Programmazione Imperativa in C++

Gilberto Filè

2008-2009

Indice

Prefazione

Questo testo intende insegnare a scrivere programmi semplici, chiari e corretti. Il linguaggio di programmazione usato è la parte imperativa classica del C++ e quindi non verrà considerata la parte orientata agli oggetti. La trattazione si prefigge di fornire al lettore un insieme di conoscenze che in ogni circostanza gli permetta di scegliere la soluzione più ragionevole per il problema dato, gli consenta di scrivere un programma che realizzi questa soluzione e, infine gli permetta di argomentare la correttezza del programma stesso. Il testo non assume alcuna conoscenza preliminare di Informatica e si propone come testo per il primo corso di Programmazione della Laurea (triennale) in Informatica o in Ingegneria Informatica. Oltre alla trattazione approfondita dei concetti fondamentali della programmazione imperativa, contiene molti esercizi, sia risolti che aperti, esercizi *da esame* e anche spunti per progetti di programmazione formativi (e anche divertenti).

Capitolo 1

Introduzione

Questo libro intende introdurre alla programmazione imperativa studenti alle prime armi. Il linguaggio scelto è la parte del C++ che non contiene le classi e gli oggetti. Non si tratta semplicemente del C, infatti considereremo diverse caratteristiche del C++ che non sono del C e che rendono più semplice e naturale il compito di scrivere programmi. Tra le caratteristiche del C++, che non fanno parte del C, che useremo, c'è l'input/output, i riferimenti, la gestione delle eccezioni, ed alcuni contenitori. Nel seguito chiameremo il linguaggio considerato **C++ imperativo** o, quando non genera confusione, semplicemente C++.

L'approccio adottato ha alcune caratteristiche peculiari che spieghiamo immediatamente. Per prima cosa, il libro non si propone di mostrare tutti i dettagli del C++ imperativo, ma focalizza l'attenzione sulle nozioni fondamentali trattandole in modo approfondito. Queste nozioni fondamentali si trovano in qualsiasi linguaggio imperativo evoluto e quindi la loro illustrazione ha un valore che va al di la del particolare linguaggio considerato. Per questo motivo abbiamo ritenuto che valesse la pena di approfondire queste nozioni in modo maggiore di quanto venga fatto nel tipico testo di introduzione alla programmazione.

Gli approfondimenti sono di vario tipo. Per esempio, confrontando costrutti del C e del C++ che hanno finalità simili, ma importanti differenze, cerchiamo di spiegare le differenze ragionando su come la concezione di *quello che i linguaggi di programmazione dovrebbero offrire* sia molto cambiata tra i primi anni '70, epoca in cui il C è stato definito, e la fine degli anni '80, periodo in cui il C++ è stato disegnato. Un altro esempio di approfondimento che permette durante l'intera trattazione di spiegare in modo più preciso molti aspetti importanti del linguaggio, è quello che riguarda la gestione dei dati manipolati dai programmi durante la loro esecuzione. Capire da subito questa gestione serve, in primo luogo, a capire a fondo la nozione di visibilità delle variabili e, in secondo luogo, a contrapporre a questa gestione *automatica* dei dati quella dinamica, che segue regole completamente diverse. Un'altra caratteristica importante e fuori dall'ordinario di questo testo è quella di discutere la correttezza di molti dei programmi presentati. Notevole sforzo viene fatto per ottenere che le dimostrazioni di correttezza siano sempre

di ausilio alla comprensione dei programmi stessi. Vale infine la pena di osservare che un importante capitolo del libro è dedicato alla programmazione ricorsiva, alle strutture dati ricorsive e dinamiche (cioè in grado di aumentare e rimpicciolire durante l'esecuzione del programma) ed alla gestione dinamica della memoria che serve per creare e gestire tali strutture. In questo capitolo vengono definite ed usate le liste concatenate (linked list), gli alberi binari e gli alberi binari di ricerca.

Per quanto riguarda quegli aspetti del C++ che non hanno trovato posto in questo libro, informazioni possono venire trovati su manuali di riferimento del linguaggio, alcuni dei quali sono disponibili su internet. Molte informazioni tecniche si possono trovare anche al sito (www.cplusplus.com).

Il libro è organizzato in 10 Capitoli. Oltre all'Introduzione, i Capitoli dal 2 al 5 presentano le nozioni di base del C++, cioè quelle necessarie a scrivere i primi programmi. Alcune importanti estensioni al linguaggio base, come i tipi definiti dall'utente e le eccezioni, sono introdotte nel Capitolo 6. Il Capitolo 7 introduce la programmazione ricorsiva, la gestione dinamica della memoria e le strutture dati ricorsive come le liste concatenate e gli alberi binari. Il Capitolo 8 approfondisce alcuni aspetti avanzati della programmazione come il sovraccaricamento di funzioni ed operatori e la gestione di programmi distribuiti su diversi file. Infine il Capitolo 9 contiene un cospicuo numero di esercizi svolti nei quali vengono usate molte delle nozioni introdotte nei Capitoli precedenti. Il Capitolo 10 contiene alcuni possibili esercizi d'esame, alcuni possibili progetti di programmazione su cui saggiare le capacità acquisite e anche molti esercizi aperti.

Concludiamo questo Capitolo introducendo alcune nozioni semplici, ma molto utili, su come è organizzato un PC e come sono fatti e come vengono eseguiti i programmi C++.

Struttura di un computer

L'architettura funzionale di un computer moderno è sorprendentemente simile a quella dei primi computer costruiti negli anni 1940. 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 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. Le operazioni che questi circuiti sono in grado di effettuare sono dette **operazioni macchina**. La memoria è costituita da una sequenza di byte ciascuno consistente di 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. La CPU è in grado di eseguire, tra le altre, operazioni che ricopiano il contenuto di byte di indirizzo dato in un registro e, viceversa, copiano il contenuto di un registro in byte di indirizzo specificato. Un valore può occupare più byte contigui e in questo caso diremo che risiede nella memoria all'indirizzo del primo di questi byte. 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.

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 successivi). 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

[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.

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 riescano a muoversi nelle diverse cartelle del file system in cui risiedono i loro programmi e possano anche lanciare i comandi di compilazione e di esecuzione dei programmi che saranno descritti dopo il seguente esempio.

Vediamo ora un esempio di programma molto semplice le cui azioni sono spiegate nel seguito.

Esempio 1.1

```
#include<iostream>
using namespace std;
main()
{
 int x, y;
 cout << ''inserire 2 interi'';
 cin >> x >> y;
 cout<<''valore di x=''<< x <<''valore di y=''<< y;
}
```

Esercizio 1.2 *Scrivete il precedente programma sorgente sul vostro editore di testo favorito (si consiglia* `xemacs`*), compilatelo ed eseguitelo utilizzando i comandi spiegati qui di seguito.*

Assumiamo che il file di testo contenente il programma sorgente si chiami `pippo.cpp` e che risieda nella cartella `pluto`. Usiamo nomi come `pippo` e `pluto` per indicare che questi nomi sono arbitrari. Al contrario, l'estensione `.cpp` in `pippo.cpp` è necessaria perchè indica al compilatore che il file contiene un programma C++. Se la cartella corrente è `pluto`, il programma `pippo.cpp` viene eseguito con il comando `g++ pippo.cpp`. Questo comando invoca il compilatore C++ che compila il programma `pippo.cpp` e produce il corrispondente programma oggetto in un file che si chiama `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 nome pippo.C`. Questo comando metterà il codice oggetto nel file `nome` che verrà eseguito con `./nome`. 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 una coppia di interi, per esempio `3` e `-2` (separati da uno spazio o da un invio), e dato l'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.

Ci sono parecchie cose da spiegare nel programma appena descritto:

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 perchè 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 bytes. In inglese sequenza si può dire "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` c'è 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`.

Capitolo 2

Tipi predefiniti e variabili

Come (quasi) tutti i linguaggi di programmazione, anche il C++ ci *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 nel rendere i linguaggi di programmazione non dei meri strumenti per codificare soluzioni trovate indipendentemente dal linguaggio, ma strumenti di supporto alla ricerca delle soluzioni e anche validi aiuti per l'individuazione degli errori che fatalmente queste soluzioni contengono.

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 per stabilire qual'è l'insieme dei valori consentiti per ciascun tipo. Infatti è importante chiarire subito che, benchè , per esempio, l'insieme degli interi sia infinito, così non può essere per gli interi che devono essere manipolati da un computer che è una macchina la cui RAM è finita. Ogni compilatore consentirà solo un insieme finito di interi la cui dimensione sarà determinata da quanti byte vengono messi a disposizione. Lo stesso si applica a qualsiasi tipo.

TIPO	KEY	OPERAZ	VALORI	BYTE	RAPPR
intero	`int`	+ -	12 -4	4	comp a 2
reale	`double`	+ -	12.3 2.3e4	8	floating
reale	`float`	+ -	12.3f 2.3e4f	4	floating
caratteri	`char`	+ -	`'a' '\t'`	1	comp a 2
booleani	`bool`	&& || !	true false	1	0 1
vuoto	`void`				

Tabella 2.1: Legenda delle colonne: KEY=keyword, OPERAZ= tipiche operazioni, VALORI= esempi di valori, BYTE= n. di byte usati dal compilatore GNU 4.1, RAPPR=rappresentazione interna

- Per il tipo intero, l'intervallo dei valori rappresentabili con 4 byte (dal compilatore GNU 4.1) è, $-2^{31}, \ldots, 2^{31}-1$; 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 varianili 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**. L'overflow può venire segnalato oppure no a seconda del computer. Se l'overflow non viene segnalato, allora il risultato dell'operazione che lo ha causato è senza senso e, nuovamente, il suo valore potrà dipendere dal computer. Naturalmente è preferibile che tali errori vengano segnalati, infatti, tali errori possono produrre valori interi plausibili che quindi mascherano l'errore.

- 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 `2.3e4` indica il numero `double` $2.3*10^4$.

- Le operazioni aritmetiche e quelle di confronto si possono applicare sia a valori interi che `float` che `double`. Conviene rendersi conto immediatamente che non si tratta veramente 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 0 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...127. 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, cf. la Tabella 2.1, come per esempio: `'v', '?'` e `'!'`. 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 rapresentati con interi è un'eredità del C. Infatti nel C semplicemente non esiste il tipo `bool` ed esso è realizzato con interi con la seguente rappresentazione: `false` è rappresentato con l'intero 0, mentre ogni intero diverso da 0 viene visto come il valore `true`. Visto che il C++ è compatibile col C, la stessa convenzione si

applica anche al C++. Come per i caratteri, a causa di questa rappresentazione interna con valori interi, si possono applicare ai valori booleani non solo le operazioni specifiche di questi valori, ma anche 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 && 0 || -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 6.2.

In generale, per sapere quanti bytes occupa un qualsiasi tipo `T` possiamo utilizzare la funzione di libreria `sizeof`, che invocata nel modo seguente, `cout<<sizeof(T);` stampa il numero di byte usati per i valori di 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: `''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 7.6. Le stringhe alla C sono sempre racchiuse tra doppi apici. Esse possono contenere qualunque carattere, anche quelli di controllo. Per esempio, se nell'Esempio 1.1 sostituissimo nella seconda riga del main la seguente stringa: `''inserire 2 interi\n''` a quella originale, potremmo osservare che l'output prodotto dal nuovo programma è un pò più ordinato di prima. È troppo presto per capire esattamente il tipo delle stringhe alla C. Lo capiremo nella Sezione 5.1 in cui vengono illustrati i puntatori.

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 dal compilatore e comunque rischiano di diventare un peso. Il C++ è sensibile alla differenza tra i caratteri minuscoli e maiuscoli. Quindi `pippo` è una variabile diversa da `Pippo` e sono entrambe diverse da `PIPPO`.

In un programma C++ ogni variabile viene associata ad un particolare tipo ed, in linea di principio, ad essa possono venire associati solo valori di quel tipo. L'associazione variabile-tipo avviene attraverso un apposito comando del C++ che si chiama **dichiarazione**. Per esempio, la dichiarazione `int x, pippo;` specifica che `x` e `pippo` sono variabili di tipo `int` (o semplicemente sono variabili intere). Possiamo (quindi non è necessario) assegnare un valore alle variabili al momento della dichiarazione 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**. Una variabile di tipo `char` è dichiarata nel modo seguente: `char y1;`. Essa può venire inizializzata per esempio nel modo seguente: `char y1='a';` I nomi delle variabili devono essere diversi dalle parole chiave (keyword) del C++ come, per esempio, `int` e `char` e molte altre che incontreremo 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'inizializzazion è sempre richiesta al momento della dichiarazione. Il valore assegnato (ovviamente) non potrà mai venire modificato (pena un messaggio d'errore del compilatore).

Il vantaggio di usare i nomi costanti rispetto ad usare direttamente i valori è duplice: in primo luogo il programma ne guadagna in leggibilità in secondo luogo, qualora ci si accorgesse in un secondo momento, che il valore costante andasse modificato, la modifica da apportare al programma sarebbe più semplice e sicura.

2.3 Che succede durante l'esecuzione di un programma

È utile, prima di iniziare a descrivere i comandi del linguaggio C++, capire cosa succede nel computer 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 dati manipolati dal 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 un'entità statica, ma al contrario essa cambia perchè in ogni momento cambia l'insieme delle variabili attive. Cerchiamo di capire brevemente in che modo. In generale un programma è costituito da blocchi annidati che con-

tengono dichiarazioni di variabili (e istruzioni). Per esempio un annidamento di 3 blocchi è il seguente:

```
(*)
{
int x=0; ...(0)
 {
  int y=1; ...(1)
   {
     int z=2;...(2)
   }
  ...(1')
 }
...(0')
}
(*')
```

Assumiamo che l'esecuzione inizi al punto (*). A questo punto nessuna variabile è attiva. Quando l'esecuzione entra nel blocco esterno, la variabile `x`, definita in questo blocco, diventa attiva, mentre le variabili `y` e `z` non sono ancora attive. Lo potranno diventare solo se l'esecuzione entra nel secondo e nel terzo blocco, rispettivamente. Ma cosa significa che una variabile diventa attiva? Significa che le viene attribuita un'area di memoria RAM in cui verrà custodito il suo valore. Questa attribuzione di memoria viene chiamata **allocazione della variabile**. Quindi quando l'esecuzione raggiunge il punto (0), sarà allocata la variabile `x`, quando raggiunge il punto (1), saranno allocate la `x` e la `y` e in (2) sarà allocata anche la `z`. É molto importante capire che in (1'), dato che l'esecuzione è uscita dal blocco più interno, la variabile `z` dovrà diventare inattiva. Questa operazione di togliere alla variabile la zona di memoria che gli era associata si chiama **deallocazione della variabile**. Quindi in (1') saranno attive solo le variabili `x` e `y`. Non sarà una sorpresa il fatto che in (0') anche `y` verrà deallocata e in (*') tutte e 3 le variabili saranno deallocate.

Da questa descrizione dovrebbe essere facile vedere che la gestione della memoria RAM dei dati segue un andamento a pila: quando si entra in un blocco, le variabili dichiarate nel blocco vengono allocate sulla cima della pila e le si dealloca dalla cima della pila quando si esce dal blocco stesso. La Figura 2.1 illustra questa gestione a pila. Visto che l'allocazione e la deallocazione delle variabili dipende solamente dal blocco in cui le variabili sono dichiarate, queste variabili sono dette `automatiche` per contrastarle con allocazioni che seguono una politica diversa e che verranno presentati in Sezione **??**. Una variabile automatica è attiva nei periodi che vanno dalla sua allocazione alla sua deallocazione. La `visibilità` di una variabile è sempre il blocco in cui è dichiarata, dalla posizione della dichiarazione in poi.

Nell'esempio di Figura 2.1, per semplicità, abbiamo considerato una sola dichiarazione per blocco. Il caso di più dichiarazioni in un blocco viene gestito in

La pila dei dati cresce e diminuisce dinamicamente

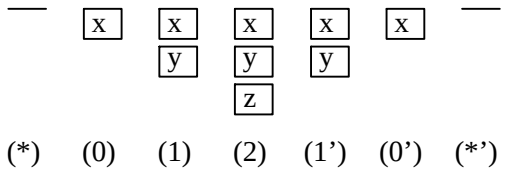

Figura 2.1: Allocazione e deallocazione della memoria durante l'esecuzione di un programma

Figura 2.2: il fenomeno dell'oscuramento delle variabili

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 2.1(2). D'altra parte, se all'interno del blocco (2) ci fosse una nuova dichiarazione di `x`, per esempio `double x;`, prima dell'istruzione che usa `x`, allora l'istruzione farebbe riferimento a questa nuova `x` che è allocata in cima alla pila, come mostra la Figura 2.2(a). Nella Figura 2.2(a) si deve anche osservare che entrambe le variabili `x` sono allocate, ma quella in cima alla pila *oscura* l'altra. Quando l'esecuzione esce dal blocco (2), la pila diventerà quella di Figura 2.2(b) e quindi se in (1') ci fosse un'istruzione che usasse `x`, essa farebbe riferimento a quella allocata in fondo alla pila.

Per una variabile è importante distinguere tra il valore associato alla variabile e l'indirizzo di memoria RAM (sulla pila dei dati) in cui quel valore viene immagazzinato. Il valore associato ad una variabile è chiamato il suo **R-valore**, mentre l'indirizzo di memoria in cui l'R-valore è immagazzinato si chiama **L-valore** della variabile. La lettera R sta per Right (destro) e la L per Left (sinistro). Il motivo di questi nomi verrà chiarito quando esamineremo l'istruzione di assegnamento. Esiste un'importante differenza tra l'R- e l'L-valore di una variabile. Mentre l'R-valore di una variabile viene "deciso" dal programma che usa la variabile e cambia (in generale) durante l'esecuzione del programma, l'L-valore viene deciso dal Software che gestisce la pila dei dati relativa al programma e quindi non può venire deciso né modificato dal programma. In compenso il programma può conoscere ed usare l'L-valore delle sue variabili. Una variabile viene detta **indefinita** quando es-

sa è dichiarata, ma non inizializzata. Essendo dichiarata, la variabile viene allocata nella pila dei dati come abbiamo visto, ma i byte a sua disposizione contengono una sequenza di bit casuale, cioè quello che casualmente è presente al momento dell'esecuzione 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 rischia di essere presa come un R-valore *buono*, senza alcuna segnalazione d'errore.

2.4 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. Sottolineamo un fatto semplice, ma importante: le espressioni nei programmi vengono sempre valutate e questo procedimento (se ha successo) produce il **valore** dell'espressione. É possibile che la valutazione di un'espressione non abbia successo? La risposta è si e questo può avvenire per vari motivi. La ragione più semplice è che l'espressione richieda di eseguire operazioni indefinite come la divisione per 0. Questo errore produce generalmente una terminazione anormale del programma in esecuzione. Un'altro possibile motivo che impedisca la valutazione di un'espressione 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 tale espressione non viene (in generale) compilato e non potrà quindi venire eseguito. Un caso ancora diverso è quello in cui l'espressione da valutare contiene variabili indefinite. In questa situazione la valutazione dell'espressione può venire 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 perchè 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. È chiaro che invece trasformare `3.14` in un intero ci fa 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 operate dal compilatore vengono chiamate **conversioni automatiche**. Maggiori dettagli su come avvengono le conversioni automatiche e su cosa significhi in pratica convertire un valore da un tipo ad un altro tipo sono dati in Sezione 7.4.

