5

LE FUNZIONI

Questo capitolo tratta in modo più approfondito l'uso delle funzioni, soffermandosi in particolare sul passaggio di argomenti e sull'uso di variabili locali e globali.

A dire il vero, dal punto di vista dell'utilizzo in C le informazioni nuove rispetto a quelle del paragrafo 3.3 non sono molte: una volta definiti correttamente gli argomenti la comunicazione tra modulo chiamante e funzione non dovrebbe creare problemi. La parte impegnativa sta piuttosto nel suddividere un programma in funzioni che eseguano parti ben definite dell'elaborazione; questo non ha molto a che vedere con la sintassi del linguaggio: occorre evitare la programmazione frettolosa, e costruirsi uno schema preciso del programma prima di impegnarsi nella traduzione in codice C.

Il mio scopo qui è guardare ancora un po' dietro le quinte, per mettere in luce i meccanismi fondamentali che vengono messi in atto. Alcuni passaggi sono decisamente tecnici, ma ormai la fase dello svezzamento dovrebbe essere ragionevolmente superata.

5.1 Comunicazione tra funzioni e dati locali

Del meccanismo di chiamata a funzioni e di come si preparano i sorgenti abbiamo già discusso nel paragrafo 3.3. Comunque, prima di discutere come passare dati tra funzioni diverse credo sia opportuno richiamare brevemente quali siano le regole fondamentali da rispettare.

5.1.1 Scrivere e richiamare una funzione

La sintassi generale è

$\langle tipo \rangle$ $\langle nome\ funzione \rangle$`(`$\langle argomenti \rangle$`)`

dove:

- il campo $\langle tipo \rangle$ si riferisce al valore restituito dalla funzione, e può essere `int`, `double`, `char` o qualunque altro tipo di dato ammesso. Se la funzione non deve restituire nessun valore tramite il nome le si assegnerà il tipo `void`. In mancanza di indicazioni il compilatore assume `int`.

- il campo $\langle nomefunzione \rangle$ è affidato all'immaginazione del programmatore, a parte l'avvertenza che il primo carattere non deve essere numerico. Il solo caso in cui la scelta del nome è obbligata è quello del modulo `main`, che definisce l'indirizzo di partenza del programma.
- il campo $\langle argomenti \rangle$ costituisce il meccanismo diretto di passaggio delle informazioni alla funzione chiamata; di fatto specifica cosa ci si aspetta di trovare nella tabella degli argomenti.

La funzione chiamante conterrà un'istruzione del tipo

`y =` $\langle nome\ funzione \rangle$`(`$\langle argomenti \rangle$`);`

il nome specifica quale debba essere la funzione chiamata, e la lista degli argomenti di fatto specifica cosa debba entrare nella tabella degli argomenti predisposta dal compilatore. Il dato `y` deve essere dello stesso tipo assegnato alla funzione, o almeno di un tipo compatibile: se necessario, il compilatore predispone una conversione di tipo come per qualunque istruzione di assegnazione.

Se la funzione è di tipo `void` allora la chiamata deve essere necessariamente della forma

$\langle nomefunzione \rangle$`(`$\langle argomenti \rangle$`);`

perché non c'è nessun valore da assegnare. Quest'ultima forma è valida anche se la funzione restituisce un valore: se il compilatore non sa dove memorizzarlo lo butta nel cestino.

È compito del programmatore garantire che ci sia esatta corrispondenza tra numero e tipi di argomenti. Fa ben attenzione a questo punto: la mancata corrispondenza tra tipo della funzione e tipo di dato che la riceve, oppure tra numero e tipo di argomenti, è uno degli errori più frequenti, ed anche difficili da scovare per un programmatore non esperto. Per ridurre i rischi al minimo il compilatore mette a disposizione due strumenti:

- la dichiarazione del *prototipo* della funzione: vedi la descrizione nel paragrafo 3.3.4; questo è uno degli strumenti standard del linguaggio C, quindi deve essere disponibile su qualunque macchina e qualunque compilatore.
- la compilazione col comando

  ```
  gcc -Wall ...
  ```

  che forza la scrittura di messaggi che segnalano tutti i casi di mancata corrispondenza; naturalmente, funziona se sono stati predisposti correttamente tutti i prototipi delle funzioni chiamate. Questa opzione è tipica del compilatore `gnu-cc`, quindi non è detto che si trovi tale e quale su tutti i sistemi; qualcosa di equivalente però c'è quasi sempre.

5.1.2 I nomi e dati locali

Come abbiamo visto negli esempi del paragrafo 3.3 la funzione può ben contenere delle dichiarazioni che riservino memoria per depositarvi dei dati. Ricorda, ad esempio, la funzione `media` del paragrafo 3.3.5:

```
double media(vet,n)
     double vet[];
     int n;
{
    double somma;
    int j;
    for(somma=0, j=0; j<n; j++) somma = somma+vet[j];
    return(somma/n);
}
```

Ti ho spiegato che le dichiarazioni `double` e `int` non sono tutte equivalenti: quelle riferite ai parametri, `double vet[]` e `int n`; servono a spiegare cosa ci si aspetta di trovare nella tabella degli argomenti, e non allocano memoria; quelle riferite a variabili locali invece chiedono esplicitamente che venga riservata della memoria per depositarci i dati.

Ma cosa avviene di questi dati? e dei nomi? Questi sono punti da chiarire, ed è il momento di farlo.

Cominciamo coi nomi. Hai visto che nessuno ti proibisce di chiamare la funzione `media` scrivendo

```
y = media(x,m);
```

che gli argomenti qui si chiamino `x,m` mentre nella definizione poco sopra si chiamavano `vet,n` non conta proprio nulla: la corrispondenza viene realizzata solo attraverso la tabella degli argomenti, e non attraverso i nomi; l'importante è che `x` sia l'indirizzo di un vettore di tipo `double` e `m` sia un dato di tipo `int`. Anzi, ti dirò di più: quando il compilatore produce il modulo rilocabile usa i nomi per predisporre la tabella degli argomenti nel programma chiamante e il prelievo degli argomenti nella funzione chiamata, e poi cancella completamente i nomi.

Un discorso analogo vale per i dati dichiarati all'interno della funzione: il compilatore usa i nomi solo per predisporre la memoria e tradurre correttamente tutte le istruzioni che fanno uso di questi dati; una volta eseguita la traduzione i nomi non esistono più. Se all'interno di un'altra funzione io uso gli stessi nomi `somma` o `j` non c'è nessuna relazione con i nomi che compaiono nella funzione `media`, e non c'è rischio di confusione: si tratta di nomi *locali*, che nascono nel momento in cui il compilatore trova le dichiarazioni, e muoiono con la parentesi graffa che chiude il corpo della funzione. Come ti ho già detto, i nomi locali sono come cenere nel vento.

I soli nomi che sopravvivono nel modulo rilocabile sono quelli *globali*, o *esterni*. Hai già fatto la conoscenza con due casi: il nome della funzione stessa, che rappresenta l'entry point, ed i riferimenti a funzioni esterne che devono essere completati dal linker. Tra poco ne vedremo un altro.

