3

LO SVEZZAMENTO

In questo capitolo cominciamo ad occuparci delle strutture fondamentali del linguaggio, mettendo in evidenza le più comuni. Procederò per esempi che illustrino volta per volta un aspetto, senza entrare nei dettagli tecnici di una descrizione completa.

Mi concentrerò su quattro punti: l'uso dell'istruzione `if` e delle sue varianti `if ... else if ... else` per controllare il flusso delle istruzioni; l'istruzione `for` per la realizzazione di cicli; l'uso di tabelle (o arrays, o vettori); la scrittura di funzioni elementari.

3.1 La ripetizione di un ciclo di istruzioni

Iniziamo con un programma molto semplice: vogliamo calcolare la somma dei quadrati dei primi n numeri interi.

Se non hai mai provato a scrivere un programmino elementare posso ben immaginare la tua prima reazione: "se dovessi farlo a mano partirei da 1, poi sommerei 4, poi 9, e così via fino ad n. Ma come faccio a spiegarlo al computer?"

La seconda reazione, dopo averci pensato un momento potrebbe essere: "Vado a cercare una formula che mi dia immediatamente il risultato, risparmiandomi tutte le somme". Detto questo, forse ti armeresti della pazienza necessaria per andare a scartabellare su qualche libro di formule numeriche, e troveresti che $\sum_{j=1}^{n} j^2 = \frac{n(n+1)(2n+1)}{6}$. E concluderesti che in fondo ti basta modificare di poco il programma `somma.c` che hai scritto come secondo esercizio per arrivare rapidamente al risultato.

Ambedue le reazioni hanno del buono. La prima perché ti sei reso immediatamente conto che scrivere un programma per il computer significa sostanzialmente spiegargli come fare una cosa che se tu fossi abbastanza stupido e abbastanza veloce potresti anche fare a mano. La seconda perché hai capito che prima di usare il computer per svolgere un numero enorme di calcoli è meglio usare la testa per vedere se non si possa ottenere lo stesso risultato in un tempo molto più breve. È una buona regola generale: il mouse può sostituire tante cose, ma non il cervello.

Il guaio della seconda soluzione è che mi costringerebbe a cercare un altro esempio per spiegarti come si costruisce un ciclo iterativo, che è il mio vero scopo. Quindi perdona la mia pigrizia, dimentica per il momento di essere una persona intelligente, e buttati sulla prima soluzione.

3.1.1 Ci vuole un algoritmo

Ci vuole un che?... un logaritmo? No, un *algoritmo*, ossia un procedimento che con un numero finito di operazioni elementari conduca alla soluzione di un problema.

Se pensi di non aver mai visto un algoritmo in vita tua, prova a tornare con la memoria alle tue scuole elementari, quando la maestra ti insegnava pazientemente a fare le addizioni. Ricorda:

(i) scrivi i due numeri su due righe, uno sotto l'altro, con le cifre ben allineate verticalmente (aiutandoti con la quadrettatura del foglio);
(ii) concentra la tua attenzione sull'ultima colonna, e addiziona le due ultime cifre;
(iii) se il risultato è minore di 10 scrivilo sulla terza riga, ben in colonna sotto le due cifre che hai preso in considerazione; se invece è maggiore di 10 scrivi solo la prima cifra a destra, e tieni la seconda come riporto;
(iv) spostati a sinistra di una colonna, ed addiziona le due cifre; se hai un riporto, addiziona anche quello;
(v) riprendi dal passo (iii), spostandoti a sinistra di una colonna ad ogni passo, fin che hai esaurito le colonne;
(vi) se ti resta un ultimo riporto scrivilo sulla prima colonna a sinistra della terza riga;
(vii) il numero sulla terza riga è il risultato.

Probabilmente la maestra non ti ha spiegato la faccenda proprio in questi termini, ma questa è la sostanza. Ebbene, questo è un algoritmo, e la maestra ti ha programmato per svolgere le addizioni. Se il termine "programmare" ti sembra troppo simile a "insegnare a fare senza capire", prova a rispondere alla domanda seguente: mentre imparavi questo algoritmo avresti saputo spiegare perché funziona?

Il solo guaio dell'algoritmo che ti ho esposto è che richiede ancora un livello di intelligenza troppo alto per essere comprensibile ad un calcolatore. Come diavolo si fa a dirgli "quando hai finito le colonne" se non ha nemmeno la più pallida idea di cosa sia una colonna di numeri?

Naturalmente, se vogliamo costruire un algoritmo da tradurre in linguaggio C dobbiamo sapere quali sono le istruzioni che il linguaggio ci mette a disposizione. Ecco quelle che ci servono in questo momento:

- sommare o moltiplicare due numeri;
- confrontare due numeri e modificare di conseguenza l'ordine sequenziale delle operazioni.

Con queste sole operazioni siamo in grado di scrivere l'algoritmo seguente (intendendo che il valore di n sia fissato):

(i) associa una cella di memoria alla somma, e azzerala;
(ii) prendi una variabile intera j, e assegnale il valore 1;
(iii) calcola il quadrato di j e aggiungilo alla somma;
(iv) incrementa j di 1;
(v) se $j \leq n$ riprendi dal passo (iii); altrimenti
(vi) hai finito: nella cella di memoria associata alla somma c'è il risultato.

Prima di procedere rifletti, e convinciti che effettivamente questo schema di lavoro conduce al risultato. Quando ti sei convinto passa alla

3.1.2 Traduzione dell'algoritmo in un frammento di programma

Per codificare l'algoritmo servono delle istruzioni che non ti ho ancora spiegato. Lasciami procedere di nuovo con la scrittura del codice C prima di commentarlo. Ecco il frammento di sorgente (dove si suppone che la variabile `n` sia stata definita in qualche modo):

```
 int somma=0;                 /* (i) */
 int j;                       /* (ii) */
 j=1;
accumula:
 somma = somma + j*j;         /* (iii) */
 j = j+1;                     /* (iv) */
 if(j <= n) goto accumula;    /* (v) */
  ...                         /* (vi) risultato in somma */
```

Tutto quanto è compreso tra i delimitatori `/*...*/` viene ignorato dal compilatore: è un commento che serve a rendere più comprensibile il codice. Qui ho usato i commenti per mettere in evidenza la corrispondenza tra le istruzioni e i passi dell'algoritmo.

Le prime due righe non introducono molto di nuovo: la sola informazione è che la dichiarazione `int somma=0` chiede al compilatore non solo di associare una cella di memoria alla variabile `somma`, ma anche di azzerarla.[1]

L'istruzione `j=1;` è quasi auto–evidente: significa che voglio che nella cella che contiene `j` venga memorizzato il valore 1. Quelle che sembrano un po' stridenti sono le istruzioni alle righe contrassegnate con `(iii)` e `(iv)`: immagino che tu sia tentato di pensare che si tratti di un refuso, perché l'eguaglianza è palesemente assurda. Il punto è che nel linguaggio C, così come in molti altri linguaggi di programmazione, il carattere `=` non deve interpretarsi come un'eguaglianza (o un'equazione), bensì come un'*assegnazione*: prendi il valore che c'è a destra del segno `=` e mettilo nella

[1] Una nota importante: associare una cella di memoria ad una variabile non significa anche metterci un valore. Ad esempio, l'istruzione `int j;` della seconda riga associa una cella di memoria a `j`, ma il valore di `j` è del tutto casuale: di solito è quello che è rimasto dentro quella cella in seguito ad un utilizzo precedente.

cella di memoria associata alla variabile che c'è a sinistra. Va da sé che a destra del segno `=` ci può ben essere un'espressione aritmetica: si calcola l'espressione e se ne memorizza il risultato. A sinistra del segno `=` invece ci può essere solo il nome di una variabile, perché non è possibile associare una cella di memoria ad un'espressione aritmetica. Conclusione: l'istruzione `j=j+1` significa: prendi `j`, incrementalo di 1 e rimetti il risultato in `j`. Come dire: incrementa `j` di 1. Quanto all'istruzione `somma=somma+j*j` il significato è analogo; credo che sia appena il caso di dire che l'asterico indica il prodotto di due numeri. [2]

La riga `accumula:` definisce una *label*, o *etichetta*. In termini precisi: associa al nome `accumula` l'indirizzo di memoria dell'istruzione immediatamente successiva. Questa definizione viene usata in coppia con l'istruzione `goto accumula;` che si trova alla riga `(v)`: il significato è: "salta all'etichetta `accumula`", o, in termini più precisi: "metti nel registro IP l'indirizzo associato all'etichetta `accumula`". L'istruzione `goto` dunque interrompe il flusso sequenziale del programma. [3]

Resta da commentare l'istruzione `if(j <= n)`. La figura 3.1 te la illustra in modo schematico. [4] La parola `if` è una delle parole chiave del linguaggio (e come tale ne è proibito l'uso come nome di variabile o di qualunque altra cosa). Il contenuto della parentesi deve essere un'espressione logica, per intenderci, un'espressione il cui valore è vero oppure falso. L'istruzione immediatamente successiva alla frase `if(...)` viene eseguita se e solo se il risultato dell'espressione logica è vero; altrimenti viene semplicemente ignorata, e il programma prosegue. Nel nostro caso il test si può leggere così: "è vero che `j` è minore o uguale ad `n`? Se la risposta è sí, allora esegui

[2] Ebbene, ... sí: contrariamente ad altri linguaggi più orientati al calcolo scientifico il linguaggio C non ha un simbolo di operazione che indichi la potenza.