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 che contengono gli operatori logici `&&` e `||` il C++ usa la cosidetta 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 il valore dell'intera espressione è `false`, 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 il suo valore non serve per decidere che il valore dell'intera espressione è vero.

2.5 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 loro. 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 2.4.

Nonostante la parte ad 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 sia dati che le funzioni per elaborare questi dati. Lo scopo è quello di proteggere i dati e di non permettere che essi vengano alterati in modo arbitrario, ma solo con le funzioni predisposte allo scopo. La nozione di classe e la relazione di ereditarietà tra di esse sono un importante ausilio alla realizzazione di programmi corretti e modificabli.

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 del C++ l'abbiamo avuto considerando i tipi predefiniti. Per esempio, abbiamo visto che in C++ i valori di tipo carattere e di tipo booleano vengono rappresentati con interi e che questo fatto rende possibile applicare a questi valori degli operatori che sarebbero altrimenti inappropriati. Infatti, mescolando tipi diversi si rischia di perdere la capacità di scoprire errori grazie ad essi. Oltre a quelle appena segnalate, nel C++ ci sono altre debolezze ben più gravi che vedremo successivamente e che rendono il C++ un linguaggio **insicuro rispetto ai tipi**, che significa che è possibile (e purtroppo anche molto facile) scrivere programmi C++ che (durante l'esecuzione) assegnano ad una variabile di un certo tipo degli R-valori di tipo diverso (e incompatibile) senza che queste violazioni siano segnalate né durante la compilazione né durante l'esecuzione.

Capitolo 3

Istruzioni di base

Questo Capitolo è dedicato alla descrizione delle istruzioni fondamentali del C++ ed in realtà di qualsiasi linguaggio di programmazione imperativo come per esempio il Fortran, il Pascal ed il C. Tratteremo in primo luogo le istruzioni di input e di output e successivamente esamineremo l'assegnazione, il condizionale ed il comando iterativo `while`.

In generale un'istruzione ha l'effetto di modificare l'R-valore di qualche variabile. Nel seguito, per spiegare il significato delle istruzioni, useremo spesso la nozione di **stato del calcolo** che intuitivamente è una fotografia dell'evoluzione del calcolo di un dato programma in un certo momento. Più formalmente, lo stato del calcolo verrà rappresentato con una funzione **S** tale che per ogni variabile `x` del programma, `S(x)` è l'R-valore di `x` nel momento fotografato da **S**. Il caso che `x` sia indefinita viene rappresentato con `S(x)`$=\perp$. Ovviamente ogni istruzione può modificare lo stato e questa modifica è proprio il significato dell'istruzione stessa.

3.1 Input/Output

I programmi devono poter scambiare informazioni con l'esterno ed a questo servono le operazioni di input/output (i/o). Ci limiteremo a spiegare le operazioni più semplici che poi useremo sempre nel testo, per operazioni maggiormente sofisticate il lettore può trovare informazioni sul sito www.cplusplus.com. Un programma comunica con l'esterno per mezzo di dispositivi molto diversi: l'input standard è la testiera, l'output standard è il video, ma un programma potrebbe voler scrivere o leggere da un dispositivo USB o da CD o DVD o semplicemente da un file nella memoria del computer (o anche su qualche altro computer raggiungibile da 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 **file**. Vediamo per prima cosa cosa sono i file e poi ci occuperemo di come un programma può avere accesso ad un file e vedremo anche le operazioni più semplici di i/o che si applicano ai file.

3.1.1 I file

Un file è semplicemente una sequenza di dati che terminano con un caarattere 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 sono fatti per essere letti dagli umani e sono una rappresentazione fedele di un testo qualsiasi. In realtà la rappresentazione dei testi è mediata dalla codifica ASCII estesa che viene compensata dagli editori di testo che noi usiamo per leggere questi file.

I file **binari** invece sono sempre sequenze di byte (tutto è una sequenza di byte in un computer), ma questi byte non vanno interpretati come caratteri, ma come rappresentazioni interne al computer di valori, per esempio valori interi, reali eccetera. A prova di questo fatto, se si cerca di aprire un file binario con un editore di testo si ottiene un testo assolutamente incomprensibile, perchè l'editore interpreta ogni byte secondo la codifica ASCII estesa anche se non lo è e quindi il risultato è una sequenza di caratteri senza senso. Per capire quanto appena detto, si deve osservare che ogni byte (e quindi anche quelli di un file binario) può venire interpretato come il codice di un carattere secondo la codifica ASCII estesa.

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 verrebbe 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 da 4 bytes 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 contento 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, la trattazione degli `stream` non verrà fatta in questo testo. Qui spiegheremo solo come usarli. In generale un oggetto contiene dati e offre funzioni per manipolare questi dati. Nel caso degli `stream` le funzioni che offrono sono le operazioni di i/o sui file.

Per usare file in un programma si devono inserire alcune istruzioni nel programma:

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

ii) per associare uno `stream` ad un file che si chiama, per esempio, *pippo* e che vogliamo usare in input, è necessario inserire nel nostro programma la dichiarazione: `ifstream XX(''pippo'');`. Questa dichiarazione crea lo `stream XX` e lo associa al file *pippo* che viene simultaneamente **aperto**, cioè i suoi dati sono ora a disposizione del nostro programma;

iii) per aprire il file *minni* per scriverci sopra informazioni, insomma se vogliamo usarlo per l'output, allora dobbiamo inserire nel programma la dichiarazione: `ofstream YY(''minni'');` che apre il file e lo mette a disposizione del programma.

È bene considerare che 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`. 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 virgolette il cammino per raggiungere il file dalla directory corrente.

Se il file *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.

Per rendere facili le operazioni di i/o che si riferiscono alla tastiera ed al video, 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 (video). Nell'Esempio 1.1, abbiamo usato questi 2 `stream`.

3.1.3 Operazioni di i/o

Gli `stream` di i/o offrono molte operazioni di i/o. Ne studieremo solo 2 per l'input e 2 per l'output.

- visto che consideriamo solo file di testo, cioè composti di sequenze di caratteri, è naturale leggere un carattere alla volta e scrivere un carattere alla volta. Se `XX` e `YY` sono, rispettivamente `stream` di input e 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`, effettuata dopo l'apertura del file, legge il primo carattere del file, poi il secondo 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`. Sottolineamo che quello che si legge con la `get` è un byte che è la codifica ASCII estesa di un carattere e lo stesso vale per quello che si scrive con la `put`. Vale la pena di osservare bene la sintassi delle 2 operazioni appena viste, per esempio, `YY.put(c)`: `YY` è l'oggetto di tipo `stream` e `put` una funzione di questo oggetto. Il punto in `YY.put(c);` indica proprio l'appartenenza di `put` a `YY` e si chiama operatore di **appartenenza**.

- 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 come funzionano. Come già detto, il contenuto del file di testo è visto come una sequenza di stringhe di caratteri separate da caratteri di separazione (spazio e invio o accapo). Ogni stringa rappresenta un valore per esempio un numero intero oppure un reale ed i separatori sono caratteri spazio oppure di accapo. L'istruzione `XX >> x;` in cui `x` è una variabile intera, causa la lettura 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 nonterminazione 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 non trovare caratteri numerici 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 non quello che succede con l'operazione `get` è discusso nell'Esempio 3.2.

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 degli interi (4 byte complemento a 2) di 12 e questo diventa l'R-valore di `x`. Questo significa che il tipo della variabile che viene letta (cioè `x` nell'esempio) determina cosa ci si deve aspettare sul file e come questo viene tradotto nella rappresentazione interna al computer della stringa letta. Quindi se nel file, anzichè `'1'` e `'2'` ci fosse `'1'`, `'.'` e `'2'`, allora l'input avrebbe un comportamento anomalo (provate leggendo 2 variabili intere) perchè il tipo `int` non prevede un punto nei suoi valori, mentre se `x` avesse tipo `double` tutto funzionerebbe normalmente. Nel caso in cui `x` avesse tipo `double` ed il file contenesse `'1' '2' ' ' '3' '4'`, allora verrebbe calcolata la rappresentazione floating point 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).

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 illeggibile.

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. È importante riuscire ad accorgersi quando questa situazione accade. Lo `stream` associato al file ci offre la funzione `eof()` (che 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 hanno un comportamento diverso rispetto al raggiungimento dell'end of file. Questo fatto è illustrato nell'Esempio 3.2.

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 del C++ è **l'assegnazione**. L'assegnazione ha la forma, `x=EXP;` in cui `x` è una variabile e `EXP` è un'espressione. 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 `EXP` sostituendo ogni occorrenza di ciascuna variabile `y` in `EXP` con il suo R-valore del momento, cioè **S**`(y)` (si osservi che tra le variabili che appaiono in `EXP` può esserci anche `x` stessa). Il valore V diventa l'R-valore di `x`, cioè V viene immagazzinato nella RAM all'indirizzo che è l'L-valore di `x`. Quindi l'effetto dell'esecuzione dell'assegnazione `x=EXP;` è quello di cambiare **S** in un nuovo stato **S'** tale che per ogni variabile `z` diversa da `x`, **S'**`(z)`=**S**`(z)`, mentre per la variabile `x`, **S'**`(x)`=V.

La descrizione di quanto avviene durante l'esecuzione di un'assegnazione spiega la scelta dei nomi R- e L-valore delle variabili. Infatti abbiamo appena visto che nell'esecuzione di `x=EXP;`, la valutazione dell'espressione `EXP`, che si trova alla *destra* (R=Right=destra) dell'assegnazione, fa uso degli R-valori delle variabili che compaiono in `EXP`, mentre il valore V di `EXP` viene immagazzinato nella RAM all'indirizzo che è l'L-valore della variabile che appare alla *sinistra* dell'assegnazione (L=Left=sinistra). Ecco quindi finalmente motivati i nomi R- ed L-valore delle variabili.

Il C++ consente di scrivere assegnazioni in forme abbreviate come, `potenza* = 2;` e `esponente++;`. L'assegnazione `potenza* = 2;` è equivalente a `potenza = potenza*2;` mentre `esponente++;` è equivalente ad `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 maggiore efficienza. 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; int x = (x++) * 2;` con quello di eseguire `int x = 1; int 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 altri tipi, per esempio con `int, char` e altri, (con `0` equivalente a `false` e tutti gli altri interi/caratteri equivalenti a `true`), è sufficiente che `EXP` abbia un valore di tipo interpretabile come booleano. L'esecuzione di questa istruzione in un certo stato del calcolo **S** ha il seguente effetto: viene calcolato il valore dell'espressione booleana `EXP` e se questo valore è `true` 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 fino a 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 come per il condizionale `EXP` rappresenta un'espressione booleana (o interpretabile come booleana) e `C` è un blocco di istruzioni. L'effetto di eseguire questa istruzione in uno stato del calcolo **S** è come segue: se in

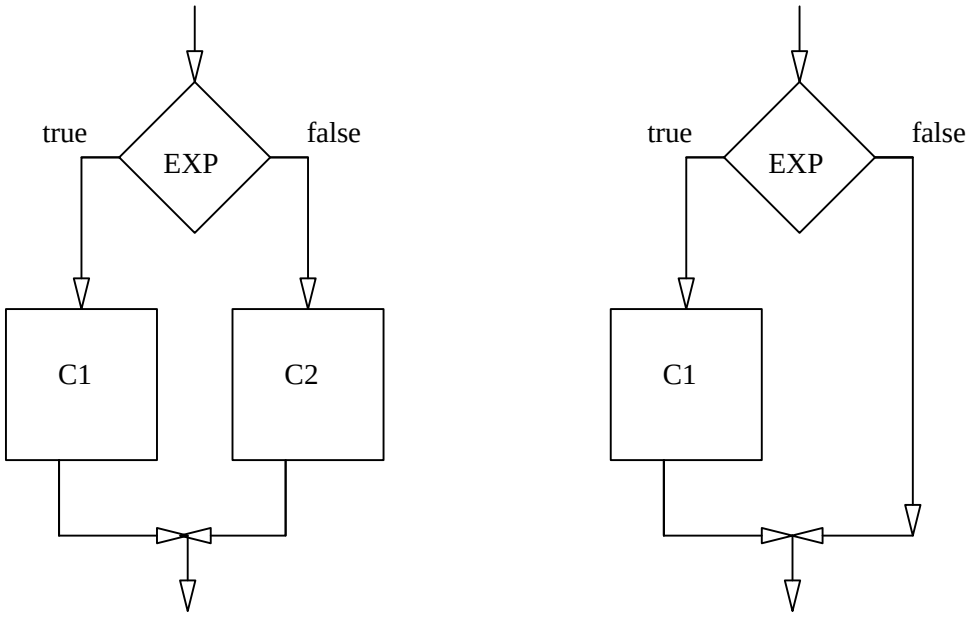

Figura 3.1: schema a blocchi dell'`if-then-else` e dell'`if-then`

S `EXP` ha valore `true` allora viene eseguito il blocco `C` (detto il **corpo** del `while`). 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 `while`. Lo stato del calcolo 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 `while` e questo fenomeno viene chiamato **un ciclo infinito**. Ovviamente quando questo succede siamo in presenza di un programma errato. Il diagramma a blocchi del `while` 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. del corpo.

3.5 Esempi e correttezza dei programmi

Illustriamo ora le nuove istruzioni appena introdotte realizzando alcuni semplici programmi. Di alcuni di essi discuteremo anche la correttezza. Iniziamo modificando l'Esempio 1.1.

Esempio 3.1 *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*

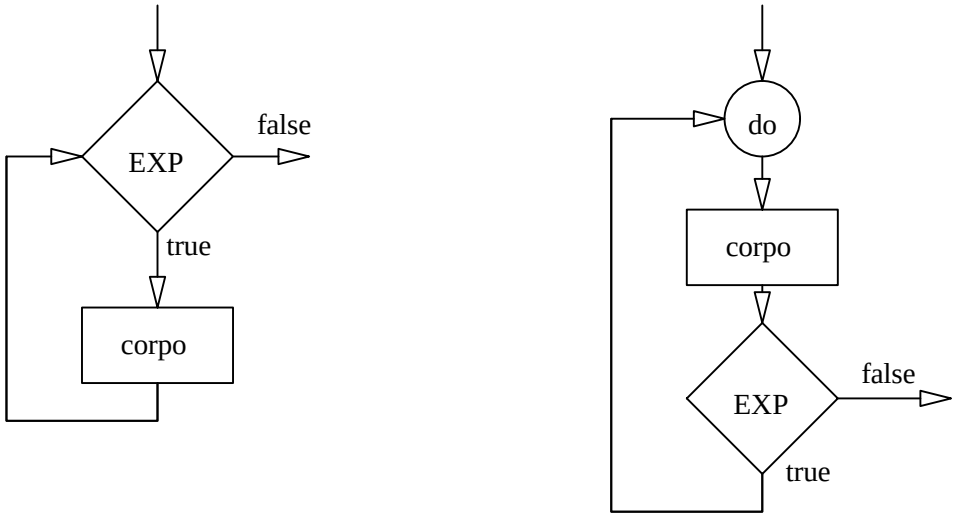

Figura 3.2: schema a blocchi del `while` e del `do-while`

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 avessimo letto i valori* `5` *e* `-4` *le cose sarebbero andate diversamente? In che modo?*

```
#include<iostream>
using namespace std;
main()
{
 int x, y;
 cout << ''inserire 2 interi'';
 cin >>x >> y;
 cout<<''valore di x=''<< x <<''valore di y=''<< y;
 int SOM;
 SOM= x+y;
 if(SOM > 0)
   y=SOM;
  else
```

```
   x=SOM;
 cout<<''valore di x=''<< x <<''valore di y=''<< y;
}
```

Esempio 3.2 *Questo esempio serve ad illustare e confrontare operazioni di i/o viste in Sezione 3.1.3. Si tratta di scrivere un programma che apre come input un file di testo* pippo *e legge il suo contenuto carattere per carattere ricopiando i caratteri letti sul file di output standard (il video, associato allo* `stream cout`*). Questa operazione di ricopiatura viene realizzata attraverso l'uso del comando iterativo* `while`*. Visto che si desidera ripetere l'operazione di ricopiatura per tutti i caratteri del file, il ciclo* `while` *usa* `PP.eof()` *come condizione di fine ciclo, cioè il ciclo termina quando il file è finito. Questo esempio ed il prossimo Esercizio 3.3 illustrano che su un file, oltre ai caratteri "buoni" ci possono essere caratteri di separazione (per esempio, enter e spazio) e che a causa della presenza di questi caratteri, le diverse operazioni trovano l'end of file in modi diversi.*

```
#include<iostream>
using namespace std;
#include<fstream>
main()
{
  ifstream PP("pippo");
  if(PP !=0) // se l'apertura ha avuto successo
   while (! PP.eof())
    {
      char c;
      c=PP.get();
      cout << c;
    }
   else
    cout<<"errore nell'apertura del file"<<endl;
}
```

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 un carattere alla volta il contento del file e lo stampa immediatamente sul video. Cambiando il nome pippo *è possibile usare questo programma per copiare a video ogni file (di testo).*

Esercizio 3.3 *Copiare il programma appena dato in un file con nome* pippo*, compilatelo ed eseguitelo. Vedrete comparire sul video il programma stesso. L'unica differenza dovrebbe essere un carattere starno alla fine della stampa. È il risultato della lettura fallita, ma necessaria a trovare l'eof. Modificate successivamente il*

programma sostituendo l'operazione di lettura con la seguente: `PP >> c;`. *Ricompilate ed eseguite. Studiate la differinza tra il nuovo output su video e quello precedente. Dovrebbe essere possibile apprezzare il fatto che la* `get` *legge tutti i caratteri del file, mentre la* `>>` *salta i caratteri di separazione (spazi e accapo). D'altro canto, questo secondo programma non dovrebbe aggiungere lo strano carattere in fondo alla stampa. Quindi, il fatto che l'operazione* `>>` *salta i separatori, permette di trovare l'eof immediatamente dopo l'ultima lettura* buona.

Capitolo 4

La Correttezza dei Programmi

In questo capitolo mostreremo come sia possibile dimostrare in modo convincente che i nostri programmi sono corretti, cioè fanno effettivamente quello per cui li abbiamo scritti. Iniziamo con un esempio. Successivamente descriveremo una regola generale per trattare la correttezza del `while` e la useremo per dimostrare la correttezza di alcuni semplici programmi.

Il lettore si chiederà quale sia il vantaggio di fare prove di correttezza dei programmi. Per rispondere conviene partire dall'osservazione di cosa succede quando le prove non vengono fatte. In questo caso il programmatore realizza il suo programma seguendo un'intuizione che in generale non è precisa e poi controlla che il programma sia corretto eseguendo alcuni test con input diversi. Questa maniera di procedere, molto comune, ha una debolezza. Essa non sviluppa in modo chiaro l'intuizione alla base del programma e solo questo chiarimento può far capire se l'intuizione è corretta oppure sbagliata. Invitiamo il lettore a riconoscere nella trattazione che segue, i momenti in cui l'idea intuitiva trova la sua formalizzazione precisa.