Veniamo invece ai dati. La memoria per variabili dichiarate localmente, nel corpo della funzione, viene assegnata al momento dell'esecuzione, e più precisamente all'atto della chiamata alla funzione stessa. Al rientro dalla funzione la memoria riservata ai dati locali viene liberata, ed è disponibile per essere riutilizzata. Questo fatto ha delle conseguenze da tenere ben presente:

- il compilatore si preoccupa di assegnare la memoria per i dati locali, ma si guarda bene dal metterci qualunque cosa. Quindi, il contenuto delle variabili al momento della dichiarazione è da considerarsi indefinito.
- se la stessa funzione viene chiamata più volte non si può contare sul fatto che tra una chiamata e l'altra il valore di una variabile sia stato conservato. [1]

Questo tipo di gestione delle variabili locali ad un modulo viene detto *allocazione dinamica*, in contrapposizione al metodo di *allocazione statica* che viene adottato da altri linguaggi. [2]

5.1.3 Una digressione: la pila di memoria, o stack

A questo punto temo di aver sollevato più dubbi di quanti non ne abbia chiariti. Come diavolo fa il compilatore a riservare e liberare memoria ad ogni chiamata di funzione senza mandare tutto in confusione? E poi, resta sempre quel meccanismo oscuro del rientro della funzione al punto di chiamata che non può essere definito dal linker — ed infatti non l'ho mai citato tra i riferimenti globali. Come funziona?

Provo a risponderti, dicendoti però subito che quello che ti spiegherò corrisponde nelle linee essenziali al meccanismo che viene effettivamente utilizzato, ma i dettagli dipendono pesantemente dal sistema, in quanto possono intervenire sia la struttura hardware della CPU, sia le convenzioni dei linguaggi o dei compilatori.

Per prima cosa devo spiegarti cosa si intende per *pila di memoria*, o, con un termine che ricorre tipicamente nei manuali e che adotterò, *stack*. [3] Per capirne il significato e l'uso pensa appunto ad una pila di dischi infilati su un asse, come nel gioco della torre di Hanoi: non ti sarà possibile estrarre un disco che si trova a metà pila senza avere estratto tutti quelli che stanno sopra. La pila di memoria funziona appunto a questo modo: l'ultimo dato

---

[1] Le due note qui sopra devono essere tenute ben presenti. Capita sovente che i programmatori assumano che una variabile di cui non hanno definito il contenuto sia stata azzerata al momento dell'allocazione della memoria — un'abitudine talvolta indotta dal fatto che alcuni compilatori in effetti azzerano la memoria allocata. Si tratta di un errore da evitare nel modo più assoluto: le conseguenze possono essere del tutto imprevedibili, perchè il valore di una variabile è semplicemente quello che per caso si trovava in quel momento nella cella di memoria che le è stata riservata.

[2] Un esempio tipico è il FORTRAN, in cui tutte le variabili vengono allocate in uno spazio predefinito dal compilatore e dal linker.

[3] Il termine stack significa appunto catasta, o pila. Ma si tratta ormai di un termine tecnico praticamente universale, per cui evito di usarne la traduzione per mantenere uniformità di linguaggio rispetto ai manuali tecnici.

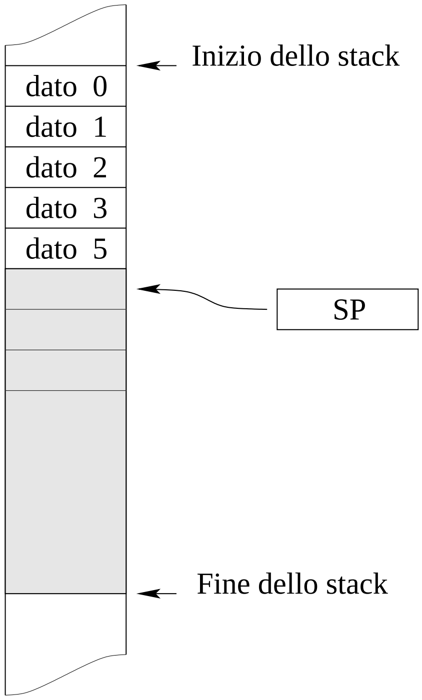

Figura 5.1. La struttura a *pila di memoria*, o *stack*. Una zona contigua di memoria viene gestita col meccanismo *LIFO*, ovvero *Last In First Out*. Nel caso dello stack di programma la gestione della struttura viene assicutrata tramite il registro *stack pointer*, che contiene sempre l'indirizzo della prima cella di stack disponibile.

inserito è il primo che viene prelevato. Questo comportamento è codificato nel nome *LIFO* che si dà ad una struttura di questo tipo: LIFO è un acronimo per *Last In First Out*.

Per la gestione dello stack si fa uso di un puntatore che tiene traccia, ad esempio, dell'indirizzo della prima cella libera: lo schema è illustrato in figura 5.1. Ogni programma è dotato di uno stack di memoria, ed il puntatore è uno dei registri generali della CPU, che viene detto *Stack Pointer*, abbreviato in SP.

La gestione dei dati richiede due operazioni: *push* e *pop*. [4] L'operazione di push richiede due passi:

(i) deposita il dato all'indirizzo contenuto nello stack pointer;
(ii) incrementa lo stack pointer in modo che punti alla cella successiva, che è la prima libera.

L'operazione di pop richiede anch'essa due passi:

(i) decrementa lo stack pointer, in modo che punti all'ultima cella riempita;

[4] Letteralmente, push significa spingere o comprimere, e pop significa far scoppiare, come ad esempio si fa saltare il tappo di una bottiglia di spumante.

