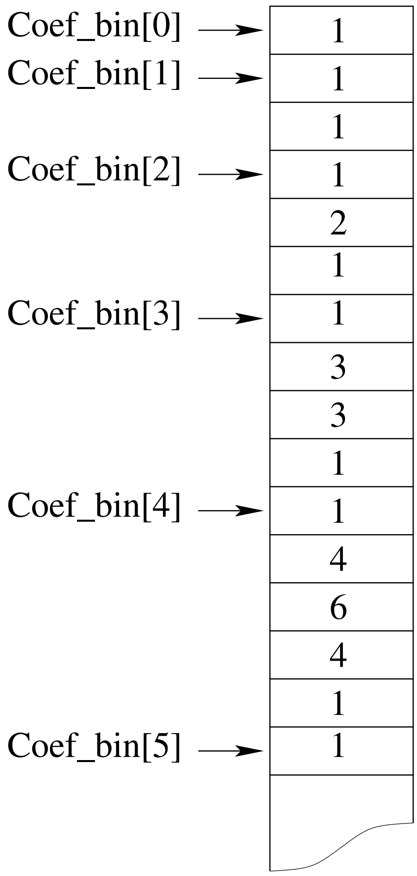


Figura 8.5. Rappresentazione in memoria della tabella dei coefficienti binomiali, o triangolo di Tartaglia. Lo schema funziona per una matrice triangolare inferiore qualsiasi.

(ii) alloca dinamicamente la memoria necessaria per memorizzare sia i puntatori alle righe che gli elementi del triangolo, disposti come in figura 8.5;
(iii) definisce tutti i puntatori e riempie la tabella;
(iv) restituisce al programma chiamante l'indirizzo della tabella.

La scrittura di questa funzione non è banale: è indispensabile saper giocare coi puntatori. Aiutati con la figura, e rifletti per il tempo necessario. Ti scrivo subito il codice; i commenti dopo.

```
int **triangolo_tartaglia(n)
     int n;
     /* Costruisce le prime n+1 righe del triangolo
        di Tartaglia*/
{
  int **pnt,j,k;
                  /* Calcola la memoria necessaria */
  j=(n+1)*sizeof(void *);       /* per i puntatori */
```

```
  j+=(n+2)*(n+1)*sizeof(int)/2; /* e per la tabella. */
                           /* Alloca la memoria: */
  pnt=malloc(j);           /* pnt indirizza i puntatori */
  pnt[0]=(int *) (pnt+n+1);  /* Definisci i puntatori */
  for(j=1;j<=n;j++) pnt[j]=pnt[j-1]+j+1;
                              /* Riempi la tabella */
  pnt[0][0]=1;
  for(j=1;j<=n;j++) {
    pnt[j][0]=pnt[j][j]=1;
    for(k=1;k<j;k++) pnt[j][k]=pnt[j-1][k-1]+pnt[j-1][k];
  }
  return(pnt);
}
```

Lungo attimo di riflessione, un respiro profondo, e veniamo ai commenti.

Per rendere l'esercizio ancor più completo ho usato la funzione `malloc` per allocare la memoria necessaria. Ma quanta ne serve? Il calcolo è facile: per $n+1$ righe servono $n+1$ puntatori e $1+2+\ldots+n+1 = (n+1)(n+2)/2$ elementi di tabella. Questo è il calcolo eseguito all'inizio della funzione, prima della chiamata alla `malloc`. Osserverai che ho usato `sizeof(...)` per tener conto della lunghezza di ciascun dato, puntatore o intero che sia.

Può darsi che ti stupisca un po' l'espressione `sizeof(void *)`. Nulla di strano: voglio solo sapere quanti bytes occupa un puntatore. Il tipo `void *` viene interpretato dal compilatore come un puntatore, indipendentemente dal tipo di dato a cui punta.

L'uso della funzione `malloc` lo dovresti già conoscere: vedi il paragrafo 5.3.2. Ti parrà strana invece l'istruzione

`pnt[0]=(int *) (pnt+n+1)`

per via di quel buffo `(int *)`. Ma è ben giustificato. Esamina attentamente l'espressione a destra, tenendo conto della dichiarazione `int **pnt`. Osserva che `pnt` è un puntatore ad un puntatore (ci sono 2 asterischi!); a `pnt` sommo l'intero `n+1`, operazione che il compilatore predispone tenendo conto dell'aritmetica degli indirizzi. Il risultato è un indirizzo, traslato rispetto a `pnt` di quanto basta per lasciare spazio ad `n+1` puntatori. Da qui voglio far partire i dati della tabella, e proprio per questo voglio mettere questo indirizzo in `pnt[0]`, il primo puntatore. Ma qui il compilatore diventa schizzinoso: se tu scrivessi semplicemente

`pnt[0]=pnt+n+1`

si metterebbe immediatamente a protestare inviandoti messaggi tipo

`warning: assignment from incompatible pointer type`

(questo almeno è il messaggio del compilatore `Gnu-cc`). Perché? Semplice: `pnt` è un puntatore ad un puntatore; `pnt[0]` invece è un puntatore ad un intero (la parentesi quadra con un indice ha in pratica l'effetto di annullare un asterisco, perché usi un elemento di un vettore di puntatori). Per lui sono due cose ben diverse, e quindi ti avverte. È ben vero che si tratta solo di un `warning`, un avvertimento che c'è qualcosa di poco pulito, ma che non impedisce la conclusione della compilazione. Ma intanto ha protestato. Quel `(int *)` che ci ho messo in mezzo significa proprio: non preoccuparti se `pnt[0]` non punta ad un intero; consideralo come tale, ché so bene quel che sto facendo. Il compilatore, per così dire, assume un atteggiamento del tipo "non capisco, ma mi adeguo".

La definizione degli altri puntatori non dovrebbe crearti problemi: non è altro che ciò che ho rappresentato in forma grafica in figura 8.5. Questo completa la disposizione degli indirizzi: adesso posso tranquillamente far riferimento ad un elemento del triangolo scrivendo `pnt[j][k]`. Se ti sembra misterioso, rifletti ancora un attimo: `pnt` punta ad un vettore di puntatori ad interi; `pnt[j]` punta ad un indirizzo intero; `pnt[j][k]` è il dato intero che si trova all'indirizzo `pnt[j]+k`.