Esercizio Risolto 4.1 *Vogliamo calcolare il minimo esponente intero tale che 2 elevato a quell'esponente sia maggiore o uguale ad un intero positivo letto dallo stream di input. Un programma che risolve questo problema segue. Nel resto dell'esempio lo chiameremo programma 4.1.*

```
#include<iostream>
using namepace std;
main()
{
 1. int X, potenza=1, esponente=0;
 2. cout<<''Digita un valore positivo:'';
 3. cin>>X;
 4. while(X>potenza)
  {
```

```
 5. potenza*=2;
 6. esponente++;
  }
 7. cout << ''l'esponente e':''<< esponente << endl;
           // endl stampa un accapo
}
```

Questo programma sembra ragionevole, si basa sull'intuizione che dobbiamo provare successive potenze del 2 fino a che non troviamo quella che uguaglia o supera `X`*. In questo modo dovremmo determinare l'esponente cercato. Ma è veramente corretto? E poi che significa in realtà che sia corretto? Cerchiamo innanzitutto di rispondere a questa seconda domanda. Poi vedremo com'è possibile dimostrare che il programma è veramente corretto. Il primo passo è quello di specificare una coppia di asserzioni: la* **Precondizione** *(***PRE***) e la* **Postcondizione** *(***POST***) del nostro programma. In generale, un'asserzione relativa ad un programma è una qualunque affermazione sulle relazioni che legano i valori delle variabili del programma. Un'asserzione viene legata ad un punto del programma e l'idea è che essa deve essere vera ogni volta che l'esecuzione raggiunge quel punto del programma. Ricordando la nozione di stato del calcolo, data nell'introduzione di questo Capitolo, un'asserzione caratterizza un insieme di stati del calcolo: gli stati nei quali le variabili del programma hanno valori che rendono vera l'asserzione.*

La precondizione **PRE** *descrive la situazione a partire dalla quale il programma viene eseguito ed è legata al punto iniziale del programma. La postcondizione* **POST** *descrive la situazione dopo che il programma termina ed è legata al punto finale del programma. Quindi la postcondizione descrive quello che vogliamo che il nostro programma calcoli e la dimostrazione della correttezza del programma consiste nel dimostrare che ogni volta che il programma termina, lo stato del calcolo raggiunto renderà vera la postcondizione. Per il nostro esempio, la precondizione* **PRE** *e la postcondizione* **POST** *sono le seguenti:*

PRE = **(lo** `stream cin` **contiene un intero maggiore di 0)** *asserisce che solo valori interi positivi verrano letti ed assegnati a* `X`*;*

POST = **(**`esponente` **è il minimo intero tale che** $2^{\texttt{esponente}} \geq$ `X`**)** *asserisce che alla fine del calcolo la variabile* `esponente` *ha il valore che vogliamo calcolare.*

Si osservi che **PRE** *considera lo stream* `cin` *e non le variabili del programma, ma visto che il contenuto di* `cin` *viene letto in* `X`*, si tratta di un'estensione ovvia. Per riportarci al caso normale basterebbe eliminare l'input ed assumere che* `X>0`*.*

Vogliamo dimostrare che qualora l'esecuzione del Programma 4.1 inizi da uno stato del calcolo che soddisfa **PRE** *, allora, se l'esecuzione termina, essa terminerà in uno stato del calcolo che soddisfa* **POST**. *Questa proprietà si esprime con la formula:* **PRE** `<Programma 4.1>` **POST**

Un passo importante per riuscire a fare questa dimostrazione è di spezzare il programma in pezzi tali che la loro dimostrazione sia semplice. Nel nostro esempio avremo 2 pezzi: la parte di inizializzazione, costituita dalle istruzioni 1, 2 e 3 ed il ciclo `while`*, costituito dalle istruzione 4, 5, e 6. L'istruzione 7 è di output,*

quindi non calcola nulla e viene ignorata dalla dimostrazione. Chiamiamo i 2 pezzi di programma `INIZ` *e* `WHILE`*. Dovremo dimostrare che:* **PRE** `INIZ` **P** *e* **P** `WHILE` **POST**, *dove* **P** *è un'opportuna asserzione. È semplice capire che la composizione delle 2 dimostrazioni prova che:* **PRE** `INIZ WHILE` **POST** *e quindi che* **PRE** `<Programma 4.1>` **POST**

Trattare i cicli è sempre la parte più dura, quindi partiamo da li. La dimostrazione di correttezza dei cicli si fonda sull'idea di **invariante**. *Un invariante di un ciclo* `while` *è un'asserzione che è vera ogni volta che l'esecuzione è sul punto di iniziare ad eseguire il ciclo. L'invariante di* `WHILE` *formalizza l'intuizione precedente. La prima volta che si esegue il ciclo è un caso speciale che consideriamo dopo, ma consideriamo la seconda, o la terza, o la n-esima volta. Se siamo all'inizio ciclo per l'n-esima volta (n>0) significa che c'è stata un'esecuzione precedente e che in quell'occasione* `potenza` *ha un valore* **v** *che è minore di quello di* `X`*. Altrimenti l'esecuzione sarebbe uscita dal ciclo e non saremmo all'n-esimo inizio ciclo. Avendo eseguito il ciclo una volta di più sappiamo che ora* `potenza` *ha valore* **v*2**, *insomma è la prossima potenza del 2. Questo è esattamente quanto scriviamo nell'invariante:*

R=`(X>0, potenza=`$2^{\texttt{esponente}}$`, X>`$2^{\texttt{esponente}-1}$`)`

Mentre il ragionamento che ci ha portato all'invariante **R** *è chiaro per le esecuzioni del ciclo dalla seconda volta in poi, un dubbio nasce pensando alla prima esecuzione del ciclo. In effetti per trattare anche questo caso, dobbiamo garantire che la parte* `INIZ` *del programma rende vera* **R**. *Questa è proprio la dimostrazione del primo pezzo del programma che avevamo trascurato per occuparci del ciclo. Si tratta di dimostrare quindi:* **PRE** `INIZ` **R**. *Da* **PRE** *e dalla istruzione di lettura 3, segue che* `(X>0)`. *Le dichiarazioni con inizializzazione della riga 1 rendono vero che* `potenza=1 =` $2^{\texttt{esponente}=0}$. *Inoltre, visto che* `(X>0)`*, segue che* `X>`2^{-1}`=0,5`. *Quindi vale* **PRE** `INIZ` **R** *che dimostra che la prima volta che si arriva al* `WHILE` *vale* **R**. *Per concludere la dimostrazione restano 2 cose da fare:*

- *Dimostrare che* **R** *è effettivamente invariante del nostro ciclo. Questo consiste nel dimostrare che se la condizione del* `WHILE` *è vera e si esegue il corpo del* `WHILE` *una volta, si arriva in uno stato che soddisfa nuovamente* **R**. *Questo significa dimostrare la seguente proprietà :*

  R **&&** `(X >` $2^{\texttt{esponente}}$`)` `<potenza*=2; esponente++;>` **R**

  Per questo basta osservare che **R** **&&** `(X >` $2^{\texttt{esponente}}$`)` *implica* `(X >` $2^{\texttt{esponente}}$`, potenza =` $2^{\texttt{esponente}}$`)` *e quindi dopo l'esecuzione di* `potenza*=2;` *vale*

  `(X >` $2^{\texttt{esponente}}$`, potenza =` $2^{\texttt{esponente}+1}$`)` *e dopo l'esecuzione di* `esponente++;` *vale di nuovo,*

  `(X >` $2^{\texttt{esponente}-1}$`, potenza =` $2^{\texttt{esponente}}$`)` = **R**.

- *Per finire si deve mostrare che all'uscita dal ciclo (se viene mai raggiunta), vale* **POST**. *All'uscita dal ciclo l'espressione booleana che regola il ciclo*

deve essere falsa, quindi vale: **R &&** `!(X > potenza)` *che implica* `(X > 2^(esponente-1)) && (X <= 2^esponente)` *da cui discende immediatamente la postcondizione:*
POST = **(**`esponente` **è il minimo intero tale che** $2^{\texttt{esponente}} \geq$ `X`**)**.

4.1 Regole di prova

Generalizziamo la dimostrazione appena discussa per avere una regola da applicare a cicli `while` qualsiasi.

In generale la regola di verifica del `while` è la seguente: per dimostrare che vale **P** `<while(B) C>` **Q** è necessario fare le seguenti 3 dimostrazioni:

1. **P** $\Rightarrow$ **R** (condizione iniziale);
2. **R** **&&** `B <C>` **R** (invarianza);
3. **R** **&&** `!B` $\Rightarrow$ **Q** (condizione d'uscita).

Il lettore attento avrà notato che nella precedente dimostrazione di correttezza abbiamo dimostrato che il programma è corretto *se termina* e che non abbiamo dimostrato che esso termina effettivamente. Tecnicamente abbiamo dimostrato la correttezza **parziale** del programma. Per essere veramente certi che il nostro programma sia completamente corretto dovremmo dimostrare anche che esso termina e solo a fronte di tale dimostrazione, potremmo asserire la correttezza **totale** del programma. Generalmente la prova di terminazione è indipendente da quella della correttezza parziale. Per il Programma 4.1 provare la terminazione è facile: la nonterminazione può dipendere solo dal ciclo `while` e la **PRE** ci assicura che `X` è positivo, grande a piacere, ma finito e quindi, calcolando potenze successive del 2, certamente questo valore prima o poi verrà uguagliato o superato e l'esecuzione uscirà dal ciclo. Purtroppo non sempre le prove di terminazione sono così semplici !

4.2 Altri esempi di prove di correttezza

In questa Sezione affronteremo alcuni problemi non triviali utilizzando gli elementi di C++ introdotti finora. Gli esercizi riguardano il passaggio dalla rappresentazione esterna a quella interna di interi e di reali. Prima di affrontare questi esercizi vogliamo però introdurre alcuni elementi della gestione degli eccezioni del C++. Si tratta di una tecnica molto utile che conviene imparare subito. Maggiori elementi sulla tecnica verranno forniti nella Sezione 7.5.

4.2.1 Esempio di gestione delle eccezioni

Spesso i programmi usano file per leggere il loro input. Nella Sezione 3.1 abbiamo visto che l'apertura di un file può fallire (quando il file non è nel posto indicato dal

nome usato nella dichiarazione che richiede l'apertura). In casi come questo è utile avere un modo di gestire in modo chiaro e semplice i casi di errore (le cosidette eccezioni) e il C++ ce lo offre (così come molti altri linguaggi tra cui Java). Il frammento che segue mostra come questo possa venire fatto usando le apposite istruzioni C++ `try, throw` e `catch` e che spiegheremo dopo il programma.

```
#include<iostream>
#include<fstream>
using namespace std;
main()
 try // apertura file
    {
      ifstream IN("input");
      if(!IN)
         throw(0);
      ofstream OUT("output");
      if(!OUT)
         throw(1);

       ..... resto del main

      IN.close(); OUT.close(); // chiusura dei file
    }
     catch(int x){
      if(x==0)
          cout<<"problemi con il file di input"<<endl;}
      else
           cout<<"problemi con il file di output"<<endl;}
```

Il corpo nel `main` è costituito da un'istruzione `try` che indica che nel blocco che segue può venire sollevata un'eccezione ed infatti in questo blocco sono presenti istruzioni `throw( )` che, qualora eseguite, producono un salto all'istruzione `catch( )` in cui viene gestita l'eccezione occorsa. Nel frammento, viene controllato che l'apertura dei due file "input" e "output" ha successo ed in caso contrario viene eseguito `throw(0)` e `throw(1)`, rispettivamente. Entrambe queste istruzioni, qualora eseguite, comportano un salto alla `catch` in fondo al `main` che stampa un opportuno messaggio. Dopo la `catch` l'esecuzione raggiunge la fine del `main` e quindi termina. Quello appena illustrato è un esempio molto semplice di gestione delle eccezioni. Una trattazione più approfondita di questo argomento è in Sezione 7.5. Il precedente frammento di programma ci mostra anche che, una volta usati i file, è necessario chiuderli con le istruzioni `IN.close();` e `OUT.close();`.

Esercizio Risolto 4.2 *Il primo esercizio che vogliamo risolvere chiede di realizzare un programma capace di eseguire la seguente sequenza di operazioni: legge dal*

file "input" una sequenza di caratteri numerici tra '0' e '9' che termina con il carattere 'a' e converte questa sequenza nel corrispondente valore intero. Esempio, se legge '2' '5' '0' '1' 'a' deve costruire il valore intero 2501. Questo valore va stampato sul file "output".

L'esercizio richiede di leggere caratteri numerici fino a che non si legge il carattere `'a'`*. Man mano che si leggono caratteri numerici, possiamo calcolare il valore intero che questi caratteri rappresentano. Per esempio se sul file troviamo 142 allora 1 verrà letto per primo. Calcolare il valore intero 1 dal carattere '1' è facile: basta calcolare '1'-'0',visto che la codifica ASCII di '1' è quella di '0'+1, cf. Sezione 2.1. Quando leggeremo il secondo carattere '4', per calcolare 14, dovremo calcolare 4 con '4'-'0' e moltiplicare l'1 calcolato prima per 10. Per l'ultimo carattere '2', dovremo moltiplicare di nuovo 14 per 10 e sommargli '2'-'0'. Vediamo come tradurre queste riflessioni in C++:*

```
char c;
int num=0;
IN >> c;
while(c!='a')
{
 num=num*10+('c'-'0');
 IN >> c;
}
OUT<<num<<endl;
```

Per esprimere l'invariante di questo ciclo usiamo la seguente notazione. Se $c_1...c_n$ *sono caratteri numerici, allora NUM(*$c_1...c_n$*) indica il valore intero rappresentato da* $c_1...c_n$*. L'invariante del ciclo è il seguente:* **R=(se si sono letti** $c_1...c_n$ **caratteri, allora num=NUM(**$c_1...c_{n-1}$**))**. *L'invariante può sembrare inutilmente complicato, ma è necessario escludere l'ultimo carattere* c_n *dall'asserzione perchè l'invariante deve essere vero prima che il test del* `while` *sia eseguito e quindi* c_n *potrebbe essere il carattere* `'a'`*. La precondizione del programma è semplicemente che il file "input" contiene una sequenza di caratteri numerici terminata da 'a' e la postcondizione è* **POST=(num=NUM(**$c_1...c_{n-1}$**) e** c_n**=='a')**. *Il fatto* **R && (**c_n**=='a')**$\Rightarrow$ **POST**, *è ovvio. Per ottenere un programma completo che risolve l'esercizio 4.2 basta inserire questo ciclo nel blocco* `try` *dello schema di* `main` *dato in precedenza.*

Esercizio 4.3 *Si tratta di leggere dal file "input" una sequenza di caratteri da '0' a '9' terminata dal carattere 'a', ma questa volta la sequenza va interpretata come un valore decimale. Quindi se si legge '3' '2' '8' 'a' va costruito il valore* `double` *.328. Questo valore va stampato sul file "output". Scrivere un programma che risolva il problema proposto e fornire anche le asserzioni necessarie a dimostrare la sua correttezza.*

La soluzione dell'esercizio 4.3 è molto simile a quella appena discussa. I caratteri letti vanno trasformati in interi come prima, ma il loro valore deve essere diviso

per 10^{-1}, 10^{-2}, 10^{-3}, ... È facile calcolare le necessarie potenze decrescenti del 10 nel ciclo di lettura. Un altro esercizio interessante è il seguente.

Esercizio 4.4 *Si scriva un programma che legga con* `>>` *dal file "input" un valore intero (direttamente in una variabile dichiarata intera) e calcoli la rappresentazione binaria (interna) di questo valore stampandola sul file "output". Per esempio se da "input" si legge il valore 30, allora la sua rappresentazione binaria è 11110 e si chiede di stampare la sequenza di caratteri numerici '0', '1','1','1','1', cioè la sequenza dei bit a rovescio, cioè dal bit meno significativo a quello più significativo. Questa richiesta serve a mantenere l'esercizio semplice, infatti la sequenza richiesta viene calcolata dividendo ripetutamente il valore intero per 2 e prendendo il resto. È utile per fare queste operazioni l'operazione di modulo che in C++ si indica con il simbolo* `%` *e quindi,* `num%2` *calcola il modulo di* `num` *rispetto a 2 e cioè il resto della divisione di* `num` *per 2. Scrivere un programma che risolva il problema proposto e fornire anche le asserzioni necessarie a dimostrare la sua correttezza.*

Nel seguito risolviamo in modo completo il seguente esercizio che presenta maggiori difficltà rispetto ai precedenti.

Esercizio Risolto 4.5 *Abbiamo un numero reale, per esempio* `0.5` *e vogliamo metterlo in forma binaria, cioè come* `0.1`*. Un altro esempio è* `0.25` *che si traduce in* `0.01`*. Queste traduzioni si basano sul fatto che in un numero binario con decimali, il primo bit alla destra del punto decimale vale* $2^{-1} = 0.5$*, il secondo bit vale* $2^{-2} = 0.25$ *e così di seguito. Vogliamo realizzare un programma che trasformi in binario un qualsiasi valore* `double` *minore di 1. Precisamente il programma che vogliamo, a fronte di* `0.5` *deve produrre il carattere '1' come output, mentre a fronte di* `0.25` *deve produrre i 2 caratteri* `'0' '1'` *in questo ordine. Vediamo di essere precisi sullo scopo del programma che vogliamo costruire. Nel seguito sia* $\mathtt{X}_1$ *il valore* `double` *(minore di 1) da trasformare e data una sequenza* $s = b_1, \ldots, b_k$ *di caratteri 0/1, sia* $VAL(s) = b_1 * 2^{-1} + \cdots + b_k * 2^{-k}$*. Vogliamo quindi un programma che da* $\mathtt{X}_1$ *produca una sequenza* s *di caratteri 0/1 tale che* $\mathtt{X}_1 = Val(s)$*. Questa relazione ci suggerisce la maniera di procedere per ottenere* s*: visto che* $\mathtt{X}_1 = Val(s)$*, allora* $\mathtt{2*X}_1 = \mathtt{2*}Val(s)$ *e ovviamente moltiplicando per 2 il numero binario* $0.s = 0.b_1 \ldots b_k$*, si ottiene* $b_1.b_2 \ldots b_k$ *e l'uguaglianza di questo valore con* `2*X` *ci permette di conoscere* b_1*; se* $\mathtt{2*X}_1 \geq 1$ *allora* $b_1 = 1$*, altrimenti* $b_1 = 0$*. Possiamo ripetere lo stesso trucco per conoscere* b_2*? Certo! Il nuovo* $\mathtt{X}_2$ *da considerare è* `if (`$\mathtt{X}_1$`*2 <1)` $\mathtt{X}_1$`*2 else` $\mathtt{X}_1$`*2-1`*. $\mathtt{X}_2$ *è uguale a* $VAL(b_2 \ldots b_k)$ *e quindi possiamo di nuovo moltiplicarlo per 2 per scoprire* b_2*. Ovviamente possiamo continuare nello stesso modo per* $\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3 \ldots$*, ma qual'è la relazione tra* $\mathtt{X}_1$*,* $\mathtt{X}_n$ *e la sequenza di 0/1* $b_1, \ldots, b_{n-1}$ *che viene prodotta negli n-1 passi che portano da* $\mathtt{X}_1$ *a* $\mathtt{X}_n$*? La relazione è la seguente:* $\mathtt{X}_1 = \mathtt{X}_n * 2^{-(n-1)} + VAL(b_1, \ldots, b_{n-1})$*. Intuitivamente,* $\mathtt{X}_n$ *rappresenta la parte di* $\mathtt{X}_1$ *che resta da tradurre, ma le* $n-1$ *moltiplicazioni per 2 fatte negli* $n-1$ *passi devono essere eliminate per avere veramente quello che resta da tradurre di* $\mathtt{X}_1$*. Sulla*

base di questa spiegazione sarà facile scrivere il programma desiderato, corredato delle asserzioni che ne provano la correttezza.

```
  //P
  double X_1=X;
  int n=1;
  while(X>0) //R
 {
   n++;
   X=X*2;
   if(X < 1)
      cout<<'0'<<' ';
   else
      {
        cout<<'1'<<' ';
        X=X-1;
      }
 }
//Q
```

Si noti che nel programma n serve solo per scrivere l'invariante, mentre X_1 serve a mantenere il valore iniziale di X*. Con queste 2 variabili le asserzione del programma sono le seguenti:* **P**=(0< X < 1), **Q**=($X_1 = VAL(output)$), *e* **R**=(|output|=n-1, $X_1 - X * 2^{-(n-1)} = VAL(output)$). *In queste formule $output$ indica la sequenza di caratteri 0/1 stampati dal programma nel momento in cui l'esecuzione raggiunge il punto associato a ciascuna formula. Per convenzione, se con c'è nessun output, $VAL(output) = 0$. Con $|output|$ indichiamo il numero di caratteri di $output$. L'invariante* **R** *esprime il fatto che l'output prodotto è solo una traduzione parziale di* X_1 *e quello che resta da rappresentare è* $X * 2^{-(n-1)}$.