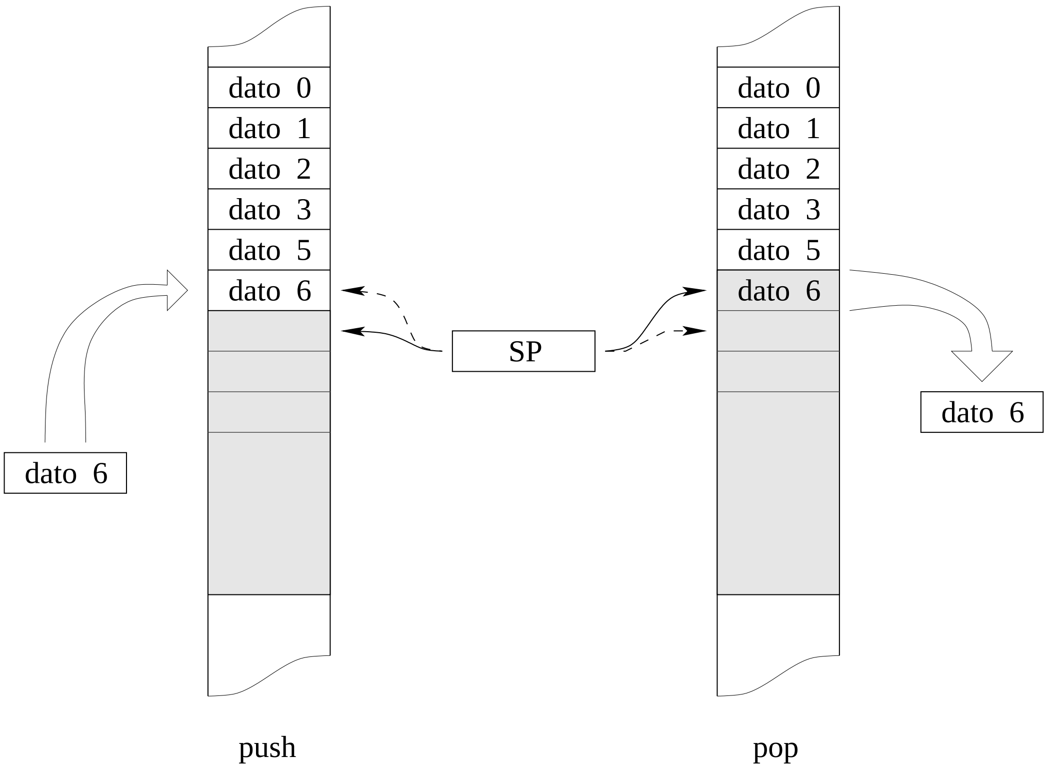

Figura 5.2. Le operazioni fondamentali per la gestione di dati nello stack. L'operazione di *push* consiste nel depositare un dato nella cella indirizzata dal registro SP, e subito dopo incrementare SP in modo che punti alla cella successiva. L'operazione di *pop* consiste nel decrementare il registro SP, e pooi prelevare il dato dallo stack. Il dato prelevato viene considerato eliminato, e potrà essere riscritto da un successivo push.

(ii) preleva il dato dalla cella indirizzata dallo stack pointer.
Nota che c'è un'inversione nell'ordine delle operazioni. [5]

Per inciso, è possibile costruire uno stack di memoria indipendente da quello di programma usando semplicemente un vettore ed un puntatore intero. Per questo scopo sono particolarmente efficienti le istruzioni di autoincremento e autodecremento degli indici. Supponiamo, ad esempio, di aver definito un vettore `vet` di lunghezza sufficiente, e di voler usare come puntatore la variabile di tipo intero `i`. Basta seguire delle semplici regole:

---

[5] Naturalmente, non è proibito riempire lo stack partendo dagli indirizzi alti invece che da quelli bassi, il che significa che un push decrementa lo stack pointer e un pop lo incrementa, o decidere che lo stack pointer punta all'ultima cella occupata invece che alla prima libera: è questione di convenzioni. La sola cosa importante è che, una volta stabilite, le convenzioni vengano uniformemente rispettate da tutti i programmi di un sistema, pena il rischio di malfunzionamenti.

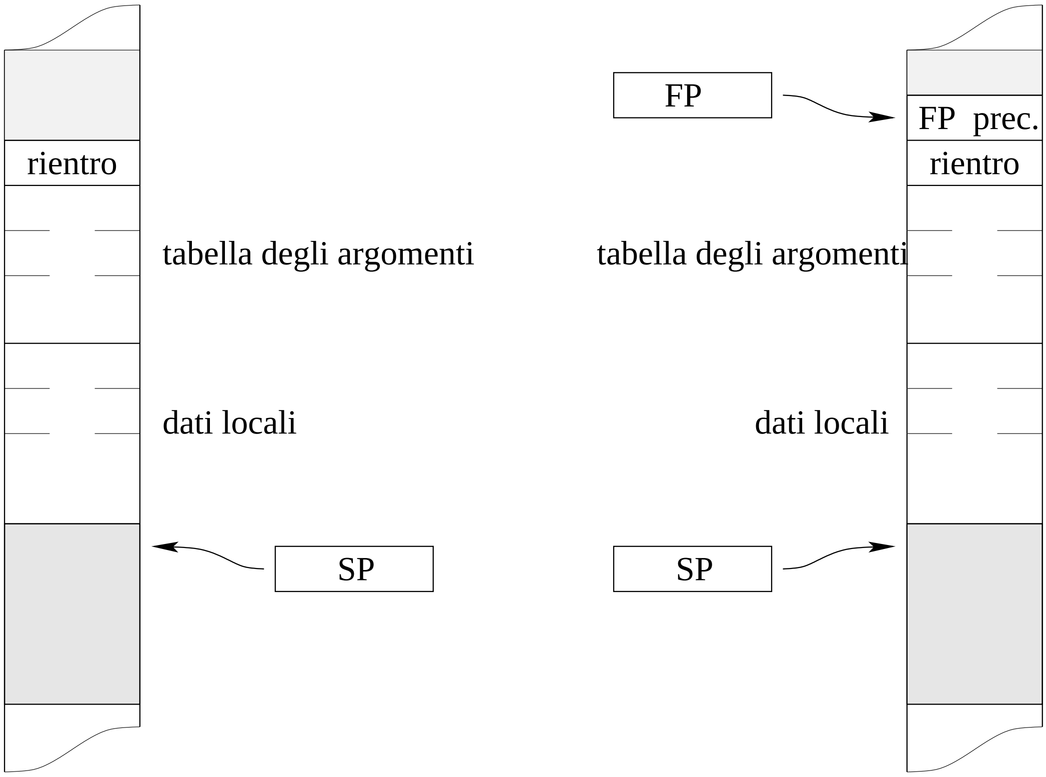

Figura 5.3. Due possibili configurazioni del call frame. Nel primo caso il frame contiene l'indirizzo di rientro, la tabella degli argomenti ed i dati locali; lo stack pointer deve essere riposizionato prima del rientro, ma non c'è modo di ricostruire la struttura del frame esaminando semplicemente lo stack. Nel secondo caso il registro FP (frame pointer) punta all'inizio del frame corrente.

- all'inizio si definisce il puntatore allo stack in modo che indirizzi il primo elemento del vettore; l'istruzione è semplicemente `i=0`;
- l'operazione di push viene eseguita con l'istruzione
  `vet[i++]=`⟨*dato*⟩;
- l'operazione di pop viene eseguita con l'istruzione
  ⟨*variabile*⟩`=vet[--i]` .

Nota bene che è cruciale il fatto di aver scritto `i++` (con `++` che segue `i`) per il push e `--i` (con `--` che precede `i`) per il pop: così si deve fare per rispettare l'ordine delle operazioni.

Va da sé che la struttura di stack non è immune da rischi: è compito del programmatore assicurarsi che non avvenga un *overflow* dello stack (il puntatore supera il limite massimo consentito) o un *underflow* (il puntatore diventa negativo). Gli effetti, al solito, sarebbero imprevedibili.

Se hai capito come funziona la struttura di stack sei pronto per apprendere come si possano realizzare contemporaneamente ed in modo semplice sia il meccanismo di chiamata di funzione che quello di assegnazione di dati locali che vengono liberati al momento del rientro al modulo chiamante. Basta usare lo stack di programma.

La parte sinistra della figura 5.3 illustra il meccanismo.

