Programmazione consapevole (semplice è bello)

Andrea Salmaso

1 aprile 2018

Indice

Prefazione Lo scopo del libro è duplice. In primo luogo si propone di insegnare un linguaggio di programmazione semplice e popolare e in secondo luogo, intende sviluppare nei lettori la capacità di usare la programmazione in modo scientifico per risolvere problemi. Questo significa, in primo luogo, definire in modo preciso problemi da affrontare e successivamente spiegare, in modo altrettanto chiaro, perché i programmi proposti per risolverli effettivamente lo fanno. Il linguaggio di programmazione è C++ senza la parte orientata agli oggetti. In pratica si tratta del linguaggio C con alcune caratteristiche aggiuntive che lo rendono più facile da usare rispetto al semplice C.

Il libro non assume alcuna conoscenza preliminare di Informatica ed è quindi pensato per un corso iniziale di programmazione, sia in una scuola superiore, sia nella Laurea (triennale) in Informatica o in Ingegneria Informatica. Il testo abbonda di esercizi, sia risolti che aperti. Ogni programma presentato nel libro è accompagnato da una dimostrazione di correttezza intuitiva, ma anche ragionevolmente precisa. La programmazione ricorsiva viene trattata diffusamente e la correttezza dei programmi ricorsivi viene dimostrata usando l'induzione.

Capitolo 1

Introduzione

1.1 Struttura di un computer

1.2 Come scrivere ed eseguire un programma

1.3 Notazione

Capitolo 2

Tipi predefiniti e variabili

2.1 Tipi predefiniti

2.2 Variabili e dichiarazioni

2.3 Espressioni e conversioni

2.4 A cosa servono i tipi

Capitolo 3

Istruzioni di base

Questo capitolo è dedicato alla descrizione delle istruzioni di base del C e C++. Chiariamo subito che queste istruzioni si trovano in qualsiasi linguaggio di programmazione della grande famiglia dei linguaggi imperativi, come per esempio Fortran, Pascal, ma anche quelli orientati ad oggetti come C++, Java eccetera.
Tratteremo in primo luogo le istruzioni di input e di output e successivamente esamineremo l'assegnazione, il comando condizionale ed il comando iterativo `while`.

In generale un'istruzione ha l'effetto di modificare il valore di qualche variabile. Per evitare di creare malintesi chiariamo subito che una variabile in programmazione ha due valori: l'R- e l'L-valore. L'R-valore è quello che viene informalmente chiamato valore e che (normalmente) è un valore del tipo della variabile in questione. L'R-valore viene però immagazzinato in memoria e l'L-valore di una variabile è l'indirizzo della memoria in cui l'R-valore è immagazzinato.

Nel seguito, per spiegare il significato delle istruzioni, useremo spesso la nozione di **stato del calcolo** che spiega l'R-valore di tutte le variabili in un dato momento. Più formalmente, lo stato del calcolo in un dato momento dell'esecuzione è rappresentato da una funzione **S** tale che ogni variabile attiva `x` del programma, **S**(x) è l'R-valore di `x` in quel momento. Il caso che `x` sia indefinita viene rappresentato con **S**(x)=$\perp$. Il significato di ogni istruzione può venire specificato attraverso le modifiche che l'istruzione opera sullo stato del calcolo.

3.1 Input/Otput

I programmi devono poter scambiare informazioni con l'esterno ed a questo servono le operazioni di lettura e stampa, dette di input/output e abbreviate in i/o.
Ci limiteremo a spiegare le operazioni più semplici che poi useremo sempre nel seguito del testo. Operazioni di i/o maggiormente sofisticate sono descritte in www.cplusplus.com. Un programma comunica con l'esterno per mezzo di dispositivi molto diversi: in un computer moderno, l'input standard è la tastiera, l'output standard è lo schermo, ma un programma potrebbe poter scrivere o leggere da un dispositivo USB o da CD o DVD o semplicemente da un file nella memoria del computer o anche un file che risiede nella memoria di qualche altro computer raggiungibile d quello su cui il programma esegue. Sarebbe troppo complicato se ci fossero istruzioni di i/o distinte per ciascun dispositivo. Quindi tutti questi dispositivi vengono visti dai programmi nello stesso modo: come un **file** Vediamo

innanzitutto cos'è un file, successivamente ci occuperemo di come un programma acceda a un file e delle principali operazioni di i/o disponibili.