È fatta: L'ultima riga restituisce al programma chiamante un puntatore a tutta la struttura che ho appena creato. Se scrivi

```
int **Coef_bin;
Coef_bin = triangolo_tartaglia(n);
```

la funzione ti restituirà un indirizzo che sarai libero di utilizzare scrivendo semplicemente `Coef_bin[j][k]`. Può darsi che ti stupisca il fatto che nella chiamata non c'è alcun riferimento alla tabella che contiene realmente i coefficienti binomiali. Ma quello che ho scritto è corretto: alla tabella si ariva in modo indiretto, tramite i puntatori. Ancora una volta, ciò che conta non sono i dati, ma gli indirizzi.

8.2 Le strutture di dati

Quello delle strutture è un argomento che non abbiamo ancora affrontato. L'ho volutamente lasciato a questo punto perché ritengo che l'uso efficace delle strutture richieda una certa familiarità con l'uso degli indirizzi e con i meccanismi di allocazione della memoria, e questo soprattutto quando si tratta di usare strutture come argomenti di funzioni.

Ma il fatto che abbia fin qui ignorato l'argomento non deve indurti a pensare che si tratti di faccenda di scarsa utilità: si tratta invece di uno strumento di grande potenza e flessibilità, per un programmatore che abbia imparato a dominarlo.

In termini semplici una struttura è un insieme di dati non necessariamente omogenei, ma tra loro collegati, che vengono raccolti in un'area contigua di memoria. Una definizione che lascia il tempo che trova, probabilmente, visto che non spiega come raccogliere questi dati. Ebbene, questo è quanto intendo spiegarti ora.

8.2.1 Creazione di strutture

Il modo più semplice – ma non il migliore – per creare una struttura fa uso della dichiarazione

```
struct {
⟨tipo⟩ ⟨nome⟩;
⟨tipo⟩ ⟨nome⟩;
...
} ⟨nome struttura⟩;
```

Significa: la mia struttura è composta dai dati che ho elencato, a ciascuno dei quali ho assegnato un tipo; a questa struttura assegno un nome, di cui farò uso tutte le volte che vorrò riferirmi a questo insieme di dati.

Un esempio semplice sarà certamente utile. Supponiamo di voler scrivere delle funzioni che eseguano operazioni su frazioni. Dal momento che il linguaggio C non prevede un dato di questo tipo sarò interessato a crearmelo. Naturalmente, mi farebbe piacere avere un modo naturale per associare numeratore e denominatore della stessa frazione. Il modo elementare consiste nel dichiarare, ad esempio

```
struct {
  int num;
  int den;
} a,b,c;
```

Significato: `a`, `b` e `c` sono strutture di dati costituite da due interi, indicati rispettivamente con `num` (il numeratore) e `den` (il denominatore). Il compilatore alloca per ciascuna struttura lo spazio sufficiente a contenere due interi.

Noterai però che la struttura che ho definito sopra non ha un tipo che la contraddistingue: è una struttura e basta. Per questo è conveniente modificare di poco la dichiarazione in

```
struct frazione {
  int num;
  int den;
} a,b,c;
```

La differenza sembra minima, ma quest'ultima scrittura ha i suoi vantaggi. Il principale sta nel fatto che la dichiarazione

```
struct frazione
```

diventa del tutto simile alle dichiarazioni `int` o `char` o `double` che hai usato fin qui: identifica un tipo di dato ben preciso. In particolare sono autorizzato ad utilizzare la notazione

```
sizeof(struct frazione)
```

per indicare la lunghezza in bytes della struttura.

Puoi usare anche un terzo modo, di poco più complesso, ma decisamente più generale ed utile. La dichiarazione

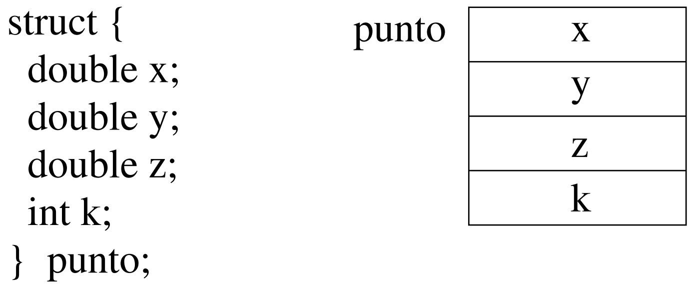


Figura 8.6. La disposizione dei dati all'interno di una struttura è rigorosamente determinata dalla dichiarazione. Il compilatore alloca la memoria in modo che i dati siano disposti uno dopo l'altro, nell'ordine che io ho indicato. Attenzione però agli allineamenti; leggi il testo, e quarda la figura 8.7.

```
typedef struct frazione {
  int num;
  int den;
} FRAZIONE;
```

non alloca memoria, ma si limita a definire un nuovo tipo di dato, identificato dal nome `FRAZIONE`. A partire da questo momento sono autorizzato a scrivere, ad esempio,

```
FRAZIONE a,b,c;
```

per indicare che `a`, `b` e `c` sono frazioni. Sono autorizzato anche a scrivere `sizeof(FRAZIONE)` per indicare la lunghezza in bytes della struttura.

