Programmazione consapevole (semplice è bello)

Andrew Sadi
Sara Righetto [AkaiSara]

15 agosto 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

Questo libro è rivolto a studenti che sanno poco o nulla di programmazione e che intendono impararla in modo approfondito. Il suo scopo non è solamente quello di illustrare un particolare linguaggio di programmazione attraverso una serie di esempi, ma anche quello di proporre un modo *consapevole* di scrivere programmi che aumenti la probabilità che siano subito corretti. Pensiamo che un buon informatico debba sapere cosa succede quando i suoi programmi sono compilati ed eseguiti e quindi il libro tratta anche alcuni concetti relativi ai linguaggi di programmazione che raramente sono trattati in un testo introduttivo. Tra questi concetti ricordiamo i tipi, la gestione dei dati durante l'esecuzione dei programmi, la gestione delle eccezioni, e la gestione dell'esecuzione delle funzioni (anche ricorsive). L'importanza di questi concetti va ben al di là del singolo linguaggio (che sia C o C++), infatti essi si applicano praticamente a tutti i linguaggi di programmazione esistenti.

Per quanto riguarda il modo di sviluppare programmi, l'idea che permea questo testo è che più un programma è semplice, più è probabile che non contenga errori. Non a caso il sottotitolo del libro è *semplice è bello*. La convinzione dell'autore (e che pare difficile confutare) è che: *non si può scrivere un programma e credere che sia corretto se non si sa dire chiaramente cosa il programma dovrebbe fare e perché lo fa*. In coerenza con questo principio, ogni programma che realizzeremo sarà accompagnato da una descrizione precisa di quello che vorremmo che calcolasse e da una dimostrazione che effettivamente lo calcola. Queste dimostrazioni sono comprensibili e quindi utili solo quando, nello scrivere i programmi, avremo seguito una logica il più possibile semplice e lineare. Insomma, il motto *semplice è bello* è una scelta obbligata se vogliamo dimostrare la correttezza dei nostri programmi.

Il linguaggio di programmazione adottato nel testo è il C++ senza le classi e gli oggetti. Non si tratta del C, infatti anche senza le classi e gli oggetti, il C++ ha alcune caratteristiche che lo rendono più espressivo del suo predecessore C, come per esempio il passaggio dei parametri per riferimento, e la gestione delle eccezioni. Per non ripetere continuamente il*C++ senza le classi e gli oggetti*, nel seguito chiameremo questo linguaggio semplicemente C++. Per quanto riguarda caratteristiche del C++ che non hanno trovato posto in questo libro, esistono molti manuali di riferimento del linguaggio, alcuni dei quali sono disponibili gratuitamente su internet. Anche il sito (`www.cplusplus.com`) è ricco di informazioni.

Il libro è organizzato in 12 Capitoli. I Capitoli 2, 3, 5, 7 e 9 trattano il linguaggio di programmazione. Ovviamente nei primi Capitoli sono illustrate le cose più semplici come i tipi predefiniti e le istruzioni di base, per passare successivamente ai tipi definiti dall'utente e ai contenitori, per finire con il sovraccaricamento e i

`namespace`. Questi Capitoli sono intercalati da Capitoli con molti esempi ognuno corredato dalla relativa dimostrazione di correttezza. Il Capitolo 4 introduce le prime dimostrazioni di correttezza, ma il Capitolo 6 riveste un ruolo particolarmente importante in quanto in esso viene formulata una semplice *ricetta*, detta *ricetta di* **indicizzazione**, che è molto utile in tutte le dimostrazioni successive. La ricorsione è introdotta nel Capitolo 10 e approfondita nei successivi Capitoli 11 e 12 che illustrano le liste concatenate e gli alberi binari. Questi Capitoli contengono molti esempi di funzioni ricorsive la cui correttezza è dimostrata con l'induzione.

In quello che resta del presente Capitolo verranno introdotte alcune nozioni molto semplici, ma comunque molto utili nel seguito del testo, su come è organizzato un Computer, su come scrivere un programma C++ e su come eseguirlo. Il Capitolo si conclude con la Sezione 1.3 in cui viene spiegata la notazione logico-matematica che è usata costantemente nel resto del libro.

1.1 Struttura di un computer

L'architettura di un computer moderno è sorprendentemente simile a quella dei primi computer costruiti negli anni quaranta. Questa architettura viene chiamata di von Neumann in onore dell'ideatore dei primi computer. Naturalmente la tecnologia di realizzazione dei computer si è enormemente evoluta nel corso di questi 70 anni, e questa evoluzione ha portato un'incredibile riduzione nelle dimensioni dei computer ed un altrettanto incredibile aumento della loro velocità di esecuzione, ma le funzionalità delle diverse parti e la loro cooperazione sono, in linea di principio, ancora simili a quelle dei tempi di von Neumann.

Un computer contiene un'unità di calcolo ed una memoria. L'unità di calcolo è detta **Central Processing Unit** (CPU) e consiste di registri capaci di contenere i valori da manipolare durante l'elaborazione e da circuiti in grado di effettuare operazioni (per esempio, la somma o la sottrazione) sui valori contenuti nei registri e anche di trasferire valori dalla memoria ai registri e viceversa. Le operazioni che questi circuiti sono in grado di effettuare sono dette **operazioni macchina**. La memoria è costituita da una sequenza di byte, ciascuno composto da 8 bit e identificato da un indirizzo. Gli indirizzi dei byte della memoria iniziano da 0 e crescono fino alla dimensione totale della memoria stessa. Un valore, per esempio un valore intero, può occupare più byte contigui e in questo caso diremo che il valore risiede nella memoria all'indirizzo del primo di questi byte. Abbiamo già detto che la CPU è in grado di eseguire operazioni che trasportano valori dalla memoria RAM ad un registro della CPU e viceversa. Queste operazioni richiedono di specificare l'indirizzo del primo byte da cui si preleva (o in cui viene copiato) il valore trasportato. Dato che tramite gli indirizzi è possibile accedere ad un qualsiasi byte della memoria, questa memoria è chiamata **Random Acccess Memory (RAM)**, cioè Memoria ad Accesso Casuale. Per quanto riguarda l'esecuzione dei programmi da parte di un computer, essa avviene in linea di massima nel modo seguente: il programma da eseguire ed i valori che esso manipola durante l'esecuzione, si trovano entrambi nella RAM (ma in parti diverse). Uno dei registri della CPU, detto **Program Counter** (PC) contiene in ogni momento l'indirizzo della prossima istruzione del programma da eseguire e la CPU *riconosce* quell'istruzione, la esegue e passa alla istruzione successiva incrementando il PC. L'esecuzione di un'istruzione in generale modifica i valori dei dati del programma.

Ovviamente questa descrizione è molto semplificata, ma sufficiente per capire i concetti relativi alla programmazione che seguiranno.

1.2 Come scrivere ed eseguire un programma

Ogni programma, scritto in un qualsiasi linguaggio di programmazione (come Pascal o C++ o Java, ecc.), per poter essere eseguito deve venire tradotto in una sequenza di operazioni macchina. Questa traduzione viene effettuata da un programma che si chiama **compilatore**. Il programma di partenza (in Pascal o C++ ecc.) viene chiamato **programma sorgente**, mentre la corrispondente traduzione in linguaggio macchina si chiama **programma oggetto**[1].

La traduzione del programma sorgente in programma oggetto ha successo solo se il sorgente è scritto seguendo alla lettera le regole sintattiche del linguaggio di programmazione adottato. Queste regole sono molto precise in quanto esse devono eliminare qualsiasi ambiguità nel significato dei programmi. Per esempio nel C++ ogni istruzione deve terminare con un ';' ed i blocchi con più di una istruzione devono essere racchiusi tra parentesi graffe. Violazioni alle regole sintattiche vengono segnalate dal compilatore con messaggi che generalmente consentono di correggere facilmente gli errori.

Per eseguire i programmi presentati in questo libro, basterà avere sul proprio computer un compilatore C++ possibilmente conforme al C++ standard. Consigliamo il compilatore GNU C++ 4.1 (e qualunque versione successiva). Si tratta di software libero che è facile trovare e scaricare dalla rete sul proprio computer (`gcc.gnu.org`). Questo compilatore è realizzato per funzionare con il sistema operativo Linux, ma per i sistemi operativi Microsoft è possibile scaricare l'ambiente **Cygwin** che simula un ambiente Linux e possiede al suo interno un compilatore C++ della GNU. Anche Cygwin è software libero (`Cygwin.com`). Nel seguito assumeremo che i nostri lettori siano in grado di aprire una finestra di comando (detta **shell**), da cui potranno spostarsi nelle diverse cartelle del file system che contengono i loro programmi e da cui potranno lanciare i comandi di compilazione e di esecuzione dei programmi, che descriveremo dopo il seguente esempio.

Esempio 1.1

```
1  #include<iostream>
2  using namespace std;
3  main()
4  {
5    int x, y;
6    cout << "Inserire 2 interi";
7    cin >> x >> y;
8    cout << "Valore di x = " << x << ", valore di y = " << y;
9  }
```

Esercizio 1.2 *Scrivere il programma dell'Esempio 1.1 usando un qualsiasi editore di testo (si consiglia* `xemacs`*), compilarlo ed eseguirlo utilizzando i comandi spiegati di seguito.*

[1] In realtà la traduzione dal linguaggio sorgente a quello macchina può consistere di vari passaggi che coinvolgono linguaggi intermedi, macchine virtuali e fasi di interpretazione, ma questi aspetti esulano dallo scopo del presente testo.

Assumiamo che il file di testo contenente il programma dell'Esempio 1.1 si chiami `pippo.cpp` e che risieda nella cartella `pluto`. Useremo spesso nomi come `pippo` e `pluto` per indicare che si tratta di nomi arbitrari. Al contrario, l'estensione `.cpp` in `pippo.cpp` è necessaria perché indica al compilatore che il file contiene un programma C++. Se la cartella corrente è `pluto`, il comando `g++ pippo.cpp`, invoca il compilatore C++ che compila il programma `pippo.cpp` e produce il corrispondente programma oggetto nel file `a.out` o `a.exe`, a seconda che il nostro computer abbia un sistema operativo Linux oppure Windows, rispettivamente. Una volta effettuata la compilazione, il comando `./ a.out` (oppure `./ a.exe`) esegue il codice oggetto. Qualora si desideri attribuire un nome meno anonimo di `a.out` o `a.exe` al file oggetto, basterà lanciare la compilazione con il comando seguente: `g++ -o paperino pippo.cpp`. Questo comando metterà il codice oggetto nel file `paperino`, che verrà eseguito con `./ paperino`. Qualunque sia il nome del codice oggetto, la sua esecuzione causerà l'apparizione sul video della scritta, `Inserire 2 interi`, con il cursore immediatamente dopo che lampeggia in attesa dei 2 interi richiesti. Una volta inserita da tastiera una coppia di interi, per esempio `3` e `-2` (separati da uno spazio o da un invio e seguiti da un invio), sul video apparirà la scritta `Valore di x = 3` seguita da `, valore di y = -2`. Il programma è tale che l'input e l'output, entrambi sul video, siano accostati in modo poco curato. Negli esempi successivi vedremo alcuni semplici accorgimenti per realizzare output di qualità migliore. Nonostante la sua semplicità, ci sono parecchie cose da spiegare nel programma dell'Esempio 1.1:

1. La prima istruzione `#include<iostream>` non è un'istruzione C++ bensì un comando, rivolto al compilatore, che richiede l'inclusione della classe di input/output `iostream` nel `namespace std`. La successiva istruzione `using namespace std;` permette al nostro programma di accedere a tutto quello che è definito nel `namespace std` e quindi anche alla classe `iostream`. Usare questa classe è necessario perché essa contiene le definizioni delle operazioni di input e di output che usiamo nel nostro programma e quindi se la classe `iostream` non fosse presente le operazioni di input e di output risulterebbero indefinite. Tutto questo sembrerà certamente *ostrogoto* al neofita. Per il momento si tratta di usare queste istruzioni senza capirle. La successiva trattazione dovrebbe renderle più comprensibili.

