1.4 Il problema delle approssimazioni

A causa del numero limitato di cifre a disposizione, un numero si può rappresentare esattamente in un calcolatore solo se la sua parte frazionaria è esprimibile come somma di un numero limitato di potenze di 2. Diversamente lo si approssima a quello a esso piú vicino. Mentre 22.75 era rappresentabile esattamente in altri casi, ad esempio 0.1, non è cosí. Per convincervene supponete di avere a disposizione un computer a 32 bit e provate a scrivere questo numero in notazione binaria.

Per lo stesso motivo è evidentemente impossibile rappresentare i numeri irrazionali (come $\sqrt{2}$, π, etc.): questi hanno infinite cifre dopo la virgola perciò non possono trovare posto nella memoria di un calcolatore. Si possono solo approssimare al numero razionale piú vicino.

Questa limitazione conduce a un problema con il quale occorre misurarsi ogni volta che si debbano eseguire calcoli con un computer: l'errore di arrotondamento. In pratica tutti i numeri non interi si approssimano al numero razionale piú vicino che ammette una rappresentazione finita, con una precisione dell'ordine di una parte su 2^{-n_m}, dove n_m è il numero di bit riservati alla mantissa: due numeri che differiscono tra loro per meno di questa quantità si considerano uguali su un calcolatore e un numero minore del piú piccolo numero rappresentabile equivale a 0 (in questo caso si parla di errore di *underflow*).

Occorre sempre prestare molta attenzione a questo tipo di approssimazione perché, benché in molti casi appaia di piccola entità, l'errore può propagarsi in maniera catastrofica in alcuni algoritmi, specie se di tipo iterativo, e diventare importante (un esempio concreto di questo effetto è descritto nel Capitolo 4). Un caso frequente di errore catastrofico si verifica quando gli operandi sono tra loro molto diversi o molto simili. Nel primo caso il piú piccolo dei due numeri perde precisione a causa dello spostamento della mantissa necessario a portarlo a una rappresentazione con lo stesso esponente del piú grande. Consideriamo la somma tra i numeri $a = 68\,833\,152$, che nella rappresentazione IEEE 754 si scrive 0 1001 1001 000 0011 0100 1001 1111 0000 e $b = 2309\,657\,318\,129\,664$ (0 1011 0010 000 0011 0100 1001 1111 0000). Per poter eseguire questa somma occorre incolonnare i numeri, pertanto il piú piccolo (a) deve essere espresso con lo stesso esponente di 2 del piú grande (b) nella notazione IEEE 754. L'esponente di a è 26, mentre quello di b è 51. La differenza tra questi due numeri è 25. La mantissa di a, dunque, si dovrebbe riscrivere spostandone le cifre di altrettanti posti verso destra. Poiché il numero massimo di cifre per la mantissa è 23, il risultato netto sarà che la mantissa di a sarà rappresentata come una sequenza di zeri:

$$\begin{array}{lll} 0\ \ 1011\,0010\ \ 000\,0000\,0000\,0000\,0000\,0000 & + & \ldots \\ \underline{0\ \ 1011\,0010\ \ 000\,0011\,0100\,1001\,1111\,0000} & = & \\ 0\ \ 1011\,0010\ \ 000\,0011\,0100\,1001\,1111\,0000 & & \end{array}$$

e la somma risulta essere $a + b = b$, un'evidente assurdità!

Il secondo genere d'errore si presenta spesso nel calcolo della differenza tra due numeri

vicini. Lo si comprende bene attraverso semplici esempi che svolgiamo nella base 10 solo perché piú familiare e dunque di comprensione piú immediata. Supponiamo di rappresentare i numeri in virgola mobile con 3 cifre significative dopo la virgola. Il numero 1000 si rappresenta come $a = 1.000 \times 10^3$, mentre il numero 999.8 come $b = 9.998 \times 10^2$.

La differenza $(a - b) = 0.2$, ma quando questa si calcola in virgola mobile il numero piú piccolo si deve esprimere con una potenza di 10 uguale a quella del piú grande, con conseguente perdita di cifre significative, quindi $b \to b' = 0.999 \times 10^3$ e la differenza diventa

$$(1.000 - 0.999) \times 10^3 = 10^{-3} \times 10^3 = 1.0\,.$$

Il valore approssimato è piú grande di un fattore 5 rispetto al valore vero, benché la precisione sul singolo numero sia dell'ordine di 10^{-3}! In taluni casi questo genere d'errore si può evitare (o almeno ridurre) riformulando l'espressione da calcolare. Supponiamo, dati $x = 3.451 \times 10^0$ e $y = 3.45 \times 10^0$, di dover eseguire l'operazione $\Delta = \left(x^2 - y^2\right) = 0.006\,901 = 6.901 \times 10^{-3}$. Se si calcolano prima i quadrati $x^2 = 11.909\,401 = 1.190 \times 10^1$ e $y^2 = 11.9025 = 1.190 \times 10^1$, si trova che $\Delta = 0$. Un risultato davvero disastroso, specialmente se Δ rappresenta il denominatore di un'espressione. Se si riformula l'espressione come $\Delta = \left(x^2 - y^2\right) = (x - y)\,(x + y)$, invece, si trova che $(x - y) = (3.451 - 3.450) \times 10^0 = 0.001 \times 10^0 = 1.000 \times 10^{-3}$, mentre $(x + y) = (3.451 + 3.450) \times 10^0 = 6.901 \times 10^0$, quindi $\Delta = 1.000 \times 10^{-3} \times 6.901 \times 10^0 = 6.901 \times 10^{-3}$.

