solo quelli pari. La condizione presente nell'istruzione `if`, infatti, è vera quando il resto della divisione per due del numero inserito è 1. In questo caso l'istruzione `continue` permette di saltare il resto delle righe e tornare alla valutazione dell'espressione di controllo `(n > 0)`. Anche in questo caso è facile trovare un sostituto; basta riscrivere la condizione per ottenere una struttura diversa:

```
int n = 1, S = 0;
while (n > 0) {
   scanf("%d", &n);
   if (!(n % 2)) {
      S += n;
   }
}
printf("%d", S);
```

In questa forma siete sicuri che il vostro programma sarà accettato da tutti come un buon programma.

4.5 Un problema di arrotondamento

In questo paragrafo discutiamo un problema apparentemente banale, ma particolarmente istruttivo: la somma di N numeri x_j, $j = 0 \ldots N-1$. La collocazione di questo paragrafo in questo capitolo non è casuale. I problemi numerici che trattiamo si verificano infatti quando il risultato si ottiene per mezzo di molte iterazioni. Per semplificare la trattazione facciamo l'ipotesi che i numeri da sommare siano tutti uguali; in particolare scegliamo $N = 10\,000\,000$ e $x_j = 7$, $\forall j$. Il risultato S che ci aspettiamo è banalmente $S = 70\,000\,000$. Ora consideriamo il Listato 4.11.

```
1 #include <stdio.h>
2
3 #define N 10000000
4
5 main() {
6   float S = 0., x = 7.;
7   unsigned int i, iS = 0, ix = 7;
8   for (i = 0; i < N; i++) {
9     S += x;
10    iS += ix;
11  }
12  printf("Using floats  : %.0f x %d = %.0f\n", x, N, S);
13  printf("Using integers: %d x %d = %d\n", ix, N, iS);
14 }
```

Listato 4.11 Un programma dai risultati inaspettati.

Prima di discutere l'algoritmo osservate il formato dell'istruzione `printf` alla riga 12: `%.0f x %d = %.0f`. Il comportamento standard degli specificatori di formato può essere alterato con modificatori posti tra il carattere `%` e lo specificatore (`f` in questo caso). Il modificatore degli specificatori per le variabili razionali è in genere della forma `n.m`, dove `n` è il numero minimo di caratteri da usare per stampare il risultato (incluso l'eventuale punto decimale) e può essere omesso. Nel caso in cui il numero da stampare sia piú corto, alla sua sinistra si aggiungono automaticamente degli spazi in modo da occupare esattamente `n` caratteri. Se `n` è omesso il valore è stampato con le cifre necessarie allineandolo a sinistra. `m` rappresenta invece il numero massimo di cifre da stampare dopo il punto decimale. In questo caso stiamo chiedendo di stampare i valori razionali senza cifre dopo la virgola. Nel caso degli interi il modificatore è il piú delle volte costituito di un numero intero `n` che rappresenta il numero minimo di caratteri da stampare. Le possibilità offerte dai modificatori sono naturalmente molte di piú; per i dettagli rimandiamo, come al solito, alla bibliografia.

Se compiliamo ed eseguiamo il programma nel Listato 4.11 troviamo il seguente risultato:

```
Using floats  : 7 x 10000000 = 77603248
Using integers: 7 x 10000000 = 70000000
```

Che succede? Come mai la somma S, nel caso in cui questa sia calcolata usando numeri razionali, è sbagliata di piú del 10 per cento? La risposta è semplice: si tratta di un problema di arrotondamento (Paragrafo 1.4). Al contrario dei numeri interi, quelli razionali non sono rappresentati esattamente nella memoria dei calcolatori, a causa della limitatezza delle cifre a disposizione. Ricordiamo che i numeri razionali si rappresentano nella notazione in virgola mobile IEEE 754.

Se si segue passo per passo il programma, si scopre che la prima deviazione rispetto al valore vero si ha già per $i = 2396\,746$, quando la variabile `S`, che rappresenta la somma S, dovrebbe valere $16\,777\,229$. Al passo precedente `S` $= 16\,777\,222$ che, nella notazione standard è

$$\texttt{0 1001 0111 (1) 000 0000 0000 0000 0000 0011}$$

in cui la cifra tra parentesi è quella sottintesa, la mantissa è costituita delle ultime 23 cifre, l'esponente di 2 si ottiene interpretando il gruppo di 8 bit a sinistra in eccesso a 127 e il segno è dato dalla prima cifra. In questo caso l'esponente vale 24 $(151 - 127)$ e la mantissa si ottiene sommando $1 + 2^{-22} + 2^{-23}$, da cui `S` $= 2^{24}(1 + 2^{-22} + 2^{-23})$. Il numero 7 invece si scrive semplicemente come $1.75 \times 2^2 = (1 + 0.5 + 0.25) \times 2^2$, cioè