3.1.1 I file

Concettualmente un file è una sequenza di dati che terminano con un carattere particolare, detto **end of file**. Ci sono 2 tipi di file: file di **testo** e file **binari**. Un file di testo è costituito da una sequenza di byte ognuno dei quali rappresenta un carattere, contiene cioè la codifica ASCII estesa di un carattere. Questi file in generale contengono un testo, per esempio un programma o un romanzo. Dato che ciascun carattere è rappresentato dalla sua codifica ASCII estesa, per leggere il testo contenuto in un file di testo, è necessario usare un programma che traduce ciascun byte nel corrispondente carattere scritto sullo schermo. Un esempio di un tale programma è un editore di testo.

I file **binari** sono ancora sequenze di byte (tutto è una sequenza di byte in un computer), ma questi byte non sono codifiche di caratteri, ma sono rappresentazioni interne al computer di valori, per esempio valori interi, reali eccetera. Ovviamente nel computer i valori di tipo carattere sono rappresentati sempre con il codice ASCII esteso e quindi un file di testo e binario che contengono valori di tipo carattere sono (fondamentalmente) uguali. Al contrario, se i valori contenuti in un file binario sono per esempio interi, se si cerca di leggerlo con un editore di testo, si ottiene sullo schermo un testo assolutamente incomprensibile. Nonostante che sequenza di 8 bit possa venire interpretata come il codice di un carattere secondo la codifica ASCII estesa, la sequenza dei caratteri ottenuti traducendo in questo modo un file binario, sarà senza senso.

Facciamo un semplice esempio per chiarire la differenza tra file di testo e file binari. Consideriamo il valore intero 8. In un file di testo il valore 8 viene rappresentato da un solo byte che contiene la codifica ASCII estesa del carattere '8', cioè 56 (per la precisione il valore binario 00111000), mentre in un file binario il valore 8 è rappresentato da 4 byte come ogni valore intero, cf. Tabella 2.1: i primi 3 composti da tutti O, mentre il quarto conterrebbe: `00001000`, cioè la codifica binaria di 8. Il valore -8 verrebbe rappresentato su un file di testo da 2 byte (uno che contiene la codifica ASCII estesa di '-' e l'altro quella di '8'), mentre in un file binario verrebbe rappresentato ancora da 4 byte che rappresentano in binario il valore 4294967288, che è la rappresentazione in complemento a 2 dell'intero -8, cioè 2^{32}-8.

Comunque ogni file è una sequenza di byte e, generalmente, la lettura di un file inizia sempre dal primo byte e successive letture leggono i byte successivi. Vedremo che è possibile leggere un byte alla volta e anche molti byte alla volta. Comunque, se, dopo alcune letture, si raggiunge l'end of file, significa che il contenuto del file è stato completamente letto. Questo modo di procedere si dice **sequenziale**. Esistono anche altre modalità di lettura non sequenziali (random) di cui non ci occuperemo. L'output inizia generalmente con un file vuoto e ogni operazione di scrittura sul file aggiunge byte in modo sequenziale, cioè successivi valori scritti sul file vengono appesi in coda a quelli scritti precedentemente In questo testo ci occuperemo solo di file di testo che sono letti e scritti in modo sequenziale. Sono più semplici e sono anche quelli usati più frequentemente.

3.1.2 Collegamento tra programma e file

Il collegamento tra un programma ed un file avviene associando al file un oggetto di tipo `stream`. Trattandosi di oggetti, nel seguito spiegheremo solo come usare gli `stream`, senza entrare nei dettagli. Un oggetto contiene dati e offre funzioni (dette **metodi**) per manipolare questi dati. Nel caso degli `stream` i metodi offerti sono le operazioni di i/o sui file. Per usare un file in un programma si devono inserire alcune istruzioni nel programma:

i) all'inizio del programma va inserita la direttiva `#include<fstream>`,

ii) inserendo, per esempio nel main, la dichiarazione,

`ifstream XX(''pippo'');`

si crea lo `stream XX` che viene associato al file pippo, il quale viene simultaneamente **aperto** per eseguire input, cioè i suoi dati sono ora a disposizione del nostro programma per operazioni sequenziali di lettura;

iii) per aprire il file `minni` in output, cioè per scriverci sopra, si deve inserire nel programma la dichiarazione: `ofstream YY (''minni'')`; che apre il file `minni`.