Nel calcolo numerico, diversamente da quello analitico, l'ordine in cui si eseguono le operazioni è di estrema importanza, cosí come l'ordine di grandezza dei termini soggetti alle operazioni aritmetiche che non devono essere troppo diversi tra loro, né troppo simili, secondo le operazioni svolte su di essi.

1.5 Non-numeri sul calcolatore

Lo svolgimento di un compito, sia esso a carattere scientifico o no, può richiedere la manipolazione di dati di natura diversa dai numeri, come immagini, suoni, etc. Oggi questo può sembrare ovvio perché costantemente sugli schermi dei computer si possono osservare immagini o seguire un film; attraverso gli altoparlanti si può ascoltare la musica; con un microfono si può comunicare con la voce con altri utenti remoti attraverso la rete; con una telecamera si può fare una videoconferenza.

Quando nacquero, negli anni '40 del secolo scorso, i computer servivano esclusivamente a eseguire calcoli. I numeri rappresentavano praticamente l'unica forma d'informazione disponibile. I programmi si impostavano azionando interruttori e i risultati erano visualizzati da lampadine il cui stato (acceso/spento) indicava la sequenza di bit nella rappresentazione binaria. Quando fu introdotta la possibilità di produrre i risultati in forma scritta (su carta e, successivamente, su schermo) e d'inserire i comandi mediante una tastiera si pose il problema di rappresentare altri tipi d'informazione come i caratteri. La necessità di rappresentare informazioni di natura diversa aumentò costantemente nel

La forma dell'operatore di *cast* è

```
(nuovo tipo) operando
```

Se consideriamo le istruzioni

```
int i = 3;
double a;
a = (double) i / 2;
```

senza l'operatore di *cast* `(double)` la variabile `a` assumerebbe il valore 0; invece la variabile intera `i` è convertita a `double` nell'ambito della sola espressione che contiene `(double)`, restando comunque intera e uguale a `3`; l'espressione è promossa a `double` e `a` vale 1.5.

3.4 La prima volta dell'Input/Output

Proviamo a compilare il programma nel Listato 3.1: la compilazione non dà errori e l'esecuzione avviene senza problemi. Tuttavia non osserviamo alcun effetto di tale esecuzione. La ragione è semplice: nel Listato 3.1 non ci sono istruzioni di *input/output*, (I/O), cioè che ricevano dati dall'esterno o passino dati dal programma all'esterno. L'estensione piú ovvia del programma nel Listato 3.1 consiste nel richiedere all'utente di fornire un valore di T_C per la variabile `tc` e stampare il valore calcolato di T_F (`tf`); del resto questo è lo scopo di un programma di conversione di unità di misura!

Prima di illustrare la forma delle istruzioni di I/O, discutiamo brevemente a cosa corrisponda in pratica ricevere input o fornire output. Nei calcolatori di oggi un programma interattivo può ricevere input da qualsiasi periferica progettata per tale scopo, per esempio mediante caratteri di tastiera o file da un disco. In entrambe i casi il programma riceve i dati attraverso un canale di comunicazione che è gestito dal sistema operativo. I linguaggi di programmazione forniscono funzioni *native* che fanno da interfaccia con i servizi di I/O del sistema. Il discorso vale anche per l'emissione di output: un programma può scrivere su periferiche, come un terminale, o su un file e lo fa usando funzioni di output del linguaggio che richiedono i servizi del sistema operativo.

```
1 main() {
2    double tc, tf, offset, conv;
3    offset = 32.;
4    conv = 5./ 9.;
5    printf("Valore in gradi Fahrenheit = ");
6    scanf("%lf", &tf);
7    tc = (tf - offset) * conv;
8    printf("Valore in gradi celsius = %f", tc);
9 }
```

Listato 3.4 Conversione di unità di misura con I/O.

Il linguaggio C usa la funzione `scanf` per ricevere input da tastiera e la funzione `fscanf` per ricevere input da file. Le funzioni di output sono `printf` per scrittura su terminale e `fprintf` per scrittura su file. Queste funzioni hanno una sintassi molto articolata che permette di gestire una grande varietà di formati: nel seguito di questo paragrafo illustriamo soltanto una parte delle opzioni esistenti, rinviando a un manuale del linguaggio C [29, 28] per una trattazione completa. Anticipiamo inoltre che nell'uso della funzione `scanf` compare un simbolo del linguaggio il cui significato è chiarito nel Capitolo 6; per adesso basta seguire la descrizione della sintassi del comando.