$$\texttt{0 1000 0001 (1) 110 0000 0000 0000 0000 0000}$$

Quando questo numero deve essere sommato a `S` $= 16\,777\,222$, si deve esprimere come un numero moltiplicato per 2^{24}. Le cifre della mantissa (incluso il bit 1 sottinteso) devono essere traslate di 22 posti e quando si svolge questa operazione la cifra piú a destra si perde perché non c'è posto per essa nei 4 byte riservati ai `float`. Il risultato è

$$0\ 1001\,0111\ (0)\ 000\,0000\,0000\,0000\,0000\,0011 = 6$$

Sommando i due numeri si ottiene

$$\begin{array}{lllll} 0\ 1001\,0111\ (1)\ 000\,0000\,0000\,0000\,0000\,0011 & = & 16\,777\,222 & + \\ 0\ 1001\,0111\ (0)\ 000\,0000\,0000\,0000\,0000\,0011 & = & 6 & = \\ 0\ 1001\,0111\ (1)\ 000\,0000\,0000\,0000\,0000\,0110 & = & 16\,777\,228 & \end{array}$$

Si ottiene un numero che differisce di un'unità rispetto al risultato corretto. Benché la differenza sia cosí contenuta il risultato finale è disastroso poiché questo errore si cumula numerose volte nel corso delle iterazioni, fino a condurre a un errore della portata di quello illustrato. In seguito a ciò si potrebbe pensare che il risultato finale dovrebbe essere inferiore a quello corretto; la perdita dei bit dovrebbe infatti condurre a sommare numeri piú piccoli (prima varie volte 6 e poi 2) alla variabile `S`. Il fatto è che nelle moderne CPU è presente un'unità detta FPU (*Floating Point Unit*) che si occupa proprio del trattamento dei dati in virgola mobile che, per diminuire la probabilità di commettere tali errori, usa 80 bit per rappresentare internamente i risultati. Questi si devono però arrotondare troncandoli a 32 bit, nel caso dei `float`, quando vengono trasferiti nella memoria.

`i`	`R7`	`S`	`P`
2396 745	16 777 222	16 777 222	0
2396 746	16 777 229	16 777 228	1
2396 747	16 777 235	16 777 236	1
2396 748	16 777 243	16 777 244	1
2396 749	16 777 251	16 777 252	1

Tabella 4.1 Contenuto delle variabili `i` e `S` e dei registri `R7` e `P` della FPU nel corso dell'esecuzione del programma listato in 4.11.

Osserviamo la Tabella 4.1 in cui sono riportati i valori delle variabili `i` e `S` con il contenuto del registro `R7` della FPU, che memorizza il risultato dell'ultima operazione svolta, e il valore del bit `P` della FPU che indica l'occorrenza di un errore di precisione. Quando `i` = 2396 745, `R7` = 16 777 222, che rappresenta il valore corretto, copiato successivamente nella variabile `S`. All'iterazione successiva si verifica la perdita dell'ultimo bit come abbiamo illustrato precedentemente; nella FPU, però, che dispone di 80 bit, il risultato è correttamente riportato come `R7` = 16 777 229. Quando si copia questo valore in memoria, il troncamento a 32 bit fa sí che la variabile `S` assuma il valore 16 777 228 e al bit `P` sia assegnato il valore 1. All'iterazione seguente nella FPU viene copiato il valore (sbagliato) di `S` e a esso si aggiunge il valore 7, portando il registro `R7` ad assumere il valore 16 777 235. Tale valore non è rappresentabile esattamente con 32 bit e il valore a esso piú vicino è 16 777 236, che è dunque assunto da `S`. Ancora una volta `P` vale 1 per effetto dell'arrotondamento. Successivamente accade qualcosa di analogo: a ogni passo in

R7 viene copiato il valore di S dell'iterazione precedente e a esso si somma 7. Il risultato però non è mai rappresentabile con 32 bit ed è approssimato al valore piú vicino espresso come somma di potenze di due, che è pari a quello corretto aumentato di 1. Analizzando ciascun passo successivo si scopre che, di fatto, alla variabile S si somma quasi sempre il valore 8 e in un solo caso il valore 4. Alla fine il risultato che si ottiene effettivamente è

$$\begin{aligned} \mathtt{S} &= 7 \times 2396\,746 + 4 + 6 + 8 \times (10\,000\,000 - 2396\,746 - 1 - 1) = \\ &= 16\,777\,222 + 4 + 6 + 8 \times 7603\,252 = \\ &= 77\,603\,248\,. \end{aligned}$$

Il motivo fondamentale per cui questo accade è che i numeri da sommare differiscono troppo tra loro (16 777 222 contro 7) e naturalmente lo stesso effetto si può verificare se i numeri x_j da sommare sono tra loro tutti diversi, ma piccoli. In questo caso il risultato è solo piú difficile da verificare e da comprendere. Il pericolo non si manifesta se

$$\log_2 S - \log_2 x_j \ll p\,,$$

dove p è la precisione della mantissa in numero di bit. Nel caso in esame $p = 23$, $N = 10^7$ e $x_j = 7, \forall j$ dunque

$$\log_2 7 \times 10^7 - \log_2 7 \simeq 23.253\,,$$

che è proprio dell'ordine di p. Usando una variabile di tipo `double` il problema sarebbe emerso per N maggiori, dell'ordine di 10^{16}, ma non sarebbe scomparso. Osservate che usando gli interi i numeri non vengono invece mai approssimati, ma poiché l'intervallo dei numeri rappresentabili è fortemente ridotto, non sempre è possibile usarli.