Fin qui non è difficile, ma lasciami, anche questa volta, sollevare un po' il velo sull'azione del compilatore. In particolare lascia che ti illustri come viene effettivamente allocata la memoria per le strutture. Ti illustro il procedimento in figura 8.6. L'ordine in cui vengono disposti i dati entro la cella di memoria che costituisce la struttura è rigorosamente fissato dalla dichiarazione: il compilatore non è autorizzato a modificarlo. Ci può essere invece qualche problema sull'allineamento dei dati. In effetti, l'architettura di alcune macchine prevede che ogni tipo di dato sia allineato ad un indirizzo che sia multiplo della lunghezza del dato stesso, o almeno di una certa quantità che dipende dal modo di indirizzamento. Ad esempio, sulle macchine di tipo PC–IBM risulta perlomeno conveniente allineare tutti i dati di lunghezza superiore a 4 bytes su indirizzi che siano multipli di 4, mentre i dati di lunghezza 2 bytes (tipicamente i dati `short int`) vengono allineati su indirizzi pari ed i dati `char` a qualunque indirizzo. [5] Al fine di rispettare queste regole di allineamento il compilatore dispone i dati entro la struttura lasciando eventualmente dello spazio libero. Quali siano le regole di allineamento, dipende dal sistema, o anche dal compilatore. Va da sé che un tal

[5] Misterioso? No: dipende semplicemente dal fatto che il processori Intel o simili usano 32 bit per l'indirizzo.

metodo comporta talvolta uno spreco di spazio di memoria.

Un esempio – strettamente dipendente dal compilatore `Gnu-CC` su macchine del tipo PC–IBM – è il seguente. Considera le due dichiarazioni

```
typedef struct c_i_c {
  char c;
  int  k;
  char d;
} CIC;
typedef struct i_c_c {
  int  k;
  char c;
  char d;
} ICC;
```

Noterai che la differenza è minima: un banale scambio di ordine nelle dichiarazioni dei dati `c` e `k`. Ma ora prova ad inserire queste dichiarazioni in un file, diciamo `strutture.c`, e ad aggiungerci il programmino

```
int main()
{
  printf("Struttura CIC: %d bytes\n",sizeof(CIC));
  printf("Struttura ICC: %d bytes\n",sizeof(ICC));
  exit(0);
}
```

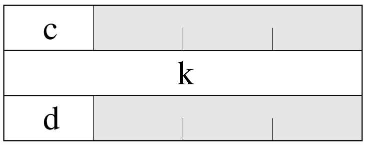
Nulla di trascendentale, come vedi: voglio semplicemente sapere quanto spazio occupano le strutture che ho dichiarato. Che ti aspetti come risultato? Ecco quello che ho trovato sulla mia macchina:

```
Struttura CIC: 12 bytes
Struttura ICC: 8 bytes
```

Perché la prima struttura occupa più spazio della seconda? Risposta: per via degli allineamenti. Ti illustro quello che è accaduto in figura 8.7.

In ambedue le strutture il dato di massima lunghezza è un `int`, che occupa 4 bytes; di conseguenza il compilatore arrotonda la dimensione totale della struttura ad un multiplo di 4 bytes. Nel primo caso, quello della struttura `CIC`, il primo dato è un `char`, ed il compilatore lo dispone ad un indirizzo che è multiplo di 4; il secondo dato è un `int`, ed il compilatore è costretto a lasciare 3 bytes vuoti per allinearne l'indirizzo ad un multiplo di 4; il terzo ed ultimo dato è ancora un `char`, ma il compilatore è costretto a lasciare altri 3 bytes liberi per arrotondare la dimensione totale ad un multiplo di 4. Totale: 6 bytes sciupati. Nel secondo caso, quello della struttura `ICC`, il compilatore allinea il primo dato, un `int`, che occupa da solo 4 bytes; il secondo ed il terzo dato, due `char`, possono essere allineati a qualunque indirizzo, e quindi il compilatore li dispone uno dopo l'altro; l'arrotondamento a multipli di 4 della dimensione totale richiede solo 2 bytes. Ti sembra poca

```
struct c_i_c {
  char c;
  int  k;
  char d;
}
```




```
struct i_c_c {
  int  k;
  char c;
  char d;
}
```

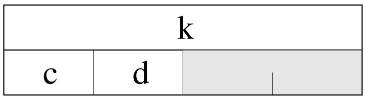


Figura 8.7. L'allineamento dei dati. Nel primo caso l'allineamento della variabile `int` posizionata in mezzo a due dai `char` costringe il compilatore a lasciare 3 bytes vuoti; inoltre l'allineamento della lunghezza totale della struttura su multipli della lunghezza di un `int` (4 bytes) provoca la perdita di altri 3 bytes alla fine. Nel secondo caso i due dati `char` possono essere allocati consecutivamente, e la sola perdita è l'allineamento dell'indirizzo finale della struttura, che richiede 2 soli bytes.

roba? Beh, prova a pensare di scrivere un programma che lavori con qualche milione di strutture di questo tipo, e scoprirai che risparmiare un buon terzo di memoria può avere i suoi vantaggi.

Morale: attenzione a come ordini i dati entro una struttura. Una buona regola è ordinare i dati per dimensione decrescente: prima quelli che occupano più spazio; i `char` per ultimi.

8.2.2 L'uso dei dati nelle strutture

Definite le strutture, vediamo come riempirle ed utilizzarle. Tutto sta nel far uso dell'operatore '`.`' (il punto), che fa riferimento ad un membro di una struttura. Torniamo all'esempio della `FRAZIONE` che abbiamo visto sopra. Se `a` è un dato di tipo `FRAZIONE`, e voglio assegnargli il valore 2/3 mi basterà scrivere

```
a.num=2;  a.den=3;
```

La scrittura `a.num` significa: il dato su cui voglio agire è il campo `num` della struttura `a`. Non c'è molto di più da sapere, a questo livello: la scrittura `a.num` può essere tranquillamente impiegata in qualunque espressione che coinvolga operazioni aritmetiche o logiche o bit–a–bit. Si tratta a tutti gli effetti di un dato di tipo `int`. Ad esempio, se voglio assegnare alla frazione `c` il risultato del prodotto tra le due frazioni `a` e `b` scriverò:

```
c.num=a.num*c.num; c.den=a.den*b.den;
```

Naturalmente, il risultato non sarà necessariamente ridotto ai minimi termini. Dovrò servirmi di una funzione `int mcd(int j,int k)` che restituisca il massimo comun divisore tra due interi, ed aggiungerò le istruzioni (tutte le variabili sono assunte di tipo intero)

```
m=mcd(c.num,c.den); c.num/=m; c.den/=m;
```

8.2.3 Le strutture come argomenti di funzioni

Le strutture, come ti ho detto, sono dati come tutti gli altri: una volta definite, il compilatore sa come trattarle. Ma...