Come per il programma 4.2, anche questo va inserito in uno schema di $main$ simile a quello descritto in precedenza e che si occupa di aprire il file di input segnalando l'eventuale mancata apertura. Dovrebbe essere semplice trovare il $main$ appropriato per questo nuovo esempio.

Usiamo la regola generale 4.1 per dimostrare la correttezza del nostro programma. Innanzitutto, la prima volta che l'esecuzione arriva al ciclo while, X_1=X *mentre $n = 1$ e, visto che nessun output è stato ancora eseguito, $VAL(output) = 0$. Quindi* **R** *è verificata. Assumiamo poi che* R *sia verificata e che valga* X > 0, *allora viene eseguito il corpo del ciclo. Le prime 2 istruzioni del corpo trasformano* **R** *in* **R'**=(|ouput|=n-2, $X_1 - (X * 2^{-1}) * 2^{-(n-2)} = VAL(output)$)=($X_1 - (X * 2^{-(n-1)} = VAL(output)$) *e l'istruzione condizionale che segue può togliere $1 * 2^{-(n-1)}$ a* $X * 2^{-(n-1)}$, *ma in quel caso stampa '1' e quindi $VAL(output \cdot 1) = VAL(ouput) + 1 * 2^{-(n-1)}$. Si noti che usiamo la condizione $|output| = n - 2$ di* **R'** *da cui segue che l'1 che viene stampato è in posizione $(n - 1)$ dell'output e quindi vale*

$1 * 2^{-(n-1)}$. *Se invece viene stampato '0', niente succede a* X *e correttamente* $VAL(output \cdot 0) = VAL(output)$. *Si osservi anche che l'aggiunta di '0' o '1' all'output, ripristina la condizione* $|output| = n - 1$ *e quindi alla fine dell'esecuzione del corpo del ciclo* **R** *vale ancora. Resta da considerare il caso di uscita dal ciclo, ma è il caso più facile. Infatti l'esecuzione abbandona il ciclo se* `X = 0` *e quindi* **R** *diventa* $X_1 = VAL(output)$ *che è esattamente la postcondizione. Quindi il programma è corretto! Certamente si, ma non si deve dimenticare che abbiamo dimostrato la correttezza parziale del programma, cioè che quando il programma arriva alla fine allora vale la postcondizione. Ma arriva sempre alla fine il nostro programma ? La risposta è no. Ci sono molti valori iniziali di* X *per cui il programma non termina. Per esempio* `X=0.4`.[1]

Questo è quindi un esempio in cui la correttezza parziale può venire dimostrata senza avere la terminazione, cioè la correttezza totale. Per rendere il programma totalmente corretto dobbiamo fissare un grado di precisione accettabile per la traduzione che stiamo facendo, cioè anzichè continuare il ciclo fintanto che `X > 0`*, possiamo consentire un piccolo errore, per esempio potremmo inserire il seguente* $X * 2^{-(n-1)} > 2^{-10}$ *come test del ciclo. Questo è equivalente a richiedere che la sequenza di caratteri stampati sia al più di 10 caratteri. Infatti dopo 10 esecuzioni del ciclo* n *diventa 11 e quindi l'errore è* $0 <= X * 2^{-10} < 2^{-10}$*, visto che il valore di* X *è minore di 1. Visto che è più semplice contare il numero di iterazioni del ciclo rispetto a calcolare potenze negative del 2, adottiamo questa maniera di procedere. Il nuovo programma diventa il seguente.*

```
  //P
  double X_1=X;
  int n=1;
  while(X>0 && n<11) //R
 {
   n++;
   X=X*2;
   if(X < 1)
      cout<<'0'<<' ';
   else
      {
        cout<<'1'<<' ';
        X=X-1;
      }
 }
//Q
```

Le asserzioni di questo nuovo programma diventano le seguenti: **P**=(`0< X <` `1`), **Q**=(X_1`-X`$* 2^{-10} = VAL(output)$), *e* **R**=(`0<n<=11, |output|=n-1,`

[1] In realtà il programma termina ugualmente, ma a causa di un errore introdotto dalla moltiplicazione floating point che inserisce decimali spuri nel valore di X.

$\mathtt{X}_1\mathtt{-X}*2^{-(n-1)} = VAL(output)$). *Vale la pena di osservare che la postcondizione esprime il fatto che la traduzione ammette un possibile errore. La terminazione del ciclo è ovvia visto che n aumenta di 1 ad ogni iterazione. Che il nuovo invariante dimostri la correttezza parziale viene lasciato come esercizio.*

Concludiamo il Capitolo con un nuovo Esercizio Risolto non banale.

Esercizio Risolto 4.6 *Quando si leggono valori da un file, una* **sentinella** *è un valore convenzionale che segnala che l'input è finito. A volte è possibile usare l'end of file come sentinella (e abbiamo visto esempi di questo nel Capitolo precedente), ma ora vogliamo invece usare come sentinella una coppia di 0 contigui. Ecco il problema. Vogliamo scrivere un programmma che legga dei valori interi dal file "input" rispettando il seguente vincolo: deve leggere al più 10 valori, ma deve fermarsi qualora legga la sentinella composta da due 0 contigui. Per ogni lettura del valore* v*, il programma deve stampare su "output"* v*, solo a condizione che esso non appartenga alla sentinella. In nessun caso il programma deve eseguire più di 10 letture. Qualche esempio di input output chiarirà l'esercizio: se su "input" si trova [0 1 0 1 0 1 0 1 0 1 0 1] (nota che sono 12 valori) il programma deve scrivere su "output" [0 1 0 1 0 1 0 1 0 1], cioè i primi 10 valori letti. Si noti che in questo caso la sentinella non viene trovata in quanto non ci sono due 0 contigui. Se su "input" si trova [0 0 1 2 3], su "output" non si deve scrivere nulla. Se su "input" c'è [1 2 3 4 5 6 7 8 9 0 0 0 0], su "output" si deve scrivere [1 2 3 4 5 6 7 8 9 0] perchè il secondo 0 è in undicesima posizione e quindi la sentinella non viene trovata.*

Ecco il primo tentativo. Usa due variabili booleane, una per indicare che alla lettura precedente è stato letto 0 e la seconda per indicare che è stato letto anche un secondo 0 di seguito al primo (insomma che la sentinella è stata letta). Il programma contiene un errore banale, ma non troppo e lo scopriremo grazie alla prova di correttezza.

```
{
  int X, n=0;
  bool uno0=false, due0=false;
  while (!due0  && n<10)
  {
   IN>>X;
   n++;
   if(X==0)
      if(uno0)
         due0=true;
      else
        uno0=true;
    else
    {
```

```
    if(uno0)
        {
         OUT << 0 <<' ';
         uno0=false;
        }
    OUT<<X<<' ';
    }

  }
}
```

La precondizione è **PRE** (input=$b_1, \ldots, b_k$, k > 9). *Dato* s=$b_1, \ldots, b_k$, *con* $k > 9$, Prefix(s) *indica* $b_1, \ldots, b_{10}$ *se questa sequenza non contiene due 0 contigui, oppure se li contiene, diciamo in posizione* j *e* $j+1$, *con* $0 < j < 10$, *allora* Prefix(s)=$b_1, \ldots, b_{j-1}$. *Adesso possiamo esprimere la postcondizione del programma che è* **POST** =(output=Prefix($b_1, \ldots, b_k$)). *L'invariante del ciclo, che come al solito chiamiamo* **R**, *consiste di varie formule che devono essere sempre tutte verificate all'inizio del ciclo:*

1. (0<=n<=10);
2. letti $b_1, \ldots, b_n$;
3. (!uno0 && !due0)$\Rightarrow$ (output=$b_1, \ldots, b_n$);
4. due0 $\Rightarrow$ ($b_{n-1} = b_n = 0$ && output=$b_1, \ldots, b_{n-2}$);
5. (uno0 && !due0) $\Rightarrow$ ($b_n = 0$ && output=$b_1, \ldots, b_{n-1}$).

R *specifica il ruolo dei 2 booleani. Se sono entrambi falsi, formula (3), allora non ci sono allarmi e scriviamo in output tutto quello che abbiamo letto. Se* due0 *è vera, formula (4), allora non abbiamo scritto le ultime due letture visto che sono due 0. Se solo* uno0 *è vero, formula (5), allora non abbiamo scritto l'ultima lettura (che è 0) in attesa di quella successiva. Non è difficile dimostrare che questa asserzione è effettivamente invariante. Si deve ragionare per casi:*

(i) Se leggiamo un valore non zero, allora dobbiamo scriverlo in output, ma prima di farlo dobbiamo scrivere anche l'eventuale 0 che era in attesa e uno0 *ci dice se è il caso e ovviamente dobbiamo mettere il booleano a* false;

(ii) Se leggiamo zero, allora di nuovo uno0 *ci dice se abbiamo trovato la sentinella (*due0 *deve diventare* true *e non si scrive nulla), oppure è il primo zero e quindi* uno0=true *senza scriverlo.*

Per concludere la prova dobbiamo dimostrare che **R** *assieme alla negazione del test del ciclo, cioè* !(!due0 && n<10)=(due0 || n>= 10) *implicano la postcondizione. Visto che ci sono 2 condizioni di uscita dal ciclo, dobbiamo considerarle separatamente:*

(`due0`*):* *in questo caso l'invariante ci assicura che abbiamo stampato la parte dell'input che precede la sentinella;*

(`n>=10`*):* *usando* **R** *(è importante ricordare che vale anche* **R** *!),* `n=10`*, quindi abbiamo letto 10 valori. Dobbiamo considerare le diverse situazioni possibili.*

- *se* `!uno0 && !due0`*, allora* **R** *ci garantisce che la postcondizione è verificata;*
- *se* `due0` *è vero, allora siamo nel caso precedente già trattato;*
- *se* `uno0 && ! due0`*, allora l'ultima lettura ha letto uno zero e quindi non l'abbiamo scritto in attesa della prossima lettura che non c'è visto che siamo a* `n=10`*, quindi dovremmo stampare questo 0, ma il programma non lo fa e quindi la postcondizione non è verificata in questo caso ! Per mettere le cose a posto, basta aggiungere la seguente istruzione all'uscita del ciclo:*

```
if(uno0) OUT<< 0 <<' ';
```

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

Il C++ permette di definire variabili che puntano ad altre variabili. Queste variabili vengono chiamate **puntatori**. Più precisamente, se `X` punta ad `Y`, significa che l'R-valore di `X` è l'L-valore di `Y`. La seguente dichiarazione `int *X;` dichiara che la variabile `X` è un puntatore ad un intero. Per il momento però `X` è indefinita cioè non punta ancora a niente. Per definirla dobbiamo assegnarle come R-valore l'L-valore di qualche variabile intera. Questo avviene nel seguente esempio.

```
#include<iostream>
using namespace std;
main()
{
 int q=10;
 int  * X=&q;
 cout<<'' l'R-valore di q e':''<< q <<
   '' il suo L-valore e':''<< &q <<endl;
 cout<<'' l'R-valore di X e':''<< X <<
   '' X punta ad una variabile con R-valore:''<< *X <<endl;
```

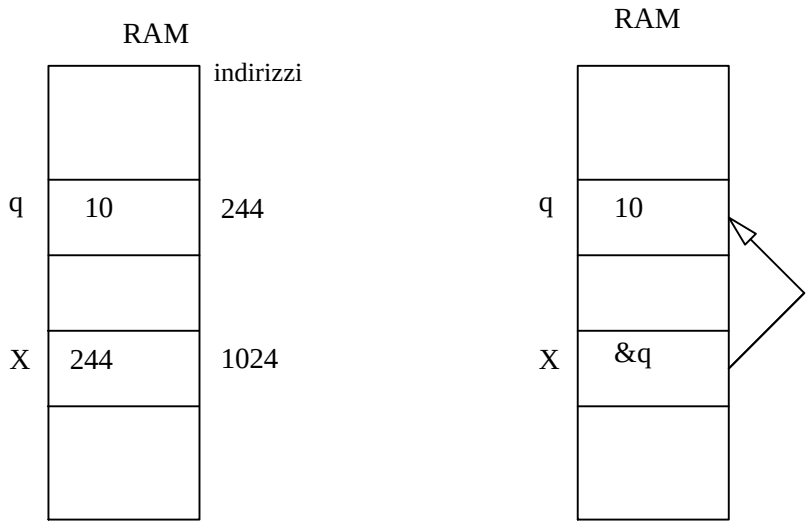

Figura 5.1: 2 modi di mostrare che X punta a q

```
}
```

Spieghiamo le novità di questo esempio. Alla variabile `X` dichiarata di tipo `int *`, si assegna l'L-valore della variabile intera `q`. Si ricordi infatti che l'espressione `&q` ha come valore l'L-valore di `q`. Per descrivere la relazione tra `X` e `q`, diremo che `X` punta a `q`. Per mostrare la relazione tra puntatore (`X`) ed oggetto puntato (`q`), nelle successive 2 istruzioni dell'esempio vengono stampati i seguenti valori:

- con `q` si stampa l'R-valore di `q` (che è `10`),
- con `&q` si stampa l'L-valore di `q` e quindi verrà stampato l'indirizzo RAM in cui è contenuto il suo R-valore `10` (si osservi che, di default, gli indirizzi vengono stampati come numeri esadecimali, cioè in base 16, e questo fatto è indicato premettendo al numero il prefisso 0x);
- con `X` si stampa l'R-valore di `X` che è l'L-valore di `q` ed è quindi lo stesso indirizzo esadecimale del punto precedente,
- con `*X` si stampa l'oggetto puntato da `X` che è naturalmente `q`, quindi questa stampa ha lo stesso effetto di `cout<<q;`, cioè stampa di nuovo `10`.

La relazione tra `X` e `q` è raffigurata in Figura 5.1 in 2 modi. Nella parte di sinistra si mostra esplicitamente che `X` ha come R-valore l'indirizzo RAM di `q` (posto arbitrariamente a 244 nell'esempio) e nella parte di destra questo fatto viene rappresentato intuitivamente con una freccia che da `X` corre a `q`. Questa figura rende anche esplicita una cosa forse ovvia, ma molto importante: `X` e `q` risiedono entrambe nella RAM.

Questo esempio ha introdotto alcune cose nuove e importanti:

1. In primo luogo l'esempio ci mostra che il simbolo `*` ha 2 funzioni diverse, entrambe collegate ai puntatori:

   - serve nella definizione di un puntatore: se `T` è un qualunque tipo, allora `T* Z;` dichiara che `Z` è un puntatore ad un valore di tipo `T`. Ovviamente questa dichiarazione non inizializza `Z` che è quindi indefinita. Nelle dichiarazioni si deve fare attenzione al fatto che l'asterisco `*` appartiene alla variabile dichiarata e non al tipo. Si consideri per esempio la seguente dichiarazione, `char * Z, P;`. Essa dichiara `Z` come puntatore a `char` e `P` come variabile di tipo `char`. Se vogliamo dichiarare 2 puntatori allora dobbiamo ripetere per ognuno l'asterisco `*` e quindi scrivere: `T * Z, *P;`. In questo modo è possibile dichiarare variabili intere e puntatori ad esse con un'unica dichiarazione, come in: `int a=2, b=3, *p=&a, q=&b;`. Si osservi che l'ordine delle variabili dichiarate è rilevante. Infatti, la dichiarazione di una variabile diventa attiva immediatamente dopo essere stata fatta. Quindi nella dichiarazione di `p` possiamo usare `&a`, ma non viceversa.
   - Nel caso di un puntatore `Z` ad un valore di tipo `T`, l'oggetto puntato da `Z` è il valore dell'espressione `*Z`. In questo caso, l'operatore asterisco applicato ad un puntatore restituisce l'oggetto puntato dal puntatore. Questa operazione è detta **derefenrenziazione** del puntatore.

2. In secondo luogo lo stesso esempio mostra che, data una variabile qualsiasi `q`, l'espressione `&q` ha come valore l'L-valore di `q`. Se `T` è il tipo di `q` allora il tipo di `&q` è `T *`. Quindi se `q` fosse di tipo `int` allora `&q` avrebbe tipo `int *`, mentre se `q` avesse tipo `int *`, allora `&q` avrebbe tipo `int **`, e la cosa funziona nello stesso modo qualsiasi sia il tipo di `q`;

Quando si dichiara una variabile di tipo puntatore, per esempio double ** X, esattamente come per le variabili di tipo predefinito, l'R-valore della variabile è indefinito (è ciò che casualmente è contenuto nei byte di RAM allocati a contenere questo R-valore) fino a che qualche istruzione del programma non le assegni un valore appropriato. Usare una variabile indefinita è sempre un grave errore, ma quando essa è di tipo puntatore, l'errore è ancora più grave. Infatti dereferenziare un puntatore indefinito significa cercare di accedere delle posizioni casuali della RAM. Questo può causare un errore a run-time del tipo *segmentation fault*, oppure può causare l'erronea modifica di altre variabili del programma che non hanno alcun rapporto logico col puntatore che causa il problema. È facile capire che questi sono errori di difficile individuazione. Per evitare questi problemi è consigliabile adottare la semplice **convenzione** di assegnare 0 ai puntatori indefiniti. Questo permette di distinguere facilmente un puntatore indefinito da uno definito ed inoltre, anche in caso ci si dimenticasse di fare il controllo, le conseguenze sarebbero limitate, dato che l'indirizzo 0 della RAM è una zona protetta e quindi ogni tentativo di accedervi porta ad un immediato errore di *segmentation fault*.