L'apertura di un file può fallire. Per esempio, la dichiarazione del punto (ii) fallirebbe se il file `pippo` aperto in input non fosse presente nella directory corrente. Ci possiamo accorgere del fallimento controllando il valore di `XX` dopo la sua dichiarazione. Infatti, in caso di fallimento, il valore di `XX` è 0. L'operazione di apertura può aprire file che si trovano in cartelle qualsiasi (anche diverse da quella corrente), specificando, tra le doppie di virgolette, il cammino per raggiungere il fi le dalla directory corrente.

Se il file le `minni` della dichiarazione del punto (iii) non fosse presente nella directory corrente, allora esso verrebbe automaticamente creato. Quindi l'apertura di un file in output difficilmente fallisce. Va tenuto ben presente però che, qualora il file `minni` esistesse già, la sua apertura in output causerebbe la cancellazione del suo contenuto. L'idea è che un file destinato a ricevere output debba essere inizialmente vuoto. Le modalità di apertura dei file che abbiamo appena presentato sono le più semplici ed esse consentono di aprire file (di testo) da usare in modo strettamente sequenziale. Altre modalità si possono trovare consultando le fonti già citate.

Quando un file ha esaurito la sua funzione, esso può venire chiuso. Se `XX` è lo `stream` associato al file da chiudere, allora XX.`close();` chiude il file associato a `XX`. Dal momento in cui questa operazione viene eseguita, `XX` non è più associato ad alcun file, anche se resta una variabile di tipo `fstream` che potrà di nuovo venire associata ad un file (lo stesso di prima o diverso), per esempio con l'istruzione `XX.open(''chi_vuoi'' );`.

Per rendere facili le operazioni di i/o che si riferiscono alla tastiera ed allo schermo, nel `namespace std`, il C++ associa ad essi 2 `stream` che quindi sono a disposizione dei programmatori. Lo `stream cin` è associato all'input standard (tastiera), mentre lo `stream cout` è associato all'output standard (schermo). Nell'Esempio 1.1, abbiamo usato questi 2 `stream`.

3.1.3 Operazioni di i/o

Gli stream sono degli oggetti e fanno quindi parte della parte orientata agli oggetti del C++. Ci limiteremo quindi ad usarli senza neppure cercare di spiegarli. Comunque una cosa dobbiamo dirla sugli oggetti: gli oggetti sono delle strutture in cui convivono dei dati e delle funzioni per manipolare questi dati. Nel gergo orientato agli oggetti le funzioni vengono chiamate **metodi**. Ovviamente i metodi definiti negli stream di i/o sono principalmente operazioni di i/o. Studieremo solo 2 metodi per l'input e 2 metodi per l'output.

- visto che consideriamo solo file di testo, cioè composti di sequenze di caratteri, è naturale leggere da questi file un carattere alla volta e scrivere su questi file un carattere alla volta. Se XX e YY sono, rispettivamente lo stream di input e quello di output, allora l'istruzione, `char c=XX.get()`, legge il prossimo carattere (dal file associato a `XX`). Questo carattere diventa l'R valore della variabile `c`. Quindi l'operazione di input cambia lo stato del calcolo per quanto riguarda la variabile letta (`c` nell'esempio). Il file è letto sequenzialmente, quindi la prima `get` legge il primo carattere del file, la seconda `get` legge il secondo carattere e così via. La lettura non cambia il contenuto del file. Quindi ad una successiva apertura si ritroverebbe il contenuto del file intatto. La scrittura di un carattere può venire effettuata con la seguente istruzione:`YY.put(c);` il valore di `c` (variabile di tipo `char`) viene appeso alla fine del file associato allo `stream YY`.

  Vale la pena di osservare bene la sintassi delle 2 operazioni appena viste. Per esempio, in `YY.put(c):YY` è l'oggetto di tipo `stream` e `put` è un metodo di questo oggetto. Il punto in `YY.put(c);` indica proprio l'appartenenza di `put` allo s `tream YY` e si chiama operatore di **appartenenza**. Lo stesso vale per `char c=XX.get()`.