Nelle prime versioni dei compilatori C c'era qualche limitazione: non era ammesso assegnare in blocco un valore alle strutture, e neppure passarle come argomenti o restituirle come valori di una funzione. Nelle specifiche attuali del linguaggio questo è ammesso, ma un po' di prudenza nel trasferire strutture tra funzioni non guasta: se le strutture hanno grosse dimensioni si corre il rischio di utilizzare la memoria in modo alquanto inefficiente, o addirittura di esaurire lo stack del programma — specie su sistemi che hanno uno stack di piccole dimensioni.

E allora devo rinunciare a passare una struttura come argomento di una funzione? No, certo: basta ricorrere agli indirizzi. Mi soffermerò su questa circostanza, perché passare direttamente una struttura come argomento o riceverla come risultato di una funzione non ha nulla di diverso dal passare o ricevere un `int` o un `double`: basta specificarne il nome.

Procediamo ancora una volta con un esempio. Supponiamo di voler scrivere una funzione

```
int confronta_frazioni(FRAZIONE *a,FRAZIONE *b)
```

che confronti due frazioni e restituisca rispettivamente il valore `-1` se `a`<`b`, `0` se `a`=`b` e `1` se `a`>`b`. Nota che ho esplicitamente richiesto che le frazioni vengano passate per indirizzo. Eccoti il codice:

```
int confronta_frazioni(a,b)
     FRAZIONE *a,*b;
{
  int j,k;
  j=a->num * b->den; k=a->den * b->num;
  if(j<k) return(-1);
  else if(j==k) return(0);
  else return(1);
}
```

Il tipo `FRAZIONE` è quello che ho definito poco fa: una struttura che contiene i due campi `num` (il numeratore) e `den` (il denominatore). La sola cosa misteriosa in questo codice dovrebbe essere quella strana notazione `a->num`, `b->den`, &c. Tu, rifacendoti a quanto ti ho detto sin qui, avresti probabilmente scritto `a.num`, `b.den`, &c. Ma avresti solo dato al compilatore un'occasione in più per protestare, e questa volta, ahimè, con piena ragione. Il fatto è che la notazione `a.num` presuppone che `a` sia una dato, e precisamente una struttura che contenga al suo interno un campo `num`. Qui invece `a` è un *puntatore* ad una struttura. Dovresti quindi scrivere `(*a).num`; le parentesi sono essenziali perché l'operatore '`.`' (membro della struttura) ha la precedenza sull'operatore '`*`' (dato all'indirizzo). La notazione non è certo né naturale né comoda; per questo nel linguaggio C è stata introdotto l'operatore '$\langle x \rangle$`->`$\langle y \rangle$' che significa: agisci sul membro $\langle y \rangle$ della struttura che si trova all'indirizzo $\langle x \rangle$.

Naturalmente, la chiamata per indirizzo implica che qualunque modifica ad un dato della struttura venga restituita al modulo chiamante.

8.2.4 Strutture, vettori e puntatori

Un giorno o l'altro probabilmente ti porrai domande del tipo: posso creare dei vettori di strutture? posso inserire in una struttura dei vettori, dei puntatori o delle altre strutture? La risposta è: ma certamente! Le cose che non ti sono concesse sono ben poche, e questo ti dà forse una vaga idea di quanto complessa possa diventare una struttura di dati.

Per creare un vettore di strutture si usa una dichiarazione che non ha nulla di diverso da quella che hai usato per dati `int` o `double`. Continuando con l'esempio che abbiamo sviluppato fin qui, un vettore di frazioni si costruisce con la dichiarazione

```
FRAZIONE vetf[100];
```

Il significato è il solito: alloca spazio per una tabella di `100` elementi, ciascuno dei quali è una struttura `FRAZIONE`. Il resto vien quasi da sé: `vetf` è l'indirizzo del primo elemento del vettore; `vetf[j]` è il `j`–esimo elemento, quindi è un dato; `vetf[j].num` è il campo `num` del `j`–esimo elemento. Se ti piace soffrire potrai sempre usare una delle notazioni `(vetf+j)->num` oppure `(*(vetf+j)).num` al posto di `vetf[j].num`: si tratta sempre del campo `num` del `j`–esimo elemento.

Quanto alle strutture nelle strutture, devi tener presente che non potrai dichiarare, ad esempio,

```
struct matrioska {
  struct matrioska;
};
```

Una tal dichiarazione genererebbe una sequenza infinita di annidamenti, che non è consentita per ovvie ragioni. È invece perfettamente lecita la dichiarazione

```
struct matrioska {
  struct *matrioska;
};
```

perché in tal caso il contenuto della struttura è un puntatore, non un dato, e come tale non richiede di essere a sua volta sviluppato nel suo contenuto.

Mi fermo qui: il solo limite alla complessità delle strutture che puoi creare è la tua immaginazione. Passo invece ad illustrarti un esempio più complesso.