- Prima della chiamata viene riservata nello stack una cella destinata a contenere l'indirizzo di rientro — quello dell'istruzione immediatamente successiva alla chiamata. La tabella degli argomenti occupa l'area successiva a quella dell'indirizzo di rientro, e viene predisposta dal modulo chiamante.
- All'atto della chiamata il valore corrente di IP (Instruction Pointer) viene copiato con un push nella cella indirizzata dallo stack pointer, che così punta all'inizio della tabella degli argomenti.
- La funzione chiamata preleva gli argomenti, e incrementa lo stack pointer quanto basta per lasciare spazio libero per le variabili locali; questo spazio viene usato dalla funzione facendo riferimento allo stack pointer solo come base di indirizzamento, senza modificarlo.
- Il rientro viene effettuato restituendo allo stack pointer il valore che aveva all'ingresso, e ripristinando il contenuto del registro IP con un pop. In tal modo lo stack si trova automaticamente riposizionato alla fine dell'area utilizzata dal modulo chiamante, e tutta l'area di stack successiva è diventata riutilizzabile.

L'area di stack utilizzata da una funzione viene identificata come *call frame*[6] Tuttavia, per quanto il frame sia ben definito come unità logica, il meccanismo che ho esposto fin qui non prevede nessuna informazione che permetta di ricostruirne la struttura semplicemente analizzando il contenuto dello stack. Per questo è diventato ormai comune l'uso di un secondo registro, detto *Frame Pointer* ed indicato con *FP*, che punta al primo indirizzo del call frame locale della funzione in esecuzione, come illustrato nella parte destra della figura 5.3.

La ricostruzione della catena dei frames generati da chiamate successive a funzioni è possibile in quanto al momento della chiamata viene generato un nuovo frame salvando nello stack il valore corrente di FP, e caricando in FP il valore corrente dello stack pointer. In tal modo si crea entro lo stack una catena di celle ciascuna delle quali indirizza la precedente, come illustrato in figura 5.4. Il registro FP punta alla prima cella dello stack corrente, che a sua volta punta alla prima cella del frame precedente &c, fino al frame di livello zero che corrisponde al modulo `main`.

5.1.4 L'allocazione delle variabili locali

La presenza di uno stack di programma, come ti ho spiegato, fornisce i meccanismi necessari sia per gestire il rientro dopo la chiamata di funzioni, che per

---

[6] Letteralmente frame significa cornice, o intelaiatura, o struttura. Nell'ambiente informatico viene ampiamente utilizzata per identificare un blocco di dati che si pensa in qualche modo come isolato.

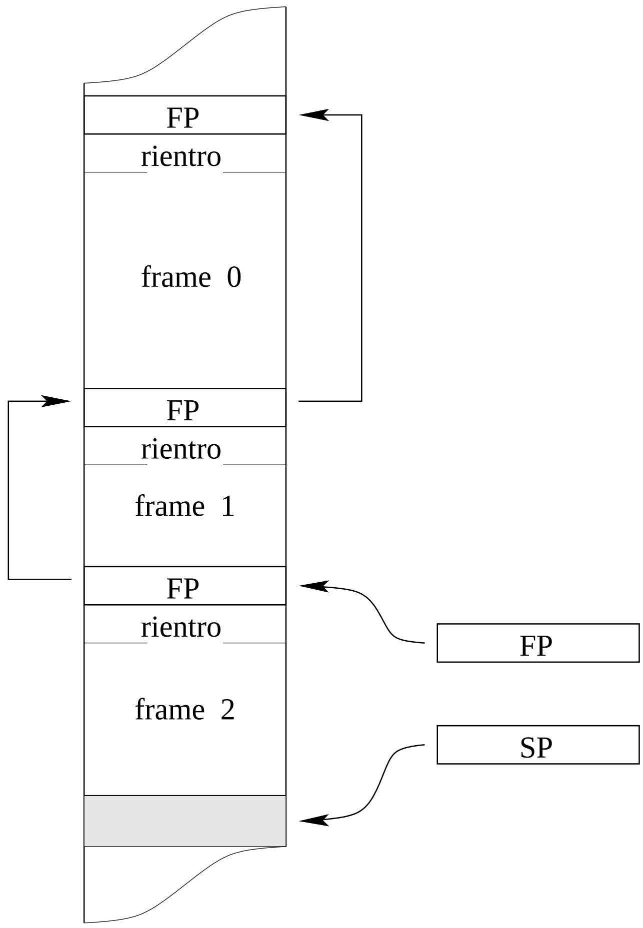

Figura 5.4. La struttura dello stack con uso dei call frames: il registro FP (frame pointer) consente di ricostruire la catena dei frames fino a quello corrispondente al main.

gestire il trasferimento di dati tra modulo chiamante e funzione, che per allocare dei dati locali la cui esistenza finisce con la il rientro della funzione: tutte queste informazioni vengono depositate nello stack del programma. Il riposizionamento dello stack pointer al momento del rientro assicura che tutta la memoria utilizzata localmente dalla funzione tornerà ad essere disponibile per utilizzi successivi — ad esempio, per chiamare un'altra funzione.

La domanda a questo punto è quasi obbligata: ma lo stack non si esaurisce mai? Sí, purtroppo, ed è una situazione abbastanza sgradevole. Ma c'è anche modo di uscirne: te lo spiego nel prossimo paragrafo.

5.2 Dati e nomi globali

Il meccanismo di allocazione dinamica di memoria nello stack del programma presenta due vantaggi:

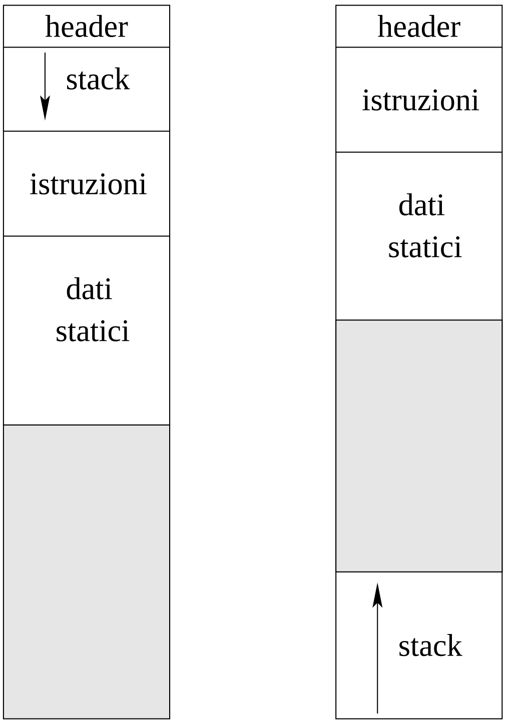

Figura 5.5. Due possibili disposizioni delle diverse sezioni del programma eseguibile.

- rende più efficiente l'uso della memoria, in quanto tende a liberare sistematicamente quella che non viene utilizzata;
- consente la scrittura di funzioni ricorsive, ossia funzioni che possono richiamare sé stesse. [7]

Accanto ai pregi ci sono però anche i difetti:

- non permette, se non in modo complicato, di creare aree di memoria che siano accessibili a funzioni che non si richiamano direttamente;
- l'allocazione di aree di memoria di grosse dimensioni non è possibile, perché si rischia di eccedere i limiti dello stack.