Rivediamo il Listato 3.1 con inserite le istruzioni di I/O. Esaminiamo l'istruzione `printf` nella riga 5. All'interno delle parentesi tonde `()` troviamo una stringa di caratteri compresa tra doppi apici `"···"`: nella riga 5 la stringa è `Valore in gradi Fahrenheit =`, e tale stringa è stampata su output (per esempio sul terminale). Lo scopo è di poter scrivere messaggi informativi che chiariscono cosa il programma stia calcolando, o quali operazioni svolga e cosí via. Nella riga 8 il formato della stringa tra parentesi tonde `()` è un po' diverso: compare infatti, oltre al messaggio da stampare, anche il termine `%f`. Tale termine si chiama *descrittore* e serve appunto a *descrivere* il formato della variabile che segue il messaggio: in questo caso il descrittore `%f` informa il compilatore che la variabile `tc` è di tipo `double`, in modo che la si possa stampare su output correttamente. Come si vede la funzione `printf` ammette un numero variabile di argomenti, separati da una virgola `,`: il suo formato generico è

```
printf("stringa di caratteri"[,esp1, esp2,...]);
```

La *`stringa di caratteri`* è sempre racchiusa tra doppi apici `"···"`, e, nel caso sia seguita da una o piú espressioni da stampare, contiene un descrittore del tipo di espressione per ciascuna espressione che segue: il descrittore ha la forma `%caratteri`. Le espressioni *`esp1, esp2,...`* sono opzionali e di numero variabile. Usiamo il termine *espressioni* perché *`esp1`* può essere una variabile semplice o un'espressione complessa: avremmo potuto scrivere `printf("Valore in gradi celsius = %f",(tf - offset) * conv);` calcolando il valore di `tc` direttamente tra gli argomenti di `printf`. Il descrittore `%f` fa riferimento a una quantità rappresentata in virgola mobile. Nella Tabella 3.7 elenchiamo i descrittori d'uso piú frequente per `printf`.

Descrittore	*Tipo*
`%f,%e,%g`	`float,double`
`%d,%i`	`int`
`%c`	`char` (singolo)
`%s`	`char` (stringa)

Tabella 3.7 Descrittori di output.

La funzione `scanf`, nella riga 6 del Listato 3.4, serve a leggere il valore di una o piú variabili da input (per esempio da tastiera). Il suo formato è simile a quello della funzione

`printf` con alcune importanti differenze: la prima è che tra doppi apici `"···"` non possono mai comparire messaggi ma soltanto i descrittori delle variabili da acquisire; la seconda consiste nel simbolo `&` che deve obbligatoriamente precedere ogni variabile da acquisire. La sintassi generica dell'istruzione `scanf` è

```
scanf("stringa di caratteri",&var1[,&var2,...]);
```

La *`stringa di caratteri`* racchiusa tra doppi apici `"···"` contiene tutti e soli i descrittori delle variabili che vanno acquisite in `var1, var2,...`. Anche in questo caso il numero di variabili che seguono la *`stringa di caratteri`* non è fissato; ciascuna variabile deve obbligatoriamente essere preceduta dal carattere `&`. Nella riga 6 del Listato 3.4 si usa il descrittore `%lf` per la variabile `tf` di tipo `double` (notare che se qui si usasse il descrittore `%f` relativo al tipo `float` il programma avrebbe un comportamento diverso). Nella Tabella 3.8 elenchiamo i descrittori di lettura d'uso piú frequente.

Descrittore	*Tipo*
`%f`	`float`
`%lf`	`double`
`%Lf, %llf`	`long double`
`%d, %i`	`int`
`%u`	`unsigned int`
`%Lu`	`unsigned long long int`
`%c`	`char` (singolo)
`%s`	`char` (stringa)

Tabella 3.8 Descrittori di input.

Le funzioni di I/O per i file sono discusse nel Capitolo 6.

Quesito 3.2 *Istruzioni di I/O*

Scrivete istruzioni `printf` e `scanf` per leggere da input una variabile `a` di tipo `double` e una variabile `k` di tipo `long int`, in base a un messaggio stampato dal programma, e quindi stampare i valori di `a` e `k` moltiplicati per la costante `3.14`, preceduti da un messaggio esplicativo.

Vogliamo infine sottolineare che il programma nel Listato 3.4, nella forma qui riportata, in realtà dà errori di compilazione[1]; il motivo è legato alle considerazioni fatte nel Paragrafo 2.4, cioè alla necessità di *collegare* il codice scritto dal programmatore (il Listato 3.4) a codice già esistente e, in questo specifico caso, alle funzioni `printf` e `scanf`.

[1] Come è spiegato nel successivo paragrafo ci sono compilatori configurati in modo da non dare questo errore. Il discorso resta però valido in generale.