- Generalmente i file di testo contengono sequenze di caratteri che rappresentano valori separati da spazi, per esempio sequenze di interi o di reali e sarebbe molto comodo riuscire a leggere (scrivere) un valore intero o reale con un'unica operazione di lettura (scrittura), senza leggere (scrivere) carattere per carattere. Gli `stream` di i/o offrono operazioni che consentono questa comodità. Esse sono le operazioni >> e << già viste nell'Esempio 1.1. Vediamo meglio come funzionano. 11 contenuto di un qualsiasi file di testo è una sequenza di stringhe di caratteri separate da caratteri di separazione (spazio e accapo), dove ogni stringa rappresenta un valore per esempio un numero intero oppure un reale. L'istruzione `XX >> x`; in cui `x` è una variabile intera, causa la lettura da `XX` della prossima stringa di caratteri numerici fino al primo separatore che si incontra. Se, anziché caratteri numerici, su `XX` si trovano altri caratteri, per esempio alfabetici, allora si verifica un errore che spesso causa la non terminazione della lettura. Da questo si può capire immediatamente che la comodità della lettura fatta con >>, causa una perdita di robustezza dei programmi, nel senso che si ottengono programmi incapaci di resistere a situazioni impreviste, come trovare caratteri inattesi su `XX`.

  C'è un'altra particolarità della >> che si deve osservare. Supponiamo che `x` e `y` siano variabili `int`. Se eseguiamo `XX >> x >> y;` quando il file associato a

`XX` contiene 12 34, allora, dopo la lettura, `x` avrà R-valore 12 e `y` 34. Tutto normale? Forse, ma occorre notare che il contenuto del file è in realtà la seguente sequenza di caratteri: `'1' '2' ' ' '3' '4'` e quindi la lettura ha saltato il carattere spazio ' ' il che è coerente col fatto che esso funge da separatore. Insomma con la >> non si possono leggere i caratteri di separazione contenuti nel file (spazi e accapo). Quanto appena descritto è vero anche se la lettura concerne una variabile di tipo `char`, come in, `char c; XX >> c;`. Anche in questo caso non vengono letti i caratteri di separazione (spazi e accapo). Un programma che mostra in modo chiaro questo fenomeno e lo contrasta con quello che succede con l'operazione `get`, è discusso negli Esercizi 3.5, 3.6 e 3.7.

Riconsideriamo l'esempio precedente. Se `x` è intera, la lettura, `XX >> x;` trasforma automaticamente i caratteri `'1'` e `'2'`, presenti sul file, nella rappresentazione interna dell'intero 12 (cioè con 4 byte in complemento a 2) e questo diventa l'R-valore di `x`. Questo significa che il tipo della variabile che viene letta (cioè `int` nell'esempio) determina quale sequenza W di caratteri ci deve essere sul file di input perché l'operazione di lettura riesca normalmente. La lettura che riesce trasforma W nella rappresentazione interna del valore rappresentato da W. Quindi se nel file, anziché `'1'` e `'2'` ci fosse `'1'` , `'.'` e `'2'`, allora la lettura avrebbe un comportamento anomalo (provare per credere) perché il tipo `int` non prevede un punto nei suoi valori. Se invece `x` avesse tipo `double` la lettura funzionerebbe normalmente e `x` assumerebbe l'R-valore `1.2` (rappresentato in modo floating point con 8 byte). Nel caso in cui `x` avesse tipo `double` ed il file contenesse `'1' '2' ' ' '3' '4'`, allora verrebbe calcolata la rappresentazione floating point su 8 byte di 12 ed assegnata come R-valore a `x`.

In conclusione, l'operazione di lettura >> si *fida* del tipo della variabile letta per sapere quali caratteri aspettarsi sul file (fragilità), e traduce la stringa di caratteri letta nella rappresentazione interna appropriata al tipo della variabile letta (comodità). Quindi l'operazione di lettura >> è comoda, ma fragile. Se le aspettative determinate dal tipo della variabile letta non si avverano, la lettura può avere risultati anomali. Da questo segue che ogni applicazione rivolta ad utenti qualsiasi non potrà mai effettuare letture con >>. Dovrà sempre leggere l'input come sequenza di caratteri e successivamente verificare che la sequenza letta corrisponda alle attese o no.

Anche l'operazione di output << esegue automaticamente una traduzione, ma in senso inverso rispetto a quella dell'input. Infatti, essa traduce l'R-valore della variabile da stampare (naturalmente si tratta della rappresentazione interna al computer del valore) nella sua rappresentazione esterna, cioè nella sequenza di caratteri che la rappresenta, ed è questa rappresentazione che viene scritta sul file. Ogni nuova operazione di output inserisce un nuovo valore dopo quelli stampati in precedenza senza lasciare spazi o accapo tra un valore ed il successivo. Se non si inseriscono esplicitamente separatori, si rischia di ottenere un file che sarebbe impossibile leggere successivamente.