Il superamento di questi problemi è possibile mediante l'allocazione di aree dati statiche.

5.2.1 Ancora una digressione: la struttura di un programma eseguibile

Lascia che ti conduca ancora una volta a guardare dietro le quinte. Un mo-

[7] Non è in queste note che può trovare spazio una discussione dettagliata sulla ricorsività. Mi limito ad osservare che l'allocazione statica di memoria rende di fatto impossibile scrivere funzioni ricorsive. Il problema è che ogni modifica ai dati locali viene resa permanente, perchè il dato locale, a qualunque livello di chiamata, si trova sempre allo stesso indirizzo. Ne segue che al rientro della funzione il modulo chiamante si trova ad operare su dati che nel frattempo sono stati modificati.

dulo assoluto è composto sommariamente da quattro aree distinte:
- un *header*, o intestazione, che contiene informazioni di controllo usate dal sistema;
- lo stack;
- l'area che contiene le istruzioni;
- l'area che contiene dati statici.

La disposizione delle varie aree di memoria nel modulo assoluto, o eseguibile, non è casuale. La figura 5.5 illustra due schemi possibili. La differenza principale sta proprio nella disposizione dello stack.

Nel caso illustrato nella parte sinistra della figura lo stack viene piazzato tra l'header e l'area delle istruzioni, ed ha ha una dimensione ridotta (tipicamente qualche Kbyte). L'area dati viene invece sistemata nella parte finale. Se durante l'esecuzione viene richiesta l'allocazione di ulteriore memoria questa andrà ad occupare gli indirizzi successivi a quelli dell'area statica, che nella figura sono rappresentati in grigio. Vada sé che la dimensione ridotta dello stack pone limitazioni considerevoli ai meccanismi di allocazione dinamica che ho descritto.

Una seconda disposizione, illustrata nella parte destra della figura 5.5, consiste nel portare lo stack in fondo alla memoria disponibile, e nel rovesciarlo, nel senso che l'occupazione parte dall'indirizzo più alto e cresce verso gli indirizzi più bassi — come potresti fare anche su se decidessi di riempire un vettore partendo dal fondo. Resta così uno spazio libero tra l'area dei dati statici e lo stack, disponibile per la richiesta di allocazione di ulteriore memoria nel corso dell'esecuzione. In questo secondo caso lo stack può raggiungere anche dimensioni ragionevolmente grosse, pur con qualche limite che esiste sempre, tipicamente di qualche Mbyte.[8]

A questo punto immagino la domanda cruciale: ma se mi hai appena spiegato che lo spazio di memoria per i dati viene allocato dinamicamente che diavolo ci va a finire nell'area dei dati statici? È appunto di questo che intendo parlarti.

5.2.2 Nomi locali e globali per i dati

Partiamo con un esempio: eccoti un frammento di codice

```
#include ...
int vet[10];
int main() {  ...  }
```

[8] Probabilmente ti verrà spontaneo chiederti perché si usi la prima disposizione, visto che la seconda sembra presentare tutti i vantaggi. La risposta mi porterebbe troppo lontano, perché la seconda disposizione fa uso in modo essenziale dei meccanismi di memoria virtuale. Pur con rammarico, sono costretto a limitare le dimensioni di queste note lasciando che per te il termine *memoria virtuale* resti poco più che un nome — a meno, ovviamente, che non tu non sia già abbastanza esperto da conoscerne il significato.

Avrai notato che la dichiarazione `int vet[10]` compare al di fuori del corpo del modulo `main`, come se fosse una faccenda del tutto indipendente. Lo è, infatti. L'istruzione significa ancora: voglio che tu mi riservi spazio per un vettore di 10 elementi di tipo `int`; ma si aggiunge di un'ulteriore richiesta: la memoria deve essere allocata al di fuori del corpo di qualunque funzione, in un'area di memoria statica, ed il nome `vet` deve essere globale.

La regola è generale: le variabili dichiarate al di fuori dal corpo di qualsiasi funzione trovano spazio nell'area dati statica del programma, ed i loro nomi sono dichiarati globali, cioè riconoscibili anche al di fuori del rilocabile, al pari dei nomi delle funzioni. Il linker provvede a far coincidere tutti gli indirizzi che si riferiscono allo stesso nome.

Un esempio più esplicito può essere utile. Supponi di aver preparato due files distinti. Il primo, che con la consueta immaginazione chiamerò `prova_riempi.c`, contiene un modulo `main` che richiama una funzione `riempi`:

```
#include <stdio.h>               /* Prototipo di riempi */
void riempi(int n);              /* Memoria statica */
int vet[10];
int main()                       /* Main */
{
  int j;
  riempi(10);
  for(j=0;j<10;j++)  printf("%5d",vet[j]);
  printf("\n");
  exit(0);
}
```

Il secondo file contiene la funzione `riempi`, che si limita a riempire un vettore intero mettendo in ciascun elemento il quadrato della sua posizione:

```
int vet[10];                    /* Memoria statica */
void riempi(n)                  /* Funzione riempi */
     int n;
{
  int j;
  for(j=0; j<n; j++) vet[j] = j*j;
  return;
}
```

La differenza rispetto a tutti gli esempi che ti ho mostrato fin qui è che il vettore non viene passato come argomento della funzione. Quando il compilatore trova il nome `vet` lo cerca prima nell'elenco delle variabili locali, poi, se non l'ha trovato, lo cerca tra quelle globali. In questo caso la funzione `riempi` va a definire il contenuto di un vettore globale; al rientro della funzione il `main` trova i dati modificati. Prova a compilare i due moduli e ad eseguire il programma, e ti comparirà la risposta

```
0    1    4    9   16   25   36   49   64   81
```

Sembra tutto facile, ma attento: anche qui ci sono i trabocchetti, ed anche pericolosi. Mettere dei dati in comune ha dei vantaggi, ma significa anche che qualunque funzione può liberamente modificare i dati semplicemente usando il nome globale — magari solo perché il programmatore ha usato un nome che intendeva come locale, ma si è dimenticato di dichiararlo entro il corpo della funzione. Non ci sono molte difese automatiche: il programma deve essere progettato in modo accurato, onde evitare sorprese. Una buona regola è mantenere l'abitudine di assegnare nomi lunghi e non banali alle variabili globali, riservando i nomi brevi per quelle locali. Si può anche giocare sul fatto che il compilatore C distingue tra caratteri minuscoli e maiuscoli: i nomi `vet` e `Vet` non sono la stessa cosa. Regole accettabili sono: tutti i nomi globali devono iniziare con una lettera maiuscola; oppure: tutti i nomi globali devono avere un certo prefisso, o un certo suffisso, eventualmente separati dal resto del nome col carattere _ (sottolineatura, o underline). Sono solo esempi: le possibilità, come insegnano gli esperti del controllo dei voti elettorali, sono tante.