Vale la pena osservare che i moderni compilatori possiedono la capacità di ottimizzare il codice per renderlo piú veloce o tale da usare meno memoria. L'ottimizzazione può in taluni casi nascondere questo problema; se, per esempio, si compila il programma nel Listato 4.11 in un sistema Linux, usando il compilatore `gcc` versione 3.3.3 con l'opzione `-O1`, il problema scompare.

Il motivo risiede nel fatto che l'ottimizzatore, tra l'altro, estrae dai cicli gli invarianti, cioè tutte quelle quantità che non dipendono dal ciclo. In altre parole l'ottimizzatore trasforma il ciclo direttamente in

```
S = x * N;
iS = ix * N;
```

Occorre però ricordare che abbiamo usato un valore costante per x solo per semplificare la nostra discussione, ma che nella pratica i valori di x_j saranno tutti diversi tra loro. In questi casi l'ottimizzatore non potrà apportare alcun beneficio.

Per ovviare a questo inconveniente si deve cercare la maniera di ridurre la differenza tra i numeri da sommare. Un modo consiste nell'eseguire le somme in passi diversi (ad esempio, sommando prima M valori di x_j per volta e poi sommando tra loro i K

risultati, con $K \times M = N$). La scelta piú estrema in questo caso consiste nel sommare le coppie di due numeri adiacenti x_j e x_{j+1} ($M = 2$) e addizionare i risultati tra loro, iterando le operazioni fino a che non rimanga che un numero. Per fare questo si pone $x_i = x_{2i} + x_{2i+1}, i = 0 \ldots N/2$ a ogni iterazione[1], riducendo cosí il numero di componenti da sommare di un fattore 2.

Un metodo piú generale consiste nell'usare l'algoritmo di somma di Kahan [26], illustrato nel Listato 4.12.

```
1  float sum=0., corr=0., x = 7.;
2  int i;
3  for(i=0; i<N; i++) {
4    float tmp, y;
5    y = corr + x;
6    tmp = sum + y;
7    corr = (sum - tmp) + y;
8    sum = tmp;
9  }
10 sum += corr;
```

Listato 4.12 L'algoritmo di somma di Kahan.

Dal punto di vista algebrico, naturalmente, la sequenza di operazioni svolte nell'algoritmo è del tutto equivalente a quella del Listato 4.11. Nel Listato 4.12 la variabile `corr` rappresenta una correzione da apportare, volta per volta, all'elemento da sommare `x` per compensare l'errore commesso al passo precedente. La valutazione di questa correzione è affetta da un errore molto minore (al limite nullo) perché i numeri da sottrarre sono dello stesso ordine di grandezza.

Fino a che le somme sono esatte il valore di `corr` resta pari a zero e dunque `y = x`. La variabile `tmp` rappresenta la somma temporanea approssimata: qui è possibile che la precisione non venga conservata a causa della perdita di informazione discussa sopra. Se il risultato fosse esatto la differenza `(sum - tmp)` valutata alla riga 7 sarebbe esattamente pari a `-y` e dunque `corr` continuerebbe a valere zero; viceversa essa assumerebbe un valore pari alla quantità persa per effetto delle approssimazioni[2].

Nel caso dell'esempio discusso sopra, quando `sum` $= 16\,777\,222$, `tmp` $= 16\,777\,228$. In questo caso `corr` $= (\texttt{sum} - \texttt{tmp}) + \texttt{y} = (16\,777\,222 - 16\,777\,228) + 7 = -6 + 7 = 1$. Al passo successivo si aggiunge alla somma il valore 8 invece del valore 7, compensando l'errore.

Ricordate sempre che l'aritmetica dei calcolatori non è cosí semplice come si può pensare e occorre prestare molta attenzione ai problemi di carattere numerico, specialmente nei casi in cui nel programma sono presenti molte iterazioni.

[1] Occorre naturalmente prendere delle precauzioni ogni volta che il numero di elementi da sommare è dispari.

[2] Non sempre si può recuperare integralmente questa quantità, ma si dimostra[18] che il risultato finale ottenibile con l'algoritmo di Kahan è esprimibile come $\sum x_j(1+\delta_j) + \mathcal{O}(Np^2)\sum|x_j|$, con $|\delta_j| \leqslant 2p$.