Esercizio 3.1 *Realizzare un programma che crea un file di testo in output, scrive, (con >>), su questo file i valori interi 11, 32, 455 e 6, chiude il file, lo riapre in input e cerca di leggere dal file i 4 valori interi appena scritti. Per finire chiude il file definitivamente. Attenzione a separare i valori quando li scrivete!*

Come spiegato in precedenza, la lettura sequenziale di un file fa in modo che ad ogni operazione di lettura venga letto il prossimo byte (o gruppo di byte). Insomma è come se sul file ci fosse un segno che avanza ad ogni lettura per indicare il prossimo byte che verrà letto. Prima o poi questo segno arriverà alla fine del file, cioè all'end of file, e chiaramente, una volta raggiunto quel punto, successive letture non avrebbero senso. Segnaliamo che tali letture, benché erronee (in generale) non causano l'interruzione dell'esecuzione del programma, ma leggono valori sballati. Lo `stream` associato al file ci offre il metodo `eof()` (eof abbrevia end of file) che restituisce il booleano `true` solo quando la fine del file è stata raggiunta. Per un `ifstream XX` si controlla di aver raggiunto la fine del file con la seguente invocazione: `XX.eof();`. Si deve fare attenzione al fatto che le diverse operazioni di lettura introdotte in Sezione 3.1.3 possono presentare un comportamento diverso rispetto al raggiungimento dell'end of file. Questo fenomeno è illustrato nell'Esempio 3.5.

Sull'input/output ci sarebbero molte altre cose da dire. Per esempio gli `stream` di input ci offrono anche altre operazioni di input come la `getline` che legge i prossimi caratteri sullo `stream` fino al primo carattere di accapo. Inoltre ci sono anche operazioni di output formattato, in cui è possibile fissare il numero di spazi disponibili per la stampa di un dato valore. È interessante anche sapere che esiste la possibilità di definire file su cui è possibile sia leggere che scrivere dati e su cui queste operazioni non sono necessariamente sequenziali. Queste cose non ci serviranno nel seguito del libro. I lettori possono trovare informazioni su questi aspetti tecnici dell'input/output su internet, per esempio all'indirizzo: `www.cplusplus.com`.

3.2 Assegnazione

L'istruzione più semplice, ma comunque più importante, del C++ è **l'assegnazione**. L'assegnazione ha la forma, `x= (2*y) / (3*z);` in cui `x`, `y` e `z` sono variabili.
L'esecuzione di una tale assegnazione produce il seguente effetto. Supponiamo che lo stato del calcolo al momento precedente l'esecuzione dell'assegnazione sia **S**, allora viene calcolato il valore `v` di `(2*S(y)) / (3*S(z))` (si ricordi che `S(y)` è l'R-valore di `y` nello stato **S**) e `v` diventa l'R-valore di `x`. Quindi, dopo l'esecuzione di questa assegnazione, il nuovo stato del calcolo sarà **S'** che è uguale a **S** per ogni variabile diversa da `x` e uguale a `y` per `x`. Si osservi che tra le variabili che appaiono nell'espressione a destra dell'uguale di un'assegnazione può esserci anche la variabile che appare alla sinistra dell'uguale. Come per esempio in `x=(2*y)/(3*x);`. L'esecuzione di una tale assegnazione non ha nulla di diverso dalla precedente.

La descrizione di quanto avviene durante l'esecuzione di un'assegnazione spiega (finalmente) i nomi R- e L-valore delle variabili. Infatti abbiamo appena visto che per la valutazione dell'espressione a destra del segno di uguale si usano gli R-valori delle variabili che vi appaiono (e destra e Right in inglese), mentre il valore v di questa valutazione viene immagazzinato nella RAM all'indirizzo che è l'L-valore della variabile che appare alla sinistra dell'assegnazione (e sinistra è Left in inglese).