[3] Sono perfettamente cosciente del fatto che esiste un nutrito numero di programmatori – e soprattutto teorici della programmazione – che considerano abominevole l'uso dell'istruzione `goto`, perché renderebbe difficile la lettura dei programmi e sarebbe sovente pericolosa in quanto esporrebbe al rischio errori non facilmente identificabili. Qualche zuzzerellone ha pure definito la programmazione col `goto` "spaghetti program". Personalmente sono convinto che la questione sia puramente estetica e professionale: un programma pieno di `goto` che saltellano allegramente su e giù per il codice è certamente brutto, difficile da leggere ed esposto ad errori. Ma la colpa non è del `goto`: è dello sprovveduto che ha scritto il programma.

[4] La figura altro non è che un esempio elementare di ciò che agli albori della programmazione veniva chiamato *flow chart*, o diagramma di flusso. In pratica, la codifica di un programma veniva preceduta dalla preparazione di uno schema di questo tipo che descriveva tutti i singoli passi. Occorreva munirsi di matita, gomma e, soprattutto, fogli molto grandi! Oggi questo tipo di rappresentazione è caduto un po' in disuso, ma per illustrare situazioni semplici è pur sempre efficace.

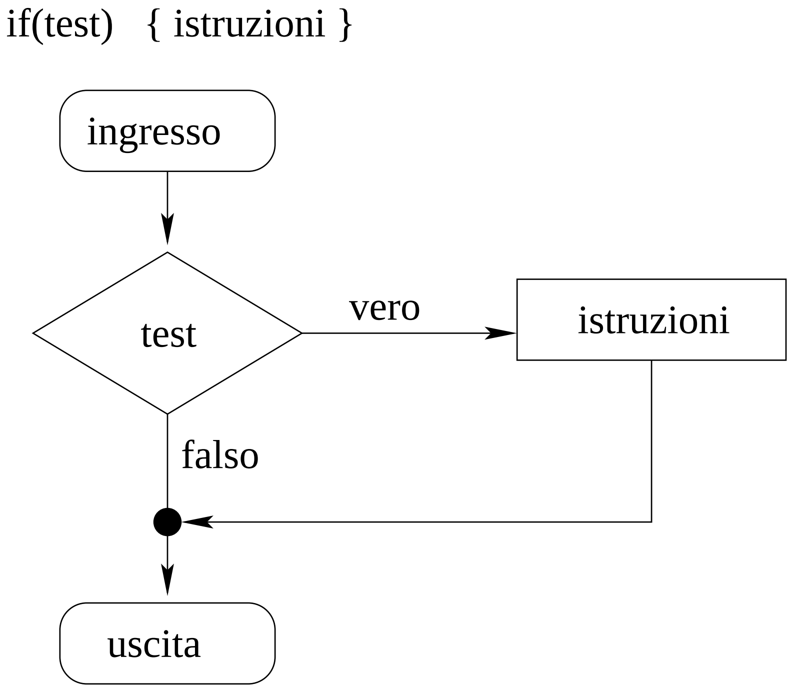

Figura 3.1. Il diagramma di flusso dell'istruzione `if`.

l'istruzione `goto accumula`; se la risposta è no interrompi il ciclo e prosegui con l'istruzione simboleggiata da tre puntini, alla riga `(vi)`".

Naturalmente, il frammento di codice che ho riportato sopra non può vivere da solo: bisogna perlomeno assegnare un valore alla variabile `n`. Se vuoi un esempio completo, eccoti un programma che:

- legge da terminale un numero n positivo;
- calcola la somma dei quadrati dei primi n numeri naturali;
- scrive il risultato sul terminale.

```
#include <stdio.h>
int main() {
  int n,j;
  int somma=0;
  printf("Dammi un numero positivo: ");
  scanf("%d",&n);
  j=1;
 accumula:
  somma = somma + j*j;
  j = j+1;
  if(j <= n) goto accumula;
  printf("La somma dei primi %d quadrati e' %d\n",n,somma);
  return(0);
}
```

Non credo che servano ulteriori commenti, tranne uno. Avrai notato che ho portato tutte le dichiarazioni `int...` all'inizio del modulo. Non è uno sfizio: le dichiarazioni che controllano l'allocazione di memoria non possono essere mescolate alle istruzioni eseguibili. [5]

La compilazione e l'esecuzione non richiedono nulla di nuovo rispetto a quanto hai fatto fin qui. Quindi a partire da questo momento ne riparlerò solo quando ci sarà qualcosa da aggiungere.

3.1.3 Una codifica diversa del ciclo

L'esempio che abbiamo discusso illustra come eseguire ripetutamente un ciclo di istruzioni. Di fatto abbiamo usato come ingredienti un contatore, un numero totale di iterazioni ed un gruppo di istruzioni da eseguire ad ogni ciclo. Questa situazione, come puoi ben immaginare, è molto frequente, al punto che è stato introdotto un modo diverso e più semplice per ottenere lo stesso risultato (evitando in particolare l'uso esplicito dell'etichetta e dell'istruzione `goto`). Si tratta dell'istruzione `for`. La figura 3.2 ti illustra il diagramma di flusso.

Ecco come si può scrivere il frammento di programma che calcola la somma dei quadrati

```
int j,somma;
for(somma=0, j=1; j<=n; j++) somma = somma + j*j;
```

Non credo di dover sottolineare la maggiore compattezza del codice sorgente. In compenso credo sia indispensabile spiegarti come funziona. Il fatto è che questa scrittura non è altro che una maniera compatta per scrivere lo stesso algoritmo che abbiamo usato fin qui, con una piccola differenza. Cerco di spiegarmi.

All'interno della coppia di parentesi che seguono la parola chiave `for` ci sono tre campi distinti, separati dal punto e virgola:

- le istruzioni di *inizializzazione* del ciclo. [6] Vengono eseguite per prime, una sola volta. Nel nostro esempio sono le istruzioni `somma = 0, j=1` Nota bene che le due assegnazioni sono separate da una virgola, non da un punto e virgola: il punto e virgola separa i campi dell'istruzione `for`.
- il *test*, e precisamente un'espressione logica che viene calcolata all'inizio di ogni ciclo, compreso il primo; se l'espressione dà come risultato vero il ciclo continua; se dà falso il ciclo di iterazioni termina. Nel nostro caso è l'espressione `j<=n`.

---

[5] Vengono dette *istruzioni eseguibili* quelle che provocano un'azione durante l'esecuzione del programma (calcolo di un'espressione aritmetica o logica, chiamata di funzione, &c), e non la semplice allocazione di memoria.

[6] Spero che mi venga perdonato l'uso dell'orribile parola *inizializzazione*, inventata da qualcuno che ha cercato di scimmiottare l'inglese *initialization*, e ormai entrata nell'uso, purtroppo, soprattutto in ambiente informatico.