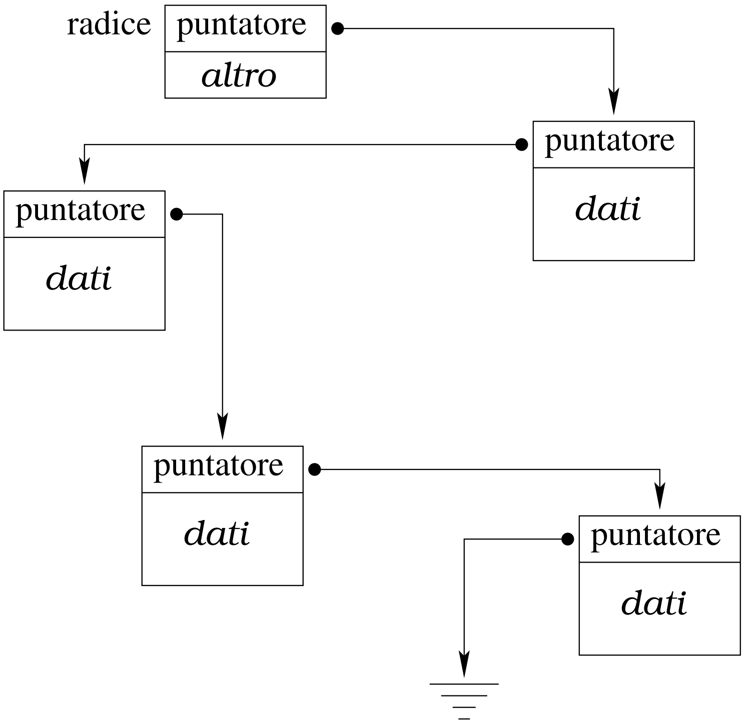


Figura 8.8. La struttura di lista nella sua forma più semplice: la radice contiene un puntatore al primo elemento, questo punta al successivo &c, fino all'ultimo elemento che contiene un'informazione di fine lista (ad esempio un puntatore `NULL`).

8.3 Organizzazione dei dati in liste

L'uso combinato di strutture e puntatori costituisce un'ottima base per l'organizzazione dei dati. Non intendo qui soffermarmi a lungo su questo argomento: ne parlerò quanto basta per farti intravedere alcune delle possibilità.

8.3.1 La lista sequenziale

Iniziamo con la struttura di lista. Il problema consiste nell'organizzare dei dati, la cui natura in questo momento non ha alcuna importanza, in un ordine sequenziale, senza per questo doverli disporre sequenzialmente in memoria – il che comporterebbe probabilmente un gran numero di spostamenti di dati. Non mi importa qui specificare di che dati si tratti: li indicherò genericamente con $\langle dati \rangle$. La struttura di lista è rappresentata in figura 8.8. Ecco il minimo necessario:

- una radice di lista che contiene il puntatore al primo elemento.
- un certo numero di strutture destinate a contenere i dati; in testa a ciascuna struttura ci sarà un puntatore che serve a gestire l'organizzazione in lista.

- Una convenzione su come indicare la fine della lista; ad esempio, si può semplicemente definire come `NULL` il puntatore dell'ultimo elemento: se per errore venisse utilizzato provocherebbe immediatamente un errore di accesso alla memoria identificato con `segmentation fault`, o `memory access violation` o qualcosa di simile. I sistemi più recenti hanno meccanismi di protezione che segnalano un tale errore interrompendo immediatamente l'esecuzione del programma.

Veniamo al codice. Cominciamo col definire la struttura destinata a contenere i dati di un singolo elemento; la chiamerò `X_DATI`.

```
typedef struct x_dati {
  struct x_dati *pnt;
  ⟨dati⟩;
} X_DATI;
```

Come puoi ben vedere, il solo componente che interessa al fine di costruire la lista è il puntatore; i dati possono essere qualunque cosa. [6]

Ora dobbiamo definire la radice della lista; con la consueta immaginazione definirò un tipo `X_ROOT`:

```
typedef struct x_root {
  X_DATI *pnt;
  ⟨altri dati, se servono⟩;
} X_ROOT;
```

Anche qui ci ho messo il minimo indispensabile: il puntatore al primo elemento della lista. Le due definizioni precedenti potranno essere inserite in un file di tipo `.h` da includere in tutti i files sorgente che fanno uso della lista. Ad esempio, potrò dare al file il nome `lista_singola.h`. Ed ecco un frammento di codice che dovrà far parte del `main` (o comunque di una funzione che ha il compito di creare la struttura iniziale della lista):

```
#include "lista_singola.h"
X_ROOT Radice;
int main()
{
  ...
  Radice.pnt=NULL;
  ...
}
```

L'istruzione `Radice.pnt=NULL` determina la situazione iniziale: la lista non contiene nessun elemento, e il puntatore contiene semplicemente un `NULL`.

Ed ora l'inserimento di un elemento. Lo schema è illustrato in figura 8.9: occorre rompere un legame tra un elemento ed il successivo, e sostituirlo con

---

[6] Potresti essere tentato di sostituire la seconda riga con `X_DATI *pnt`, ma molti compilatori – incluso il compilatore `Gnu-cc` – ci troverebbero da ridire.

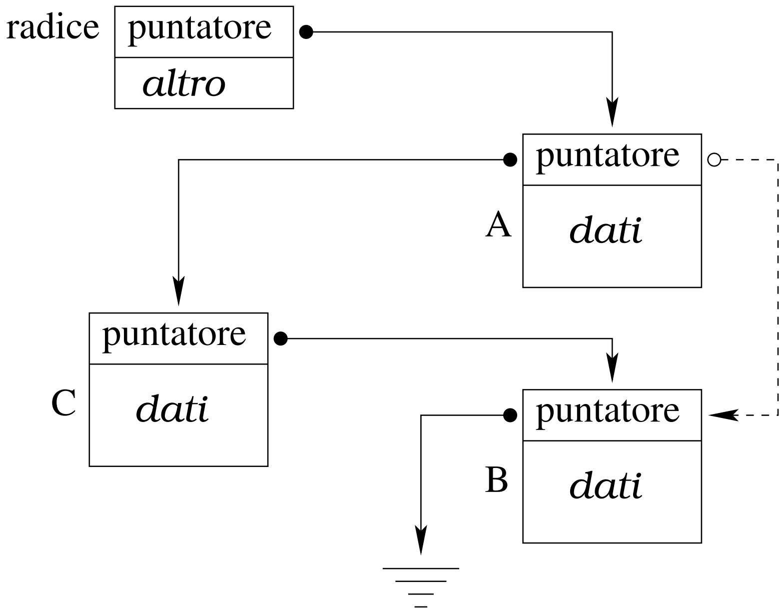


Figura 8.9. Inserimento di un nuovo elemento in una lista, in una posizione assegnata. L'elemento C viene inserito dopo l'elemento A mediante una semplice ridefinizione dei puntatori: il contenuto del puntatore di A viene copiato in C; nel puntatore di A viene inserito l'indirizzo di C. Il legame precedente tra A e B (tratteggiato) viene automaticamente rimosso come conseguenza delle operazioni di ridefinizione dei puntatori.

un percorso che passa per l'elemento da inserire. Osserva bene la figura:

- Situazione precedente: l'elemento A punta a B (nella figura, la freccia tratteggiata).
- Operazioni da compiere per inserire C tra A e B:
  - copio il puntatore di A in C;
  - scrivo l'indirizzo di C nel puntatore di A.
- Situazione finale: A punta a C, che punta a B (nella figura, le frecce a tratto continuo).