Il C++ consente di scrivere assegnazioni in forme abbreviate come, `potenza*= 2;` e `esponente++;`. Esse corrispondono alle seguenti assegnazioni normali: `potenza* = 2;` ⇒ `potenza = potenza*2;` e
`esponente++;` ⇒ `esponente = esponente + 1;`
Lo stile abbreviato di queste assegnazioni proviene dal C ed aveva originalmente il duplice scopo di permettere di scrivere codice conciso e anche di ottenere codice

oggetto più efficiente. In realtà i compilatori moderni non hanno bisogno delle forme abbreviate per ottimizzare il codice che producono e quindi la forma abbreviata è ora motivata solo dalla ricerca della concisione del codice e forse dall'abitudine. Nel seguito useremo spesso gli operatori `++` e $--$ sia in forma postfissa che prefissa, come per esempio in `esponente++;` e `++esponente;`. Entrambi questi comandi effettuano l'assegnazione `esponente = esponente + 1;` e non c'è alcuna differenza tra i 2 comandi se essi sono usati da soli, mentre c'è differenza se essi vengono usati all'interno di un'espressione più complessa. Consideriamo un semplice esempio: confrontiamo l'effetto di eseguire `int x=1; x=(x++)*2;` con quello di eseguire `int x=1; x=(++x)*2;`. Nel primo caso, dopo l'esecuzione, il valore di `x` è 3, mentre nel secondo caso il valore di `x` diventa 4. li motivo è che nel primo caso l'incremento richiesto da `x++`, viene eseguito dopo l'assegnazione `x=1*2` e quindi il valore finale di `x` è 3, mentre nel secondo caso l'incremento richiesto da `++x` viene fatto immediatamente e quindi l'assegnazione diventa `x = 2*2`.

3.3 Condizionale

La prossima istruzione da considerare è l'istruzione condizionale. Essa ha la forma, `if (EXP) C1 else C2;` dove `C1` e `C2` sono blocchi di istruzioni e `EXP` è un'espressione booleana, cioè un'espressione il cui valore è di tipo `bool`. In realtà, visto che il C++ *confonde* `bool` con `int` (con 0 equivalente a `false` e tutti gli altri valori interi equivalenti a `true`), `EXP` può avere anche un valore `int`. L'esecuzione di questa istruzione in un certo stato del calcolo **S** ha il seguente effetto: viene calcolato il valore dell'espressione booleana `EXP` nello stato del calcolo **S** e se questo valore è `true` (o un qualsiasi intero diverso da 0), allora viene eseguito il blocco di istruzioni `C1`, altrimenti viene eseguito `C2`. In entrambi i casi, dopo l'esecuzione di `C1` o `C2` il calcolo continua con l'istruzione che segue il condizionale. La Figura 3.1 contiene il diagramma a blocchi del condizionale nella sua forma completa con entrambi i rami `C1`, detto tradizionalmente il ramo `then`, e `C2`, detto il ramo `else` ed anche nella forma con il solo ramo `then`.

3.4 Cicli while

Ogni linguaggio di programmazione che si rispetti deve possedere un'istruzione iterativa che permette di ripetere un certo blocco di istruzioni per tutto il tempo che una data condizione è verificata. Senza una tale istruzione iterativa (o qualcosa di analogo) un linguaggio è incapace di realizzare calcoli realmente interessanti.

L'istruzione iterativa più semplice del C++ è il `while` che ha la seguente forma: `while (EXP) C;`, in cui `EXP` rappresenta un'espressione booleana (o interpretabile come booleana) e `C` è un blocco di istruzioni. `EXP` è detta la **condizione** di permanenza nel ciclo, mentre `C` è il **corpo** del `while`.

L'effetto di eseguire questa istruzione in uno stato del calcolo **S** è come segue: viene calcolato il valore dell'espressione booleana `EXP` nello stato del calcolo **S** e se esso ha valore `true` (o intero diverso da 0) allora viene eseguito il blocco `C`. L'esecuzione del corpo generalmente cambia lo stato da **S** in **S'** ed in questo nuovo stato si torna ad eseguire `while (EXP) C;`. Se in **S'** `EXP` è ancora `true` allora si esegue di nuovo il corpo e così via. Prima o poi (sperabilmente) si raggiunge

uno stato del calcolo **S"** tale che in questo stato `EXP` abbia valore `false` e allora l'esecuzione del `while (EXP) C;` termina ed il calcolo continua dall'istruzione che segue immediatamente il corpo del `false`. Lo stato del calcolo alla fine del ciclo sarà **S"**.