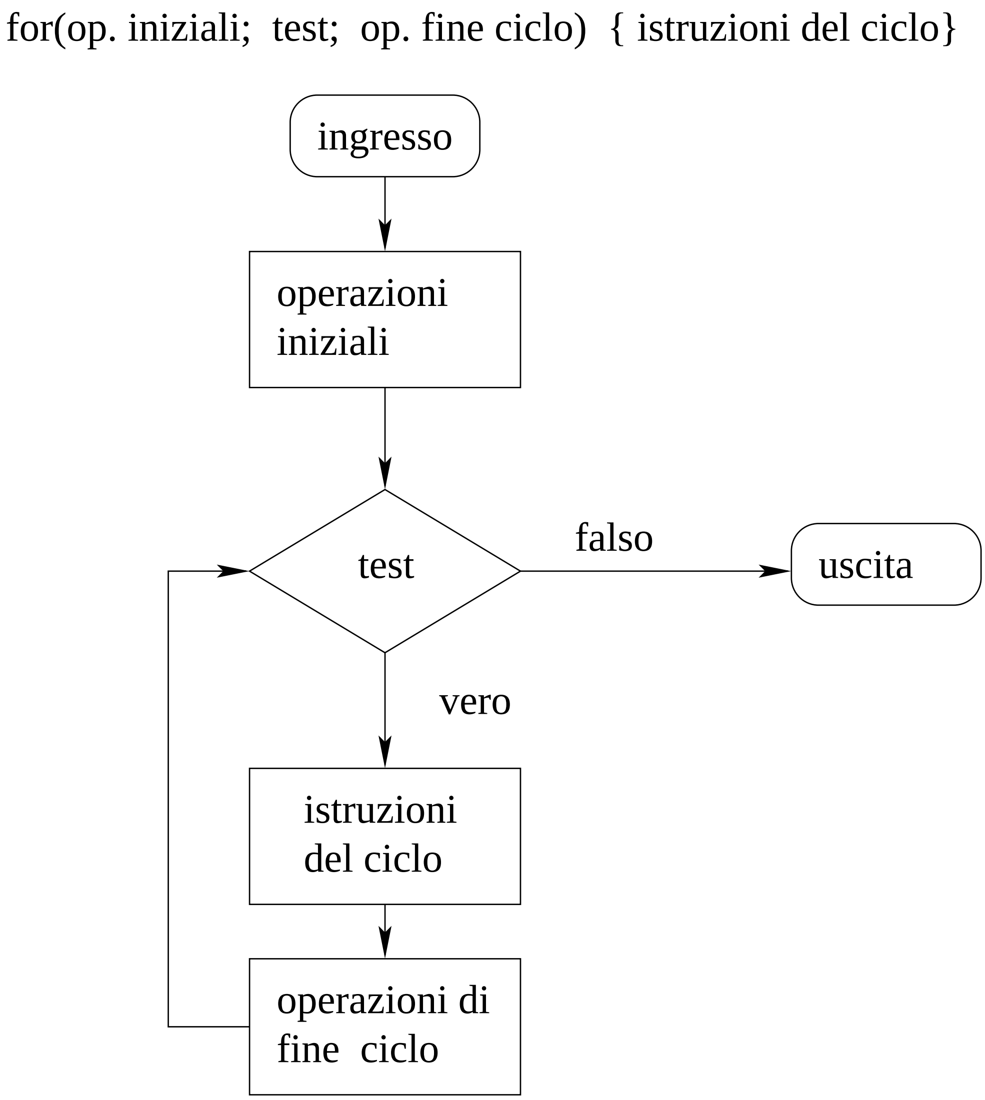

Figura 3.2. Il diagramma di flusso dell'istruzione `for`.

- un'istruzione che viene eseguita alla fine di ciascun ciclo, prima di iniziare il successivo. Nel nostro caso è l'istruzione `j++`, che significa: incrementa `j` di 1. Si tratta di una scrittura sintetica perfettamente equivalente a `j=j+1`.

Dopo la frase `for(...)` ci può essere una singola istruzione terminata dal punto e virgola, oppure un gruppo di istruzioni racchiuse tra parentesi graffe. Queste costituiscono il cosiddetto *corpo* del ciclo, ossia le istruzioni che vengono eseguite ad ogni iterazione.

Permettimi di essere un po' pedante. L'esecuzione del ciclo del nostro esempio avviene così:

(i) vengono eseguite le assegnazioni `somma=0` e `j=1`;
(ii) viene eseguito il test per verificare se `j<=n`;
(iii) se il test dà come risultato falso (ossia `j` ha superato `n`) il ciclo viene

interrotto; altrimenti

(iv) viene eseguita l'istruzione `somma = somma+j*j;`

(v) viene incrementato `j`, e si torna al passo (ii) per l'esecuzione del test.

Se confronti questo algoritmo con quello del paragrafo precedente osserverai che la piccola differenza esiste: qui il test viene eseguito all'inizio del ciclo, e non alla fine. Di conseguenza, nella codifica del paragrafo precedente il ciclo viene eseguito almeno una volta, qualunque sia il valore di `n`. Nella codifica con `for` invece il ciclo viene eseguito solo se `n` è positivo.

Esercizio 3.1: Completa il programma come nel paragrafo precedente, aggiungendo la lettura del valore di `n` e la stampa del risultato. Poi:

- esegui ambedue i programmi col valore `n=0`: avrai modo di osservare l'effetto della differenza nel controllo del ciclo.
- prova ad eseguire uno dei due programmi con `n=10000`. Che succede?

3.2 Un problema da liceo

Come secondo esercizio ti propongo un problema che conosci benissimo dal liceo: scrivere un programma che calcoli la soluzione di un'equazione di secondo grado. Naturalmente dovrai saper distinguere e trattare correttamente i tre casi possibili: soluzioni reali e distinte, reali coincidenti o complesse.

3.2.1 Pensiamo all'algoritmo

Concentriamoci sul frammento di programma che calcola la soluzione dell'equazione, supponendo che i coefficienti `a`, `b` e `c` siano stati determinati in qualche modo. Ecco l'algoritmo, dettagliato quanto basta:

(i) calcola il discriminante $\Delta = b^2 - 4ac$;

(ii) se $\Delta > 0$:

  (ii.a) calcola le due soluzioni reali $x_1 = \frac{-b+\sqrt{\Delta}}{2a}$, $x_2 = \frac{-b-\sqrt{\Delta}}{2a}$;

(iii) altrimenti, se $\Delta = 0$:

  (iii.a) calcola l'unica soluzione reale $x = -\frac{b}{2a}$;

(iv) altrimenti (non serve controllare ancora il segno di Δ):

  (iv.a) calcola la parte reale delle soluzioni, $\mathrm{Re}\, x = -\frac{b}{2a}$;

  (iv.b) calcola la parte immaginaria $\mathrm{Im}\, x = \sqrt{-\Delta}$.

3.2.2 Scriviamo il programma

Per scrivere l'intero programma ti servono diverse informazioni che non ti ho ancora dato. Quindi anche questa volta te lo propongo per intero, rimandando i commenti a più tardi.

```
#include <stdio.h>
#include <math.h>
int main() {
  double a,b,c,delta,x1,x2,x,rex,imx;
  printf("Dammi i coefficienti a,b,c: ");
```

```
    scanf("%lf %lf %lf",&a,&b,&c);
    delta = b*b - 4.*a*c;
    if(delta > 0.) {
      x1 = 0.5*(-b + sqrt(delta))/a;
      x2 = 0.5*(-b - sqrt(delta))/a;
      printf("Le radici sono reali e distinte:\n");
      printf("x_1 = %e\n",x1);
      printf("x_2 = %e\n",x2);
    }
    else if(delta == 0.) {
      x = -0.5*b/a;
      printf("Le radici sono reali e coincidenti:\n");
      printf("x = %e\n",x);
    }
    else {
      rex = -0.5*b/a;
      imx = 0.5*sqrt(-delta)/a;
      printf("Le radici sono complesse coniugate:\n");
      printf("Re x = %e\n",rex);
      printf("Im x = %e\n",imx);
    }
    exit(0);
  }
```

E veniamo ai commenti, cominciando con le faccende semplici, che si risolvono in poche righe.

La seconda riga contiene una mezza novità: oltre al file `stdio.h` viene incluso anche uno strano `math.h`, che ricorda la matematica. La ragione è semplice: dobbiamo calcolare una radice quadrata, che si calcola mediante la funzione `sqrt()`. Il file `math.h` serve per garantire un uso corretto di questa funzione. Se ti stai chiedendo che succederebbe se lo dimenticassi, la risposta è molto semplice: prova!

La quarta riga contiene una dichiarazione `double a,b,...` Significa che i nomi `a,b,...` sono associati a variabili che sono numeri reali detti *in doppia precisione*. Per il momento ti basti sapere che i numeri reali vengono rappresentati con una precisione di circa 15 cifre. Ne saprai di più quando avremo parlato della rappresentazione dei dati.

La linea `scanf(...)` contiene una piccola novità: nella stringa del formato di conversione compare un `%lf`. Significa che il dato in lettura è un numero in doppia precisione, quindi è ammesso l'uso del punto decimale.