Un secondo punto a cui fare attenzione riguarda la lunghezza dei vettori dichiarati come globali: le dichiarazioni devono essere inserite in tutti i moduli, ed è indispensabile che le lunghezze dichiarate siano consistenti: l'invasione di memoria è in agguato più che mai.

5.2.3 Prima le variabili locali o quelle globali?

Nessuno proibisce di far uso di variabili locali e globali che hanno lo stesso nome. Ma che succede in questo caso? Un esempio: prova a modificare il sorgente della funzione `riempi` in questo modo:

```
{
  int vet[10];
  int j;
    ....
}
```

Una modifica un po' bizzarra, d'accordo, ma un esempio è un esempio. Prova a compilare ed eseguire, e vedrai che il risultato cambia. Sul mio sistema ho trovato

```
0    0    0    0    0    0    0    0    0    0
```

Non è quello che mi sarei aspettato. Perché?

La regola seguita dal compilatore te l'ho enunciata sopra, ma lasciamela ripetere in dettaglio:

- quando il compilatore trova una dichiarazione per una variabile (singola o vettore, poco importa) riserva la memoria, se necessario, e si compila una tabella personale in cui accanto al nome mette l'indirizzo. Più precisamente, di tabelle ne ha (almeno) due: una è quella locale del modulo che sta compilando, la seconda è quella globale. I nomi dichiarati esternamente vengono elencati nella tabella globale, lasciando al linker il compito di assegnare definitivamente gli indirizzi; i nomi dichiarati all'interno del corpo delle funzioni restano nella tabella locale.

- quando il compilatore trova il nome di una variabile ne cerca l'indirizzo nella tabella locale; se nella tabella locale non lo trova – e solo in questo caso – lo cerca nella tabella globale. Se anche in quella non lo trova, segnala un errore e se ne lava le mani.

Con queste informazioni è facile capire cosa sia successo al programma che ti ho illustrato sopra. Avendo trovato il nome `vet` nella tabella locale, il compilatore non è neppure andato a cercare in quella globale. La funzione dunque ha riempito il vettore locale, sistemato nello stack, che al rientro della funzione è stato buttato nel cestino, e non ha minimamente toccato quello globale, che invece interessava al `main`.

5.2.4 Qualche strumento comodo

Ho già sottolineato il fatto che le dichiarazioni di variabili globali devono essere ripetute in tutti i files sorgenti i cui moduli facciano uso di quelle variabili. Semplice da dire, ma basta, anche per una sola volta, scrivere un programma che presenti un minimo di complessità per rendersi conto di quanto possa essere noiosa e pericolosa un'operazione del genere.

Il problema non è tanto la scrittura del programma, quanto piuttosto le correzioni o le modifiche successive. Un esempio banale: seguendo i buoni consigli degli esperti hai scritto un programma spezzando l'elaborazione in un dozzina di funzioni distinte. Siccome c'erano molti dati che dovevano essere utilizzati – o talvolta modificati – da diverse funzioni hai provveduto a piazzare dei vettori nella memoria statica, sfruttando il meccanismo dei nomi globali. Risultato: una quindicina di linee di dichiarazioni ripetute tali e quali in una dozzina di files diversi.

Meriti un plauso per avere seguito tutte le migliori regole, ma forse il plauso ti sembrerà una presa in giro il giorno in cui ti renderai conto che avevi sottostimato le dimensioni dei vettori. Quindi dovrai correggere tutte le dichiarazioni in una dozzina di files, stando ben attento a non dimenticarne nessuno e a non commettere errori. Commenti?... preferisco non riferirli.

C'è un modo più semplice e meno pericoloso? Certamente, anzi, più di uno, e cerco di spiegarteli. Basta ricorrere alla linea `#include`, come hai già fatto per i files `stdio.h` e `math.h`.

Primo modo. Supponiamo che io voglia far uso di due vettori globali di 100 elementi ciascuno. Mi basta creare un file, al quale potrò assegnare il nome `robamia.h` che contenga

```
#define NDATI 100
double Vettore_1[NDATI],Vettore_2[NDATI];
```

Fatto questo, all'inizio di ciascun file che fa uso di questi vettori inserisco la linea

```
#include "robamia.h" .
```

Inoltre, farò ben attenzione a scrivere `NDATI` anziché `100` ogni volta che nel sorgente vorrò far riferimento alla dimensione dei vettori.
Se per qualche motivo dovrò modificare la dimensione dei vettori mi basterà correggere il file `robamia.h` e ricompilare tutti gli altri files. Comodo, no? Ma forse ti serve qualche spiegazione.

- Ho assegnato al file il tipo `.h` perché questa è una convenzione del linguaggio C: i files destinati ad essere inclusi da altri, e non ad essere compilati indipendentemente, hanno tipo `.h`.
- La linea `#define NDATI 100` è un ordine al compilatore: ogni volta che trovi il nome `NDATI` devi prima di tutto sostituirlo con 100, poi sarai autorizzato a fare tutto il resto del tuo lavoro. Nota bene che non c'è il punto e virgola dopo `100`. Vedi qui sotto qualche spiegazione aggiuntiva.
- La dichiarazione
  ```
  double Vettore_1[NDATI],Vettore_2[NDATI];
  ```
  non ha nulla di speciale: il compilatore sostituisce `NDATI` con 100, e la linea diventa una normale richiesta di allocazione di memoria per i vettori.
- La linea
  ```
  #include "robamia.h"
  ```
  ha la sola stranezza di includere il nome del file tra virgolette, anziché tra `<...>`. Il motivo è questo: se il nome del file è racchiuso da `<...>` il compilatore cerca il file nella sua libreria di moduli da includere; se invece è chiuso tra virgolette lo cerca nel directory corrente. Ti basta mettere il file `robamia.h` nello stesso directory che contiene tutti i tuoi files sorgente.

Due parole in più sulla linea `#define`. La sintassi generale è:

`#define` $\langle nome \rangle$ $\langle definizione \rangle$

Il significato è: nel resto di questo file, ovunque trovi $\langle nome \rangle$ sostituisci il resto della riga – che qui ho indicato genericamente con $\langle definizione \rangle$, fino alla fine, senza preoccuparti di cosa ci trovi; poi sarai autorizzato a interpretare il risultato. Comodissimo, vero? Sí, ma attento al trabocchetto, che questa volta è più subdolo che mai. Guarda bene queste due righe di codice:

```
#define PARZIALE a+b+c
      ...
   z = z*PARZIALE;
```

Il tuo intento era probabilmente di spiegare al compilatore che deve eseguire la somma `a+b+c`, moltiplicarne il risultato per `z` e rimettere il tutto in `z`. Ma adesso prova a metterti nei panni del compilatore: non ragionare, e sostituisci il nome `PARZIALE` con la sua definizione, esattamente come ti ho detto poco fa. Troverai

```
z = z*a+b+c;
```

che non sembra corrispondere alle tue intenzioni. Soluzione: scriverai

```
#define PARZIALE (a+b+c)
```

Non ripetermi ancora una volta che questa macchina è proprio stupida: la risposta la conosci già. Piuttosto, prova a domandarti perché non ho messo il punto e virgola alla fine della linea `#define`.