I puntatori hanno molti usi nel C++ (ed ancora di più nel C). Nella prossima sezione ne vedremo uno molto importante: il passaggio dei parametri alle funzioni. Un'altra nozione del C++ è utile per il passaggio dei parametri alle funzioni: le variabili **riferimento**. Intuitivamente, una variabile riferimento non è che un nuovo nome per una variabile (normale) già esistente. Una variabile riferimento ad intero `R` viene dichiarata nel modo seguente: `int & R = X;` in cui `X` è una variabile normale di tipo `int` che deve essere dichiarata prima di questa dichiarazione. Dopo quest'istruzione, `R` e `X` sono la stessa cosa, diremo che `R` e `X` sono **alias**, cioè esse hanno lo stesso L- ed R-valore. È importante ricordarsi che, nel caso dei riferimenti **è necessario** che essi siano inizializzati al momento della dichiarazione, come nell'esempio appena mostrato. Per le variabili ordinarie l'inizializzazione è invece opzionale, ma certo un valore deve essere sempre attribuito ad una variabile prima di usarla. Illustriamo la nozione di riferimento con un esempio.

```
#include<iostream>
using namespace std;
main()
{
 char X='x';
 char & RX=X;
 cout<<''L-valore di X=''<<&X<<
   '' R-valore di X=''<<X<<endl;
 cout<<''L-valore di RX=''<<&RX<<
   '' R-valore di RX=''<<RX<<endl;
}
```

Eseguendo questo programma si puó verificare che l'L- e l'R-valore di `X` e di `RX` sono identici.

Per sicurezza ripetiamo che il seguente modo di definire la variabile riferimento è un errore che verrebbe segnalato dal compilatore:

```
#include<iostream>
using namespace std;
main()
{
 char X='x';
 char & RX; // ERRORE manca l'inizializzazione
 RX=X; // troppo tardi

}
```

Alcuni lettori (particolarmente impazienti) potrebbero chiedersi a cosa mai servano le variabili riferimento. La loro utilità diventerà chiara nella Sezione 6.1 dove introdurremo le funzioni.

Esercizio 5.1 *Scrivere un programma che definisca una variabile intera* `x` *inizializzata a 10 e successivamente definisce una variabile* `p` *di tipo* `int*` *che punta a* `x`*, una* `pp` *di tipo* `int**` *che punta a* `p`*, una variabile* `ry` *di tipo* `int &` *alias di* `x` *e infine una variabile* `rq` *di tipo* `int** &` *che sia alias di* `pp`*. Successivamente si cerchino 4 diversi modi di aggiungere 1 al valore di* `x` *ciascuno dei quali usa esattamente una di queste 4 variabili (*`p, pp, ry` *e* `rq`*) senza mai usare* `x`*.*

Consideriamo anche un paio di altri esercizi di tipo diverso.

Esercizio 5.2 *Si consideri il seguente frammento di programma:*

```
char x='a', y='z', *p=&x, *q, **Q=&q;
q=p ;
p=*Q ;
q=&y ;
cout << **Q << *p << *q << endl ;
```

Cosa viene stampato? E cosa stampa invece il seguente programma ?

```
char x='d', y='e', *p=&x, *q, **Q=&q;
*Q=p ;
q=*Q ;
q=&y ;
cout << **Q << *p << *q << endl ;
```

5.2 Array e for

Molti linguaggi di programmazione offrono una struttura dati che permette di trattare in modo uniforme molti valori dello stesso tipo. Si tratta della struttura dati **array**. In C++ un array di 10 interi viene definito con la seguente dichiarazione: `int pippo[10];`. Essa definisce una sequenza di 10 interi che hanno nome `pippo[0]....pippo[9]`. Naturalmente possiamo definire array di elementi di qualsiasi tipo: `char, float, bool` e anche di altri array. Per esempio, `int A[20][100];` definisce `A` come un array a 2 dimensioni che ha 2000 elementi organizzati in 20 righe di 100 elementi ciascuna. Il primo elemento di `A` è `A[0][0]`, mentre `A[4][99]` è l'ultimo elemento della **quinta** riga. È necessario fare attenzione al fatto che righe e colonne sono sempre numerate a partire da 0.

Nel caso dell'array `pippo` diremo che ha 10 elementi e dimensione 1, mentre la matrice `A` ha 20×100 elementi e 2 dimensioni, la prima dimensione ha limite 20 e la seconda 100. Si possono definire array con qualsiasi numero di dimensioni, per esempio con 4 dimensioni come in `char K[3][5][10][20];`. Il C++ Standard richiede che nella dichiarazione di un array i limiti siano specificati da costanti. Questa richiesta deriva dal fatto che, in corrispondenza della dichiarazione, il compilatore produce codice che riserva lo spazio RAM per l'array e quindi deve conoscere il tipo ed il numero dei suoi elementi. Il compilatore C++ GNU 4.1 non

segue questa direttiva e accetta anche dichiarazioni con dimensioni non costanti. In modo trasparente all'utente, questi array sono gestiti in modo meno efficiente del normale, usando l'allocazione dinamica della memoria, cf. Sezione **??**. Attenersi al linguaggio C++ Standard è importante per la portabilità dei programmi.

In ogni caso, in C++ gli elementi di un array vengono sempre allocati in posizioni contigue della RAM. Per controllarlo basta dichiarare un array, per esempio `int A[10];` e poi stampare gli indirizzi dei suoi elementi: `&A[0], &A[1],`.... Tra un indirizzo e il successivo ci sarà una differenza di 4, sono i 4 bytes occupati da un intero. Attenzione al fatto che in C++ la stampa di indirizzi viene automaticamente fatta in esadecimale (i.e., in base 16). Per evitarlo basta specificare che si vuole un intero in base 10 come nel seguente esempio: `cout<< (int) &A[1];`. Visto che al giorno d'oggi la memoria RAM dei PC può essere di vari miliardi di byte, anche gli indirizzi possono essere più grandi di $2^{31} - 1$ (circa 2 miliardi) e quindi è più appropriato convertire i valori esadecimali nel tipo `long int` che permette di rappresentare valori più grandi di `INT_MAX`.

Nel caso di un array a 2 dimensioni come, `int M[5][6];` gli elementi sono allocati in posizioni contigue della RAM ordinati per righe, cioè prima la prima riga, poi la seconda e così via.. Per controllarlo, basta stampare gli indirizzi `&M[0][0]` e `&M[1][0]`. Tra questi 2 indirizzi c'è una differenza pari alla lunghezza di una riga dell'array che consiste di 6 interi e quindi di $6 * 4$ byte. La Figura 5.2 mostra schematicamente l'allocazione della matrice `M` nella RAM.

In C++ si possono dichiarare array di 3, 4, n dimensioni, per un n a piacere e in ogni caso gli elementi dell'array saranno allocati in posizioni contigue della RAM nella maniera seguente: consideriamo `double K[10][12][14][16];` allora questo array viene alloggiato nella RAM come una sequenza di 10 array `double [12][14][16]` contigui, ognuno dei quali è aloaggiato nella RAM come una sequenza di 12 array `double [14][16]` contigui, ognuno dei quali è una sequenza di 14 array `double [16]` contigui. Ovviamente un elemento di `K` viene individuato da 4 indici, per esempio `K[8][10][11][0]` è un elemento. Si deve tenere a mente che tutti gli indici partono da 0 e arrivano al limite relativo alla dimensione considerata diminuito di 1. Per esempio, il terzo indice può variare da 0 a 13.

Vediamo subito alcuni esercizi con gli array. Come al solito useremo gli esempi anche per introdurre nuove notazioni come per esempio il modo di inizializzare un array al momento della sua dichiarazione.

Esempio 5.3 *Il primo programma che consideriamo determina la posizione del massimo in un array di interi. L'invariante R del ciclo* `while` *di questo programma segue:* **R**=`(i<=7 && max = MAX(A[0..i-1]), max=A[pos_max])`. *Intuitivamente,* **R** *esprime il fatto che ogni volta che l'esecuzione arriva all'inizio del ciclo è stato calcolato il massimo in* `A[0...i-1]`*, inoltre, il fatto che* **R** *sia invariante significa che un'ulteriore esecuzione del ciclo calcola il massimo in* `A[0...i]`*. Ovviamente l'esecuzione abbandona il ciclo solo quando* `i` *supera la posizione dell'ultimo elemento dell'array e quindi* **R** *implica che il massimo*

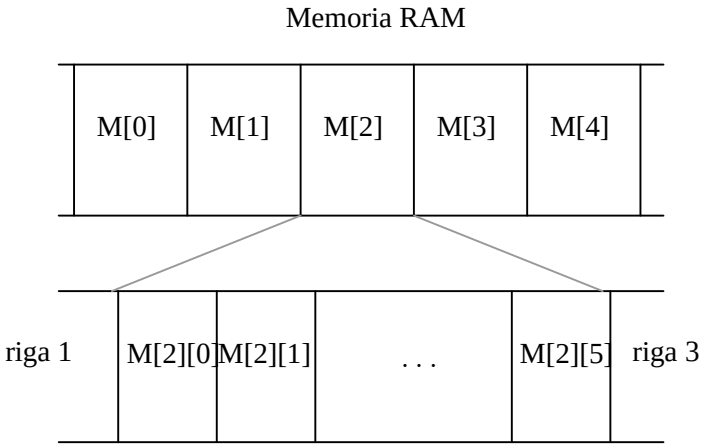

Figura 5.2: l'allocazione delle matrici nella memoria nel C++

di tutto l'array è stato individuato. Invarianti come **R** *sono molto frequenti e ne vedremo parecchi altri nel resto della dispensa.*

```
#include<iostream>
main()
{
 int A[]={10,0,5,-12,45,-9,23}; // A ha 7 elementi
   //inizializzati come specificato
 int i=1, max=A[0], pos_max=0; // inizialmente il massimo e'
   //il primo elemento di A
 while(i<7)
 {
  if(A[i]> max)
     {
        max=A[i];
        pos_max=i;
      }
  i++;
 }
 cout<<''Il massimo e':''<<max<<'' in posizione:''
   <<pos_max<<endl;
}
```

L'array `A` *è inizializzato al momento della dichiarazione. In questo caso, se il numero dei valori specificati in parentesi graffa ci va bene, non è necessario specificare la dimensione di* `A`*. Per questo abbiamo* `A[]=...`*. Se avessimo* `int A[10]=`$\{10, 0, 5, -12, 45, -9, 23\}$*; allora i primi 7 elementi di* `A` *riceverebbero i valori specificati mentre i restanti 3 sarebbere inizializzati a 0. Nel caso di dichiarazione senza inizializzazione come,* `int A[10];` *il C++ standard non richiede*

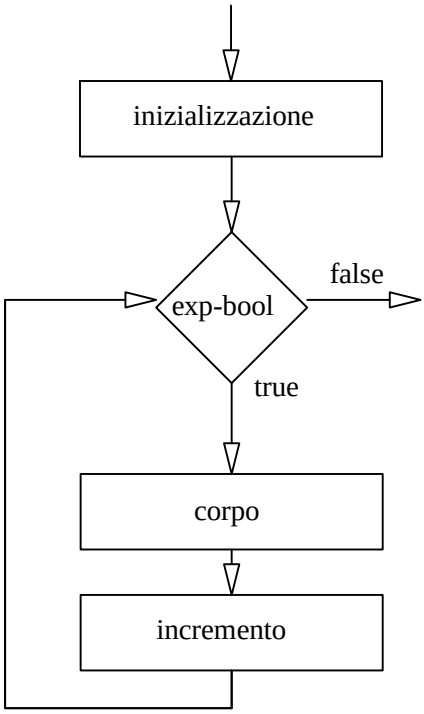

Figura 5.3: il costrutto iterativo `for`

alcuna inizializzazione di default. Quindi gli R-valori degli elementi di A vanno considerati indefiniti.

Esercizio 5.4 *Scrivere un programma che calcoli il numero di elementi dell'array* A *che sono minori di 0. Usate la dichiarazione ed inizializzazione di* A *dell'esempio 5.3.*

Per esaminare tutti gli elementi di un array, come viene fatto dal programma dell'esempio 5.3, è comodo usare al posto del ciclo `while` un altro comando iterativo: il ciclo `for` che ha la forma seguente.

```
for(inizializzazione; exp-bool; incremento) C;
```

La Figura 5.3 mostra il diagramma a blocchi del `for` da cui dovrebbe essere semplice capirne il funzionamento. Un punto importante da ricordare è che l'incremento viene effettuato solo dopo l'esecuzione del corpo `C` del `for`. Si noti inoltre che l'inizializzazione (con eventuale dichiarazione di nuove variabili) viene eseguita solo alla prima entrata nel `for`.

Usando il `for` possiamo riscrivere il programma dell'esempio 5.3 nel modo seguente:

Esempio 5.5

```
#include<iostream>
```

```
using namespace std;
main()
{
 int A[]={10,0,5,-12,45,-9,23}; // A ha 7 elementi
   //inizializzati come specificato
 int  max=A[0], pos_max=0; // inizialmente il massimo e'
   //il primo elemento di A
 for(int i=1;i<7;i++)
   if(A[i]> max)
     {
        max=A[i];
        pos_max=i;
      }
  cout<<''Il massimo e':''<<max<<'' in posizione:''
   <<pos_max<<endl;
}
```

Nel `for` usato in questo esempio, l'inizializzazione è `int i=1;`. La presenza del tipo `int` indica che si tratta della dichiarazione di una nuova variabile `i`. Dalla Sezione 2.3, sappiamo che ogni dichiarazione è visibile solo entro il blocco in cui è dichiarata. È importante qui renderci conto che il `for` introduce un nuovo blocco contenuto nel corpo del main e che `i` è visibile solo nel `for`. Quindi, se dopo il corpo del `for` introducessimo l'istruzione: `cout<<i<<endl;` il compilatore ci darebbe un errore perchè la variabile `i` è sconosciuta all'esterno del `for`. Esaminiamo ora il seguente esempio più complesso.

```
main()
{
 int i=4;
 cout<<i<<endl;  // stampa 4
 for(int i=5;i<10;i++)  // stampa 5 0, 6 0, 7 0, 8 0, 9 0
 {
   cout<<i<<endl;
   int i=0;
   cout<<i<<endl
   i++;
  }
 cout<<i<<endl; // stampa ancora 4 !!!
}
```

Tutte le dichiarazioni di `i` presenti in questo esempio sono in blocchi diversi (e annidati uno dentro l'altro) e quindi dichiarano variabili diverse. Quindi da questo esempio si capisce che l'intestazione del `for` forma un blocco che contiene propriamente un altro blocco che è il corpo del `for`. La struttura a blocchi del programma è quindi la seguente:

```
{ int i=4;
  {int i=0   // intestazione del for
    {
      int i=0; i++;
    }
   i++; // incremento del for
  }
}
```

Questo annidamento spiega perchè le 3 dichiarazioni non entrano in conflitto. Dai commenti a lato delle isruzioni di stampa del corpo del `for`, è facile capire che la prima stampa del ciclo stampa la `i` dell'intestazione del `for` che aumenta infatti ad ogni iterazione, mentre la seconda stampa stampa la `i` dichiarata nel corpo del `for` che ad ogni iterazione riparte da 0 e quindi l'assegnazione `i++` del corpo del `for` non ha alcun effetto, mentre quello dell'intestazione ha effetto.

Esercizio 5.6 *Scrivere un programma che dichiara un array* `C` *di 10* `char` *e legge 10 caratteri dall'input negli elementi di* `C`*. Dopo di che esamina* `C` *per vedere se tra i suoi elementi è presente il carattere* `'a'`*. Se c'è stampa un messaggio che annuncia il successo della ricerca ed anche la posizione in* `C` *in cui è stato trovato. Altrimenti stampa un messaggio di insuccesso. Si chiede di fare due versioni del programma, una con un ciclo* `for` *ed una con un ciclo* `while`*.*

Esercizio 5.7 *Modificare l'esercizio precedente nei seguenti modi:*

1. *deve calcolare quanti caratteri* `'a'` *ci sono nell'array;*

2. *deve calcolare quanti caratteri alfabetici minuscoli sono presenti in* `C`*;*

3. *deve calcolare il numero di occorrenze in* `C` *di ciascun carattere alfabetico minuscolo.*

4. *infine deve determinare se* `C` *contiene* `'b'` *e in una posizione successiva anche* `'2'`*.*

Esercizio 5.8 *Scrivere un programma che dichiara un array* `C` *di 20* `int` *e legge interi dall'input standard in C fino a che l'array è pieno oppure viene letto il valore -1 che segnala appunto la fine degli input. In ogni caso il programma deve calcolare il numero di valori letti senza contare il -1 qualora sia stato letto.*

Esercizio 5.9 *Scrivere un programma che dichiara due array* `A1` *e* `A2` *di 10* `int` *ciascuno, legge 20 interi dall'input standard inserendoli alternativamente nell'uno e nell'altro. Infine calcola quanti valori sono sia in un array che nell'altro.*

5.3 Il tipo degli array

Data una dichiarazione di array, come `int A[15];` che tipo ha `A`? Il tipo di `A` è `int [15]`, ma il C++ considera questo tipo equivalente a `int [ ]`, che significa che il numero di elementi (15) non è rilevante per il tipo e che quindi tutti gli array ad 1 dimensione di interi sono equivalenti. Se eseguissimo l'istruzione di output `cout << A << &A[0];` ci accorgeremmo che l'R-valore di `A` è l'indirizzo del primo elemento `A[0]`. Quindi in realtà in C++ il tipo `int [ ]` è molto vicino a `int*`. I 2 tipi sono simili ma non equivalenti perchè `A` è costante (provate a modificare l'R-valore di `A`, per esempio con `A++`), mentre un valore di tipo `int*` non lo è .

Il fatto che `int [ ]` e `int [15]` siano equivalenti è positivo quando vorremo passare array alle funzioni, cf. Sezione 6.2 . Questa equivalenza tra tipi ci permetterà di usare la stessa funzione per ordinare (per esempio) array (di un dato tipo) con 10, 20 o 10000 elementi. Il fatto che `int [ ]` è molto simile a `int *` è di nuovo significativo in relazione alle funzioni: implica che non verrà mai passata alla funzione una copia di un array, ma solo il puntatore al suo primo elemento, cf. Sezione 6.2.

La possibilità di "dimenticare" il numero di elementi (`int [ ]` e `int [15]` sono equivalenti) non "scala" ad array a più dimensioni. Infatti il tipo di `double K[10][12][20]` è `double [ ][12][20]`. Possiamo "dimenticare" il primo indice, ma non gli altri. Questo è vero per array con numero di dimensioni qualsiasi. Il motivo di nuovo è legato al problema di passare gli array alle funzioni e verrà spiegato nella Sezione 6.2. Un'altra maniera assolutamente equivalente di scrivere in C++ questo tipo è : `double (*)[12][20]`. Interessante notare (e facile da dimostrare) che `cout << K << &K[0][0][0];` di nuovo mostra che l'R-valore di `K` è l'indirizzo del primo elemento dell'array (primo in relazione alla maniera in cui un tale array viene immagazzinato nella RAM). Quindi `K` è un puntatore e il tipo `double (*)[12][20]` lo mostra più chiaramente di `double [ ][12][20]`, ma, ripetiamo, i 2 tipi sono equivalenti.