Analogamente, in alcune delle chiamate alla funzione `printf(...)` compare un `%e` nel formato di conversione. Significa che vuoi la conversione di un numero reale, scritta, si dice, in formato `e`. Non spaventarti: è solo la scrittura in notazione scientifica del tipo 1.2345×10^6. Dato che il terminale

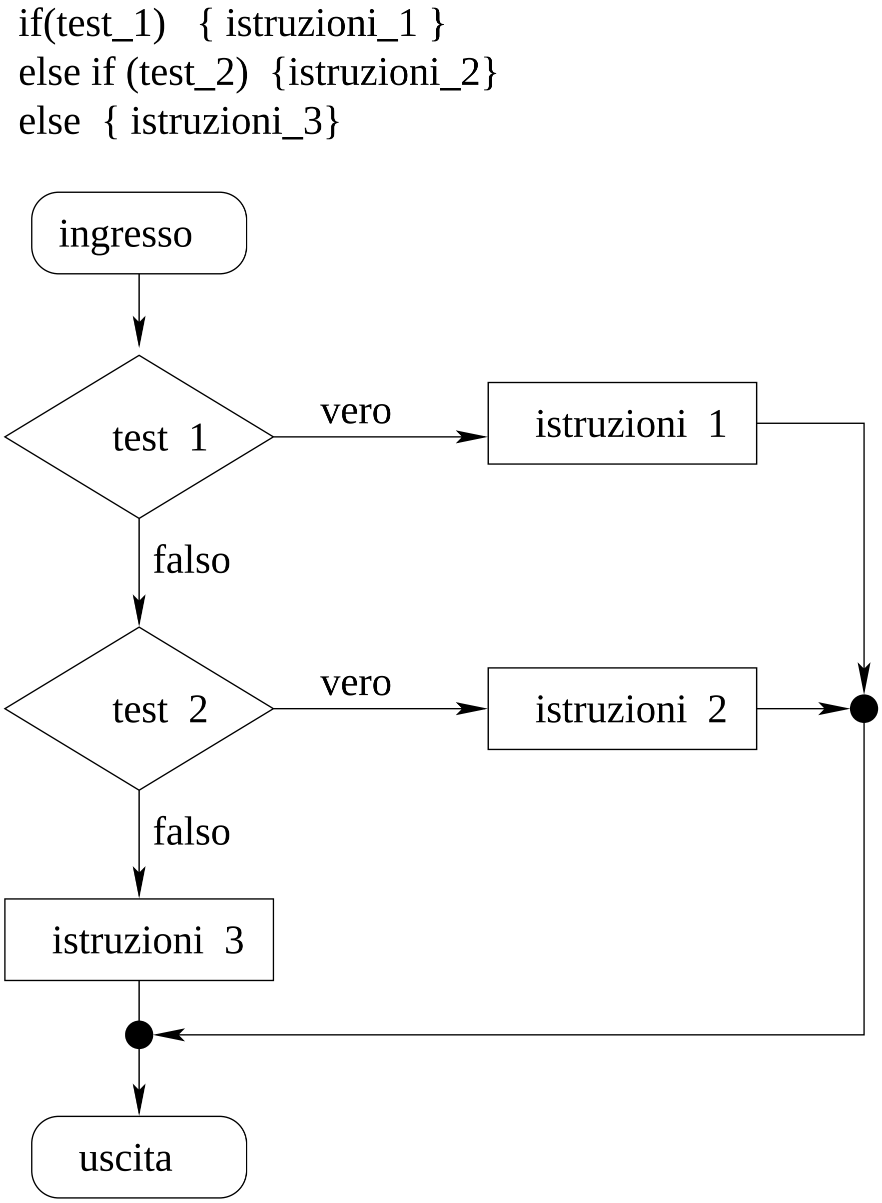

Figura 3.3. Il diagramma di flusso dell'istruzione `if ... else`.

tipicamente non è in grado di stampare gli esponenti il numero viene scritto come `1.2345e+06`. Tutto quello che segue la lettera `e` è esponente di 10.

Infine, avrai probabilmente notato che la divisione si denota con `/`. Su questo lasciami aggiungere una piccola nota. Se devi calcolare, ad esempio, $\frac{b}{2a}$ potresti essere tentato di scrivere `b/2.*a`. E sarebbe un grosso errore! Il compilatore esegue le moltiplicazioni e le divisioni nell'ordine in cui le trova, e quindi interpreterebbe la tua espressione come: prendi `b`, dividilo per 2 e poi moltiplica il risultato per a. Non è quello che intendevi! Scriverai invece `b/(2.*a)` oppure `b/2./a`.

E veniamo alla parte un po' più complicata: l'uso dell'istruzione `if`, che presenta qualche novità rispetto al paragrafo 3.1.2. Il diagramma di flusso lo trovi nella figura 3.3. Lo schema generale del blocco di istruzioni è:

```
if (test 1) {istruzioni 1}
else if (test 2) {istruzioni 2}
else if (test 3) {istruzioni 3}
...
else if (test n-1) {istruzioni n-1}
else {istruzioni n}
... (continua il programma)
```

Il significato è il seguente:

- se la prima condizione test 1 è vera, esegui il blocco di istruzioni `{`*istruzioni 1*`}` e poi salta tutto il resto del blocco `if`, ossia salta alla riga *continua il programma*;
- se la prima condizione è falsa controlla la seconda; se è vera esegui il blocco di istruzioni `{`*istruzioni 2*`}` e poi salta tutto il resto del blocco `if`;
- continua così fino al test *n-1*. Se nessuna delle condizioni precedenti è risultata vera esegui il blocco di istruzioni `{`*istruzioni n*`}`.

Puoi mettere tutte le linee `else if(...)` che vuoi. Puoi anche omettere la linea `else` finale: significa che se non è verificata nessuna delle condizioni precedenti non se ne fa nulla. Lo schema `if` che abbiamo usato nel paragrafo 3.1.2 è un caso particolare di questo schema generale: semplicemente in quel caso non c'è nessuna linea `else`.

Permettimi solo di richiamare la tua attenzione su un tranello, che ti illustro con un esempio:

```
if(j > 2) {ba...ba...ba}
else if (j>=0) {bo...bo...bo}
else {bu...bu...bu}
```

Interpretazione ingenua: supponiamo che `(j>2)` sia vero; a maggior ragione è vero `(j>=0)`. Quindi verranno eseguite sia le istruzioni *ba...ba...ba* che le istruzioni *bo...bo...bo*, mentre verranno saltate le istruzioni *bu...bu...bu*. Nulla di più sbagliato: essendo risultata vera la prima condizione, la seconda non viene neppure presa in considerazione; vengono eseguite solo le istruzioni *ba...ba...ba*. Rileggi la spiegazione, e, se ancora non ne fossi convinto, prova!

Con queste indicazioni non ti dovrebbe essere difficile capire come funziona il programma. Non ti resta che inserirlo in macchina col text editor, assegnandogli un nome. Che ne dici di `liceo.c`?

3.2.3 Compilazione e link

Dovresti ormai sapere cosa fare. Con grande sicurezza batti il comando

```
gcc -Wall -o liceo liceo.c
```

e... con tua grande sorpresa trovi che la perfida macchina ha qualcosa da ridire, perché protesta energicamente lanciandoti insulti del tipo

```
/tmp/ccdWjLfq.o: In function ‘main’:
/tmp/ccdWjLfq.o(.text+0x72): undefined reference to ‘sqrt’
/tmp/ccdWjLfq.o(.text+0xa1): undefined reference to ‘sqrt’
/tmp/ccdWjLfq.o(.text+0x16d): undefined reference to ‘sqrt’
collect2: ld returned 1 exit status
```

Che diavolo vuole ancora? Semplice: tu hai richiamato una funzione, per la precisione `sqrt`, che lui non è riuscito a trovare da nessuna parte; quindi se ne lava le mani dicendoti che è rimasto un riferimento esterno indefinito. Devi solo spiegargli dove cercare il pezzo mancante. Per questo ti basta modificare di poco il comando di compilazione:

```
gcc -Wall -o liceo liceo.c -lm
```

Quel `-lm` che aggiungi alla fine mette a posto tutto. Preferisco non riferire i tuoi commenti, e passo alla spiegazione.

La funzione `sqrt` fa parte della libreria di funzioni matematiche che il linguaggio C mette a disposizione come standard, ma non viene inclusa automaticamente dal linker nelle sue ricerche dei moduli mancanti. La regola generale è la seguente: aggiungendo al comando di compilazione

`-l`$\langle xxx\rangle$

il linker capisce: “tra le librerie standard ce n'è una che si chiama `lib`$\langle xxx\rangle$`.a`. Cerca anche lí dentro i moduli che ti mancano per completare il lavoro”. Nel nostro caso il linker va a cercare la libreria `libm.a`. Se sei proprio curioso mi chiederai anche: ma dove trova questo file? Risposta: dipende dall'installazione. Sui sistemi Linux di solito lo trovi nel directory `/usr/lib/` .