Naturalmente, l'operazione di inserimento non tocca i dati che fanno parte della struttura `X_DATI`: qualcuno dovrà provvedere ad aggiornarli, se necessario.

Ed ora, veniamo al codice: ci proponiamo di tradurre l'algoritmo che abbiamo appena enunciato in una funzione

```
void inserisci_lista(X_DATI *prec,X_DATI *elm);
```

dove `elm` è l'indirizzo dell'elemento da inserire, e `prec` è l'indirizzo dell'elemento di lista dopo il quale vogliamo che sia inserito il nuovo elemento (per rifarci all'esempio della figura, `elm` è l'indirizzo di C, e `prec` è l'indirizzo di A). Sembra un'operazione complicata, ma il codice è sorprendentemente semplice:

```
void inserisci_lista(prec,elm)
     X_DATI *prec, *elm;
{
  elm->pnt = prec->pnt;
  prec->pnt = elm;
  return;
}
```

Tutto qui? Certo: le istruzioni che ho scritto sono proprio la traduzione in linguaggio C dello schema della figura 8.9.

Forse un esempio più completo non farebbe male. Colgo l'occasione per sottolineare il fatto che nello schema della struttura di lista l'accento si deve porre sugli indirizzi, e non sui dati contenuti nei singoli elementi. Per questo ti propongo di scrivere un programmino molto semplice che crei una lista di elementi e stampi le informazioni contenute nei puntatori; non mi preoccuperò per nulla di metterci dei dati. Eccoti la funzione che stampa la struttura della lista:

```
void stampa_lista(ls)
     X_ROOT *ls;
{
  X_DATI *le;
  printf("Radice: indirizzo %p; pnt %p\n",ls,ls->pnt);
  for(le=ls->pnt; le!=NULL; le=le->pnt)
    printf("Elemento: indirizzo %p; pnt %p\n",le,le->pnt);
  return;
}
```

L'apparenza è alquanto semplice, ma non sottovalutare la difficoltà: se stai vedendo per la prima volta la struttura di lista ti conviene soffermarti un momento. Noterai che il ciclo `for` ha un aspetto un po' anomalo: non ci sono contatori. Questo è il bello: nella gestione della struttura contano solo gli indirizzi. Il ciclo è costruito così:

(i) inizializzazione: prelevo dalla radice l'indirizzo del primo elemento della lista;
(ii) test: l'indirizzo non deve essere un `NULL`, altrimenti significa che sono arrivato alla fine della lista;
(iii) istruzione da eseguire prima dell'iterazione successiva: sostituisco l'indirizzo dell'elemento corrente col puntatore all'elemento successivo.

Il resto è elementare: stampo l'indirizzo dell'elemento che sto considerando ed il contenuto del puntatore. Se la lista è consistente il puntatore dovrà coincidere con l'indirizzo dell'elemento che compare sulla riga successiva. Per inciso: il formato di scrittura `%p` significa proprio che quello che voglio stampare è un indirizzo. La stampa è in esadecimale, ma controllare se due puntatori siano uguali o no non comporta particolari difficoltà.

Ed ora ci vuole un main che esegua il test. Per farla breve, accontentiamoci di creare una lista di 10 elementi. Eccoti il file sorgente:

```
#include <stdio.h>
#include "lista_singola.h"
void inserisci_lista(X_DATI *prec,X_DATI *elm);
void stampa_lista(X_ROOT *ls);
X_ROOT Radice;
X_DATI Dati_lista[10];
int main()
{
  int j;
  Radice.pnt=NULL;
  for(j=0;j<10;j++)
    inserisci_lista((void *) &Radice,Dati_lista+j);
  stampa_lista(&Radice);
  exit(0);
}
```

Non sembra particolarmente impegnativo, ma un paio di trucchi del mestiere ci sono.

Anzitutto, noterai che ho posto tutti i dati relativi alla lista nell'area dei dati globali. Questo ha un duplice vantaggio: i dati da inserire in lista restano permanentemente in memoria, ed allo stesso tempo non rischio l'esaurimento dello stack di programma nel caso di tabelle di dati troppo estese. Naturalmente, l'allocazione dinamica dei dati tramite malloc sarebbe un ottimo metodo, ma qui non ho voluto esagerare.

Il secondo punto che intendo sottolineare è una faccenda più delicata, e nasconde il rischio di un errore subdolo. Se rileggi il sorgente della funzione inserisci_lista noterai che il primo argomento dovrebbe essere l'indirizzo di un elemento di lista, mentre qui io gli passo l'indirizzo della radice della lista. Sono due cose ben diverse. Come mai funziona? Ci sono due ottimi motivi. Il primo motivo, ed anche quello cruciale, è che in ambedue le strutture X_DATI ed X_ROOT il puntatore che gestisce la lista è il primo dato; in altre parole, l'indirizzo del puntatore coincide con l'indirizzo della struttura. Dal momento che la funzione inserisci_lista usa solo il puntatore, la differenza tra le due strutture non ha nessuna influenza. Permettimi di richiamare la tua attenzione su questo punto: sto usando pesantemente il fatto che la sistemazione dei dati entro la struttura rispetta in modo rigoroso l'ordine che io ho specificato. Il secondo motivo riguarda quel (void *) che ho inserito nell'istruzione

```
inserisci_lista((void *) &Radice,Dati_lista+j);
```

Significa semplicemente: so benissimo che sto passando un puntatore a qualcosa di diverso da quello che ho dichiarato nel prototipo, ma so bene quel che faccio.