5.4 Stringhe alla C e array di caratteri

Le stringhe alla C, introdotte nella Sezione 2.1, sono degli array di caratteri con alcune peculiarità. La caratteristica più importante è che i suoi elementi sono costanti. Quindi la seguente dichiarazione `const char * S=''pippo'';` introduce l'array `S` di caratteri costanti. È possibile omettere `const` dalla dichiarazione, ma questo non altera il fatto che la stringa sia costante. Come sempre negli array, l'R-valore di `S` è un puntatore al primo elemento dell'array. Un'altra caratteristica delle stringhe alla C è che esse possiedono un carattere 'invisibile' alla fine. Si tratta del carattere nullo (codice ASCII 0, si denota con `'\0'`). Questo carattere ha la funzione di sentinella di fine stringa. Serve per esempio nel caso dell'istruzione `cout<<S;` in cui si produce l'output della sequenza di caratteri

"pippo" e la sentinella serve a comunicare che la sequenza di caratteri da stampare è finita.

Gli array di caratteri sono simili alle stringhe alla C, ma non sono costanti (a meno di richiederlo esplicitamente nella dichiarazione) e non possiedono sentinelle alla fine (a meno di inserirla esplicitamente). Questa vicinanza ci permettere di inizializzare gli array di caratteri con stringhe alla C, per esempio nel modo seguente: `char A[]= ''pippo'';`. I caratteri di `''pippo''` sono semplicemente copiati negli elementi di `A`, cioè `A[0]=='p'`, `A[1]=='i'` e così via. Chi fosse sorpreso che una stringa costante come `''pippo''` possa venire usata per l'inizializzazione appena descritta, deve riflettere che copiare i caratteri della stringa alla C nell'array `A` non modifica la stringa alla C. Si osservi comunque che l'array `A` avrebbe 6 elementi e non 5: non dobbiamo dimenticare la sentinella che sempre termina le stringhe alla C. Dopo quanto detto, dovrebbe apparire ovvio che l'esecuzione di `cout<<A;` produca la stampa della stringa "pippo". Ma cosa succederebbe invece per l'array `A` definito e stampato come segue: `char A[]={'p','i','p','p','o' }; cout<<A;`? L'array `A` ha solo 5 elementi (non ha la sentinella). Essendo `A` un array di caratteri, l'istruzione `cout<< A;` produce la stampa di tutti gli elementi dell'array fino a che non si incontra un byte a 0 (la sentinella). Ovviamente visto che la sentinella non è stata inserita, la stampa potrebbe continuare oltre l'ultimo elemento di `A`. Provate ad eseguire le istruzioni appena discusse per osservare questo fenomeno con i vostri occhi.

È importante notare che solo per gli array di caratteri (e le stringhe alla C) vale che la stampa del nome di un array causi la stampa di tutto l'array (e anche di più). Per gli array di tutti gli altri tipi, la stampa del nome dell'array produce solo la stampa dell'R-valore del nome e cioè l'indirizzo del primo elemento dell'array.

Gli array di caratteri sono speciali anche per l'input: `char C[10]; cin >> C;` inserisce i prossimi caratteri che si trovano su `cin` nei successivi elementi di `C`. Quando si fanno operazioni di input come queste, si deve essere certi che la stringa di caratteri letta da `cin` non sia più lunga della dimensione dell'array in cui viene letta (10 posizioni nel nostro esempio). Se la stringa fosse più lunga delle dimensioni dell'array, la lettura invaderebbe la memoria RAM che si trova dopo l'ultimo elemento dell'array, causando un errore.

Può essere utile sapere che il linguaggio C (e C++) comprende la libreria `string.h` che consiste di molte funzioni per manipolare le stringhe alla C. Queste funzioni sfruttano molto spesso la sentinella che segnala la fine di queste stringhe. La descrizione di questa libreria è disponibile all'indirizzo http://www.cplusplus-.com/doc/.

Il fatto che il C++ permetta di inizializzare un array di caratteri con una stringa alla C, potrebbe indurre a pensare che il C++ permetta anche l'assegnamento tra array di qualsiasi tipo, cioè per esempio: `int A[]={1,2,3}, B[]={4,5,6}; A=B;`. Si potrebbe pensare che questa assegnazione produca lo stesso effetto delle assegnazioni: `A[0]=B[0]; A[1]=B[1];` e `A[2]=B[2];`, ma non è così. Infatti, questo programma non compila poichè il C++ Standard non permette l'assegnazione tra array neppure tra array di caratteri.

5.5 L'aritmetica dei puntatori

L'aritmetica dei puntatori è tutta riassunta nella seguente frase: *dato il puntatore* `T* p, p+1` *è un'espressione il cui R-valore è uguale a* `p+sizeof(T)` *e* `p+k` *ha R-valore uguale a* `p+sizeof(T)*k`. Se `p` punta ad un elemento di un array di tipo `T`, allora, visto che il C++ garantisce che gli elementi degli array sono allocati nella RAM consecutivamente, l'aritmetica dei puntatori ci offre un modo molto semplice per accedere all'elemento successivo a quello puntato da `p` o anche a quello che si trova `k` posizioni dopo di esso. Ci sono 2 cose però che complicano la faccenda e bisogna capirle bene per evitare errori.

La prima cosa da osservare è che quanto detto funziona sulla semplice ipotesi che `p` sia un puntatore ad un oggetto di tipo `T`: non è richiesto per nulla che `p` punti ad un elemento di un array. Supponendo che appunto `p` punti ad un valore di tipo `T`, allora `p+1` e `p+k` sono espressioni il cui valore è l'R-valore di `p` sommato a `sizeof(T)` e `k*sizeof(T)`, rispettivamente. Il linguaggio C++ non garantisce in alcun modo che nella RAM, in corrispondenza di questi indirizzi, ci sia effettivamente un valore di tipo `T`. È responsabilità del programmatore fare in modo che sia così.

La seconda cosa cui fare attenzione, quando si usa l'aritmetica dei puntatori, è il vero tipo del puntatore a cui la stiamo applicando. Tutto è semplice quando consideriamo puntatori a tipi predefiniti oppure consideriamo un array ad una dimensione di tipo predefinito (che, come visto nella Sezione 5.3, in pratica è un puntatore ad un tipo predefinito). Le cose si complicano quando si considerano array a più dimensioni, infatti i loro tipi sono più complicati.

Facciamo un esempio: assumiamo la dichiarazione `double K[8][10][8];` nella Sezione 5.3 abbiamo visto che il tipo di `K` è `double (*)[10][8]` e quindi se l'R-valore di `K` è `I`, allora `K+2` ha valore uguale a `I+2*(10*8)*8` infatti, come rivela il suo tipo, `K` punta ad un array `double[10][8]` e quindi un tale array consta di 10*8 elementi `double` che occupano 8 byte ciascuno. Visto che consideriamo `K+2` questo sposta `I` di `2*(10*8)*8`. Ovviamente lo stesso vale per qualsiasi intero che sommiamo o sottraiamo a `K`.

A questa difficoltà si aggiunge quella di riconoscere il tipo per esempio di `*(K+2)`. La maniera di ragionare è la seguente: **se dereferenziamo un puntatore ad un oggetto otteniamo l'oggetto puntato**. Applicando questo ragionamento al nostro esempio, visto che `K` e quindi anche `K+2` hanno tipo `double (*)[10][8]`, il tipo di `*(K+2)` è quello di un array `double[10][8]` e cioè è `double (*)[8]`. Da questo segue che la distanza tra `(K+2)` e, per esempio, `*(K+2)-3` è uguale a `3*sizeof(*(K+2))= 3*8*8`, cioè vengono "saltati" 3 array di 8 `double`, dove ogni `double` occupa 8 byte. Quindi, considerando gli R-valori di `(K+2)` e `*(K+2)-3`, abbiamo che `(K+2)-(*(K+2)-3)=3*8*8`.

Continuando l'esempio potremmo considerare il tipo di `*(*(K+2)-3)`. Applichiamo di nuovo il ragionamento di prima. Visto che il tipo di `*(K+2)-3` è `double (*)[8]`, dereferenziandolo otteniamo un array `double [8]` il cui

tipo è `double []` o `double *`. Quindi `*(*(K+2)-3)+4 -(*(K+2)-3)=4*8`, cioè il puntatore si sposta di 4 `double`.

Concludiamo la Sezione osservando che il C++, oltre a quella appena vista, offre un'altra notazione per l'aritmetica dei puntatori. La notazione con gli indici e le parentesi quadrate detta **subscripting**, cf. Sezione 5.2. Facendo nuovamente riferimento all'esempio precedente, possiamo scrivere `K[2]` al posto di `*(K+2)`. Il significato è equivalente. Quindi il subscripting "sposta" il puntatore (esegue il `+2`) e contemporaneamente lo dereferenzia. Il tipo di `K[2]` è lo stesso di `*(K+2)`, cioè `double (*)[8]`. In modo simile possiamo scrivere `K[2][-3]` che ha tipo `double *` e finalmente `K[2][-3][4]` che ha tipo `double`. Al lettore di determinare quale elemento di `K` individui questa scrittura.

Capitolo 6

Funzioni

6.1 Funzioni

Finora tutti i nostri esempi di programmi consistono di un unico blocco, il main che contiene tutte le operazioni da eseguire. Nel momento in cui la sequenza delle istruzioni da eseguire diventa lunga e complessa anche il main diventerebbe lungo e complesso e quindi difficile da capire e modificare. In generale è quindi necessario organizzare i nostri programmi come un insieme di pezzi, ognuno con un compito ben chiaro ed un nome (che possibilmente richiami il suo compito). Questi pezzi che formano il programma si chiamano **funzioni** ed esse vengono eseguite per mezzo di un'apposita istruzione detta di **invocazione** della funzione in cui appare il nome della funzione invocata. Chiariamo subito un punto: anche il main è una funzione, ma essa è invocata automaticamente al momento dell'esecuzione del programma (ogni programma deve avere uno ed un solo main). Le altre funzioni devono invece essere invocate esplicitamente dal main o da qualche altra funzione (che a sua volta è stata invocata e così via). Una funzione in generale restituisce un risultato e, attraverso i suoi parametri, riceve dei valori da chi l'invoca. Vediamo un esempio semplice di funzione usata in un programma che determina il massimo di un array:

```
#include<iostream>
bool maggiore(int primo, int secondo)
{
 if(primo > secondo)
    return true;
   else
    return false;
}

main()
{
  int A[7];
```

```
  for(int i=0;i<7;i++)
   {
    cout<<''Digita l'elemento:<<i<<'')'';
    cin>>A[i];
   }
  int max=A[0], pos_max=0;
  for(int i=1;i<7;i++)
    if(maggiore(A[i],max))   //invocazione
     {
        max=A[i];
        pos_max=i;
      }
  cout<<''Il massimo e':''<<max<<'' in posizione:''
   <<pos_max<<endl;
}
```

La funzione `maggiore` restituisce un valore di tipo `bool` ed ha 2 argomenti di tipo `int`. Essa è invocata dal main all'interno dell'istruzione condizionale. In questo caso il main è detto il **chiamante** di `maggiore` e l'invocazione `maggiore(A[i],max)` causa l'esecuzione della funzione `maggiore` che alla fine della sua esecuzione restituisce al chiamante (con un'istruzione di `return`) un valore booleano, chiamiamolo `V`, che prende il posto dell'invocazione e quindi l'esecuzione continua con l'esecuzione di `if(V)...` del chiamante. Si deve capire che l'invocazione della funzione `maggiore` è un'espressione il cui valore è quello restituito dalla funzione invocata.

Nell'esempio appena visto abbiamo inserito la definizione della funzione `maggiore` prima del main per un motivo preciso: visto che main invoca `maggiore`, quest'ultima deve essere nota al compilatore quando esso arriva a compilare l'invocazione. Il compilatore ovviamente considera le istruzioni del programma nell'ordine in cui sono nel file. In realtà, non è necessario avere l'intera definizione della funzione prima della sua invocazione, ma è sufficiente il **prototipo** o la **dichiarazione** o anche la **segnatura** della funzione e cioè la sua prima riga. L'intera definizione può seguire l'invocazione nello stesso file oppure essere addirittura su un altro file. Chiaramente, il compilatore la dovrà trovare prima o poi, altrimenti darà errore dichiarando che la funzione non è definita. Un esempio di prototipo della funzione `maggiore` è il seguente: `bool maggiore(int , int);` Si osservi che nel prototipo non è necessario inserire i nomi degli argomenti. In caso si inseriscano i nomi degli argomenti, non ha alcuna importanza che essi siano gli stessi che sono usati nella definizione della funzione. In un programma ci può essere un numero arbitrario di prototipi di una funzione, ma **una sola** definizione della funzione stessa!

Nella dichiarazione di una funzione, alla sinistra del nome c'è il tipo del valore che la funzione restituisce come risultato, mentre alla destra del nome tra parentesi si trovano i cosidetti **parametri formali**. Invece i parametri tra parentesi usati nelle

invocazioni della funzione sono detti **parametri attuali**. Ovviamente il numero dei parametri formali deve essere lo stesso di quello dei parametri attuali di ogni invocazione della funzione. Inoltre, il tipo di ogni parametro formale deve essere lo stesso del tipo del corrispondente parametro attuale. In realtà differenze di tipo tra corrispondenti parametri attuali e formali sono ammesse, ma è inutile affrontare ora questo aspetto, lo tratteremo nella Sezione 7.4.

Nell'esempio precedente la funzione `maggiore` è invocata con 2 parametri attuali: `A[i]` e `max`. Quest'invocazione fa si che gli R-valori di `A[i]` e `max` diventino gli R-valori dei corrispondenti parametri formali `primo` e `secondo` della funzione.

Questa maniera di "passare" valori ad una funzione quando la si invoca è detta **passaggio dei parametri per valore**. È importante capire che il parametro formale `primo` è una variabile del corpo di maggiore che è completamente distinta da `A[i]`. Essa riceve l'R-valore di `A[i]`, ma ha L-valore diverso da `A[i]`. Lo stesso vale per il parametro formale `secondo` ed il corrispondente parametro attuale `max`. Quindi, ad ogni invocazione della funzione `maggiore`, il suo corpo viene eseguito nello stato in cui `primo` e `secondo` hanno come R-valori i valori dei parametri attuali dell'invocazione che infatti vanno visti come espressioni.

La funzione `maggiore` restituisce un risultato e lo fa eseguendo l'istruzione di `return` seguita dall'espressione il cui valore va restituito. Ogni esecuzione di una funzione che restituisce un risultato deve terminare con un `return` che effettivamente restituisce un valore del tipo dichiarato nell'intestazione della funzione. L'istruzione di `return` ha anche un altro effetto: esegue un salto che riporta l'esecuzione all'istruzione immediatamete successiva all'invocazione da cui si ritorna. Esistono anche funzioni che non restituiscono alcun risultato, per esempio, funzioni che eseguono delle stampe. L'esecuzione di una tale funzione termina o quando viene eseguito un `return;` (senza valore da ritornare) oppure quando l'esecuzione arriva semplicemente alla fine del corpo della funzione. Funzioni che non restituiscono risultato dichiarano nell'intestazione di restituire un risultato di tipo `void`, cf. Sezione 2.1. Anche per funzioni che non restituiscono un risultato, al termine della loro esecuzione, si ritorna all'istruzione del chiamante che segue l'invocazione.

Abbiamo già detto che una funzione viene invocata, esegue e poi ritorna al punto immediatamento dopo l'invocazione (con o senza risultato). Visto che una funzione possiede ed usa delle variabili proprie, per esempio i parametri formali, è naturale chiedersi come queste variabili vengano gestite durante l'esecuzione del programma. Ovviamente finchè la funzione non è invocata sarebbe inutile assegnare spazio RAM alle sue variabili. Quindi le variabili di una funzione vengono allocate in cima allo stack dei dati ogni volta che la funzione viene invocata e sono deallocate quando l'esecuzione della funzione termina. Tutto questo avviene automaticamente e per questo motivo queste variabili sono dette **automatiche**. In effetti invocare una funzione significa eseguire il blocco che costituisce il corpo della funzione. Quindi quanto appena descritto è coerente con quanto detto nella Sezione 2.3. In particolare, quando la funzione termina, le sue variabili vengono dealloca-

te dallo stack e lo spazio liberato diventa disponibile a contenere i valori di altre variabili, per esempio variabili di altre funzioni che potranno venire chiamate successivamente. Quando una funzione è invocata essa è ovviamente invocata da una particolare istruzione di invocazione. Chiamiamo I l'istruzione immediatamente successiva a questa invocazione. Assieme alle variabili della funzione invocata, vengono inserite nello stack varie altre informazioni tra cui anche I che è detto **indirizzo di ritorno** dell'invocazione. Questo serve per realizzare l'operazione di ritorno della funzione che sarà infatti un salto all'istruzione I (cioè all'indirizzo RAM in cui si trova l'istuzione I). Nel prossimo esempio mostriamo una funzione che al suo interno dichiara delle variabili. Queste variabili, assieme ai parametri formali, vengono chiamate variabili **locali** della funzione.

Esempio 6.1 *La funzione* `divisore` *che segue, riceve per valore un parametro* `int x` *e calcola e restituisce al chiamante il massimo divisore intero di* `x` *(minore di* `x` *stesso).*

```
int divisore(int x)
{
 int y=x/2;
 while (x % y != 0)
   y--;
 return y;
}
```

La funzione oltre al parametro formale `x` *dichiara una variabile locale* `y`. *La funzione deve restituire un risultato intero e quindi termina sempre con l'esecuzione dell'istruzione* `return y;` *con l'espressione* `y` *di tipo intero e quindi coerente con quanto dichiarato nell'intestazione della funzione. Ogni volta che la funzione* `divisore` *viene invocata, verrano allocate le due variabili* `x` *e* `y` *della funzione ed alla prima sarà assegnato come R-valore il valore del parametro attuale dell'invocazione. L'R-valore di* `y` *viene invece inizializzato da* `y=x/2;` *e poi modificato nel ciclo while dall'istruzione* `y--;`. *Le 2 variabili verranno deallocate quando verrà eseguita l'istruzione finale* `return y;`

È importante avere ben chiaro che nel caso di passaggio di parametri per valore, non è necessario che i parametri attuali siano variabili. Essi possono essere espressioni qualsiasi purchè il loro valore sia dello stesso tipo di quello del corrispondente parametro formale (o almeno di un tipo convertibile in quello del corrispondente parametro formale). Per esempio la funzione `divisore` dell'esempio precedente potrebbe essere invocata nel seguente modo: `int x=235, y=divisore(12*x);`. Il fatto che espressioni siano accettate come parametri attuali nel caso di passaggio per valore dovrebbe essere facilmente comprensibile dalla definizione stessa di passaggio dei parametri per valore: quello che serve è semplicemente un R-valore da assegnare ai parametri formali. Su questo fatto si basa la tecnica dei **parametri di default** delle funzioni che illustriamo con

il seguente esempio: `int f(int x, int y=10);`. In questa funzione viene specificato il valore di default 10 per il parametro `y`. Questo permette di invocare la funzione `f` sia con 2 parametri, per esempio con l'invocazione: `f(w*2, z);`, che con 1 solo parametro, come per esempio in: `f(w*2);`. In quest'ultimo caso, il valore di `w*2` viene assegnato al parametro `x`, mentre il parametro `y` riceve il valore di default 10. Per usare i parametri di default si devono rispettare 2 regole:

- i valori di default si possono applicare solo a parametri passati per valore (tra poco vedremo che esiste anche un'altra modalità di passare i parametri);
- i parametri con valori di default devono sempre essere accostati alla destra della lista dei parametri.