3.2.4 *Una tabella di numeri*

Ancora un problema elementare: devo raccogliere tanti numeri in una tabella, e poi devo eseguire qualche operazione su questi numeri. Trasformiamolo in un esempio.

Proponiamoci di:

(a) riservare lo spazio per una tabella di 1000 numeri reali;
(b) riempire la tabella con dei numeri più o meno a caso;
(c) calcolare la media di questi numeri.

Ancora una volta, se dovessi farlo con carta e matita il problema sarebbe semplice: prendi la riga, e disegni una tabella con 1000 caselle; poi ci metti dei numeri tirando i dadi; poi sommi tutti i numeri e dividi per 1000. Mi dirai che si tratta di un lavoro lungo e noioso, ma, secondo te, cosa facevano i ragionieri prima di essere soppiantati dagli informatici? I dadi probabilmente non li usavano, ma le colonne di numeri sí.

3.2.5 *La codifica in linguaggio C*

Separiamo il problema nelle sue parti elementari, che del resto ho già isolato, grosso modo, nei punti (a), (b) e (c) qui sopra.

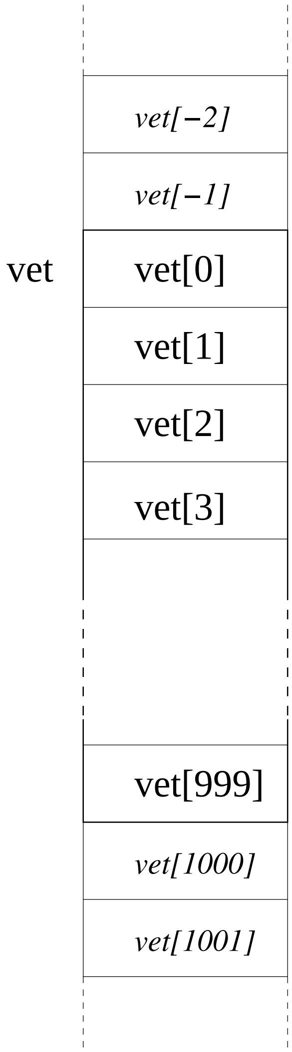

Figura 3.4. L'allocazione dello spazio di memoria per una tabella o *array*, o vettore.

Il punto (a) richiede un'informazione nuova. Eccoti l'istruzione:

```
double vet[1000];
```

Sto ordinando al compilatore: voglio che tu mi riservi uno spazio contiguo di 1000 celle di memoria, ciascuna delle quali deve essere sufficiente a contenere un dato di tipo `double`; a questa tabella devi associare il nome `vet`. Il compilatore procede secondo lo schema illustrato in figura 3.4: associa al nome `vet` un indirizzo di memoria, e a partire da quell'indirizzo riserva 1000 celle ciascuna delle quali può contenere un dato `double`. Se vorrò far riferimento all'intera tabella userò il nome `vet`; se vorrò far riferimento ad un singolo elemento della tabella scriverò `vet[0]`, per il primo elemento, `vet[1]` per il secondo, e così via, fino a `vet[999]` per l'ultimo elemento. Non hai le traveggole, e non ho fatto confusione; è proprio così: gli elementi di tabella in C si numerano a partire da 0, e dunque una tabella di n elementi ha indici che vanno da 0 a $n-1$.

Guardiamo anche dietro le quinte. La scrittura `vet[4]` significa: prendi l'indirizzo `vet`, sommagli 4 volte lo spazio occupato da un singolo dato di tipo `double`, e prendi il dato di tipo `double` che si trova a quell'indirizzo. [7] Posso anche scrivere `vet[j]` per indicare il `j`–esimo elemento della tabella, ma è compito mio assicurare che `j` sia compreso tra 0 e 999. Che succederebbe se `j` valesse, ad esempio, -2 oppure 1001? La figura 3.4 te lo spiega: il calcolatore applicherebbe beluinamente l'algoritmo di calcolo dell'indirizzo che ti ho spiegato, senza accorgersi che l'indice è fuori dai limiti, e andrebbe a leggere o scrivere in una zona di memoria che non ha nulla a che fare con la tabella. Si dice di solito: è un'*invasione di memoria*. Gli effetti possono essere imprevedibili, perchè si utilizzano o addirittura si modificano dati che non fanno parte della tabella.

Permettimi di insistere. La dichiarazione

```
double x
```

significa: `x` è un dato di tipo `double`, e `&x` è l'indirizzo a cui si trova il dato `x`. La dichiarazione

```
double vet[1000]:
```

significa: `vet` è l'indirizzo di una tabella di 1000 dati di tipo `double`, e `vet[j]` è il dato contenuto nel `j`–esimo elemento di questa tabella. Come vedi, l'informazione associata direttamente al nome è ben diversa.

Veniamo al punto (b). Se tieni conto di quanto ti ho già detto, questo non dovrebbe crearti difficoltà. Semmai ti potresti chiedere come inventarti dei numeri più o meno a caso senza prenderti la briga di metterti a lanciare dadi o ad aprire l'elenco telefonico ad una pagina qualunque. Per il momento ti può bastare un metodo di questo tipo: assegna ad x un valore qualsiasi tra 0 e 1 (estremi esclusi); poi genera una successione di numeri, tutti tra 0 e 1, cambiando x in $4x(1-x)$. Puoi memorizzare nella tabella i primi 1000 numeri di questa successione. Ecco il frammento di programma corrispondente:

```
double x;
int j;
x=0.7615234;
for(j=0; j<1000; j++) {
  x = 4.*x*(1.-x);
  vet[j] = x;
}
```

Il punto (c) è ancora più semplice: una piccola variazione sull'esercizio del paragrafo 3.1. Passo direttamente al frammento di programma:

---

[7] Pensa alla memoria–cassettiera: un dato `double` occupa 8 cassetti (fidati: te lo dico io). Supponi che `vet` sia 48: significa che la tabella `vet` occupa spazio a partire dal cassetto 48. Il primo elemento delle tabella, `vet[0]`, occupa i cassetti da 48 a 55; `vet[1]` occupa i cassetti da 56 a 63, &c.

```
double somma,media;
for(somma=0, j=0; j<1000; j++) somma = somma+vet[j];
media = somma/1000.;
```

Tutto qui? Sí, tutto qui.

Per completare il lavoro non ti resta che aggiungere il contorno, compilare ed eseguire. A te il compito, questa volta.

3.3 Divide et impera

Per eseguire un lavoro complesso non c'è spesso metodo migliore del dividerlo in parti più piccole, ed eventualmente suddividere ciascuna parte in altre parti, &c, fin che si è arrivati a ridurre il tutto ad una serie di compiti affrontabili singolarmente. Inoltre, è frequente, nella soluzione di un problema, scoprire che ci sono operazioni che vengono ripetute più volte, in contesti diversi: sarebbe utile disporre di un personaggio che le esegue.

Abbiamo già incontrato alcuni esempi, fin qui: la stampa di messaggi o risultati, la lettura di dati, il calcolo della radice quadrata. E in questi casi abbiamo trovato il personaggio disposto ad eseguire queste operazioni: una funzione di libreria. Abbiamo anche visto che per chiamare una funzione si usa una frase del tipo

`ciccio(`*argomenti*`);`

qui, `ciccio` è il nome della funzione, e tutto quel che si trova tra parentesi è l'informazione che noi passiamo alla funzione. Nel caso della radice quadrata poi la funzione ci ha restituito un valore. Per questo abbiamo usato un'istruzione di assegnazione

```
y = sqrt(x);
```

il valore calcolato dalla funzione viene assegnato ad `y`.

La domanda è spontanea: posso costruirmi io stesso una funzione della mia libreria personale? Sí, ovviamente. Basta sapere come fare. È quello che intendo spiegarti ora, senza esagerare.

3.3.1 A domanda risponde

Cominciamo, al solito, con un esercizio elementare. Scriviamo una funzione che converta una lunghezza da pollici (o inches, l'unità di misura anglosassone) a centimetri. La domanda è precisa per noi, ma non per il calcolatore. Aggiungiamo un po' di dettagli. Dobbiamo decidere:

- che nome dare alla funzione;
- che dati le dobbiamo fornire, e come;
- che operazioni deve eseguire al suo interno;
- cosa ci deve restituire come risultato.

Cominciamo. Per il nome: che ne dici di `cm`? Sorvoliamo sulla risposta, ed usiamolo.

I dati: dobbiamo passare alla funzione una lunghezza in pollici; quindi ci serve un solo argomento. Non basta: dobbiamo anche precisare che si tratta di un argomento di tipo `double` — per noi è ovvio, ma per il calcolatore no.