Non ti piace? Beh, non ti resta che usare come radice una struttura X_DATI, sacrificando eventualmente un po' di spazio, oppure trovare il modo di fare a meno della radice. A te l'esercizio.

Ed ora prova a raccogliere tutti i pezzi del programma, compilare ed eseguire. Ecco il risultato che ho trovato sulla mia macchina:

```
Radice:   indirizzo 0x80497c0; pnt 0x80497b0
Elemento: indirizzo 0x80497b0; pnt 0x80497a0
Elemento: indirizzo 0x80497a0; pnt 0x8049790
Elemento: indirizzo 0x8049790; pnt 0x8049780
Elemento: indirizzo 0x8049780; pnt 0x8049770
Elemento: indirizzo 0x8049770; pnt 0x8049760
Elemento: indirizzo 0x8049760; pnt 0x8049750
Elemento: indirizzo 0x8049750; pnt 0x8049740
Elemento: indirizzo 0x8049740; pnt 0x8049730
Elemento: indirizzo 0x8049730; pnt 0x8049720
Elemento: indirizzo 0x8049720; pnt (nil)
```

Quali siano gli indirizzi effettivi, non ha grande importanza: dipende dal compilatore e dalla macchina. Importa invece che il puntatore `pnt` di ogni riga coincida con l'indirizzo dell'elemento che compare alla riga successiva, altrimenti la struttura di lista non sarebbe consistente. L'ultimo puntatore `(nil)` indica che la lista finisce lì.

E se volessi rimuovere un elemento dalla lista? Qui occorre un attimo di attenzione. Per capire che operazioni devi fare potrai far riferimento ancora alla figura 8.9: questa volta si tratta di sostituire il percorso a tratto continuo che collega in successione A, C e B con il percorso tratteggiato che collega A con B. Eccoti il procedimento:

- Situazione precedente: A punta a C, che punta a B (le frecce a tratto continuo).
- Operazione da compiere per rimuovere C: copio il puntatore di C in A;
- Situazione finale: l'elemento A punta a B (la freccia tratteggiata).

Sembra ancora più semplice dell'inserimento, ma sorge un piccolo problema: per rimuovere l'elemento C devo sapere dove si trova A, e questa informazione non si trova tra i dati che formano l'elemento C. Soluzione rapida: dovrò specificare che devo rimuovere l'elemento che segue A. Il codice, ancora una volta, è decisamente semplice:

```
void rimuovi_lista(prec)
      X_DATI *prec;
{
  prec->pnt = (prec->pnt)->pnt;
  return;
}
```

Intricato? Beh, un pochino sì, ma non più di tanto. Probabilmente ti si confondono le idee nel leggere `(prec->pnt)->pnt`. Ragiona con calma:

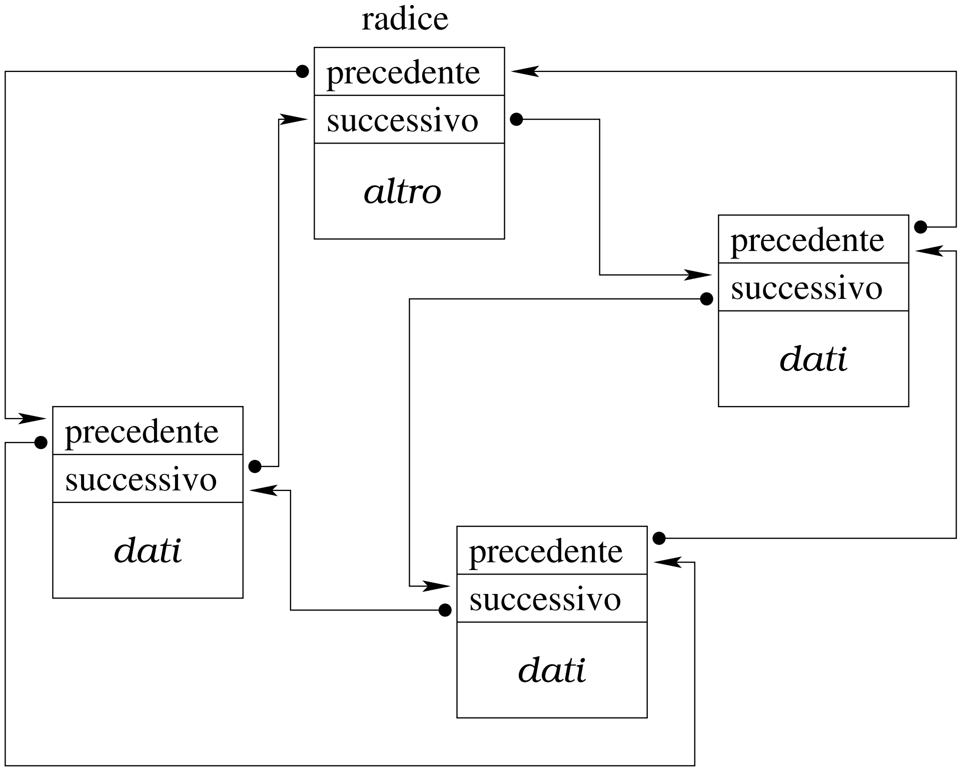


Figura 8.10. La struttura di lista bidirezionale. I puntatori formano due catene che percorrono la lista nelle due direzioni, e si chiudono ad anello sulla radice.

`prec` è l'indirizzo di un elemento; `prec->pnt` punta all'elemento successivo; `(prec->pnt)->pnt` prende il campo `pnt` dall'elemento che si trova all'indirizzo `prec->pnt`. Le parentesi non sono essenziali: le ho messe solo per eliminare qualunque dubbio sull'interpretazione dell'istruzione.

Se volessi eliminare il primo elemento della lista dovresti scrivere semplicemente

```
rimuovi_lista((void *) &Radice);
```

Non potrai invece specificare nella chiamata l'indirizzo dell'ultimo elemento: l'errore di accesso alla memoria sarebbe garantito!

8.3.2 Liste bidirezionali

L'organizzazione in lista che ti ho spiegato fin qui è molto semplice, ma ha i suoi difetti – che del resto sono già risultati abbastanza evidenti. Uno schema più completo consiste nel creare una lista che possa essere percorsa sia in avanti che all'indietro, e che non abbia salti nel vuoto come quello implicito nel puntatore `NULL` dell'ultimo elemento.