La prima regola è ovvia: i valori di default sono appunto valori e non variabili e quindi possono sostituirsi solamente a parametri attuali passati per valore (che passano valori). La seconda regola è causata dal fatto che in qualunque invocazione i parametri attuali sono associati ai formali da sinistra a destra e quindi i parametri formali che non hanno corrispondenti parametri attuali sono quelli accostati alla destra della lista dei parametri. Un esempio ci aiuterà a capire la faccenda.

```
int f(int x, int y, int z=2){....}
..................
f(4,3);
```

quest'invocazione ha meno parametri attuali dei formali della definizione di `f`. I parametri attuali vengono associati ai formali da sinistra a destra: 4 si associa a `x` e 3 a `y`. Resta "scoperto" `z` alla destra della lista dei parametri di `f`. Se `z` non avesse un valore di default la funzione `f` non verrebbe associata dal compilatore all'invocazione `f(4,3)`. Si osservi che se la definizione di `f` fosse `int f(int x=3, int y= 1, int z=2)`, l'invocazione con 2 parametri attuali verrebbe considerata valida con la stessa associazione descritta prima. Naturalmente quest'ultima funzione `f` può venire invocata senza parametri, con uno, con 2 e con 3 parametri attuali.

Il passaggio dei parametri per valore visto finora ha un grosso limite: la funzione invocata può avere un solo effetto sul chiamante e cioè ritornargli il valore che ha calcolato. In alcuni casi questo è troppo restrittivo, per esempio, se la funzione invocata calcola più di un risultato oppure quando si desidera che essa possa avere un effetto sui valori di alcune variabili del chiamante. Questi effetti non visibili della funzione invocata sulle variabili del chiamante si dicono **side-effects**. Essi si possono ottenere in due modi.

- passando per valore puntatori a variabili anzichè variabili semplici. In questo modo la funzione possiede l'L-valore di alcune variabili della funzione chiamante e quindi può modificarne l'R-valore a piacimento;

- attraverso il **passaggio dei parametri per riferimento**. In questo modo (che illustreremo tra poco) i parametri formali diventano riferimenti (alias) di quelli attuali (che devono quindi essere solo variabili). Quindi ogni cambiamento che la funzione invocata effettua su un parametro formale si riflette automaticamente sul corrispondente parametro attuale (che è una variabile della funzione chiamante).

Illustriamo questi nuovi concetti con qualche esempio:

Esempio 6.2 *Supponiamo di voler definire una funzione che abbia come effetto quello di scambiare tra loro i valori di due variabili del main (il chiamante). Vediamo innanzitutto che se passiamo per valore le variabili da scambiare non riusciamo a fare lo scambio. Dato che la funzione* `scambia` *che segue cerca (inutilmente) di produrre un side-effect e non restituisce alcun risultato, nella sua intestazione il tipo del risultato è* `void`.

```
#include<iostream>
void scambia(int primo, int secondo)
{
 int T=primo;
 primo=secondo;
 secondo=T;
 cout<<primo<<secondo<<endl;
}
main()
{
 int x=10, y=20;
 cout<<x<<y<<endl; // stampa 10 e poi 20
 scambia(x,y);     // stampa 20 e poi 10
 cout<<x<<y<<endl; // stampa ancora 10 e poi 20
}
```

Il problema è ovviamente che la funzione `scambia` *scambia tra loro gli R-valori dei suoi parametri formali che però hanno L-valori completamente indipendenti dagli L-valori dei parametri attuali (*`x` *ed* `y`*) e quindi al ritorno dalla funzione, i valori di* `x` *ed* `y` *sono gli stessi di prima. La situazione della RAM è rappresentata in Figura 6.1.*

Vediamo ora come ottenere l'effetto desiderato passando per valore come parametri attuali puntatori alle variabili da scambiare.

```
void scambia(int *primo, int *secondo)
{
 int T=*primo;
 *primo=*secondo;
 *secondo=T;
```

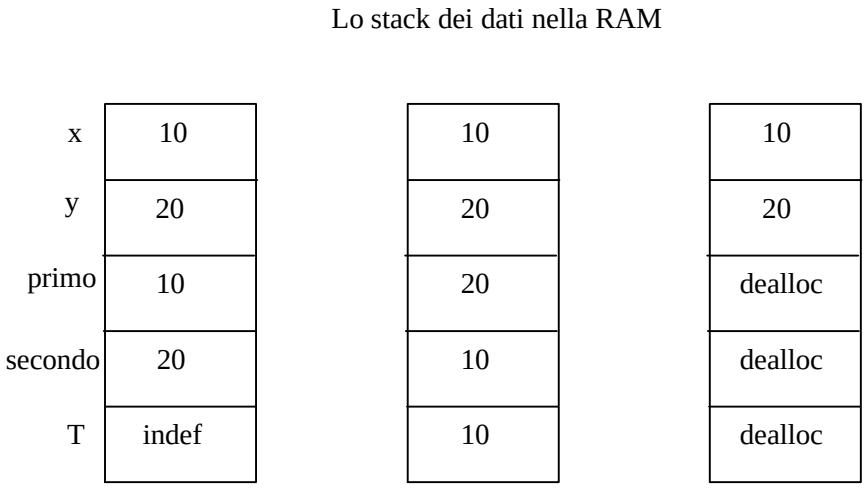

Figura 6.1: Da sinistra a destra: lo stack dei dati quando `scambia` viene invocata, alla fine della sua esecuzione e dopo il ritorno

```
 cout << *primo << *secondo << endl;
}
main()
{
 int x=10, y=20;
 cout << x <<y << endl; // stampa 10 e poi 20
 scambia(&x,&y);     // vengono passati gli L-valori
                     //di x e y, stampa 20 e 10
 cout << x << y << endl; // stampa  20 e poi 10
```

Ci sono varie cose da osservare:

- *i parametri formali di* `scambia` *sono di tipo puntatore ad* `int`.
- *il passaggio dei parametri avviene ancora per valore ed infatti i parametri attuali sono espressioni il cui valore è l'L-valore di* `x` *ed* `y` *che hanno il tipo* `int *` *e quindi vengono assegnati a* `primo` *e* `secondo`*, rispettivamente.*
- *la funzione* `scambia` *dereferenzia* `primo` *e mette questo valore nella variabile* `int` *locale* `T`*. L'R-valore di* `T` *è quindi uguale a quello di* `x`*, cioè 10. La derefenziazione è necessaria anche per le operazioni successive di scambio e di stampa. Se si stampasse per esempio* `primo` *anzichè* `*primo`*, otterremmo l'L-valore di* `x` *anzichè il suo R-valore.*
- *L'uso della variabile locale* `T` *di tipo* `int` *è necessario per effettuare lo scambio correttamente. Infatti se avessimo scritto in* `scambia`*:*

  ```
  *primo=*secondo;
  ```

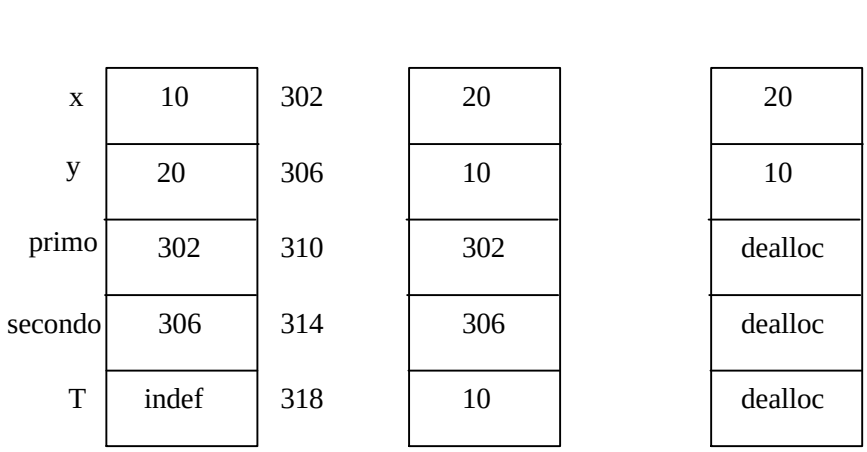

Figura 6.2: Il comportamento di `scambia` quando riceve per valore puntatori alle variabili da scambiare

```
*secondo=*primo;
```

non avremmo ottenuto l'effetto desiderato, ma .... lasciamo al lettore il compito di pensarci.

La situazione della RAM in questo caso è rappresentata in Figura 6.2.

Consideriamo ora la seconda soluzione corretta al problema dello scambio, quella che usa il passaggio dei parametri per riferimento.

```
#include<iostream>
void scambia(int & primo, int & secondo) //primo e secondo
                                    //passati per riferimento
{
 int T=primo;
 primo=secondo;
 secondo=T;
 cout<<primo<<secondo<<endl;
}
main()
{
 int x=10, y=20;
 cout<<x<<y<<endl; // stampa 10 e poi 20
 scambia(x,y);     // stampa 20 e poi 10
 cout<<x<<y<<endl; // stampa  20 e poi 10
}
```

Il programma è più leggibile del precedente e la spiegazione è anche più semplice: avendo dichiarato i parametri formali come riferimenti, `primo` *e* `secondo`

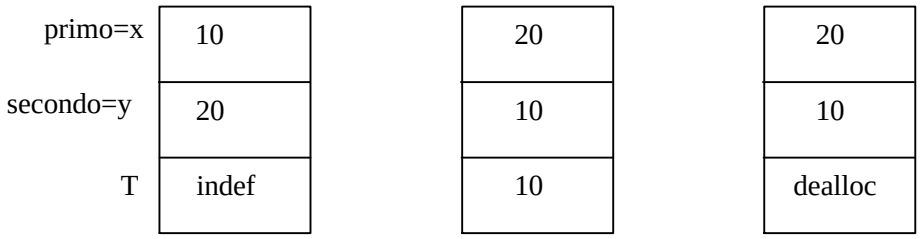

Figura 6.3: Comportamento di `scambia` quando riceve le variabili da scambiare per riferimento

diventano alias di `x` *ed* `y` *e quindi tutto ciò che facciamo a* `primo` *e* `secondo` *è come lo facessimo a* `x` *e* `y`. *La situazione della RAM per questa soluzione è in Figura 6.3 nella quale per mostrare che* `x` *è alias di* `primo` *e che* `y` *è alias di* `secondo`, *abbiamo scritto in modo intuitivo* `primo=x` *e* `secondo=y`. *Questo non chiarisce come vengano effettivamente realizzati i riferimenti:* `primo` *e* `secondo` *sono variabili di* `scambia` *o no? Una spiegazione è discussa alla fine di questa Sezione.*

Esercizio 6.3 *Scrivere una funzione C++ che compie la seguente azione sulle variabili* `x` *ed* `y` *di tipo* `int` *della funzione che la invoca: dopo l'esecuzione della funzione la variabile* `x` *deve avere come R-valore il valore maggiore tra l'R-valore di* `x` *e quello di* `y` *mentre* `y` *avrà il minore dei 2 R-valori. Realizzare questa funzione in due modi: in primo luogo passando per valore puntatori alle 2 variabili* `x` *e* `y` *e in secondo luogo passando le 2 variabili per riferimento.*

Visto che ogni parametro passato per riferimento crea una variabile "in comune" tra chiamante e chiamato, vedi Figura **??**, questi parametri possono servire alle funzioni anche per restituire i loro risultati al chiamante. Questa tecnica ci sarà spesso utile nel seguito in quanto (per il momento) sappiamo restituire il risultato di una funzione solo col `return` e questo comando serve a restituire un solo risultato. In Sezione 7.2 vedremo che questa limitazione è meno grave di quanto ci possa sembrare ora (potremo definire strutture contenenti un numero arbitrario di informazioni), ma il ritorno dei risultati attraverso il passaggio per riferimento resterà utile in molti casi. Supponiamo di voler definire una funzione `P` che sia capace di restituire al suo chiamante due valori, per esempio di tipo `double`. Questo effetto lo possiamo raggiungere facilmente con il passaggio per riferimento. Basta dichiarare la funzione: `..P(double & r1, double & r2,....)`. Un qualsiasi chiamante di `P` dovrà dichiarare due variabili `double` che, opportunamente usate nell'invocazione di `P`, diventeranno alias di `r1` e `r2` e quindi, dopo la fine dell'esecuzione dell'invocazione di `P`, essi avranno come R-valore i due risultati calcolati da questa esecuzione della funzione `P`.

Per concludere la Sezione discutiamo di come vengono realizzati i riferimenti in C++. I riferimenti sono dei puntatori *nascosti* (al programmatore). Cioè quando si dichiara un riferimento, come in `int x, &y=x;` in realtà il compilatore crea un puntatore `int * px =&x;` e ogni volta che nel programma viene usato `y`, il compilatore lo sostituisce con `(*px)`. Insomma i riferimenti sono puntatori nascosti al programmatore. Ma resta la domanda di perchè si è deciso di nascondere i puntatori nel C++. Già mezza risposta è data dal fatto che questo è fatto nel C++ e non nel C. Infatti il C è nato nei tardi anni 60 per disegnare il sistema operativo Unix e quindi la parola d'ordine del C è stata l'efficienza e la massima libertà d'utilizzo per consentire a programmatori molto esperti di fare cose complicate di cui erano consapevoli. Dagli anni 60 agli 80, in cui è stato definito il linguaggio C++, la visione di quello che deve essere un buon linguaggio *general purpose* è cambiata molto. Negli anni 80, e ancora di più oggi, un buon linguaggio deve aiutare programmatori non necessariamente esperti a risolvere problemi e quindi si deve assolutamente evitare che il linguaggo adottato, anzichè facilitarlo, complichi il compito del progettista del Software. Questo risultato viene raggiunto anche nascondendo al programmatore "inutili" dettagli tecnici su come vengono realizzati i costrutti del linguaggio. A mio avviso la visione è così drammaticamente cambiata dagli anni 60 ad oggi a causa della cosiddetta crisi del software, cioè del fatto che si è presa coscienza di quanti errori siano contenuti nel software che viene normalmente prodotto e di quali problemi e costi questo causi. A conferma di questa tendenza, il linguaggio Java, definito degli anni 90 e attualmente molto usato, non possiede puntatori espliciti, ma solo riferimenti.

6.2 Funzioni ed arrays

La differenza tra le due modalità per il passaggio dei parametri alle funzioni, per valore e per riferimento è insita nel loro nome: il passaggio per valore passa dei valori che diventano gli R-valori dei parametri formali, mentre il passaggio per riferimento passa delle variabili di cui i parametri formali diventano alias. Insomma nel passaggio per riferimento il chiamante "concede" alla funzione invocata di intervenire direttamente (attraverso gli alias) su alcune delle sue variabili.

Quando passiamo array alle funzioni non abbiamo scelta, passiamo per valore il puntatore al primo elemento dell'array, cioè l'R-valore del nome dell'array, cf. Sezione 5.2. Quindi non viene mai fatta una copia dell'array, ma si permette alla funzione invocata di accedere allo stesso array posseduto dal chiamante. Questo fatto è coerente con quanto osservato in Sezione 5.2 che il tipo di un array è molto simile ad un tipo puntatore (costante).

Questa scelta è motivata da considerazioni di efficienza: passare alla funzione una copia dell'intero array che ne avrebbe quindi una sua copia locale, causerebbe uno spreco di tempo e spazio.

Vediamo la sintassi C++ del passaggio degli array alle funzioni. Iniziamo con array ad una dimensione. Successivamente consideremo anche quelli a più dimen-

sioni.

Se vogliamo passare l'array `double K[1000];` ad una funzione `f`, il prototipo della funzione deve avere un parametro formale in grado di accoglierlo e che in pratica deve avere tipo `double []` o `double *`. Come anticipato nella Sezione 5.2, si potrebbe anche usare il tipo `double [1000]`, ma il 1000 non avrebbe alcun significato in quanto il compilatore lo ignora. Quindi meglio usare il tipo `double[]` che esprime più precisamente quello che succede e cioè che `f` è in grado di accogliere array di `double` di ogni lunghezza. Per questo motivo, spesso, accanto al parametro formale che accoglie l'array si prevede un parametro intero che accoglie la lunghezza dell'array passato. Spesso chiameremo questo parametro `limite` oppure, quando avremo a che fare con array a più di una dimensione, denoteremo i limiti delle varie dimensioni con `limite_1`, `limite_2`, eccetera. Continuando l'esempio dell'array di 1000 `double`, la funzione `f` potrebbe avere prototipo: `...f(double A [], int limite)` e un'invocazione che passa i primi 100 elementi di `K` (che ha in tutto 1000 elementi) sarebbe, `f(K,100)`. Entrambi i parametri sono passati per valore.

Segue un esempio di passaggio di un array ad una funzione. Si tratta di una funzione che cerca il massimo in un array di caratteri. Questo esempio illustra anche i diversi tipi che può assumere un parametro formale destinato a ricevere un array.