Operazioni da eseguire: questa è la parte che personalizza, per così dire, la funzione. Di solito richiede un po' di lavoro: ricondurre la soluzione ad una serie di operazioni elementari, formulare un algoritmo e tradurlo in codice C. Ma qui è facile: basta sapere quanti centimetri ci stanno in un pollice. Un vecchio libro di fisica, un'enciclopedia, o una telefonata ad un ministro che ti consiglia di cercare su internet, e trovi la risposta: 1 in = 2.54 cm.

Risultato da restituire: una lunghezza in centimetri, ossia un dato di tipo `double`.

Stabilito questo, si tratta di scrivere il tutto in C. Al solito, prima la risposte e poi i commenti. Ecco il codice:

```
#include <stdio.h>
double cm(x)
    double x;
{
  double risposta;
  risposta = 2.54*x;
  return(risposta);
}
```

Come vedi, non è poi tanto diverso da quello che hai scritto fin qui.

La seconda riga e la terza riga significano: qui inizia un modulo – una funzione – il cui nome è `cm`. Questa funzione si aspetta di ricevere come argomento un dato `x`, che deve essere di tipo `double`; inoltre la funzione restituisce come risultato un dato di tipo `double`. La coppia di parentesi graffe alla quarta ed all'ultima riga delimita il corpo della funzione: tutto succede lí dentro.

L'istruzione `return(risposta)` è quella che restituisce il dato calcolato. Naturalmente, il dato è dichiarato `double`, consistentemente col tipo di dato che deve essere restituito.

Forse ti stai domandando (o faresti bene a farlo): perché la dichiarazione `double x` sta fuori dalle graffe, mentre `double risposta` sta dentro? Perché `x` è un argomento della funzione, non ha nulla a che vedere col corpo della funzione stessa. Inoltre, il dato `x` sta già da qualche parte (in qualche cassetto della memoria); il compilatore deve solo sapere di che tipo di dato si tratta. Invece `risposta` è un dato che riguarda il corpo della funzione, e non ha nulla a che vedere con gli argomenti: io chiedo al compilatore di riservarmi una cella di memoria per metterci questo dato.

Come vedi, scrivere una funzione non è poi tanto diverso dallo scrivere il modulo `main`. Anzi, vien quasi il dubbio che il main per il compilatore sia in pratica una funzione come tutte le altre. Così è, infatti. Di più: non sei neppure obbligato a chiudere il `main` con un `exit(0)`. Se ci mettessi un `return(0)` funzionerebbe lo stesso: alla chiamata ai moduli che chiudono il programma (e che fin qui ho indicato genericamente come modulo `exit`) ci penserebbe comunque il sistema al momento del rientro dal `main`.

3.3.2 Ed ora, chiamiamo la funzione

Adesso dobbiamo scrivere il programma che chiama la nostra funzione. Questo è proprio facile: leggo la lunghezza in pollici, chiamo la funzione che la converte in centimetri e scrivo la risposta. Ecco il codice C:

```
int main()
{
  double a,c;
  printf("Dammi una lunghezza in pollici: ");
  scanf("%le",&a);
  c = cm(a);
  printf("La lunghezza in centimetri e' %e\n",c);
  exit(0);
}
```

Come vedi, se `cm` fosse una funzione standard di libreria scriveresti esattamente la stessa cosa. Unica osservazione: forse hai notato che l'argomento della funzione qui si chiama `a`, mentre nel sorgente della funzione si chiamava `x`. Nessun problema: quello che viene passato è il dato, non il nome.

Ora viene il bello: e come dico al linker che oltre al `main` c'è anche la funzione `cm`? Semplice: prendi il file che contiene il sorgente della funzione, ci aggiungi in coda il `main`, ed esegui compilazione e link esattamente come hai fatto fin qui. Proviamoci.

Creo un file che contiene:

```
#include <stdio.h>
/* cm: converte pollici in centimetri */
double cm(x)
    double x;
{
        vedi le istruzioni nel paragrafo 3.3.1
}
/* main */
int main()
{
        vedi le istruzioni qui sopra
}
```

Ho solo aggiunto un paio di linee di commento: se fra sei mesi mi capiterà di rileggere quel file mi serviranno. Devo anche dare un nome al file. Non dire nulla: lo chiamo `cm.c`. Passo a compilare il file col comando

```
gcc -Wall -o cm cm.c
```

e lo eseguo:

```
[...]$ ./cm
Dammi una lunghezza in pollici: 1
la lunghezza in centimetri e' 2.540000e+00
```

Direi che funziona.

Ma non posso escludere che tu abbia pensato di esercitare la tua libertà decidendo di scambiare il posto del `main` con quello della funzione `cm`. In altre parole, il tuo file potrebbe contenere:

```
#include <stdio.h>
/* main */
int main()
{
        vedi le istruzioni qui sopra
}
/* cm: converte pollici in centimetri */
double cm(x)
    double x;
{
        vedi le istruzioni nel paragrafo 3.3.1
}
```

Se non l'avessi fatto (bravo, significa che sei uno studente diligente!) provaci. Tanto, che differenza fa? Baldanzoso, esegui il comando di compilazione, proprio quello che c'è poche righe sopra. E arriva la sorpresa: [8]

```
cm.c: In function 'main':
cm.c:8: warning: implicit declaration of function 'cm'
cm.c: At top level:
cm.c:14: warning: type mismatch with previous implicit
    declaration
cm.c:8: warning: previous implicit declaration of 'cm'
cm.c:14: warning: 'cm' was previously implicitly declared
    to return 'int'
```

Resti un po' perplesso, e poi pensi: "bah, in fondo sono solo dei warnings. L'eseguibile l'ha fatto lo stesso". Ed esegui. Ecco cosa ti potrebbe accadere:

```
[...]$ ./cm
Dammi una lunghezza in pollici: 1
La lunghezza in centimetri e' 1.000000e+00
```

Ma allora è proprio stupido! Non esattamente: è un calcolatore, e quindi ha un modo tutto particolare di essere stupido. Forse è bene che ti dica qualcosa in più sul meccanismo di chicmata delle funzioni.

[8] Una precisazione d'obbligo: i messaggi qui sotto dipendono dal sistema e dal compilatore, quindi non è detto che siano identici a quelli che ti dà un sistema diverso, o anche una versione diversa dello stesso compilatore.

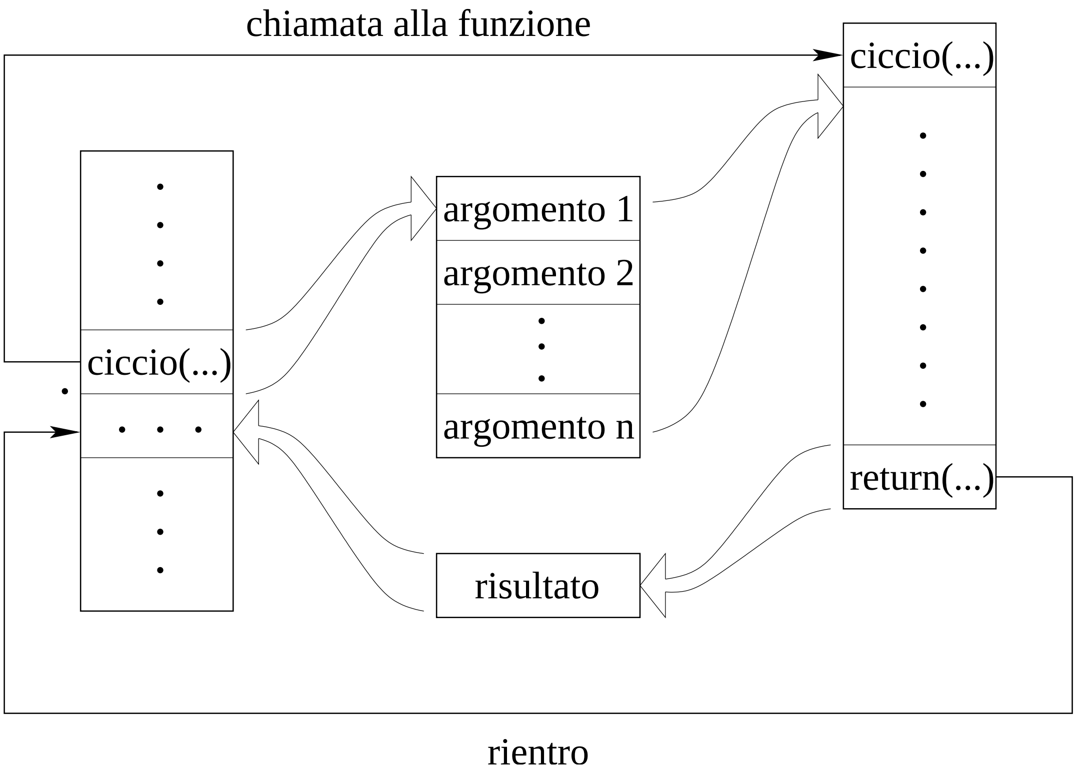

Figura 3.5. Il meccanismo di chiamata ad una funzione, col passaggio di argomenti e la restituzione del risultato.