2. `main()` è il nome di una **funzione** e quello che segue, racchiuso tra parentesi graffe, è il **corpo** della funzione, cioè le istruzioni che la compongono. Nel seguito una sequenza di istruzioni racchiuse tra graffe viene chiamato un **blocco**. In inglese `main` significa *principale*. Ogni programma deve contenere esattamente una funzione `main` e l'esecuzione di ogni programma inizia sempre dalla prima istruzione della funzione `main`. Le parentesi tonde `()` che seguono il nome `main` indicano che si tratta di una funzione e che (in questo caso) non ha argomenti. In realtà il C++ consente di passare parametri al `main`, ma è decisamente troppo presto per questo genere di dettagli tecnici. Rinviamo il lettore inguaribilmente curioso al sito `www.cplusplus.com` per maggiori informazioni su questo.

3. Nel C++ l'input e l'output sono visti come sequenze di byte (come la memoria RAM). In inglese sequenza si dice "stream" ed infatti i nomi legati all'input ed all'output sono variabili di tipo `stream`. L'operazione di output più comune è

<<. Il dispositivo standard di output è il video ed il nome dello `stream` che gli viene associato nella libreria `iostream` è `cout`. Per l'input, l'operazione base è >>, il dispositivo standard di input è la tastiera ed il nome dello `stream` associato è `cin`.

4. Nel programma che stiamo considerando usiamo sia l'operazione di scrittura << che quella di lettura >>. Queste operazioni hanno entrambe 2 argomenti: lo `stream cout` o `cin` ed il valore da mandare in output oppure, rispettivamente, la variabile a cui assegnare il valore letto da `cin`. Un esempio di un comando di output è la seconda riga del corpo del `main`: questa istruzione (quando viene eseguita) produce la stampa su video del valore stringa di caratteri `"Inserire 2 interi"`. Nella terza riga del `main` troviamo invece un esempio di un'istruzione di lettura. Questa istruzione va vista in realtà come `(cin » x) » y;` in cui la prima parte `(cin » x)` legge il prossimo valore dallo `stream` di input `cin`, lo assegna alla variabile `x` e restituisce `cin` in modo da consentire la seconda lettura, `(cin » y)`, che assegna il prossimo valore letto a `y` e restituisce `cin` che però non serve a nulla dato che le letture sono finite. Per finire, la quarta istruzione del corpo del `main` è un output che stampa di seguito le stringhe `"Valore di x = "` e `", valore di y = "`, ciascuna seguita dal corrispondente valore (appena letto) di `x` e di `y`.

1.3 Notazione

Per descrivere in modo formale i concetti ed in particolare per scrivere le asserzioni di correttezza dei programmi, useremo una notazione logico-matematica che conviene spiegare una volta per tutte in questa Sezione.

Principalmente useremo le operazioni logiche di congiunzione (e), di disgiunzione (o), di negazione, d'implicazione e di equivalenza. Per completezza, ricordiamo che la congiunzione di 2 affermazioni (A e B) è vera quando entrambe sono vere, mentre la loro disgiunzione (A o B) è vera se almeno una tra A e B è vera. La negazione di un'affermazione è vera se l'affermazione è falsa. Un'affermazione A implica B, quando B è vera ogni volta che A è vera. L'equivalenza di 2 affermazioni è lo stesso che richiedere che ciascuna delle 2 implichi l'altra e quindi 2 affermazioni sono equivalenti se sono sempre entrambe vere o entrambe false.

I simboli che verranno usati per rappresentare queste operazioni sono vari:

- La congiunzione è rappresentata spesso da && (come nel C++), ma spesso viene usata semplicemente la virgola per evitare di appesantire troppo le formule. Quindi verrà scritto (A && B) oppure (A, B).
- La disgiunzione viene rappresentata con la doppia sbarra verticale ‖, come nel C++, e anche per la negazione useremo il punto esclamativo ! come nel C++. Quindi scriveremo A ‖ B e !A.
- Si rappresenta l'implicazione con $\Rightarrow$, mentre l'equivalenza la si rappresenta con $\Leftrightarrow$ (è la doppia implicazione!), ma anche con **se e solo se**, spesso abbreviato in **sse**.

Avremo bisogno di considerare insiemi S di oggetti e di affermare che tutti gli elementi di S soddisfano una certa proprietà $P(\cdot)$. Questa affermazione è descritta

dalla seguente formula: $\forall a \in S$,*P(a)*. Invece $\exists a \in S$,*P(a)* significa che esiste (almeno) un $a \in S$ tale che valga *P(a)*. Vale la pena di osservare che qualora $S = \emptyset$, allora $\forall a \in S$,*P(a)* è vero perché richiedere che per tutti gli elementi di S (cioè nessuno) valga *P(·)* è banalmente vero. Al contrario, se $S = \emptyset$, allora $\exists a \in S$,*P(a)* è, altrettanto banalmente, falso perché essa richiede che ci sia qualche elemento $a \in S$ per cui *P(a)* sia vero.

Spesso verranno considerati gli insiemi composti da tutti gli interi tra 2 estremi i e j. Un tale insieme è rappresentato con *[i..j]*. Qualora $i > j$, *[i..j]* rappresenta l'insieme vuoto. In molti casi *[i..j]* rappresenta indici di un array A e con *A[i..j]* si indica la sequenza degli elementi di A dalla posizione i alla j.
Se *[i..j]* è tale che $i > j$, allora *A[i..j]* rappresenta la sequenza vuota di elementi dell'array A.

Capitolo 2

Tipi predefiniti e variabili

Come (quasi) tutti i linguaggi di programmazione, anche il C++ *offre* dei tipi belli e pronti per l'uso. Si chiamano tipi **predefiniti** e naturalmente servono a rappresentare valori praticamente onnipresenti nei problemi che si affrontano con la programmazione e cioè gli interi, i reali, i caratteri, e i booleani. Cercheremo di presentarli in modo particolarmente compatto e mettendo in luce immediatamente i problemi che nascono dalla coesistenza nei nostri programmi di valori di tipo diverso e che di conseguenza sono rappresentati nella RAM in modo diverso. In particolare, discuteremo di conversioni tra valori di tipi diversi e accenneremo anche al concetto di *sovraccaricamento o overloading* degli operatori. Introdurremo anche le **variabili**, che in C++ devono sempre avere un tipo, ed inizieremo ad illustrare la nozione di **visibilità** o **scope** delle variabili stesse. Chiuderemo il capitolo illustrando l'importanza che i tipi hanno per l'individuazione degli errori che sono spesso presenti nei programmi che vengono scritti.

2.1 Tipi predefiniti

I tipi predefiniti del C++ sono i seguenti: il tipo intero, due tipi per i reali, un tipo carattere, il tipo booleano ed il tipo `void`. La Tabella 2.1 contiene alcune utili informazioni su questi tipi. La colonna `BYTE` di questa tabella specifica quanti byte vengono usati dal compilatore GNU C++4.1 per rappresentare nella RAM i valori dei diversi tipi. Questa quantità è importante perché essa fissa l'insieme dei valori supportati per ciascun tipo. È importante chiarire subito che, benché, per esempio, l'insieme degli interi sia infinito in matematica, gli interi che un computer supporta saranno sempre in numero finito, visto che ogni computer ha una memoria finita.

- Per il tipo intero, l'intervallo dei valori rappresentabili con 4 byte (dal compilatore GNU 4.1) è, -2^{31} , ... , 2^{31} - 1. Le tipiche operazioni applicabili ai valori interi sono quelle aritmetiche (+ - / * e % che rappresenta il modulo). Oltre a queste sono applicabili ai valori interi anche le operazioni di confronto, cioè il test di uguaglianza ==, il test di disuguaglianza ! =, il test di maggiore >, il test di maggiore o uguale >=, eccetera. L'ambiente C++ fornisce 2 costanti `INT_MAX` e `INT_MIN` che hanno come valore, rispettivamente, il massimo ed il minimo intero rappresentabile. Useremo spesso queste costanti per inizializzare variabili intere con un valore che sia sicuramente il più grande o il più piccolo possibile. Visto che gli interi rappresentati nel computer sono finiti, è naturale chiedersi cosa succede se sommando o moltiplicando 2 interi

si esce dall'intervallo dei valori rappresentabili. In questo caso diremo che si è verificato un errore di **overflow** ed il risultato dell'operazione è privo di senso. Purtroppo l'overflow non viene, in generale, segnalato automaticamente dal computer.

- Per i reali il C++ offre 2 tipi: `double` e `float`. Il tipo `double` che occupa 8 byte offre maggiore precisione del `float` per cui si usano solo 4 byte. Una costante reale come `12.3` viene considerata dal compilatore di tipo `double`. È necessario aggiungere `f` alla fine per renderla `float`: `12.3f`. La scrittura 2.3^4 indica il numero `double` $2.3*10^4$.

- Le operazioni aritmetiche e quelle di confronto si possono applicare sia a valori `int`, che `float`, che `double`. Chiariamo immediatamente che non si tratta delle stesse operazioni, ma solo di simboli uguali per indicare operazioni diverse. La diversità è causata dalla diversa rappresentazione interna degli interi (complemento a 2) rispetto a quella dei reali (floating point con 4 o 8 byte). Questo fenomeno di indicare con lo stesso simbolo operazioni simili concettualmente, ma che sono diverse a causa del tipo degli operandi, si chiama **sovraccaricamento** o **overloading**. Vedremo nel seguito del capitolo come il compilatore decide qual'è l'operazione che deve venire effettivamente usata a fronte di un'operazione sovraccaricata nel programma che sta traducendo.

- L'insieme di valori del tipo `char` è costituito da 256 caratteri. Il compilatore C++ 4.1 della GNU usa un byte per rappresentare questi 256 valori ed infatti con otto bit si rappresentano 256 valori diversi che visti come interi in complemento a 2 sono gli interi da -128 a 127. Quindi ogni carattere è rappresentato nella RAM da uno di questi interi. I caratteri con codifica da O a 127 sono i caratteri ASCII e sono i caratteri alfabetici (minuscoli e maiuscoli), i numeri, i segni di punteggiatura, le parentesi e diversi caratteri che servono a controllare il cursore come il carattere di tabulazione e quello di invio. I caratteri con codifica negativa dipendono invece dal sistema operativo che si utilizza. In generale tra questi sono presenti molti caratteri accentati, il segno di insieme vuoto ed altri caratteri utili. L'insieme dei 256 caratteri viene chiamato **extended ASCII**. In `www.cplusplus.com` si possono trovare informazioni più precise.