Ti avevo promesso di spiegarti un secondo modo per semplificare la preparazione dei sorgenti. È un po' più complesso, ma ci sono dei buoni motivi. Nel file `robamia.h` scrivi

```
#define NDATI 100
extern double Vettore_1[],Vettore_2[];
```

L'istruzione

```
double Vettore_1[NDATI],Vettore_2[NDATI];
```

la inserisci solo nel file che contiene il `main`, sempre al di fuori dal corpo della funzione `main`. Se devi fare correzioni alle dimensioni dei vettori correggi `robamia.h` e ricompila tutto. Funziona? Sí, e ti spiego perché.

La dichiarazione

```
extern double Vettore_1[],Vettore_2[];
```

significa: "`Vettore_1` e `Vettore_2` sono nomi globali che corrispondono a vettori di tipo `double` — in altre parole, sono indirizzi di vettori. Dove siano sistemati e quanto siano lunghi i vettori non sono affari tuoi: sono vettori e basta". Il compilatore predispone la memoria nel modulo che contiene la dichiarazione

```
double Vettore_1[NDATI],Vettore_2[NDATI];
```

senza la parola chiave `extern`, e segnala al linker che quel nome globale è definito in quel modulo; per gli altri moduli prende un atteggiamento del tipo "non capisco ma mi adeguo": compila tutto facendo riferimento ad un indirizzo specificato da un nome globale, e passa la palla al linker. Il linker trova questo nome globale in tutti i moduli, e ne cerca uno in cui questo nome sia effettivamente definito: lo trova nel modulo che contiene il `main`, e tanto gli basta.

Detto così, mi farai notare che questo secondo modo è un po' come mangiare tenendo il piatto in equilibrio su un bastone appoggiato sulla punta del naso. E in effetti, non se ne vede ancora l'utilità. Ma forse ti ricrederai quando avremo parlato di inizializzazione di dati globali.

Un'ultima nota: avrai notato che ho usato solo caratteri maiuscoli per tutti i nomi definiti mediante una linea `#define`. Non è obbligatorio; come convenzione personale lo trovo comodo per evitare confusioni con le variabili, nel cui nome metto sempre caratteri minuscoli.

5.2.5 L'inizializzazione dei dati globali

A volte fa comodo avere un programma che parte con dei dati già definiti. È possibile? Risposta: sí, per i dati globali. Basta aggiungere i valori iniziali direttamente alla dichiarazione. Esempio:

```
int primi[10]={1,2,3,5,7,11,13,17,19,23};
```

inizializza una tabella di 10 numeri primi. Potresti ottenere un risultato analogo inserendo all'inizio del `main` le istruzioni

```
primi[0]=1; ...; primi[9]=23;
```

ma con l'effetto collaterale di aumentare il codice del programma senza alcun beneficio reale. L'inizializzazione come è fatta sopra viene gestita direttamente dal compilatore, ed il programma entra in memoria con l'area dati già definita. Uno strumento comodo, ma... poteva mancare il trabocchetto? No, naturalmente.

Supponi di aver preparato due files diversi. In uno hai inserito la linea

```
int primi[10]={1,2,3,5,7,11,13,17,19,23};
```

mentre nel secondo hai scritto

```
int primi[10]={23,19,17,13,11,7,5,3,2,1};
```

Domanda: nel momento in cui inizia l'esecuzione del programma che valore avrà `primi[0]` ?

Risposta: dipende da come si comportano il compilatore ed il linker. Il compilatore *Gnu–CC* marca come definiti i vettori globali in ambedue i moduli; il linker trova due nomi globali a cui corrispondono due definizioni diverse, e non sapendo cosa scegliere ti manda una segnalazione d'errore rifiutandosi di completare il suo lavoro. Gentile, tutto sommato, ma non contarci troppo: certi linker prendono il primo modulo che capita, ed ignorano il resto.

Come uscirne? Ebbene, il secondo modo di procedere che ti ho spiegato nel paragrafo 5.1.4 ti risolve il problema: l'inizializzazione la metti nel modulo in cui hai allocato la memoria, che deve essere uno solo. Il linker carica il vettore da quel modulo, con i valori che hai definito. Negli altri moduli trova solo l'indicazione che deve cercare l'indirizzo da qualche altra parte, e lo trova nel modulo che ha caricato.

5.3 Le funzioni di libreria

Il linguaggio C mette a disposizione una serie di funzioni prefabbricate, quelle che svolgono operazioni di uso comune. Ce ne sono tante, ed elencarle tutte servirebbe solo a compilare un elenco destinato a diventare obsoleto in tempi brevissimi. Mi limiterò qui alle funzioni matematiche — quelle più usate nell'ambiente scientifico. Aggiungerò le funzioni di allocazione di memoria, perché sono effettivamente utili.

Una sola nota, prima di cominciare: tutte le funzioni di libreria sono documentate, sia pure in computerese. Un linguaggio che col tempo, e con robuste dosi di pazienza ed autocontrollo, potrai anche dominare, ma che all'inizio non mancherà di mandarti su tutte le furie. Il consiglio è: prova a leggere la documentazione, e per prima cosa scrivi un programmino, il più breve possibile, che usi la funzione che ti interessa e ti dia modo di controllare che il risultato sia proprio quello che ti aspetti. Se non capisci, chiedi a chi ne sa più di te. Ma non andare a dire: io ho un programma che non funziona. Sforzati di formulare la domanda in modo preciso. È il solo modo di imparare. [9] Ah, dimenticavo, su Linux gran parte della documentazione si consulta col comando `man` ⟨*nome della funzione*⟩.

5.3.1 Funzioni matematiche

Eccoti un breve elenco:

- `double acos(double);`
  Arcocoseno di un valore reale, in radianti, tra 0 e π.
- `double asin(double);`
  Arcoseno di un valore reale, in radianti, tra $-\pi/2$ e $\pi/2$.
- `double atan(double);`
  Arcotangente di un valore reale, in radianti, tra $-\pi/2$ e $\pi/2$.
- `double atan2(double x, double y);`
  Arcotangente di `y/x`, in radianti, tra $-\pi$ e π. Il quadrante del risultato è calcolato in funzione dei segni di `x` e `y`.
- `double ceil(double);`
  Arrotondamento all'intero superiore.
- `double cos(double);`
  Coseno di un angolo; l'argomento è in radianti.
- `double cosh(double);`
  Coseno iperbolico di un numero reale.
- `double exp(double);`
  Esponenziale di un numero reale.
- `double fabs(double);`
  Valore assoluto di un numero reale.
- `double floor(double);`
  Arrotondamento all'intero inferiore.
- `double fmod(double, double);`
  Resto della divisione tra numeri reali.
- `double j0(double);`
  Funzione di Bessel del primo tipo di ordine `0`.
- `double j1(double);`
  Funzione di Bessel del primo tipo di ordine `1`.
- `double jn(int, double);`
  Funzione di Bessel del primo tipo di ordine `n`.
- `double ldexp(double, int);`
  Moltiplica un numero reale per una potenza intera di 2.