3.3.3 Apriamo il giocattolo

Quello che accade quando si chiama una funzione è illustrato nella figura 3.5. A sinistra trovi il modulo che esegue la chiamata; nel nostro caso è il `main`. A destra c'è la funzione. Mi è sembrato più simpatico chiamarla `ciccio` — `cm` sa troppo di unità di misura. Quando il compilatore legge il sorgente dal `main` trova una chiamata alla funzione `ciccio`, con degli argomenti. Ecco, in sintesi, cosa fa:

- predispone una tabella con gli argomenti da passare alla funzione; in questa tabella inserisce gli argomenti, uno dopo l'altro, occupando per ciascuno lo spazio necessario.
- predispone lo spazio per ricevere il risultato; per questo deve sapere di che tipo di risultato si tratta.
- inserisce lo spazio per il salto all'entry point della funzione; non può ancora definirlo correttamente, perché non sa dove si trovi la funzione.
- predispone l'indirizzo di rientro, quello dell'istruzione immediatamente successiva alla chiamata.
- All'istruzione successiva alla chiamata preleva il risultato e lo deposita nella cella di memoria che gli ho indicato io.

Quando il compilatore legge il sorgente della funzione:

- predispone il riferimento all'entry point, la prima istruzione eseguibile del corpo della funzione.

- preleva gli argomenti dalla tabella preparata dal modulo chiamante.
- traduce il codice della funzione.
- inserisce le istruzioni per depositare il risultato nella cella predisposta dal modulo chiamante.
- rientra all'indirizzo predisposto dal modulo chiamante.

Il linker non fa altro che definire correttamente la chiamata. Il flusso di istruzioni e dati nel corso dell'esecuzione dovrebbe essere chiaro dalla figura. Ma...

Il passaggio di dati tramite tabella di argomenti e valore restituito comporta dei rischi da non sottovalutare. Torniamo un attimo sui nostri passi.

Quando il compilatore prepara la tabella degli argomenti – mentre compila il modulo chiamante – ha a disposizione come informazione la lista di argomenti che trova nell'istruzione di chiamata. L'ordine ed il tipo di dati che inserisce in tabella è determinato da questa lista.

Quando il compilatore preleva gli argomenti – mentre compila la funzione – ha a disposizione come informazione la lista di argomenti che trova nella dichiarazione del nome della funzione. L'ordine ed il tipo di dati che preleva dalla tabella è determinato da questa lista.

Conclusione ovvia: è mia responsabilità fare in modo che le due operazioni siano consistenti. In termini precisi, il numero, il tipo e l'ordine degli argomenti devono essere identici nel modulo chiamante e nella funzione chiamata. I nomi, ... sono come cenere nel vento: una volta passato il compilatore non esistono più.

Una considerazione analoga vale per il dato restituito dalla funzione. Quando il compilatore legge il modulo chiamante deve sapere di che tipo è il risultato. Se non lo sa, assume `int`. E come faccio a dirgli che la mia funzione restituisce un dato `double`, e non `int`? Semplice: devo trovare il modo di dirglielo prima che trovi l'istruzione di chiamata alla funzione. Ho due modi per farlo.

1. Inserisco il sorgente della funzione nello stesso file che contiene il modulo chiamante, ma *prima* del sorgente del modulo chiamante. In altre parole, l'ordine di inserimento dei moduli nel file sorgente deve essere opposto rispetto all'ordine delle chiamate. Va da sé che il `main` deve essere l'ultimo.
2. Uso un *prototipo* per la funzione (*function prototype*):

   $\langle tipo\rangle\ \langle nome\rangle$ `(` $\langle tipo\rangle\ \langle argomento\rangle, \ldots, \langle tipo\rangle\ \langle argomento\rangle$ `);`

   il prototipo va inserito nel file sorgente che contiene il modulo chiamante, prima delle istruzioni del modulo.

Nel caso della funzione `cm` il prototipo è

```
double cm(double x);
```

Non dimenticare il punto e virgola: da qui il compilatore capisce che si tratta solo di un prototipo, perché non ci sono le graffe che delimitano il corpo della funzione. Prendi il file `cm.c` che contiene il `main` prima della funzione `cm`; inserisci il prototipo prima della riga `int main()`. Riprova a compilare ed eseguire, e vedrai che tutto si sistema.

3.3.4 Un altro modo per preparare i sorgenti

Ma se avessi deciso di usare la mia preziosa funzione `cm` in diversi programmi sarei obbligato a copiarne il sorgente i tutti i files che scrivo? No, naturalmente, altrimenti dovresti sapere dove si trova il sorgente delle funzioni `printf`, `scanf`, `sqrt` o di tutte le funzioni di libreria che userai in futuro per copiarli nel tuo file sorgente! Potrai procedere così.

Prepara un file `cm.c` che contiene solo

```
/* cm: converte pollici in centimetri */
double cm(x)
    double x;
{
        vedi le istruzioni nel paragrafo 3.3.1
}
```

Non spaventarti se manca la linea `#include <stdio.h>`: in questo modulo non ci sono operazioni di I/O, quindi non serve. Poi prepara un file `prova_cm.c` che contiene

```
#include <stdio.h>
/* Prototipo della funzione cm */
double cm(double x);
/* main */
int main()
{
        vedi le istruzioni nel paragrafo 3.3.2
}
```

Nota che ho inserito il prototipo. Che succederebbe se lo dimenticassi? Non hai che da provare per saperlo, e da riflettere per capire perché: le informazioni le hai tutte, questa volta. Compila col comando

```
gcc -Wall -o prova_cm prova_cm.c cm.c
```

ed esegui.

A questo punto dovresti essere in grado di immaginare perché lo scambio di ordine tra `main` e funzione `cm` ha provocato un tal disastro: il compilatore ha trovato la chiamata alla funzione nel modulo `main`, e in mancanza di altre informazioni ha inteso che `cm` restituisse un dato `int`. Quindi ha trasferito un dato `int` in una cella destinata a contenere un `double`. Qualcosa di sporco deve essere accaduto. Cosa? Aspetta il prossimo capitolo.

Immagino la domanda successiva: “ma quando ho usato la funzione `sqrt` mica gli ho messo il prototipo; come ha fatto a sapere che `sqrt` restituisce un `double`, e non un `int`?” Risposta: ti sbagli; il prototipo ce l’hai messo, eccome: sta nel file `math.h`.

3.3.5 Un altro esempio

Ora che hai imparato a costruire una funzione a cui passi un dato, vediamo come fare a passare una tabella (o array, o vettore). Facciamo un esempio, al solito. Proviamo a scrivere una funzione che riceve come argomento un vettore `double` di lunghezza qualunque (purché positiva!) e calcola la media dei dati contenuti nel vettore. Precisiamo (rileggi l'inizio del paragrafo 3.3.1):

- il nome della funzione: con la consueta fantasia, proporrei di chiamarla `media`;
- gli argomenti devono essere due: il vettore `double` e il numero di elementi nel vettore, un `int`;
- l'algoritmo per il calcolo della media l'abbiamo già visto nel paragrafo 3.2.5: basta riutilizzarlo tale e quale;
- il risultato da restituire è un dato double.

Detto questo, la codifica è quasi obbligata:

```
/* media: calcola la media degli elementi di un vettore
   Parametri: vet[] : un vettore double
              n     : il numero di elementi
*/
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);
}
```

Un solo commento: la dichiarazione `double vet[]` significa: l'argomento `vet` che ti è stato passato è un vettore, non un semplice dato. Quanto è lungo? Non importa: è un vettore e basta. Questo lo dice la coppia di parentesi quadre. La memoria necessaria per memorizzare il vettore è già stata riservata dal modulo chiamante; quindi nella funzione non occorre riservare ancora della memoria: basta sapere dove si trova il vettore. Per questo basta conoscere l'indirizzo del primo elemento, e non la lunghezza.