  Visto che la rappresentazione interna dei caratteri è fatta attraverso interi(sebbene di 1 solo byte), ai caratteri si possono applicare le operazioni aritmetiche. Naturalmente, in questo modo è molto facile ottenere risultati che non rappresentano più caratteri, cioè che sono fuori dall'intervallo -128... 27. Come per gli errori di overflow è il programmatore che deve fare attenzione a questi possibili errori.

  I valori di tipo carattere si rappresentano nei programmi tra apici, come per esempio in `'v'` , `'?'` e `'!'`, cf. la Tabella 2.1. Ci sono caratteri che non sono visibili sullo schermo, ma hanno la funzione di spostare il cursore, per esempio, il carattere `'\n'` serve per spostare il cursore all'inizio della prossima riga del video, mentre `'\t'` muove il cursore al prossimo punto di tabulazione. Con la crescente importanza di paesi che adottano alfabeti diversi dal nostro, sono nati tipi in grado di rappresentare insiemi di caratteri più ampi di 256. A questo scopo il C++ prevede il tipo `wchar_t` i cui valori occupano 2 o 4 byte.

- Il tipo booleano, denotato `bool`, consiste solamente di 2 valori, `true` e `false`. Ai valori di tipo `bool` si applicano gli operatori booleani che sono la congiunzione (AND), rappresentata con `&&`, la disgiunzione (OR), rappresentata con `||` e la negazione (NOT), rappresentata con `!`. I valori booleani sono rappresentati internamente al computer con gli interi 0 (`false`) e 1 (`true`). Il fatto che i valori del tipo `bool` siano ripresentata con interi è un'eredità del C. Visto che il C++ è compatibile col C, la stessa convenzione si applica anche al C++. Come per i caratteri, il fatto che i valori booleani siano rappresentati internamente con interi, permette di applicare ai valori booleani tutte le operazioni applicabili agli interi. Per esempio, `true - true == false` è un'espressione corretta del C++ ed ha valore `true` (con rappresentazione interna 1). Visto che i valori booleani sono in realtà interi, anche la seguente espressione, `2 && o || -4`, è un'espressione valida in C++ ed ha valore `true`.

- Il tipo `void` non ha valori né operazioni. Esso viene usato proprio per indicare la mancanza di valori. Vedremo il suo uso nelle funzioni che non restituiscono alcun risultato, cf. Sezione 7.3.

In generale, per sapere quanti byte occupano i valori di un qualsiasi tipo `T` possiamo utilizzare la funzione di libreria `sizeof`. L'istruzione, `cout << sizeof(T);` stampa sul video il numero di byte usati per un qualsiasi valore del tipo `T`. Oltre ai tipi predefiniti elencati nella Tabella 2.1, il C++ permette anche i tipi `short int`, `long int` e `long double`. Il significato dei qualificatori `short` e `long` è ovvio, ma in molte realizzazioni i valori di questi tipi occupano lo stesso numero di byte del tipo base a cui il qualificatore è applicato. Per esserne certi si può usare la funzione `sizeof()` descritta prima.

Un altro tipo di valore, usato molto frequentemente nei programmi, è la stringa (di caratteri). Un tale valore è usato nell'Esempio 1.1 che scrive sul video la stringa, `"inserire 2 interi"`. Queste stringhe si chiamano **stringhe alla C**, per contrasto rispetto a quelle **alla C++** che sono realizzate dal tipo contenitore string di cui parleremo nella Sezione 9.6. Le stringhe alla C sono dei valori costanti formati da sequenze di caratteri racchiuse tra doppi apici. Esse possono contenere qualunque carattere, anche quelli di controllo. Per esempio, se nell'Esempio 1.1 sostituissimo nel la seconda riga del ma in la seguente stringa: `"inserire 2 interi \n"` a quella originale, potremmo osservare che l'output prodotto dal nuovo programma cambierebbe rispetto a quello del programma originale. Le stringhe alla C sono menzionate anche nella Sezione 5.1.

2.2 Variabili e dichiarazioni

Come tutti i linguaggi di programmazione, il C++ permette di usare dei nomi per rappresentare i dati che i programmi manipolano. Questi nomi si chiamano **variabili**(o anche identificatori). Le variabili devono iniziare sempre con un carattere alfabetico (minuscolo o maiuscolo) oppure con il carattere di sottolineatura _ (underscore) ed i caratteri successivi possono essere o alfabetici o numerici oppure il carattere _. La massima lunghezza delle variabili non è fissata, ma nomi troppo lunghi possono venire abbreviati in fase di compilazione e comunque rischiano di appesantire la programmazione. Il C++ distingue i caratteri minuscoli da quelli

maiuscoli. Quindi `pippo` è una variabile diversa da `Pippo` ed entrambe sono diverse da `PIPPO`.

Le variabili vengono introdotte nei programmi C++ attraverso le operazioni di **dichiarazione** che specificano il tipo delle variabili dichiarate. Per esempio, la dichiarazione `int x, pippo;` specifica che `x` e `pippo` sono variabili di tipo `int`(o semplicemente sono variabili intere). In generale, se il programma è corretto, una tale dichiarazione garantisce che i valori che, durante l'esecuzione del programma, verranno associati alle due variabili `x` e `pippo` saranno di tipo `int`. Al momento della dichiarazione di una variabile, è anche possibile (non necessario) assegnare un valore alla variabile stessa, come per esempio in: `int x=0, pippo= -1;`. In questo caso parliamo di **inizializzazione** delle variabili. Una variabile dichiarata, ma non inizializzata, è **indefinita**. Esempi di dichiarazioni di variabili di tipo char sono: `char y1, z;` e `char y1=' a' , z;` in cui `y1` viene inizializzata al carattere `'a'`. I nomi delle variabili devono in ogni caso essere diversi dalle parole chiave (keyword) del C++ come, per esempio, `int` e `char` e altre che saranno introdotte nel seguito.

A volte nei programmi c'è l'esigenza di usare dei valori **costanti**, per esempio costanti numeriche, stringhe particolari, eccetera. Piuttosto che usare direttamente i valori costanti nel testo del programma conviene dichiarare nomi costanti a cui assegnare questi valori e poi usare i nomi nel testo del programma dovunque servano i valori. Le dichiarazioni di costante hanno la seguente forma: `const double pi=3.14;` oppure `const char inizio='a', fine='z';`. Quindi si tratta semplicemente di premettere la parola chiave `const` ad una normale dichiarazione. Importante notare che nel caso delle costanti l'inizializzazione è sempre richiesta al momento della dichiarazione. Il valore assegnato(ovviamente) non potrà venire modificato nel programma (pena un messaggio d'errore del compilatore). Usare dichiarazioni di costanti, rispetto ad usare direttamente i valori costanti nel testo del programma, fa guadagnare in leggibilità, in robustezza ed in facilità di modifica dei programmi. Il guadagno in leggibilità è ovvio se i nomi costanti sono scelti con intelligenza. La robustezza deriva dal fatto che accidentali modifiche di un valore che dovrebbe essere costante vengono segnalate. Per la modificabilità, è ovvio che, in caso un valore costante sia da cambiare, è certamente più semplice e sicuro modificare la corrispondente dichiarazione di costante piuttosto che modificare tutte le occorrenze del valore nel testo del programma.

2.3 Espressioni e conversioni

Un'espressione è composta nel caso più semplice da costanti, per esempio, `2*( 5+24)`, il cui valore è 58. Espressioni possono contenere variabili, per esempio, `x*(pippo+24)`, il valore di questa espressione lo si calcola sostituendo ad ogni variabile il suo R-valore. Supponiamo che x abbia R-valore 5 e `pippo` abbia R-valore 2, in questo caso l'espressione ha valore 130. Sottolineiamo un fatto semplice, ma importante: le espressioni nei programmi vengono sempre valutate e questo procedimento (se ha successo) produce il **valore** dell'espressione. E' possibile che la valutazione di un'espressione non abbia successo? La risposta è affermativa e il fallimento della valutazione può avvenire per vari motivi. La ragione più semplice è che l'espressione richieda di eseguire operazioni indefinite come la divisione per O. Questo errore produce generalmente una terminazione anormale del programma in esecuzione. Un altro possibile motivo che può impedire la valutazione di un'espres-

sione è che l'espressione contenga operandi ed operatori di tipi incompatibili. Una tale espressione è `"pippo" * 13` in cui si chiede di moltiplicare una stringa alla C (il cui tipo non conosciamo ancora) con un intero. Questa espressione ovviamente non ha senso e questo fatto ci verrà segnalato dal compilatore. Quindi un programma che contiene una take espressione non viene (in generale) compilato e non potrà quindi venir eseguito. Un caso ancora diverso è quello in cui l'espressione da valutare contiene variabili indefinite. In questa situazione la valutazione dell'espressione può venir eseguita senza errori apparenti ed il solo sintomo dell'errore è un valore finale casuale.

Nel seguito, considereremo cosa avviene nel caso di espressioni che mescolano valori e variabili di tipi diversi, ma non incompatibili tra loro, cioè espressioni che il compilatore riesce a tradurre in codice oggetto senza dare errori. Questa situazione è problematica perché le operazioni macchina si applicano su valori dello stesso tipo. Per esempio, i computer possono eseguire l'operazione di somma di 2 operandi interi e anche la somma di 2 operandi reali, ma non la somma tra un intero ed un reale. Quindi cosa fa il compilatore se deve tradurre un'espressione come `2 + 3.14`? Ovviamente ci sono 2 possibilità: o trasformare l'intero 2 in un reale (scrivendo 2 in forma floating point) oppure trasformare il reale 3.14 in un intero, per esempio troncando la parte decimale riducendo 3.14 a 3. Il compilatore C++ sceglie sempre la prima possibilità. Il motivo è semplice e logico: un intero occupa meno byte (4) di un `double` (8 byte) e quindi la conversione dell'intero in `double` non comporta mai una perdita di informazione: ogni intero tra -2^{31} e $2^{31}-1$ ha una rappresentazione precisa in floating point con 8 byte. E' chiaro che invece trasformare 3.14 in un intero ci faccia perdere i decimali .14.

Quindi il compilatore C++ quando compila espressioni con valori di tipi diversi e compatibili, trasforma alcuni di questi valori in modo che ogni operazione si applichi a valori dello stesso tipo e nel farlo segue il seguente semplice principio: **vengono applicate le conversioni che producono il minimo rischio di perdita di informazione**. Le trasformazioni di tipo che soddisfano questo principio, vengono chiamate **promozioni**. In Sezione 9.4 si possono trovare maggiori dettagli sulle promozioni e in generale su tutti i tipi di conversione e su cosa comporti in pratica convertire un valore da un tipo ad un altro tipo.