[9] Ricordi quando hai imparato a camminare? Nessuno ti ha insegnato la teoria: ci hai provato, hai sfruttato a fondo l'elasticità dei tuoi glutei e l'imbottitura del pannolone, ed alla fine hai imparato; ed ora hai persino dimenticato la fatica che hai fatto. Imparare il computerese non è poi tanto diverso.

- `double log(double);`
  Logaritmo naturale di un numero reale.
- `double log10(double);`
  Logaritmo volgare (o decimale, o di Briggs) di un numero reale.
- `double modf(double, double *);`
  Separa la parte intera e frazionaria di un numero reale.
- `double pow(double, double);`
  Calcola la potenza.
- `double sin(double);`
  Seno di un angolo; l'argomento è in radianti.
- `double sinh(double);`
  Seno iperbolico di un valore reale
- `double sqrt(double);`
  Radice quadrata di un numero reale.
- `double tan(double);`
  Tangente di un angolo; l'argomento è in radianti.
- `double tanh(double);`
  Tangente iperbolica di un numero reale.
- `double y0(double);`
  Funzione di Bessel del secondo tipo di ordine `0`.
- `double y1(double);`
  Funzione di Bessel del secondo tipo di ordine `1`.
- `double yn(int, double);`
  Funzione di Bessel del secondo tipo di ordine `n`.

Per usare tutte queste funzioni occorre inserire all'inizio del modulo l'istruzione

```
#include <math.h>
```

Inoltre occorre specificare nel comando di compilazione l'opzione `-lm` per ordinare al linker di inserire la libreria matematica (vedi il paragrafo 3.2.3).

5.3.2 Allocazione di memoria

È abbastanza comune il caso di programmi che necessitano di grandi quantità di memoria. [10] È altrettanto comune che il programmatore non possa conoscere in anticipo le effettive necessità del programma al momento dell'esecuzione. Ad esempio, un programma che debba elaborare dei dati statistici dovrà usare una quantità di memoria dettata dal numero dei dati da elaborare, e questo non può essere noto a priori.

---

[10] A dire il vero, la fame di memoria dei programmatori cresce almeno di pari passo con l'incremento della disponibilità garantito dallo sviluppo della tecnologia, che riduce sia le dimensioni che il costo dei componenti. Spesso si ha la sensazione che la disponibilità sempre maggiore induca i programmatori a curare sempre meno l'efficienza e l'ottimizzazione del codice: macchine sempre più potenti che non producono un effettivo incremento delle prestazioni.

Una soluzione possibile è: dimensionare tutte le aree di memoria al massimo compatibile con la macchina che si usa. Non particolarmente efficiente, ma pratico. Ma... che macchina si usa? Con quanta memoria disponibile? Probabilmente un metodo che consentisse di allocare memoria a programma avviato, magari facendo un ragionevole compromesso tra le necessità del programma e la disponibilità, sarebbe benvenuto. Ed infatti questo metodo esiste.

Un esempio completo forse non farebbe male. Tanto vale prendere un programma che abbiamo già scritto, e modificarlo in questo senso. Propongo di usare il programma che calcola il minimo ed il massimo tra gli elementi di un vettore reale, che abbiamo usato come esempio nel paragrafo 3.3.6. Eccoti il testo sorgente del `main`, modificato in modo da allocare durante l'esecuzione un vettore di dati di tipo `double`.

```
#include <stdio.h>
#include <stdlib.h>
#define NDATI 10000
/* Prototipo della funzione minmax */
void minmax(double vet[],int n,double *vmin,double *vmax);
/* Modulo main */
int main()
{
  double *vet;
  double x,xmin,xmax;
  int j;
                      /* Allocazione della memoria */
  vet=malloc(NDATI*sizeof(double));
  x=0.7615234;        /* Riempimento della tabella */
  for(j=0; j<NDATI; j++) {
    x = 4.*x*(1.-x);
    vet[j] = x;
  }
                      /* Calcolo di minimo e massimo */
  minmax(vet,NDATI,&xmin,&xmax);
  printf("Minimo: %e ;  Massimo: %e\n",xmin,xmax);
                      /* Libero la memoria allocata */
  free(vet);
  exit(0);
}
```

Se confronti questo testo con quello del paragrafo 3.3.6 noterai che le modifiche sono poche. Te le metto in evidenza.

- Ho usato un parametro `NDATI` invece che un numero per definire la dimensione del vettore. Nota anche che `NDATI` è usato ovunque io voglia indicare la lunghezza del vettore `vet`.

- La dichiarazione `double vet[1000]` è stata sostituita da `double *vet`. La differenza qui è sostanziale. Nel primo caso, non mi sono ancora stancato di scriverlo, io specifico che voglio riservare memoria sufficiente a contenere 1000 dati di tipo `double`, e che `vet` è l'indirizzo iniziale del vettore. Nel secondo caso la dichiarazione `double *vet` significa semplicemente: il dato `vet` è l'indirizzo di un dato `double`; riservami una cella per metterci questo indirizzo. Cosa ci mette il compilatore? Nulla: lascia quel che c'era. Il dato `vet`, ossia l'indirizzo contenuto nella cella `vet`, resta indefinito fin che io non ci metto qualcosa di sensato.
- L'istruzione
  ```
  vet=malloc(NDATI*sizeof(double));
  ```
  è quella che fa tutto: *al momento dell'esecuzione* la funzione `malloc` alloca la memoria e ne restituisce l'indirizzo come valore della funzione. [11] Insisto: tutto questo avviene non in compilazione, ma in esecuzione.
- Osserva che la chiamata alla funzione `minmax` non è cambiata: la funzione vuole l'indirizzo iniziale di un vettore, e io le passo l'indirizzo iniziale di un vettore.
- L'istruzione `free(vet)` libera la memoria allocata con `malloc`.

A questo punto ti potrebbe prendere una crisi di nervi. Immagino la domanda: ma non mi hai detto che se io scrivo `double vet[1000]` dentro il corpo del `main` la memoria viene allocata all'atto dell'esecuzione? E allora cosa è cambiato? Molto, e per due ottimi motivi.

Il primo motivo è che la memoria allocata da `malloc` non viene sottratta allo stack di programma; viene invece riservata allargando dinamicamente l'area dei dati statici. La conseguenza è notevole: si superano i limiti della dimensione dello stack. Il sistema concede memoria fin che ne ha.

Il secondo motivo è che la memoria, essendo allocata staticamente, è accessibile a tutte le funzioni che ne conoscano l'indirizzo, ed il suo contenuto è permanente. Nell'esempio qui sopra la funzione `minmax` può accedere al vettore `vet` perché l'indirizzo le viene passato come argomento; volendo, potrei usare il vettore per restituire dei dati al modulo chiamante, perchè l'area riservata al vettore non viene cancellata al momento del rientro dalla funzione.

Ti sembrano piccole differenze? Beh, allora vuol dire che hai poca esperienza di programmazione. Poco male: non ti mancheranno le occasioni per imparare, se ne vorrai approfittare.

---

[11] Ti ricordo che `sizeof(double)` restituisce la lunghezza in bytes del tipo `double`.