Lo schema non presenta grosse differenze rispetto a quello che hai visto: si tratta solo di realizzare una doppia catena di puntatori, come ti illustro nella figura 8.10. Noterai le due differenze principali:

- i puntatori ora sono due, uno all'elemento precedente, l'altro al successivo;
- le due catene dei puntatori in avanti ed all'indietro si chiudono sulla radice, realizzando una struttura circolare.

Naturalmente lo schema che ti sto illustrando non è l'unico possibile. Una variazione minima consisterebbe nel creare una lista che abbia una testa ed una coda distinte, invece che farle coincidere nella radice. Altre variazioni ti si presenteranno in modo naturale, quando ti serviranno.

Veniamo, questa volta rapidamente, al codice C. Per prima cosa, sarà conveniente creare un file `lista_doppia.h` che contenga le definizioni delle strutture. Ecco un esempio:

```
typedef struct y_dati {
  struct y_dati *pn_prec;
  struct y_dati *pn_succ;
  double x;
} Y_DATI;
typedef struct y_root {
  Y_DATI *pn_prec;
  Y_DATI *pn_succ;
} Y_ROOT;
```

Non dovresti avere difficoltà ad interpretare le istruzioni: sia la radice che il pacchetto dei dati contengono una coppia di puntatori, rispettivamente all'elemento precedente ed al successivo.

La creazione della lista richiede ancora la definizione dei puntatori localizzati nella radice. Qui c'è una differenza da non trascurare: dal momento che le catene si devono chiudere sulla radice, ambedue i puntatori della radice dovranno contenere l'indirizzo della radice stessa. Ecco le istruzioni che servono:

```
Y_ROOT Radice;
...
Radice.pn_prec = Radice.pn_succ = (void *) &Radice;
```

Anche qui, quel `(void *)` è necessario perchè l'assegnazione coinvolge puntatori ad oggetti diversi; la consistenza è garantita dal fatto che i puntatori sono posizionati sempre all'inizio della struttura, e nello stesso ordine.

E veniamo all'operazione di inserimento, che ti illustro nella figura 8.11. Qui ci sono quattro puntatori da aggiornare. Ecco il codice:

```
void inserisci_lista(prec,elm)
     Y_DATI *prec, *elm;
{
  elm->pn_prec = prec;
  prec->pn_succ->pn_prec = elm;
  elm->pn_succ = prec->pn_succ;
```

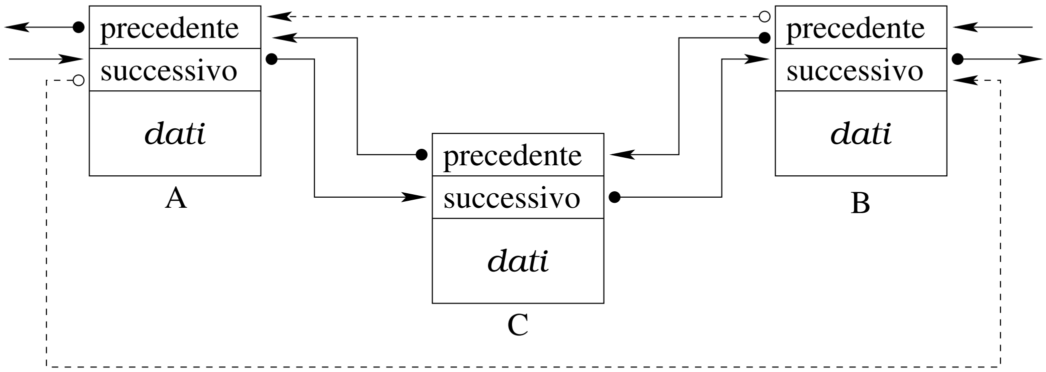


Figura 8.11. Inserimento di un elemento in una lista bidirezionale: l'elemento C viene inserito dopo l'elemento A mediante una semplice ridefinizione dei puntatori: il contenuto del puntatore all'elemento successivo in A viene copiato in C, e sostituito dall'indirizzo di C. Analogamente, il contenuto del puntatore all'elemento precedente in B viene copiato in C, e sostituito dall'indirizzo di C.

```
  prec->pn_succ = elm;
  return;
}
```

Non particolarmente complicato, come puoi ben vedere. Il solo punto che ti richiederà un attimo di riflessione è l'istruzione

```
prec->pn_succ->pn_prec = elm;
```

Si tratta semplicemente di aggiornare il puntatore all'indietro nell'elemento B. L'indirizzo di B non è passato come parametro, e del resto non servirebbe; lo posso prelevare utilizzando l'informazione contenuta in A.

La rimozione di un elemento è un'operazione simile, e questa volta la struttura bidirezionale ci mette in condizione di specificare direttamente quale elemento debba essere rimosso. Ecco il codice:

```
void rimuovi_lista(elm)
      Y_DATI *elm;
{
  elm->pn_succ->pn_prec = elm->pn_prec;
  elm->pn_prec->pn_succ = elm->pn_succ;
  return;
}
```

Con questa funzione sarai in grado di rimuovere qualunque elemento da un qualunque punto della lista. Attenzione pero': non rimuovere la radice!

Utilizzando come base le funzioni che ti ho illustrato potrai essere in grado di gestire un buon numero di situazioni: ad esempio, non ti sarà difficile ordinare la lista secondo un qualche criterio – ti basterà avere un buon algoritmo di ordinamento, o estrarne gli elementi che soddisfano a determinate condizioni. Il giorno in cui userai questo modo di organizzare i dati ti renderai conto di quanto sia flessibile ed utile.

A conclusione di questo capitolo lascia che ti ricordi ancora una volta quello che ho cercato di comunicarti fin dall'inizio: la capacita di lavorare direttamente sugli indirizzi costituisce uno degli strumenti più utili della programmazione. Quello che ho cercato di spiegarti in queste note è solo un rapido sguardo.