Espressioni molto usate nei programmi sono le espressioni booleane, cioè quelle il cui valore è di tipo booleano. Queste espressioni generalmente usano gli operatori relazionali come $>$ $<$ $>=$ $<=$ $==$ (uguale) `!=` (diverso) e gli operatori logici `&&` `||` `!`. Un esempio di espressione booleana è la seguente. Si assuma che `x` sia una variabile di tipo char: `(x>='a') && (x<='z')`, ha valore `true` se `x` ha come R-valore un carattere alfabetico (tra 'a' e 'z') ed altrimenti l'espressione ha valore false. Si noti che questa espressione usa il fatto che la codifica ASCII dei caratteri assegna ai caratteri alfabetici dei valori interi contigui e coerenti con l'ordine alfabetico. Più precisamente, 'a' ha codice ASCII 97, 'b' 98 e così via. Naturalmente il codice ASCII codifica in modo simile anche i 10 caratteri numerici e le maiuscole.

Per le espressioni booleane che contengono gli operatori logici `&&` e ||, il C++ usa la cosiddetta valutazione **abbreviata** (o **shortcut**). Consideriamo l'espressione esaminata prima `(x>='a') && (x<='z')`. La valutazione procede da sinistra a destra, ma se la condizione di sinistra `x>='a'` è falsa allora non viene valutata la condizione di destra. Infatti l'intera espressione è falsa, indipendentemente dal valore della condizione di destra. Nel caso di un'espressione che contiene l'operatore `||`, come per esempio, `(x<'a') || (x>'z')`, se la condizione di sinistra è vera non

viene valutata quella di destra perché l'intera espressione è vera indipendentemente dal valore della condizione di destra.

2.4 A cosa servono i tipi

In C++, così come in moltissimi linguaggi di programmazione, ogni variabile deve venire dichiarata prima di poter essere usata e la dichiarazione specifica il tipo della variabile. Questa associazione implica (se il programma è corretto rispetto ai tipi) che, durante ogni esecuzione del programma, ogni variabile assumerà solo R-valori appartenenti al tipo dichiarato per quella variabile. Un tale comportamento è una restrizione che vincola la libertà del programmatore, ma è una restrizione *a fin di bene* per chi programma. Infatti, grazie al tipo di ciascuna variabile, diventa più facile evitare di inserire errori nei propri programmi (per esempio, scrivendo espressioni prive di senso che mescolano variabili e valori di tipi incompatibili). Un vantaggio ancora più importante è che il compilatore usa i tipi per controllare automaticamente che i nostri programmi usino le variabili solo in modo coerente rispetto al tipo dichiarato per esse. Inoltre i tipi servono al compilatore per risolvere i problemi di overloading. Si immagini che il compilatore debba tradurre l'espressione `x + y`. Quale operazione di somma dovrà inserire nel codice macchina che deve produrre? Quella che somma 2 interi o quella che somma 2 reali? I tipi di `x` e `y` dicono al compilatore cosa deve fare. Il caso di tipi diversi tra loro è discusso in Sezione 2.3.

Nonostante la parte orientata agli oggetti del C++ non venga trattata in questo testo, quando si discute dell'utilità dei tipi è obbligatorio accennarvi. Gli oggetti sono istanze di tipi (chiamati classi) che racchiudono al loro interno dati assieme a funzioni per manipolarli. Lo scopo è quello di proteggere i dati e di non permettere che essi vengano manipolati in modo arbitrario con funzioni diverse da quelle previste a questo scopo nella classe. La nozione di classe e la relazione di ereditarietà tra classi sono un importante ausilio alla realizzazione di programmi corretti e modificabili.

Occorre osservare però che il C++ (soprattutto a causa dell'eredità del C) non è un linguaggio *virtuoso* nella gestione dei tipi. Un anticipo di questo approccio troppo liberale rispetto ai tipi, l'abbiamo già avuto per i tipi predefiniti. Per esempio, in C++ i valori di tipo carattere e di tipo booleano vengono rappresentati internamente come interi e questo fatto rende corretto applicare a valori di tipo carattere e booleano degli operatori che una disciplina più rigorosa sui tipi considererebbe errori. Quindi, abbattendo le differenze tra tipi diversi, si perde la capacità di scoprire errori grazie ai tipi. Oltre a quelle appena segnalate, nel C++ ci sono altre debolezze, ancora più gravi, che vedremo successivamente e che rendono il C++ un linguaggio **insicuro rispetto ai tipi**, cioè un linguaggio nel quale è possibile (e purtroppo anche molto facile) scrivere programmi che (durante l'esecuzione assegnano ad una variabile di un certo tipo degli R-valori di tipo diverso (e incompatibile) senza che queste violazioni vengano segnalate né durante la compilazione né durante l'esecuzione.

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 0, 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. Il 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;` $\Rightarrow$ `potenza = potenza*2;` e
`esponente++;` $\Rightarrow$ `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. Il 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:*

```
#include<iostream>
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.*

```
#include<iostream>
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.*

```
#include<iostream>
using namespace std;
main() {
  int x, i = 0, max = INT_MIN, min = INT_MAX;
  while (i < 10) {
    cout << "inserire il prossimo intero";
    cin >> x;
    if(x > max)
      max = x;
    else
      if(min > x)
        min = x;
    i = i+1;
  }
  cout << "il valore massimo e' " << max << ", il valore minimo e' " << min;
}
```

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.*

```
// SOLUZIONE 1
#include<iostream>
using namespace std;
main() {
  int x, i = 0, max = INT_MIN, min = INT_MAX;
  while(i < 10) {
    cout << "inserire il prossimo intero";
    cin >> x;
    if(x > max) {
      max = x;
```

```
      if (i == 0)
        min = x;
    }
    else
      if(min > x) {
        min = x;
        if(i == 0) //($) questo commento e' usato nel
        max=x; //($) primo esercizio del prossimo Capitolo
      }
    i = i+1;
  }
  cout << "il valore massimo e' " << max << ", il valore minimo e' " << min;
}
```

È corretto? la risposta è affermativa, ma non è facile convincersene. Nell'Esercizio Risolto 4.2, mostreremo una prova di correttezza per questo programma. Resta la seconda strada per risolvere il nostro problema: leggere il primo valore ed assegnarlo sia a `max` *che a* `min` *e poi leggere gli altri 9 valori.*

```
// SOLUZIONE 2
#include<iostream>
using namespace std;
main() {
  int x, i = 1, max, min;
  cout << "Inserire il primo intero";
  cin >> x;
  max = min = x;
  while(i < 10) {
    cout << "Inserire il prossimo intero";
    cin >> x;
    if(x > max)
      max = x;
    else
      if(min > x)
        min = x;
    i = i+1;
  }
  cout << "il valore masimo e' " << max << ", il valore minimo e' " << min;
}
```

Praticamente è la prima soluzione proposta, quella che si è rivelata sbagliata. Siamo sicuri che questa diversa inizializzazione la renda corretta? Non è evidente e solo con la prova di correttezza che verrà sviluppata nel Capitolo 4 potremo esserne sicuri.

Esercizio Risolto 3.5 *Questo esercizio ha lo scopo di illustrare e confrontare alcune operazioni di i/o viste nella Sezione 3.1.3. L'Esempio è strettamente collegato ai due esercizi seguenti che andrebbero infatti considerati immediatamente dopo questo. Si tratta di scrivere un programma che apre come input un file di testo pippo.cpp e legge il suo contenuto carattere per carattere ricopiando i caratteri letti sul file di output standard (il video, associato allo* `stream cout`*). La coppia di operazioni di lettura e stampa è ripetuta grazie all'uso del comando iterativo* `while` *la cui condizione di permanenza è ! PP.eof(), cioè le due operazioni sono ripetute fino a che l'end of file di* `PP` *non è raggiunto.*

```
#include<iostream>
using namespace std;
main() {
```

```
4     ifstream PP("pippo.cpp");
5     if(PP != 0) { //se l'apertura ha avuto successo
6       while(!PP.eof()) {
7         char c;
8         c = PP.get();
9         cout << c;
10      }
11      PP.close();
12    }
13    else
14      cout << "errore nell'apertura del file" << endl;
15  }
```

Questo programma controlla se l'apertura del file ha avuto successo e nel caso sia fallita, stampa sul video un apposito messaggio d'errore e termina senza fare altro. Solo nel caso l'apertura abbia avuto successo, legge, carattere per carattere, il contenuto del file e stampa sul video ciascun carattere immediatamente dopo averlo letto. Cambiando opportunamente il nome pippo.cpp è possibile usare questo programma per copiare a video ogni file (di testo).

Esercizio 3.6 *Si chiede di memorizzare il programma descritto nell'Esercizio precedente in un file di nome pippo.cpp, di compilarlo e di eseguirlo. L'esecuzione di questo programma farà comparire sul video il testo del programma stesso seguito da un ultimo carattere strano. Questo carattere strano è causato dal fatto che, dopo che viene letto l'ultimo carattere buono del file,* `PP.eof()` *non diventa immediatamente* `true`*. È necessaria un'ulteriore operazione* `get` *per far scattare* `PP.eof()` *a* `true`*. Visto che questa lettura non trova nulla da leggere, l'R-valore di* `c` *diventa il carattere strano che viene stampato per ultimo.*

Esercizio 3.7 *Si chiede di modificare il programma dell'Esercizio precedente, sostituendo l'operazione di lettura con la seguente:* `PP >> c;`*. Sia ancora pippo.cpp il file che contiene il programma modificato. L'esecuzione di questo nuovo programma stampa sul video tutti i caratteri del file pippo.cpp omettendo gli spazi e gli accapo. In sostanza il contenuto del file viene scritto tutto di seguito. Questo strano comportamento è dovuto a quanto detto in precedenza sull'operazione »: essa non legge i caratteri di separazione (spazio e accapo), ma li usa solo per separare le stringhe di caratteri buoni. Invece l'operazione get legge tutti i caratteri, anche quelli di separazione. Va notato anche che ora nessun carattere strano conclude la stampa. Il motivo è sempre lo stesso: l'operazione >> salta i separatori e quindi, dopo la lettura dell'ultimo carattere buono, salta qualunque carattere di separazione lo segua e attiva subito l'end of file.*

3.6 Sulla visibilità delle variabili

È utile capire cosa succede alle variabili ed ai loro valori quando un programma viene eseguito. Abbiamo già visto che il programma sorgente viene tradotto dal compilatore in un equivalente programma oggetto che consiste di istruzioni macchina. Quando il programma oggetto viene eseguito, esso deve risiedere nella memoria RAM del computer ed un'altra area della RAM contiene i valori delle variabili del programma. Eseguire un programma significa che in ogni momento una particolare

istruzione del programma oggetto viene eseguita e che un particolare sottoinsieme delle variabili del programma sono **attive**. Quindi la zona di memoria RAM che contiene i valori delle variabili del programma non è statica, ma al contrario essa cambia sia nella sua dimensione sia nei valori che contiene perché in ogni momento cambia l'insieme delle variabili attive. Cerchiamo di capire quali sono le variabili attive in ogni momento dell'esecuzione e come vengono gestite. In generale, un programma è costituito da blocchi annidati gli uni negli altri, dove ogni blocco può contenere dichiarazioni di variabili (oltre a istruzioni). Ecco un main che consiste di 3 blocchi annidati con dichiarazioni e operazioni di output:

```
1  main()
2  //(*)
3  {
4    int x = 0; //...(0)
5    {
6      int y = 1; //...(1)
7      {
8        int z = 2; //...(2)
9      }
10   cout << x << y << endl; // (1')
11   }
12 cout << x << endl; // (0')
13 }
14 //(*')
```