E ora il programma di prova. È molto semplice:

```
#include <stdio.h>
/* Prototipo della funzione media */
double media(double vet[],int n);
/* Modulo main */
int main()
{
  double x,vet[1000];
  int j;
```

```
  x=0.7615234;              /* Riempimento della tabella */
  for(j=0; j<1000; j++) {
    x = 4.*x*(1.-x);
    vet[j] = x;
  }
                            /* Calcolo della media */
  x = media(vet,1000);
  printf("Media : %e\n",x);
  exit(0);
}
```

Non ti resta che copiare, compilare ed eseguire. Ma, a pensarci bene...

C'è qualcosa di strano. Gli argomenti non dovevano essere dei dati? Come fa il `main` a passare un vettore ad una funzione? Lo copia tutto nella tabella degli argomenti? Risposta: no.

Ricorda cosa ti ho spiegato a proposito dei vettori. La dichiarazione `double vet[1000]` spiega al compilatore che deve riservare spazio per 1000 elementi di tipo `double`, e che il nome `vet` identifica l'indirizzo del primo elemento. Ora pensaci un momento: che differenza c'è tra un dato ed un indirizzo? Semplice: l'uso che ne fa la CPU. L'indirizzo non è altro che un numero che identifica una cella di memoria, senza preoccuparsi del contenuto della cella. Ma è un numero come gli altri: non è fatto di bit di un colore diverso da quello dei dati (i bit, si sa, sono come l'idrogeno: incolori, insapori e inodori).

Quando passo il vettore `vet` come argomento di una funzione il compilatore fa una cosa molto semplice: prende il valore di `vet` e lo mette nella tabella degli argomenti. Ma il valore di `vet` è l'indirizzo del vettore; quindi ciò che viene passato è l'indirizzo. In altre parole, la funzione sa dove sta il vettore.

Passiamo alla codifica della funzione. Scrivendo `double vet[]` io ho proprio dichiarato che `vet` è l'indirizzo di un vettore: perfettamente consistente. Quando scrivo `vet[j]` la CPU prende l'indirizzo iniziale e determina la posizione del `j`–esimo elemento di `vet`; poi lo usa. Perfetto. Salvo una piccola ma non trascurabile informazione: se scrivessi, ad esempio, `vet[3]=17.5` andrei a modificare il contenuto di un elemento del vettore. In altre parole, la funzione può modificare dei dati memorizzati dal programma chiamante. Utilissimo come meccanismo di restituzione di un numero più o meno elevato di informazioni — mentre il valore restituito direttamente dalla funzione è uno solo. Ma estremamente pericoloso se attuato all'insaputa di chi scrive il modulo chiamante.

Non c'è modo di uscirne: tutte le cose utili hanno le loro controindicazioni. L'unico modo per garantire che non vi siano spiacevoli conseguenze è rispettare rigorosamente un paio di regole auree:

- chi scrive una funzione deve documentarla accuratamente, specificando

con chiarezza se la funzione usi o no gli argomenti per restituire dei dati al programma chiamante;
- chi usa una funzione scritta da altri deve leggere con attenzione la documentazione.

3.3.6 Avanti e indietro coi dati

Un terzo esempio: proviamo a scrivere una funzione che restituisca più di un valore. Anche qui, un problema molto semplice; quello che mi importa non è il contenuto della funzione, ma il traffico dei dati: voglio restituire informazioni mediante gli argomenti, e non tramite il nome della funzione. Proponiamoci di scrivere una funzione che riceva in ingresso un vettore reale, e restituisca i valori minimo e massimo tra i dati contenuti nel vettore. Anche qui, precisiamo:
- il nome della funzione è quasi obbligato: `minmax`;
- gli argomenti: in ingresso l'indirizzo del vettore (`double`) e la sua lunghezza (un `int`); in uscita due valori `double` che devono passare mediante gli argomenti;
- l'algoritmo: prova a fare uno sforzo di immaginazione e ad inventarne uno; potrebbe anche essere migliore di quello che ti illustrerò tra poco;
- il risultato da restituire in modo diretto: non sapendo quale dei due valori associare al nome della funzione decido salomonicamente di non dargliene nessuno;

Cominciamo col costruire lo scheletro della funzione, senza occuparci dell'algoritmo. Ecco il codice sorgente:

```
/* minmax: calcola il minimo ed il massimo di un vettore
   Parametri: vet[] : un vettore double
              n     : il numero di elementi
              vmin  : *indirizzo* di un double che
                      riceve il minimo
              vmax  : *indirizzo* di un double che
                      riceve il minimo
*/
void minmax(vet,n,vmin,vmax)
     double vet[];
     int n;
     double *vmin,*vmax;
{
     qui devi inserire il calcolo
  return;
}
```

Permettimi di richiamare la tua attenzione sui commenti: sono irrilevanti ai fini dell'esecuzione, ma sono preziosi come documentazione per chi volesse usare questa funzione. Noterai che in corrispondenza agli argomenti `vmin` e `vmax` ho ben specificato che la funzione deve ricevere (nella tabella degli argomenti) l'indirizzo dei due dati, e non i dati: la funzione deve restituire dei valori, e quindi deve sapere dove andare a depositarli. Ripensa alla differenza tra le funzioni `printf`, che vuole il dato da stampare, e `scanf`, che deve restituire un dato.

La dichiarazione della funzione è `void minmax(...)` : significa semplicemente che la funzione non restituisce nessun valore tramite il suo nome: non potrai richiamarla nella forma `y = minmax(...)` come per la funzione `media` del paragrafo 3.3.5, ma solo nella forma `mimnax(...)` come per `printf` o `scanf` .

La dichiarazione `double *vmin,*vmax` significa: nella tabella degli argomenti trovi gli `indirizzi` delle variabili `vmin` e `vmax`; nei manuali tecnici trovi talvolta le espressioni *address of*, *pointer to* o *puntatore a* `vmin` o `vmax`: sono tutte equivalenti. Che i nomi corrispondano ad indirizzi e non a dati è specificato dall'asterisco che precede i nomi.

E come faccio a depositare dei dati a quegli indirizzi? Semplice: invece di scrivere `vmin =` $\langle$*qualche cosa*$\rangle$ scrivi `*vmin =` $\langle$*qualche cosa*$\rangle$: l'asterisco di fronte al nome `vmin` significa proprio: il dato che mi interessa è quello che si trova all'indirizzo di memoria `vmin`.

Devi scrivere anche un programma che chiama la funzione, ovviamente. Il grosso del programma lo puoi copiare da quello che hai scritto per la funzione `media`. Eccoti il testo sorgente:

```
#include <stdio.h>
/* Prototipo della funzione minmax */
void minmax(double vet[],int n,double *vmin,double *vmax);
/* Modulo main */
int main()
{
  double x,xmin,xmax,vet[1000];
  int j;
  x=0.7615234; /* Riempimento della tabella */
  for(j=0; j<1000; j++) {
    x = 4.*x*(1.-x);
    vet[j] = x;
  }
/* Calcolo di minimo e massimo */
  minmax(vet,1000,&xmin,&xmax);
  printf("Minimo: %e ;  Massimo: %e\n",xmin,xmax);
  exit(0);
}
```

Le sole righe che devi osservare sono il prototipo della funzione, dove è specificato che gli ultimi due argomenti sono indirizzi e non dati, e la linea dove si fa la chiamata. In quest'ultima riga gli argomenti sono indicati con `&xmin` e

`&xmax`, proprio per spiegare al compilatore che nella tabella degli argomenti deve mettere l'indirizzo, e non il dato.

E veniamo all'algoritmo:

(i) definisci due variabili $v_{\min}$ e $v_{\max}$ assegnando loro come valore il primo elemento del vettore;

(ii) fa scorrere tutti gli elementi del vettore partendo dal secondo:

  (ii.a) se $v_{\min}$ è maggiore dell'elemento corrente riassegna a $v_{\min}$ il valore dell'elemento corrente;

  (ii.b) se $v_{\max}$ è minore dell'elemento corrente riassegna a $v_{\max}$ il valore dell'elemento corrente;

(iii) restituisci i valori $v_{\min}$ e $v_{\max}$.

Questa è la codifica in linguaggio C:

```
int j;
*vmin = *vmax = vet[0];
for(j=1; j<n; j++)
  if(*vmin > vet[j]) *vmin = vet[j];
  if(*vmax < vet[j]) *vmax = vet[j];
```

Osserva bene l'uso dell'asterisco per indicare che faccio riferimento ai dati che si trovano agli indirizzi specificati da `vmin` e `vmax`.

Completare il corpo della funzione è ormai una faccenda banale. Batti i sorgenti, compila ed esegui. Se saltassero fuori errori, ... ti ho già detto anche troppe volte cosa dovresti fare.