Può capitare che `EXP` non diventi mai `false`. In questo caso (a meno di istruzioni di salto contenute nel corpo) l'esecuzione non esce mai dal `false` e questo fenomeno viene chiamato un **ciclo infinito**. Ovviamente quando questo succede siamo in presenza di un programma errato. Il diagramma a blocchi del `false` è in Figura 3.2. Nella stessa Figura 3.2 viene anche illustrata la variante `do-while` del `while` che esegue il corpo sempre almeno una volta visto che valuta l'espressione `EXP` solo dopo questa prima esecuzione del corpo.

3.5 Esempi

Illustriamo ora le nuove istruzioni appena introdotte realizzando alcuni semplici programmi. Iniziamo modificando l'Esempio 1.1.

Esercizio Risolto 3.2 *Dopo aver letto 2 valori dallo `stream` di input `cin` ed averli assegnati alle variabili `x` e `y`, vogliamo sommare i due valori ed assegnare questo valore ad una nuova variabile intera `SOM` e per ultimo, usando l'istruzione condizionale, se il valore di `SOM` è maggiore di 0 vogliamo assegnare il valore di `SOM` ad `y`, altrimenti, (cioè se `SOM` $\leq$ 0), vogliamo assegnare `SOM` a `x`. Il programma che compie queste azioni segue. Esso ci mostra alcune cose interessanti. Per esempio che le dichiarazioni possono essere in qualsiasi punto di un blocco, cf. la dichiarazione di `SOM`. Inoltre il programma illustra l'uso del condizionale che ci consente di fare cose diverse a seconda dello stato del calcolo in cui ci troviamo. Per esempio assumiamo che i valori letti per `x` e `y` siano 2 e -4. Allora `SOM` assume il valore 2-4=-2 e quindi la condizione `SOM>0` in questo stato del calcolo è falsa e quindi viene eseguito il ramo `else` del condizionale cioè l'assegnazione `x=SOM`. Per cui lo stato finale **S** è come segue: `S(x)=-2, S(y)=-4 e S(SOM)=-2`. Se invece di 2 e -4 dallo `stream` di input vengono letti i valori 5 e -4 le cose sarebbero andate diversamente? In che modo?*

```
1  #include<iostream>
2  using namespace std;
3  main() {
4    int x, y;
5    cout << "inserire 2 interi";
6    cin >> x >> y;
7    cout<<"valore di X="<< x <<"valore di y="<< y;
8    int SOM;
9    SOM= x+y;
10   if (SOM > 0)
11     y=SOM;
12   else
13     x=SOM;
14   cout<<"valore di x="<< x <<"valore di y="<< y;
15 }
```

Esercizio Risolto 3.3 *Consideriamo ora di voler leggere degli interi dall'input standard fino a che non si legge il valore 0 e dei valori letti si vuole calcolare quanti sono positivi e quanti negativi (quindi lo 0 non viene contato). Benché l'esercizio*

sia semplice proponiamo due diverse soluzioni, una che usa un ciclo `while` *ed una seconda con un* `do-while`*, vedi Figura 3.2. La prima soluzione è la seguente:*

```
using namespace std;
main() {
  int pos=0, neg=0, x; cin>>x;
  while (x != 0) {
    if(x < 0)
      neg=neg+1;
    else
      pos=pos+1;
    cin>>x;
  }
  cout<<"neg="<< neg <<" pos="<<pos<<endl;
}
```

Un punto da notare è che `neg` *e* `pos` *vanno inizializzati a 0 prima di essere usati, altrimenti otterremmo dei valori casuali o comunque sbagliati. Si noti anche che la prima lettura viene fatta prima di entrare nel ciclo in modo da avere la variabile* `x` *pronta per il test di permanenza nel ciclo. Le successive letture sono eseguite alla fine del corpo del ciclo cioè immediatamente prima di tornare alla valutazione del test di permanenza. Il fatto che serva una lettura fuori dal ciclo ed una nel ciclo, indica che probabilmente il problema può essere risolto in maniera più naturale con un ciclo* `while-do` *che permette di fare anche la prima lettura nel corpo del ciclo visto che il test di permanenza è alla fine del ciclo stesso. Questa soluzione alternativa segue.*

```
using namespace std;
main() {
  int pos=0, neg=0, x; cin>>x;
  do {
   cin>>x;
   if (x < 0)
     neg=neg+1;
   else
     if(x>0)
       pos=pos+1;
  } while(x!=0);
  cout<<"neg="<< neg <<" pos="<<pos<<endl;
}
```