L'esecuzione inizia al punto (*) e in quel momento nessuna variabile è ancora attiva. Quando l'esecuzione entra nel blocco (0), la variabile `x`, dichiarata in questo blocco, diventa attiva, mentre le variabili `y` e `z` non sono ancora attive. Lo potranno diventare solo quando l'esecuzione entra nel secondo e nel terzo blocco, rispettivamente. Ma cosa significa in pratica che una variabile diventa attiva? Significa che le viene attribuita un'area di memoria RAM in cui verrà custodito il suo R-valore. Questa attribuzione di memoria viene chiamata **allocazione della variabile** e l'indirizzo di memoria attribuito è l'L-valore della variabile. Quindi quando l'esecuzione raggiunge il punto (0), sarà allocata la variabile `x`, quando raggiunge il punto (1), verrà allocata anche `y` e in (2) sarà allocata anche la `z`. É molto importante capire che in (1') l'esecuzione è uscita dal blocco più interno e quindi la variabile `z` non è più attiva e l'area di memoria che le era stata attribuita viene tolta e torna disponibile per l'allocazione di altre variabili. Questa operazione si chiama **deallocazione della variabile**. Quindi in (1') saranno attive solo le variabili `x` e `y` e quindi la stampa avrà successo e farà apparire 0 1 sullo schermo. Il tentativo di stampare `z` in (1') avrebbe causato un errore di compilazione con una spiegazione del tipo: variabile `z` non esiste. A questo punto non sarà una sorpresa il fatto che in (0') anche `y` verrà deallocata e solo `x` resta attiva, mentre in (*') tutte e 3 le variabili saranno deallocate.

Da questa descrizione dovrebbe essere facile vedere che la gestione dei dati manipolati da un programma segue un andamento a pila: quando l'esecuzione entra in un blocco, le variabili dichiarate nel blocco vengono allocate sulla cima della pila e vengono deallocate dalla cima della pila quando l'esecuzione esce dal blocco stesso. La Figura 3.3 illustra questa gestione a pila. Visto che l'allocazione e la deallocazione delle variabili dipende solamente dal blocco in cui le variabili sono dichiarate, esse sono dette **automatiche** per sottolineare la loro differenza con variabili i cui valori sono gestiti in un modo diverso che verrà illustrato in Sezione 11.1. Una variabile

automatica è attiva nel periodo che va dalla sua allocazione alla sua deallocazione. La **visibilità** di una variabile è sempre il blocco in cui è dichiarata, dalla sua dichiarazione in poi. Va ricordato che la pila delle variabili attive è realizzata in una parte della memoria RAM.

Nell'esempio di Figura 3.3, per semplicità, abbiamo considerato una sola dichiarazione per blocco. Il caso di più dichiarazioni in un blocco viene gestito in modo analogo: lo spazio necessario per tutte le variabili dichiarate viene allocato sulla pila nel momento in cui l'esecuzione entra nel blocco e quella stessa zona di memoria viene deallocata all'uscita dal blocco. Non dobbiamo dimenticare la relazione che intercorre tra la pila dei dati ed il codice oggetto che sta eseguendo. Supponiamo che nel blocco (2) ci sia un comando che usi la variabile `x`, allora la variabile usata è quella allocata in fondo alla pila in Figura 3.3(2). D'altra parte, se all'interno del blocco (2) ci fosse una nuova dichiarazione di `x`, per esempio `double x;` che precede l'istruzione che usa `x`, allora l'istruzione farebbe riferimento a questa nuova `x` che sarebbe allocata in cima alla pila, come mostra la Figura 3.4(a). Nella Figura 3.4(a) si deve anche osservare che entrambe le variabili `x` sono allocate, ma quella in cima alla pila oscura l'altra visto che si trova più vicina alla cima della pila (si ricordi che la pila cresce verso il basso). Quando l'esecuzione esce dal blocco (2), la pila diventerà quella di Figura 3.4(b) e quindi l'output in (1') stamperà l'R-valore della `x` che sarà tornata visibile visto che non ci sono più altre `x` sopra di lei nella pila.

Una variabile attiva ha un L-valore ed un R-valore. L'L-valore è l'indirizzo di memoria RAM (sulla pila dei dati) in cui l'R-valore viene immagazzinato. Ricordiamo che la lettera R sta per Right (destro) e la L per Left (sinistro). La spiegazione di questi nomi è stata data in Sezione 3.2.

Esiste un'importante differenza tra l'R- e l'L-valore di una variabile attiva. Mentre l'R-valore di una variabile viene "deciso" dalle istruzioni che definiscono la variabile e quindi in generale cambia durante l'esecuzione del programma, il suo L-valore viene deciso dal programma che gestisce la pila dei dati e quindi non può venire deciso né modificato dal programma in esecuzione. In compenso il programma può conoscere ed usare l'L-valore delle sue variabili. Una variabile viene detta **indefinita** quando essa è attiva, ma non è inizializzata. Essendo attiva, memoria è stata allocata nella pila dei dati per contenere il suo R-valore, ma i byte a disposizione contengono una sequenza di bit casuale, cioè quello che casualmente è contenuto in quel momento in quei byte della RAM. Fare uso dell'R-valore di una variabile indefinita è un errore che può anche essere insidioso da scoprire. Infatti, ogni tipo di valore all'interno di un computer è rappresentato come una sequenza di bit, quindi una sequenza casuale di bit viene sempre interpretata come un R-valore coerente con il tipo della variabile, causando, senza alcuna segnalazione d'errore, il calcolo di valori erronei.

Il corpo di un `while` o il ramo `then` o `else` di un condizionale costituiscono un blocco e naturalmente questi blocchi al loro interno ne possono contenere altri con livello di annidamento arbitrario. Inoltre ogni blocco può contenere dichiarazioni. Ogni dichiarazione appartiene ad un solo blocco e bisogna essere precisi nel definire quale sia. Il blocco di appartenenza di una dichiarazione D è il blocco che contiene D ed è tale che non ci sia alcun altro blocco, annidato in esso, che contenga D (attenzione: proprio D, non un'altra dichiarazione uguale a D). Quindi, delle dichiarazioni che appartengono ad un blocco, non fanno parte quelle che compaiono nei blocchi annidati.

La visibilità o usabilità di una variabile inizia dal punto del programma imme-

diatamente successivo alla sua dichiarazione e si estende fino alla fine del blocco cui la dichiarazione appartiene. Eventuali blocchi annidati sono inclusi nella visibilità. Quindi le variabili dichiarate in un certo blocco sono, in linea di principio, visibili anche nei blocchi annidati.

Il C++ (così come ogni linguaggio di programmazione con tipi) fissa una regola molto semplice per le dichiarazioni di variabili, la **Regola della dichiarazione singola** che è la seguente: **in uno stesso blocco non ci può essere, più di una dichiarazione di una variabile**. In uno stesso blocco non sono ammesse neppure dichiarazioni di una variabile con tipi diversi. Il motivo è facile da capire: se ci fossero 2 dichiarazioni diverse di uno stesso nome in uno stesso blocco, ci sarebbero 2 variabili con lo stesso nome contemporaneamente attive nel blocco e quindi, qualora questo nome venisse usato, sarebbe impossibile sapere a quale delle due variabili si intende fare riferimento. D'altra parte in un programma è consentito definire diverse volte variabili con lo stesso nome purché queste dichiarazioni appartengano a blocchi diversi, anche annidati. Nel caso di ridefinizioni in blocchi annidati, la ridefinizione di una variabile in un blocco annidato produce l'oscuramento della variabile con lo stesso nome dichiarata nel blocco esterno. Questo è necessario per avere sempre una sola variabile con un dato nome visibile. Questo fenomeno di oscuramento è stato già descritto in Figura 3.4 che mostra come il meccanismo dell'oscuramento viene facilmente realizzato tramite la gestione dei dati automatici con una pila. Altre informazioni sulla visibilità delle variabili si trovano nelle Sezioni 5.3, 7.4 e 9.7.

Concludiamo la Sezione con un'osservazione importante sulle dichiarazioni che si trovano nel blocco costituito dal corpo di un comando iterativo. Ad ogni iterazione, l'esecuzione entra nel blocco, producendo l'allocazione delle variabili dichiarate al suo interno, giunge alla fine del blocco, producendo la deallocazione delle stesse variabili e così via per ogni iterazione. Quindi il risultato del seguente programma è la stampa di 1 0 per 1 0 volte: ad ogni iterazione si alloca una nuova `x` con R-valore 1 0 e la successiva somma non si trasmette da un'iterazione all'altra.

```
1  int i = 0;
2  while(i < 10) {
3    int x = 10;
4    cout << x << endl;
5    x = x+10;
6    i++;
7  }
```

Capitolo 4

La correttezza dei programmi

In questo capitolo mostreremo come sia possibile dimostrare in modo convincente che i nostri programmi sono corretti, cioè che fanno effettivamente quello per cui li abbiamo scritti. Inizieremo dall'Esercizio 3.3 che ci aiuterà a definire una regola generale per dimostrare la correttezza dei cicli while e do-while. Con questa regola affronteremo l'Esercizio 4.2 che già presenta maggiori difficoltà.

Molto spesso il programmatore realizza il suo programma seguendo un'idea intuitiva più o meno precisa. Successivamente controlla il programma eseguendo alcuni test con input diversi. Questa maniera di procedere, molto comune, non garantisce che il programma sia corretto. Sviluppare una prova di correttezza consiste nel precisare l'idea intuitiva, evidenziando eventuali errori. Nel seguito della trattazione cercheremo di sottolineare i passaggi in cui la sua formalizzazione rivela gli errori nascosti sotto un'intuizione troppo superficiale.

Esercizio Risolto 4.1 *Nell'Esercizio Risolto 3.3 abbiamo presentato 2 soluzioni per risolvere uno stesso esercizio Vogliamo ora dimostrare che sono entrambe corrette. Per dimostrare che un programma è corretto, è necessario prima scrivere in maniera precisa cosa il programma dovrebbe fare. Si tratta di specificare due asserzioni, la pre- e la postcondizione del programma. La precondizione specifica la situazione dello stato del calcolo che si assume valga quando il programma inizia la sua esecuzione. La postcondizione invece specifica la situazione dello stato del calcolo dopo che il programma termina. La dimostrazione di correttezza di un programma è sempre in relazione ad una coppia di pre- e postcondizioni e consiste nel dimostrare che se il programma esegue partendo da uno stato del calcolo che soddisfa la precondizione, allora, se esso terminerà, lo farà in uno stato che soddisfa la postcondizione. Vediamo di specificare pre- e postcondizioni per il problema in esame:*

- **PRE = (`cin` contiene l'intero 0 preceduto da `n` interi diversi da 0, con `n`≥0)**
- **POST = (`neg+post=n` e `neg` è il numero degli interi negativi che precedono 0 su `cin`, mentre `pos` è il numero degli interi positivi che precedono 0)**

La precondizione asserisce che su `cin` *c'è certamente uno 0 e che esso è preceduto da zero o più interi diversi da 0. Questo è importante perché ci assicura che il*

programma terminerà sempre. La postcondizione invece asserisce che il programma deve calcolare in `neg` *il numero dei negativi e in* `pos` *quello dei positivi che precedono lo 0. Ovviamente i nomi* `pos` *e* `neg` *sono arbitrari. Per dimostrare che la prima soluzione dell'Esercizio Risolto 3.3), quella che usa il* `while`*, è corretta, dobbiamo dimostrare che il suo ciclo* `while`*, iterazione dopo iterazione arriva ad attribuire a* `neg` *e* `pos` *il valore giusto rispetto ai valori letti da* `cin`*. Possiamo dire che ogni volta che stiamo per ripetere il ciclo* `while`*, avremo letto* `neg+pos+1` *interi da* `cin` *e i primi* `neg+pos` *sono diversi da 0 e ce ne sono* `neg` *negativi e* `pos` *positivi, e inoltre, se* `x` *è 0 allora* `neg+pos=n`*, mentre se è diverso da 0, allora* `neg+pos<n`*.*

$$\mathrm{R} = \begin{cases} (1) & \textbf{(letti } \texttt{neg+pos+1} \textbf{ valori da } \texttt{cin} \textbf{ (neg e pos non negativi))} \\ (2) & \textbf{(i primi } \texttt{neg+pos} \textbf{ valori letti sono tutti diversi da 0} \\ & \quad \textbf{e ce ne sono } \texttt{neg} \textbf{ negativi e } \texttt{pos} \textbf{ positivi)} \\ (3) & \textbf{x=0} \Rightarrow \texttt{neg+pos=n)} \\ (4) & \textbf{x} \neq \Rightarrow \texttt{neg+pos<n)} \end{cases}$$

Osserviamo alcune cose interessanti su questa **R**:

1. *la prima volta che l'esecuzione arriva al ciclo (cioè prima di valutare il test di permanenza del ciclo),* ***R*** *vale: abbiamo letto un solo valore e infatti* `neg+pos=0` *(vale(1) di* ***R****) e se* `x=0`*, allora n è 0 e quindi* `neg+pos=0` *(e quindi vale (3) di* ***R****), mentre se* `x`$\neq$*0, allora* `n>0` *(e quindi vale (4) di* ***R****). Il punto (2) di* ***R*** *vale in modo banale.*

2. *Se* ***R*** *vale all'inizio del ciclo e il test di permanenza è vero allora dopo aver eseguito il corpo del ciclo una volta di più, vale ancora* ***R****. Vista l'ipotesi che il tesi di permanenza è vero,* `x` *non è 0 e quindi è un valore che precede 0 per cui il test aumenta correttamente o* `neg` *o* `pos`*) ((2) di* ***R****). Poi viene letto un nuovo valore, ma visto che* `neg+pos` *è aumentata di 1,* `neg+pos+1` *è ancora uguale al numero di letture (punto (1) di* ***R****). (3) e (4) di* ***R*** *valgono banalmente. Questo significa che* ***R*** *vale ogni volta che l'esecuzione arriva all'inizio del ciclo. Per questo motivo* ***R*** *è detto* **invariante** *del ciclo.*

3. *Se all'inizio del ciclo vale* ***R*** *e il test di permanenza è falso, allora vale* ***POST****. Che il test di permanenza sia falso significa che* `x=0` *e in questo caso i punti (2) e (3) di* ***R*** *implicano* ***POST****.*

I tre punti appena visti formano lo schema che useremo sempre per dimostrare la correttezza dei cicli. Chiaramente il cuore della dimostrazione è l'invariante ***R*** *che descrive quello che il ciclo calcola dopo ogni numero di iterazioni. Ma possiamo seguire questo schema anche per dimostrare la correttezza della seconda soluzione dell'Esercizio Risolto 3.3, quella che usa il* `do-while`*?* ***PRE*** *e* ***POST***

4.1 Regola di prova del `while`

4.2 Altri esempi di prove di correttezza

4.2.1 Esempio di gestione delle eccezioni

Capitolo 5

Puntatori, array ed aritmetica dei puntatori

I puntatori sono una nozione fondamentale per il C. Nel C++ essi sono affiancati dai riferimenti. In questo capitolo illustreremo entrambe le nozioni sottolineando la stretta relazione esistente tra di esse e spiegando il motivo per cui i riferimenti hanno soppiantato i puntatori nei linguaggi più moderni come Java. Successivamente introdurremo gli array assieme al costrutto iterativo FOR che si usa spesso per percorrere gli array. Cura particolare sarà dedicata a spiegare il tipo degli array a più dimensioni. Aver chiarito questo argomento tornerà molto utile per capire *l'aritmetica dei puntatori* che è descritta nella sezione finale del capitolo.

5.1 Puntatori e riferimenti

5.2 Array e for

5.3 Il tipo degli array

5.4 L'aritmetica dei puntatori

5.5 Stringhe alla C e array di caratteri

Capitolo 6

Alla Ricerca dell'Invariante Perduto

In questo Capitolo intendiamo rispondere ad una domanda che tipicamente gli studenti si e soprattutto ci (a noi docenti) rivolgono quando viene loro chiesto di costruire la prova di correttezza dei loro programmi e cioè, "come faccio a trovare le asserzioni ed, in particolare, gli invarianti dei cicli?". La prima risposta che viene in mente è "bisogna pensarci su", ma in realtà è possibile dire qualcosa di più utile. Quando dobbiamo realizzare un programma per eseguire una certa operazione, il punto di partenza ineludibile è la chiara comprensione sia della situazione del calcolo nella quale il programma verrà eseguito sia della situazione del calcolo che dovrà essere realizzata alla fine della sua esecuzione. Specificare queste 2 cose coincide con la scrittura della pre- e della postcondizione. È innegabile che la scrittura precisa di queste due asserzioni è necessaria prima di iniziare a scrivere il codice della funzione stessa. Mostreremo con qualche esempio che la postcondizione può guidarci nella determinazione degli invarianti dei cicli contenuti nel programma. In sostanza illustreremo una semplice tecnica che permette di derivare automaticamente dalla postcondizione un'asserzione che in alcuni casi è l'invariante che cerchiamo e, nei casi meno fortunati, è comunque utile a trovare l'invariante. Spiegheremo questa tecnica risolvendo alcuni esercizi che riguardano le seguenti nozioni.

Definizione 6.1 *Dati 2 insiemi A e B, diremo che A è contenuto in B se ogni elemento di A è anche in B. Si noti che anche nel caso A e B fossero uguali,* ***A è contenuto in B****. Se invece consideriamo 2 multi-insiemi A e B, cioè insiemi che possono contenere occorrenze multiple dei valori, allora diremo che* ***A è m-contenuto in B****, quando per ogni valore v in A, se il numero delle occorrenze div in A è A(v), allora v è contenuto anche in B e B(v)>=A(v).*

Dato un array `C[r][k]` sia R una sua riga e K una sua colonna. Possiamo vedere R e K come insiemi e chiederci se R è contenuto in K, oppure possiamo vedere R e K come multi-insiemi e chiederci se R è m-contenuto in K. Per essere certi che la differenza sia chiara diamo il seguente esempio:

Esempio: sia R=[0,1,0,2,0] e K=[2,0,1,0]. R è contenuto in K, mentre non è m-contenuto in K in quanto R(0)>K(0). Invece K è sia contenuto che m-contenuto in R.

Esercizio Risolto 6.1 *Il primo problema che vogliamo affrontare è il seguente: dato un array* `int C[6][5]` *vogliamo riempire un array* `bool B[6][5]` *in modo che* $\forall i \in [0..5]$ *and* $\forall j \in [0..4]$, `B[i][j]=true` *se e solo se (sse) la riga* `C[i]` *è contenuta nella colonna* `C[..][j]`[1].

Si osservi che viene considerato il contenimento semplice tra insiemi e non l'm-contenimento, vedi la Definizione 6.1. Prima di occuparci della definizione di `B`*, dobbiamo riempire di valori interi l'array* `C`*. A questo fine, facciamo l'ipotesi di avere un file che si chiama input che contiene almeno 30 interi che useremo per riempire C. L'operazione di lettura dei 30 interi dal file è un esempio veramente semplice e lo sfruttiamo per iniziare ad introdurre, nel modo più semplice possibile, la tecnica di costruzione degli invarianti che abbiamo annunciato all'inizio di questo Capitolo. L'ipotesi fatta in precedenza sul file input si traduce nella seguente precondizione:* **PRE=(il file `input` contiene almeno 30 interi)** *dalla quale segue che non dobbiamo preoccuparci di controllare se il file termina prima di aver letto i 30 interi. Dobbiamo "aprire" il file e dichiarare l'array* `C` *delle dimensioni previste:*

```
1 ifstream IN("input");
2 int C[6][5];
```

e procedere a leggere i 30 interi da `IN` *in* `C` *senza preoccuparci di altro. È chiaro che ci servono 2 cicli annidati, il primo che scorre le righe e quello interno che riempie ciascuna riga scorrendo le colonne. Supponiamo che l'indice che scorre le righe sia* `i` *e quello che scorre le colonne sia* `j`*. Deriveremo gli invarianti di questi cicli, in modo meccanico, dall'asserzione* ***POST*** *che deve essere vera dopo la lettura. La speranza (che sarà mantenuta) è che in questo modo sarà facile dimostrare che la* ***POST*** *è vera quando la lettura termina. Questa tecnica viene chiamata* ***ricetta di indicizzazione****. Vediamo in cosa consiste.*

Iniziamo col precisare cosa vogliamo sia vero dopo la lettura. Ovviamente vogliamo che `C` *sia riempita di interi. Esprimere questa proprietà è facile:*
POST=($\forall$a $\in$ [0..5] e $\forall$b $\in$ [0..4], `C[a][b]` è definito).

Il primo ciclo, che scorre le righe con l'indice `i`*, dovrà dire che il riempimento di* `C` *è arrivato a riempire fino alla riga* `i-1` *ed infatti esso è ricavabile da* ***POST*** *sostituendo a 5 il valore* `i-1`*, ottenendo il seguente invariante:*
R1=($\forall$a $\in$ [0..i-1] e $\forall$b $\in$ [0..4], `C[a][b]` è definito), *a cui dobbiamo aggiungere il campo di variabilità di i, cioè* **(0$\leq$i$\leq$6)**. *Quindi* ***R1*** *lo si ottiene da* ***POST*** *sostituendo a 5 l'indice del ciclo* `i-1`*. Ecco spiegato il nome di ricetta di indicizzazione. La relazione che questa ricetta istituisce tra* ***POST*** *e* ***R1*** *rende automatica una parte della prova di correttezza del ciclo.*

La regola di verifica del while (vedi Sezione 4.1) dice che la prova di correttezza si articola in 3 parti. Concentriamoci sulla parte (3), che considera l'uscita dal ciclo. La regola dice che in questo caso dobbiamo dimostrare che:
R1 && !(i<6) $\Rightarrow$ POST.