Come previsto questa soluzione necessita di una sola lettura anziché due come nella soluzione precedente. Però questa semplificazione la si paga con la necessità di un condizionale più complicato nel corpo del ciclo. Infatti, dopo la lettura, `x` *potrebbe essere 0 nel qual caso non dobbiamo cambiare né* `neg` *né* `pos` *e, per garantire questa condizione, siamo obbligati ad avere il test* `x>0` *nel ramo* `else` *in modo da escludere il caso* `x=0`*. Forse per questo semplice problema si può pensare che sia ovvio che le nostre 2 soluzioni siano giuste, ma in generale non è cosi e ci sarà bisogno di ragionare per arrivare ad una soddisfacente convinzione della correttezza dei nostri programmi. Nel prossimo Capitolo, questo esempio ci servirà da apripista per mostrare come questi ragionamenti possono venire organizzati.*

Esercizio Risolto 3.4 *Si vuole realizzare un programma che legge 10 interi dall'input standard, memorizza il minimo e il massimo tra i valori letti e alla fine li stampa sull'output standard li programma deve avere un ciclo* `while` *che esegue 10 volte e che dopo aver letto il prossimo intero lo confronta con il massimo ed il minimo*

tra i valori letti in precedenza e li modifica a seconda dei casi. Chiamiamo le due variabili i cui R-valori sono, rispettivamente, il massimo ed il minimo tra i valori letti in precedenza, ***max*** *e* ***min****. Sembra semplice, ma c'è un problema. Alla prima esecuzione del* ***while****, non abbiamo ancora letto alcun valore e quindi che valore dovranno avere* ***max*** *e* ***min****? Questo problema si presenterà spesso in altri esercizi proposti nel seguito. Lo affronteremo in modo completo subito. Possiamo assegnare a* ***max*** *e* ***min*** *dei valori ad hoc che fanno funzionare le cose, oppure leggere il primo valore (sappiamo che c'è visto che sono previste 10 letture) e assegnarlo sia a* ***max*** *che a min e dopo leggere gli altri 9 valori. Pensiamo prima alla soluzione con i valori ad hoc. Per* ***max*** *abbiamo bisogno di un valore che sarà certamente minore o uguale di ogni valore intero che potremo leggere. Questo valore è ovviamente* INT_MIN*: il minimo intero rappresentabile. Analogamente, per min, il valore giusto è* INT_MAX*. Vediamo un programma che segue questa idea.*

```
1  using namespace std;
2  main() {
3    int x, i=0, max=INT_MIN, min=INT_MAX;
4    while (i<10) {
5      cout << "inserire il prossimo intero";
6      cin >> x;
7      if(x > max)
8        max=x;
9      else
10       if(min>x)
11         min=x;
12     i=i+1;
13   }
14   cout << "il valore massimo e' " << max << ", il valore minimo e' " << min;
15 }
```

Questo programma non fa quello che vogliamo. Basta infatti considerare il caso in cui i valori letti siano, 1, 2, ... , 10, per vedere che alla fine ***max=10****, il che è giusto, ma che* ***min*** *sarà rimasto tristemente* INT_MAX*! Certo se i 10 valori non fossero sempre crescenti, allora il programma funzionerebbe, ma non possiamo accontentarci di un programma che è quasi sempre corretto! Possiamo correggerlo? Certamente, ma non è possibile considerare* ***min*** *solo quando* ***max*** *non cambia. Perché i valori ad hoc con cui li abbiamo inizializzati vanno sempre cambiati, quindi dobbiamo essere sicuri che ciascuna delle due variabili sia confrontata almeno una volta con uno dei valori letti. Un programma che corregge l'errore segue.*

```
1  #include<iostream>
2  using namespace std;
3  main() {
4    int x, i=0, max=INT_MIN, min=INT_MAX;
5    while(i<10) {
6      cout << "inserire il prossimo intero";
7      cin >> x;
8      if(x > max) {
9        max=x;
10       if (i==0)
11         min=x;
12     }
13     else
14       if(min>x) {
15         min=x;
16         if(i==0) //($) questo commento e' usato nel
```

```
17          max=x; //($) primo esercizio del prossimo Capitolo
18        }
19      i=i+1;
20    }
21    cout << "il valore massimo e' " << max << ", il valore minimo e' " << min;
22  }
```