Da ***(0$\leq$i$\leq$6)****, contenuta in* ***R1****, assieme a* ***!(0<i<6)$\equiv$(0>i>6)****, segue che* ***i=6*** *e quindi, da* ***R1****, abbiamo che* $\forall$***a*** $\in$ ***[0..5] e*** $\forall$***b*** $\in$ ***[0..4],*** `C[a][b]` ***è definito****, che è uguale alla nostra* ***POST*** *e quindi l'implicazione è verificata! In effetti, applicando*

[1] Con la notazione `C[..][j]`, nel seguito indicheremo la colonna `j` di C.

*la **ricetta di indicizzazione**, abbiamo costruito **R1** da **POST** proprio per fare in modo che esse coincidessero quando tutte le righe fossero riempite!*

Il nostro programma continua con

```
for(int i = 0; i < 6; i++){ //R1
  comandi che riempono la riga i;
    //qui vogliamo che sia vero Q=(riempita la riga i)
}
```

Precisiamo l'asserzione **Q=(∀b ∈ [0..4], C[a][b] è definito)** *e avremo bisogno di un ciclo, che esegue 5 letture riempiendo con i 5 valori letti la riga i-esima di* `C` *in modo da arrivare in uno stato in cui vale* ***Q****. Di nuovo usiamo la ricetta di indicizzazione per ricavare l'invariante di questo ciclo da* ***Q****, cioè sostituiamo in* ***Q*** *a 4 l'indice* `j-1` *(si ricordi che* `j` *scorre le colonne). L'invariante che otteniamo è:* **R=(∀b ∈ [0..j-1], C[a][b] è definito)&&(0≤j≤5)**.

A questo punto dovrebbe essere facile capire che, quando si uscirà dal ciclo, `j=5` *e quindi* ***R*** *con* `j=5` *diventa* ***Q****. Il ciclo che scorre le righe è:*

```
for(int j = 0; i < 5; j++) //R
  IN >> C[i][j];
//Q
```

e la correttezza della sua condizione di uscita dal ciclo è automatica grazie alla ricetta di indicizzazione. La parte (2) della correttezza, cioè l'invarianza di ***R****, è vera perché la lettura rende vera* **(∀b ∈ [0..j], C[i][b] è definito)** *e l'incremento* `j++` *rende di nuovo vero* ***R****. Nota anche che quando eseguiamo il corpo, la condizione di permanenza è vera e allora* `j<5` *e* `j++` *rende vero* `j<=5` *come richiesto da* ***R****. La condizione iniziale del ciclo è facile: la prima volta* `j=0` *e quindi* **R = (∀b ∈ [0..0-1],..)&&(0≤j≤5)** *e dato che [0..-1] è l'intervallo vuoto, è banalmente vero che per tutti i b in questo intervallo (cioè nessuno)* `C[i][b]` *è definita. Inoltre j=0 è nell'intervallo richiesto da* ***R****.*

Ora cerchiamo di dimostrare la parte (2) del primo ciclo (invarianza di ***R1****). Usiamo la prova del ciclo interno che ci dice che dopo il ciclo interno vale* ***Q****, e* ***R1****&&****Q*** *implica che abbiamo riempito una riga in più rispetto a quelle riempite in* ***R1****, quindi vale:* **(∀a ∈ [0..i] e ∀b ∈ [0..4], C[a][b] è definito)&&(0≤i<6)** *e dopo aver eseguito l'incremento* `i++` *(del primo* `for`*), questo diventa* ***R1****.*

La condizione iniziale del primo ciclo è simile a quella del ciclo interno. Il fatto che si parta con i=0, rende ***R1*** *banalmente verificata.*

*Dalla parte di esercizio appena sviluppata, abbiamo formalizzato la **ricetta di indicizzazione** che continueremo ad usare nel resto dell'esercizio e anche nel resto del libro. Deve essere chiaro che questa **ricetta** non è sempre sufficiente a ricavare l'invariante, ma in ogni caso essa ci fornisce un utile punto di partenza per ottenerlo. Nel seguito useremo questa ricetta per costruire e dimostrare la correttezza della seconda parte dell'esercizio, cioè il calcolo di* `B`*. Iniziamo con descrivere cosa deve essere vero dopo la parte di programma che definisce* `B`*.*

POST1=(B è calcolata bene) *che si può scrivere in modo preciso come segue,* **(∀a ∈ [0..5] e ∀b ∈ [0..4], B[a][b] sse C[a] è contenuta in C[..][b])**.

Visto che B viene riempita per righe, servono anche qui 2 cicli `for` *annidati, il più esterno che scandisce le righe di* `B` *e quello interno che scandisce le colonne. Assumiamo che gli indici dei due cicli siano* `i` *e* `j`*, rispettivamente. Allora l'invariante del ciclo più esterno (secondo la ricetta delineata prima) lo ricaviamo modificando* ***POST1*** *nel modo seguente:* **R1=(∀a ∈ [0..i-1] e ∀b ∈ [0..4],**

`B[a][b]` **sse** `C[a]` **è contenuta in** `C[..][b]`)**&&(0≤i≤6)** *e quello del* ***for*** *interno avrà invariante:* **R2=(∀b ∈** `[0..j-1]`**,** `B[i][b]` **sse** `C[i]` **è contenuta in** `C[..][b]`**)&&(0≤j≤5)** *e alla fine di questo ciclo dovrà valere l'asserzione,* **POST2=(ho messo a posto la riga** `i`**)** *che è espressa in maniera precisa nel modo seguente,* **R2=(∀b ∈** `[0..4]`**,** `B[i][b]` **sse** `C[i]` **è contenuta in** `C[..][b]`**)**. *Si osservi che* ***POST2*** *è ottenuta da* ***R2*** *con la solita* ***ricetta****, ma applicata nella direzione opposta rispetto al solito. Quindi il programma che riempie B comincia così:*

```
for(int i = 0; i < 6; i++); //R1
  for(int j = 0; j < 5; j++); //R2
```

Il corpo del ciclo interno deve mantenere ***R2*** *invariante e quindi deve inserire il valore giusto in* ***B[i][j]*** *e questo lo possiamo esprimere così: il blocco deve consistere nel calcolo di una variabile booleana, diciamo* ***OK****, che godrà della seguente proprietà:* **POST3=(**`OK` **sse ∀b ∈** `[0..4]` **∃a ∈** `[0..5]`**,** `C[i][b]=C[a][j]`**)**.

Per calcolare una tale variabile ***OK****, la formula esprime chiaramente che abbiamo bisogno di un ciclo che scorra la riga (useremo l'indice* ***k****) ed un secondo* ***for*** *annidato che scorra la colonna (useremo l'indice* ***z****) per cercare l'elemento corrente della riga.*

L'invariante del primo ciclo lo otteniamo di nuovo facilmente da **POST3:**
R3=(`OK` **sse ∀b ∈** `[0..k-1]` **∃a ∈** `[0..5]`**,** `C[i][b]=C[a][j]`**)&&(0≤k≤5)** *in sostanza* ***OK*** *ci da l'informazione che vogliamo fino all'elemento* ***k-1****. Quindi il corpo del terzo* ***for*** *deve calcolare l'informazione se* ***C[i][k]*** *è in* ***C[..][j]*** *o no, in altre parole deve produrre un booleano, chiamiamolo* ***trovato****, che alla fine del blocco soddisfi:*
POST4=(`trovato` **sse ∃a ∈** `[0..5]`**, tale che** `C[i][k]=C[a][j]`**).**

Da **POST4** otteniamo nel solito modo **R4** invariante dell'ultimo(quarto) ciclo:
R4=(`trovato` **sse ∃a ∈** `[0..z-1]`**, tale che** `C[i][k]=C[a][j]`**)&&(0≤z≤6)**
La relazione tra ***R4*** *e* ***POST4*** *è quella della nostra* ***ricetta*** *e quindi garantisce che, all'uscita del ciclo, allorché vale* ***(z==6)****, sia vera* ***POST4****. Quindi il ciclo più annidato è come segue:*

```
bool trovato = false;
  for(int z = 0; z < 6; z++) //R4
    if(C[i][k] == C[z][j])
      trovato = true;
//POST4
```

Il fatto che ***R4*** *è invariante è facile: fa avanzare di 1 la porzione controllata della colonna* ***j****, cioè, dopo il test vale:* **(**`trovato` **sse ∃a ∈** `[0..z]`**, tale che** `C[i][k]=C[a][j]`**)&&(0≤z≤6)** *e quindi* ***z++*** *rende nuovamente vero* ***R4****.*

Vale anche la pena di controllare che ***R4*** *sia vera la prima volta che si arriva a valutare il test di permanenza del quarto ciclo: in questo caso* ***trovato=false*** *quindi* ***R4*** *diventa* **(**`false` **sse ∃a ∈** `[0..0-1]`**, tale che** `C[i][k]=C[a][j]`**)&&(0≤z≤6)** *e visto che [0,-1] è vuoto, ne segue che* **(∃a ∈** `[0..-1]`**, tale che** `C[i][k]=C[a][j]`**)** *è falso. Quindi abbiamo* ***(falso sse falso)*** *che è vero.*

Mettiamo ora assieme i quattro cicli annidati:

```
for(int i = 0; i < 6; i++)
{
  for(int j = 0; j < 5; j++)
  {
    bool OK = true;
    for(int k = 0; k < 5; k++)
```

```
7       {
8         bool trovato = false;
9         for(int z = 0; z < 6; z++)
10          if(C[i][k] == C[z][j])
11            trovato = true;
12        if(!trovato)
13          OK = false;
14      }
15      B[i][j] = OK; //POST3 ci dice che e' la cosa giusta
16    }
17  }
18  //POST1 = (per ogni a in [0..5] e per ogni b in [0..4], B[a][b]
19  //sse C[a] e' contenuta in C[..][b])
```

Il programma appena presentato può fare dei calcoli inutili nei due cicli più interni. Per il terzo ciclo è inutile continuare a cercare successivi elementi di `C[i]` *dopo averne trovato uno che non è presente in* `C[..][j]`*. Per il quarto ciclo è inutile continuare a cercare* `C[i][k]` *quando lo si è già trovato in* `C[..][j]`*. Le modifiche sono semplici: basta aggiungere una seconda condizione d'uscita a questi due* `for`*, come segue:*

```
1   for(int k = 0; k < 5 && OK; k++) //R3
2   {
3     bool trovato = false;
4     for(int z = 0; z < 6 && !trovato; z++) //R4
5       if(C[i][k] == C[z][j])
6             trovato = true;
7     if(!trovato)
8       OK = false;
9   }
```

Gli invarianti trovati prima continuano a valere. Le dimostrazioni dell'inizializzazione e invarianza di questi 2 cicli non cambiano, ma la condizione d'uscita dal ciclo va riconsiderata visto che ora si può uscire da questi due cicli anche con `k<5` *e* `z<6`*. Consideriamo quindi la dimostrazione del nuovo caso d'uscita:*
(R3 && !OK ⇒ POST3) *(si ricordi che fondamentalmente* ***POST3*** *asserisce che (*`!OK`* **va bene**)):*
R3 && !OK ⇒ (false sse ∀b ∈ [0..k-1], ∃a ∈ [0..5], C[i][b]=C[a][j])
che è equivalente ad asserire che:
(*) (∃b ∈ [0..k-1] tale che !∃a ∈ [0..5] tale che C[i][b]=C[a][j])
e questo implica che ***POST3*** *è vera, visto che* ***POST3*** *con* `OK` *uguale a* `false` *è* **(false sse ∀b ∈ [0..4] !∃a ∈ [0..5], C[i][b]=C[a][j])** *e la parte destra di questa equivalenza è falsa a causa di* ***(*)****.*

Sarebbe anche abbastanza facile eliminare la variabile booleana `trovato` *ed usare* `OK` *al suo posto.*

```
1   bool OK = true;
2   for(int k = 0; k < 5 && OK; k++)
3   {
4     OK = false;
5     for(int z = 0; z < 6 && !OK; z++)
6       if(C[i][k] == C[z][j])
7             OK = true;
8   } //risparmiando un condizionale
```

Basta riscrivere le asserzioni ***R4*** *e* ***POST4*** *sostituendo* `trovato` *con* `OK`*. Le prove restano valide.*

Il prossimo esercizio risolto estende il precedente considerando l'm-contenimento al posto del contenimento insiemistico normale. Nonostante l'esercizio sia molto simile al precedente, la sua prova di correttezza richiede un notevole sforzo aggiuntivo. L'esercizio illustra alcune tecniche di prova più complesse di quelle viste finora. Il lettore dovrebbe capire che non c'è mai una sola strada per produrre una dimostrazione di correttezza. Armati di tecniche, ma soprattutto di intelligenza e fantasia, dobbiamo di volta in volta trovare la notazione e l'approccio più adatto ad ottenere una dimostrazione convincente e semplice.

Esercizio Risolto 6.2 *Consideriamo ora brevemente la realizzazione di un programma che risolve un esercizio simile al precedente, ma in cui la condizione da verificare è la seguente:* `B[i][j]` *è* `true` *se la riga* `i`*-esima di* `C` *è m-contenuta nella colonna* `j`*-esima di* `C` *e altrimenti è* `false`*, vedi la Definizione 6.1 per la nozione di m-contenimento.*

Si tratta di lavorare sui due cicli più interni, quelli che eseguono il test di contenimento della riga nella colonna che deve diventare un test di m-contenimento.

L'idea è la seguente: ogni volta che trovo che `C[i][k]` *è uguale a* `C[z][j]`*, segno in qualche modo* `C[z][j]` *in modo da non usarlo più quando cercherò i successivi elementi di* `C[i][k+1..]`*. Come possiamo realizzare questo marcaggio? Una tecnica è di dichiarare un array di booleani* `bool T[6]` *(lungo quanto una colonna di* `C`*, riempito inizialmente di* `false` *e che, qualora si trovi che* `C[i][k]` *è uguale a* `C[z][j]`*, ricordi che* `C[z][j]` *non deve più essere usato nel futuro, assegnando a* `T[z]` *il valore* `true`*. Vediamo:*

```
1  bool T[6] = {};
2  bool OK = true;
3  for(int k = 0; k < 5 && OK; k++) //R3'
4  {
5    OK = false;
6    for(int z = 0; z < 6 && !OK; z++) //R4'
7      if(!T[z] && C[i][k] == C[z][j])
8      {
9        OK = true;
10       T[z] = true;
11     }
12     //POST4'
13 }
14 //POST3'
```

POST3'=(`OK` **sse** $\forall$**b** $\in$ `[0..4]`, $\exists$ **un distinto a** $\in$ `[0..5]` **tale che** `C[i][b]=C[a][j]`).
Questa asserzione ci fa capire che la tecnica adottata richiede di abbandonare il ciclo più interno non appena trovato(non marcato) l'elemento corrente della riga, altrimenti si marcherebbe non uno solo, ma tutti gli elementi della colonna che sono uguali ad esso.

Proviamo ad ottenere ***R3'*** *da* ***POST3'*** *applicando la solita ricetta:*
(`OK` **sse** $\forall$**b** $\in$ `[0..k-1]`, $\exists$ **un distinto a** $\in$ `[0..5]` **tale che** `C[i][b]=C[a][j]`)**&&**(**0**$\leq$`k`$\leq$**5**).
Questa asserzione ci sarà utile, ma certamente non è completa, infatti non dice nulla su `T`*, mentre l'invariante di questo ciclo deve asserire qualcosa su* `T` *che viene usato per realizzare il test di m-contenimento. L'invariante* ***R3'*** *completo segue. Conviene dividerlo in 3 parti in quanto, in caso* `OK` *sia falso, non si vuole asserire nulla su* `T`*:*
R3'=

- `OK` $\Rightarrow \exists[a_0,...,a_{k-1}]$, **a due a due diversi e tutti** $\in$ **[0..5] tali che** (`C[i][0]` `=C[`a_0`][j] ... C[i][k-1]=C[`a_{k-1}`][j]` && `T[`a_0`]=true ... T[`a_{k-1}`]=true)` **&&(gli altri elementi di** `T` **sono** `false`);
- `!OK` $\Rightarrow \nexists[a_0,...,a_{k-1}]$, **a due a due diversi e tutti** $\in$ **[0..5] tali che** (`C[i][0]` `=C[`a_0`][j] ... C[i][k-1]=C[`a_{k-1}`][j];`
- **(0**$\leq$`k`$\leq$**5)**

Nonostante non sia stato ricavato con la ***ricetta****, è facile convincersi che* ***R3'*** *assieme a* `(k=5 || !OK)` *implica* ***POST3'****:* `OK` *riflette correttamente il risultato del test di m-inclusione.*

A questo punto, quale potrebbe essere fa postcondizione ***POST4'*** *del ciclo interno (quello con indice* `z`*)? Questa asserzione deve asserire che se* `OK` *è diventato* `true` *allora in* `T` *è stato aggiunto esattamente un* `true`*(rispetto a prima). Per farlo dobbiamo usare un trucco che ci servirà anche altre volte. Con* `'T` *indichiamo il valore di* `T` *al momento in cui il quarto ciclo(quello con* `z`*) viene raggiunto per la prima volta. Non asseriamo nulla sul suo valore, ma introduciamo la notazione* `'T` *che permette di confrontare il valore di* `T` *all'uscita dal ciclo con quello all'inizio del ciclo,* `'T`*.*
POST4' *consiste di due parti:*

- (`OK` $\Rightarrow \exists$a, $\in$ **[0..5] tale che** (`C[i][k]=C[a][j]` && `T[a]=true` && `T[a]`$\neq$`'T[a]` && $\forall$b $\in$ **[0..5] con** b$\neq$a `T[a]='T[a]`));
- `!OK` $\Rightarrow \nexists$a, $\in$ **[0..5] tale che** (`C[i][k]=C[a][j]` && `'T[a]=false`);

POST4' *asserisce che il ciclo più interno assegna ad* `OK` *un valore che permette di estendere in modo corretto l'invariante* ***R3'*** *al prossimo elemento* `C[i][k]`*. L'invariante* ***R4'*** *si ottiene da* ***POST4'*** *applicando la ricetta e sostituendo* `z-1` *a 5. Che* ***R4'*** *sia invariante del ciclo più interno è facile da vedere. La condizione d'uscita è banale grazie alla* ***ricetta****. La condizione iniziale è altrettanto semplice a causa del fatto che* `z=0` *e* `OK=false`*.*

Capitolo 7

Funzioni

Capitolo 8

Esercizi sulle funzioni

8.1 Problemi (1) e (2)

8.1.1 Soluzione del problema (1)

8.1.2 Soluzione del problema (2)

8.2 Problemi (3) e (4)

8.2.1 Soluzione del problema (3)

8.2.2 Soluzione del problema (4)

8.3 Problema (5)

8.3.1 Soluzione del problema (5)

Capitolo 9

Estensioni del C++

Capitolo 10

Ricorsione

10.1 Programmazione ricorsiva

10.2 Esempi di ricorsione

10.2.1 Le prove induttive

Capitolo 11

Gestione dinamica dei dati e liste

Le variabili viste finora sono dette automatiche perchè la memoria RAM per contenere il loro R-valore viene allocata e deallocata automaticamente secondo la disciplina a pila descritta nella sezione 3.6. Ci sono casi in cui esrvono dati la cui vita e le cui dimensioni possono venire gestiti a seconda del bisogno ed al di fuori delle regole automatiche. L'abilità di creare e distruggere tali dati si dice **gestione dinamica dei dati** ed i dati prodotti in questo modo sono detti **dinamici**. Utilizzando la gestione dinamica dei dati costruiremo strutture dati ricorsive come le **liste concatenate** e gli **alberi binari** sulle quali vedremo moltissimi esempi di funzioni ricorsive. La gestione dinamica dei dati viene realizzata con alcune semplici istruzioni C++ che verranno introdotte nella prossima sezione. Inoltre, visto che i dati dinamici non seguono le regole di gestione dei dati automatici, essi non possono venire allocati nella pila dei dati automatici, ma vengono invece allocati in un ' area di memoria RAM diversa della pila che si chiama **heap**.

11.1 Gestione dinamica dei dati

Le istruzioni del C++ per la gestione dinamica dei dati sono le seguenti. L'operazione di allocazione della memoria è la funzione `new T`, in cui `T` è un tipo qualsiasi. Una tale invocazione richiede al sistema operativo di allocare una zona di memoria RAM sufficiente a contenere un valore di tipo `T`. In caso la richiesta venga soddisfatta, la memoria viene allocata in una parte della RAM chiamata **heap** ed essa resta a disposizione del programma finchè il programma stesso non la dealloca eseguendo il comando di deallocazione `delete`. Lo heap è un'area di memoria RAM diversa da quella che contiene la pila in cui vengono gestite le variabili automatiche. La funzione `new` ha il seguente prototipo: `T* new(T);`. L'invocazione di `new`, se ha successo, riserva sullo heap una sequenza di byte della dimensione giusta a contenere un valore di tipo `T` e restituisce l'indirizzo del primo byte di questa sequenza. Se invece l'allocazione non ha successo, il valore restituito è 0. In programmi in cuci la robustezza è importante conviene, dopo ogni invocazione di `new`, testare se il valore ritornato è 0 oppure no. Una `new` può fallire perchè la sua richiesta di memoria supera la disponibilità di memoria del PC al momento della richiesta; è raro che un tale fallimento avvenga per programmi come quelli che svilupperemo in questo corso. Vediamo un esempio di allocazione dinamica.

```
1     struct S{int x,y; char a,b;};
```

```
2     main(){
3         S* p = new S[10];
4         if(p==0) throw(..errore..) //errore d'allocazione, gestiscilo
5         p[0].x = 10;
6         p[2].a = 'h';
7     }
```

In questo frammento di programma allochiamo un array di 10 strutture `S`. L'array viene acceduto attraverso il puntatore `p` a cui possiamo applicare l'operatore di subscripting

11.2 Liste concatenate

11.3 Ricorsione ed iterazione

Capitolo 12

Alberi binari

12.1 Nozione di basse sugli alberi

12.2 Esercizi sugli alberi binari

12.3 Alberi binari di ricerca

12.4 Un esercizio da esame