Esempio 6.4 *Nella seguente funzione si sfrutta il fatto che i caratteri sono in effetti rappresentati da interi e quindi tra loro esiste un ordine totale. Il primo parametro formale della funzione è l'array di* `char` *in cui trovare il massimo. Il secondo parametro formale serve ad accogliere il numero di elementi dell'array passato che vanno considerati. In questo modo la funzione può lavorare su array di lunghezza qualsiasi. La funzione restituisce il massimo dell'array (un carattere) con il* `return max;` *finale.*

```
char massimo(char C[], int limite)
{
 char max=C[0];
 for(int i=1; i<limite;i++)
    if(C[i]>max)
      max=C[i];
 return max;
}
```

Visto che passare ad una funzione un array di caratteri significa passarle un puntatore al primo elemento dell'array, cioè un valore di tipo `char *`*, potremmo definire la nostra funzione* `massimo` *nel modo seguente senza cambiarne il corretto funzionamento:*

```
char massimo(char *C, int limite)
{
```

```
 char max=C[0];
 for(int i=1; i<limite;i++)
    if(C[i]>max)
      max=C[i];
 return max;
}
```

Al posto della notazione `C[i]` *per riferirsi all' i-esimo elemento di* `C` *è possibile usare l'equivalente notazione* `*(C+i)` *che rivela che il subscripting fa 2 cose: applica l'aritmetica dei puntatori (*`C+i`*), vedi Sezione 5.5, e poi dereferenzia il puntatore ottenuto.*

Naturalmente, ogni funzione deve assolvere un particolare compito, diciamo calcolare dei risultati partendo da certi valori. Si pone quindi il problema di dimostrare la sua correttezza (parziale e totale). Per ogni funzione definiremo una precondizione **PRE** *ed una postcondizione* **POST** *e poi cercheremo di dimostrare che* **PRE** *¡corpo della funzione¿* **POST** . *Nel caso della funzione* `massimo` *di questo Esempio avremo:*

- **PRE** =**(**`C` **ha** `limite` **elementi definiti)***;*
- **POST** = **(il valore di** `max` **è il valore massimo in** `C[0..limite-1]`**)***;*

L'invariante del ciclo `for` *è:* **R=(1** $\leq$ `i` $\leq$ `limite`**,** `max` **è il massimo in** `C[0..i-1]`**)**. *Può sembrare strano che in* **R** *si asserisca di aver trovato il massimo in* `C[0..i-1]` *anzichè in* `C[0..i]`*, ma basta osservare che, all'inizio del ciclo,* `i` *è l'indice dell'elemento di* `C` *che sta per essere considerato (l'incremento di* `i` *avviene dopo l'esecuzione del corpo), quindi gli elementi considerati fino a quel momento sono proprio* `C[0..i-1]`*. Inoltre, la prima volta che l'esecuzione arriva al ciclo,* `i=1` *e quindi è stata considerata la porzione di* `C` *che consiste di* `C[0..1-1]`=`C[0]` *e quindi* `max=C[0]` *soddisfa* **R***. Visto che ogni esecuzione del ciclo considera il prossimo elemento di* `C` *e, se è il massimo tra quelli esaminati, cambia* `max` *nel modo appropriato,* **R** *è effettivamente invariante. Dato che il ciclo termina quando* `i=limite`*, è immediato che in questo caso* **R** *implica* **POST** *. Quindi la funzione è parzialmente corretta (rispetto a* **PRE** *e* **POST***). La correttezza totale segue dall'osservazione che* `limite` *è finito (maggiore di 0) e ogni esecuzione del ciclo avvicina* `i` *a* `limite`*.*

Consideriamo ora il problema di passare come parametro un array a più dimensioni. Per esempio, consideriamo l'array `float A[10][20][30];`. Il parametro formale capace di accogliere `A` deve avere tipo `float [][20][30]` oppure `float (*)[20][30]` che sono 2 modi equivalenti di scrivere il tipo di `A`, cf. Sezione 5.2. Quindi, mentre per array ad una dimensione possiamo scrivere funzioni capaci di accogliere array di lunghezza qualsiasi, questo non vale più per array a più dimensioni. Insomma, non possiamo scrivere funzioni, per esempio di calcolo del massimo, capaci di svolgere il loro compito su array a 3 dimensioni in

cui ogni dimensione abbia lunghezza qualsiasi. La libertà sulla lunghezza rimane solo per la prima dimensione, mentre le successive dimensioni devono avere lunghezza fissa. Il motivo di questo vincolo, che spesso si rivela piuttosto fastidioso, non è quello della sicurezza, cioè che in questo modo il codice può venire controllato in modo più stringente (ricordate che questo è uno degli scopi più importanti dei tipi), ma piuttosto esso è dovuto al fatto che senza questo vincolo il compilatore non **potrebbe funzionare**! Cerchiamo di capire perchè. Consideriamo per esempio una funzione `f` a cui vogliamo passare l'array `float A[10][20][30]` e supponiamo per un momento di poter accogliere questo array con un parametro formale del tipo `float X [][][]`[1], che specifichi cioè solo che si tratta di un array a 3 dimensioni e basta. Supponiamo ora che nel corpo di `f` venga acceduto un elemento `X[i][j][z]` dell'array. Sappiamo che, col passaggio dei parametri, l'R-valore di `X` sarà (durante l'esecuzione di `f`) un puntatore al primo elemento di `A`. Questa è la sola conoscenza su cui il compilatore può contare, oltre, naturalmente, al fatto che durante l'esecuzione `i,j` e `z` avranno un R-valore. Il compilatore deve poter costruire un'espressione che calcoli la distanza di `X[i][j][z]` dall'inizio di `X` che conosce. Proviamo a scrivere noi questa espressione ricordando che ogni valore `float` occupa 4 byte nella RAM: l'indice `i` ci fa saltare `i*(20*30)*4` bytes dall'inizio di `X`, dall'indirizzo raggiunto in questo modo, il secondo indice ci fa saltare `j*30*4` byte ed infine l'ultimo indice ci fa saltare solo `z*4` byte. Quindi la distanza di `X[i][j][z]` da `X` è il valore dell'espressione `i*(20*30)*4+j*30*4+z*4`. Ora è chiaro che per produrre questa espressione il compilatore ha bisogno di sapere che la lunghezza delle dimensioni 2 e 3 dell'array che viene passato a `f` sono 20 e 30 e questa è esattamente l'informazione contenuta nel tipo che il C++ richiede di specificare per il parametro formale che accoglie l'array!

Vediamo un esempio di passaggio di una matrice a 2 dimensioni ad una funzione.

Esempio 6.5 *Questa funzione riceve una matrice di dimensioni* `limite_1 * 10` *di* `int` *e restituisce il puntatore al primo elemento della riga la somma dei cui elementi è massima tra tutte le righe della matrice.*

```
int * riga_max(int M[][10], int limite_1)
   // limite_1 = lunghezza prima dimensione.
   //10 = limite seconda dimensione
{
 int somma_max=0;
 int * punt_riga=M[0]; // equivalente a = & M[0][0]
 for(int j=0;j<10;j++) // somma della riga 0 per avere un valore
  somma_max+=M[0][j];  // iniziale per somma_max

 for(int i=1;i<limite_1;i++)  // considera le righe da 1 a limite_1 - 1
```

[1] Sottolineiamo che questo non è un tipo C++ e che stiamo usandolo solo per fare capire il nocciolo della questione.

```
  {
   int somma=0;  // var dichiarata nel corpo del for
   for(int j=0;j<10;j++) // calcola la somma della riga i
      somma+=M[i][j];
   if(somma>somma_max)
    {
     somma_max=somma;
     punt_riga=M[i]; // equivalente a = & M[i][0]
    }
   }
 return punt_riga;
}
```

La funzione calcola la somma della riga 0 e la assegna a `somma_max` *per inizializzare questa variabile con un valore sensato. Coerentemente, il puntatore* `punt_riga` *viene fatto puntare alla prima riga di* `M`*. Si tiene conto che la prima riga è già considerata, facendo partire il ciclo da* `i=1`*. Sarebbe possibile anche inizializzare* `somma_max` *a* `INT_MIN` *cambiando il ciclo* `for` *in modo che parta da* `i=0`*. L'idea è che la somma degli elementi di una riga qualsiasi di* `M` *sarà maggiore di* `INT_MIN` *(che è il più piccolo intero rappresentabile) e quindi il valore iniziale di* `somma_max` *verrà sostituito con la somma di qualche riga di* `M`*. Per avere la correttezza anche dell'improbabile, ma non impossibile, caso limite in cui tutte le righe di* `M` *hanno somma uguale a* `INT_MIN`*, è necessario inizializzare comunque* `punt_riga` *a* `M[0]`*. Si osservi la variabile* `somma` *dichiarata nel corpo del* `for`*: essa "rinasce" con valore 0 ad ogni esecuzione del corpo del* `for` *esterno (cioè quello con indice* `i`*). Nel ciclo interno* `somma` *diventa la somma degli elementi della riga* `j`*-esima e, all'uscita da questo ciclo, viene controllato se la riga appena scorsa ha somma maggiore della massima trovata delle righe precedenti e, in questo caso, aggiorna* `somma_max` *e* `punt_riga`*. L'assegnazione* `punt_riga=M[i]` *è equivalente a* `punt_riga=&M[i][0];`*, come spiegato nella Sezione 5.5.*

È facile dimostrare la correttezza di questa funzione. Innanzitutto la pre- e la post-condizione sono ovviamente come segue:

- **PRE** =**(**`M` **è un array** `int [limite_1][10], con limite_1>0`*)*;
- **POST** =**(**`punt_riga` **punta alla riga di** `M` **che ha somma massima tra le** `limite_1` **righe di** `M`**)**[2].

Le prime istruzioni fino al `for` *con indice* `j` *compreso, inizializzano* `somma_max` *e* `punt_riga` *come descritto prima e questo soddisfa l'invariante del successivo ciclo che è:* **R=(**$1 \leq$ `i` $\leq$ `limite_1`**, la riga** `punt_max` **di** `M` **ha la somma degli elementi uguale a** `somma_max` **che è la massima somma tra le prime** `i-1`

[2] Nel caso ci siano in `M` più righe con somma uguale tra loro e maggiore di quella delle altre righe, la funzione restituisce il puntatore alla riga a indice minimo tra quelle a somma massimale. Per semplicità nella prova ignoreremo questo caso che comunque non presenta alcuna insidia.

righe di `M`**)**. *Per quanto riguarda il ciclo interno, è facile vedere che esso calcola in* `somma` *la somma della riga* `i`*-esima di* `M` *e quindi ogni volta che l'esecuzione ritorna all'inizio del ciclo esterno,* **R** *sarà ancora verificato. Visto che si esce dal ciclo esterno quando* `i=limite_1`*, questa condizione, assieme ad* **R** *implica immediatamente* **POST**. *La correttezza totale segue immediatamente dal fatto che* `limite_1` *è maggiore di 0 e non viene mai modificato.*

L'esercizio seguente ha lo scopo di mostrare che programmi corretti e che risolvono lo stesso problema possono avere prove di correttezza di complessità molto diversa. La soluzione che ammette una semplice dimostrazione della sua correttezza è (generalmente) anche quella più semplie da capire. È ovviamente preferibile scegliere sempre questa soluzione.

Esercizio 6.6 *Dimostrare la correttezza della versione della funzione dell'Esempio 6.5 in cui* `somma_max` *viene inizializzata a* `INT_MIN`.

Esercizio 6.7 *Per ciascun esercizio che segue, la funzione da produrre va corredata di pre-, post-condizione, invarianti per i cicli e prova di correttezza.*

1. *Scrivere una funzione che riceve una matrice di tipo* `char [][5]` *ed il numero di righe* `limite_1` *e determina se c'è un carattere che è presente in tutte le righe della matrice.*

2. *Come nel punto precedente, abbiamo una matrice* `char [][5]`*, ma questa volta vogliamo determinare se tra le colonne dalla seconda alla quinta di questa matrice, ce n'è una che contiene tutti gli elementi che compongono la prima colonna (possibilmente in un ordine diverso). Si osservi che se la prima colonna è composta da soli 1, allora basta che una delle altre colonne contenga un 1 per rispondere si alla domanda. Nel caso la risposta sia si, la funzione deve restituire al chiamante l'indice della colonna trovata. Se la risposta è no la funzione restituisce 0.*

3. *Considerando ancora una matrice* `char [][5]`*, ora vogliamo determinare se una delle colonne dalla 2 alla 5 ha esattamente gli stessi elementi della prima colonna (in ordine possibilmente diverso). Come nel caso precedente, in caso di risposta si, la funzione deve restituire l'indice della colonna trovata e altrimenti restituisce 0.*

4. *Data ancora una matrice* `char [][5]`*, vogliamo una funzione che determini se essa contiene 2 righe tali che tutti gli elementi dell'una sono anche nell'altra e viceversa. Osservare che questo problema è diverso da quello di determinare se una riga è una permutazione di un'altra. In caso la risposta sia si, la funzione deve anche restituire al chiamante i puntatori alle 2 righe in questione. Altrimenti deve restituire dei valori che avrete scelto in modo da mostrare che non è stata trovata alcuna soluzione.*

6.3 Variabili globali

Finora abbiamo visto solamente variabili dichiarate nel `main` oppure in una funzione. Esistono variabili dichiarate fuori da ogni funzione. Esse sono dette **globali** perchè il loro campo di visibilità è l'intero file in cui sono dichiarate. Quindi le variabili globali sonovisibili in ogni funzione che sia definita nel file dopo la dichiarazione delle globali stesse. Sulle variabili globali è necessaio essere coscienti di due problemi:

(i) L'evoluzione del valore di una variabile globale può diventare difficile da controllare in quanto esso può, in linea di principio, venire modificato in qualsiasi funzione ma se una funzione dichiara una variabile locale con lo stesso nome della globale allora quella locale oscura la globale e quindi considerare semplicemente i punti in cui una variabile globale cambia di valore può trarre in inganno, nel senso che il cambiamento potrebbe avere effetto su un'altra variabile (locale e con lo stesso nome).

(ii) Quando in una funzione viene usato un nome che non sia dichiarato localmente alla funzione, allora questo nome è quello di una variabile globale dichiarata prima della dichiarazione della funzione. L'esempio che segue illustra questo fatto.

Esempio 6.8
```
#include<iostream>
using namespace std;
int x=1;
void f()
{
 cout<<x<<endl;
}
main()
{
 int x=10;
 f();   // stampa 1
 cout<< x <<endl; //stampa 10
}
```

Questo esempio illustra il punto (ii) precedente. La `x` *usata nel corpo di* `f` *è quella globale e non quella dichiarata nel* `main` *che invece è quella vista stampata dal* `main`.

Può valere la pena di dichiarare una variabile globale quando questa variabile è usata in molte funzioni diverse. Dichiararla globale evita di doverla passare come parametro a ciascuna funzione, ma, per limitare il problema descritto nel punto (i), è buona norma usare poche variabili globali e inoltre dare loro nomi che indicano in modo chiaro il loro compito e che quindi non siano nomi che potrebbero venire usati anche per variabili locali di funzioni. Spesso vengono dichiarate globali

delle quantità costanti che servono nel programma. Un esempio classico è `const double pi_greco=3,14;`. Come questo esempio mostra, per dichiarare costanti è sufficiente premettere la keyword `const` e inizializzare immediatamente la costante. Maggiori dettagli sulle costanti si trovano nella Sezione 7.1.

6.4 Funzioni e valori restituiti

Quando le funzioni restituiscono un risultato, questo può essere un semplice valore o un riferimento. Nel primo caso la funzione restituisce solo un R-valore che può avere un tipo qualsiasi (predefinito o puntatore), mentre, nel secondo caso, essa restituisce una variabile e cioè un'entità che possiede sia un R- che un L-valore (di tipo predefinito o puntatore). Abbiamo già visto che nel C++ il tipo del risultato viene specificato nel prototipo di una funzione alla sinistra del nome della funzione. Quando questo tipo è `void`, allora la funzione non restituisce alcun risultato esplicito. Se il tipo del risultato è `T` (diverso da `void`), allora ogni possibile esecuzione della funzione deve terminare con l'esecuzione di un comando `return e;` dove `e` deve essere un'espressione il cui valore sia sempre un valore del tipo `T` (o convertibile in `T`).

Esempio 6.9 *In questo esempio consideriamo una funzione che trova il massimo in un'array di* `int`*, ma anzichè ritornare il suo valore o la sua posizione, ritorna un puntatore a quell'elemento dell'array. In questo modo il* `main` *(che nel nostro esempio invoca la funzione), può modificare l'elemento dell'array che contiene il massimo.*

```
int * massimo(int * A, int limite_1)
{
 int max=A[0], pos_max=0;
 for(int i=1;i<limite_1;i++)
    if(A[i]>max)
     {
      max=A[i];
      pos_max=i;
     }
 return &A[pos_max];
}
main()
{
 int A[20];
 // ciclo di lettura dall'input di A e' omesso

 *(massimo(A,20))+=5;
}
```

Il `main` *aggiunge 5 all'elemento massimo dell'array. Lo stesso effetto è ottenibile ritornando un riferimento, come nel seguente programma.*

```
#include<iostream>
int & massimo(int * A, int limite_1)
{
 int max=A[0], pos_max=0;
 for(int i=1;i<limite_1;i++)
    if(A[i]>max)
     {
      max=A[i];
      pos_max=i;
     }
 return A[pos_max];
}
main()
{
 int A[20];
 // il ciclo di lettura dall'input di A e' omesso
 massimo(A,20)+=5;
}
```

Questi esempi mostrano che, come nel caso del passaggio dei parametri, anche nel caso del risultato restituito dalle funzioni, puntatori e riferimenti ci consentono di lavorare con gli L-valori delle variabili e quindi anche di manipolare i loro R-valori. I riferimenti spesso consentono di scrivere codice pù leggibile.

In una funzione che ritorna un puntatore o un riferimento si deve avere molta cura di evitare un errore molto grave. Nel caso di puntatore restituito dalla funzione, il puntatore non deve puntare ad una variabile locale della funzione perchè questa variabile sarà deallocata dopo la fine dell'esecuzione della funzione, vedi Sezione 6.1. Lo stesso vale quando si ritorna un riferimento: non si deve ritornare alias di variabili locali. Ritornare il puntatore o il riferimento ad una variabile deallocata viene detto **creare un puntatore o un riferimento penzolante (dangling pointer/reference)**.

Come esempio di una funzione che crea una **dangling reference**, trasformiamo opportunamente la seconda delle funzioni `massimo` dell'esempio precedente. È sufficiente sostituire `return A[pos_max];` con `return max;` per introdurre l'errore: la nuova versione restituisce un riferimento a `max` che viene deallocata al momento stesso dell'esecuzione del `return`. Dopo la funzione sbagliata ripetiamo lo stesso main dell'esempio precedente per sottolineare il fatto che restituire un dangling reference crea "solo" i presupposti dell'errore, ma che c'è bisogno di usare la dangling reference perchè l'errore sia veramente commesso e questo avviene appunto nel main, come sottolinea il relativo commento.

```
// funzione sbagliata !!
```

```
int & massimo(int * A, int limite_1)
{
 int max=A[0], pos_max=0;
 for(int i=1;i<limite_1;i++)
    if(A[i]>max)
     {
      max=A[i];
      pos_max=i;
     }
 return max; // restituisce dangling reference
}
main()
{
 int A[20];
 // il ciclo di lettura dall'input di A e' omesso
 // ERRORE: accede ad un'indirizzo della RAM appena deallocato
 massimo(A,20)+=5;
}
```

È bene osservare che errori di dangling pointer/reference sono particolarmente insidiosi in quanto essi non vengono segnalati dal compilatore e molto spesso non vengono segnalati neppure durante l'esecuzione del programma. Inoltre ad un dato istante del calcolo, un dangling pointer/reference fa riferimento ad un'indirizzo RAM che in precedenza è servito a contenere valori di variabili locali di una funzione che nel frattempo è terminata. Successivamente questo indirizzo potrebbe venire usato per contenere R-valori di altre variabili. Quindi questo errore in esecuzioni diverse può causare l'erronea modifica di variabili diverse rendendo quindi difficile trovare l'errore esaminando il comportamento del programma.

Esercizio 6.10 *Risolvere i seguenti esercizi specificando pre- e post-condizione per ciascuna funzione prodotta:*

1. *Scrivere una funzione che riceve una matrice di tipo* `char [][5][10]`*, il limite* `limite_1` *ed un valore* `char c` *e cerca il primo elemento della matrice uguale a* `c` *(percorrendola per righe dalla prima alla* `limite_1 - 1`*-esima) e calcola 2 cose. Se* `c` *viene trovato nella matrice, la funzione deve restituire* `true` *(con un apposito parametro passato per riferimento) e deve ritornare (col* `return`*) il riferimento all'elemento della matrice che contiene* `c`*. Altrimenti, se un tale elemento non c'è, la funzione restituisce* `false` *e col* `return` *il riferimento all'ultimo elemento della matrice.*

2. *Modificare l'esercizio precedente in modo che restituisca un riferimento al primo elemento della riga che contiene il massimo numero di valori uguali a* `c`.

Concludiamo la Sezione con un'osservazione che riguarda la possibilità di restituire array come risultato di una funzione. Visto che la funzione riceve solo il puntatore al primo elemento dell'array e quindi "lavora" sull'array del chiamante, non ha alcun senso restituire questo array al chiamante: il chiamante l'ha già ! Naturalmente sarebbe possibile per una funzione dichiarare un array locale sul quale lavorare ed alla fine restituirlo al chiamante. Ovviamente una tale funzione commetterebbe l'errore di restituire un dangling pointer: il pointer al primo elemento dell'array che, essendo locale, verrà deallocato al momento del `return`. In conclusione il C++ non consente in nessun caso di restituire array con il `return` in quanto una tale operazione non ha senso. Quanto appena detto è vero per i soli array che conosciamo a questo punto della dispensa (cioè gli array automatici), non si applica invece agli array allocati dinamicamente che vedremo più avanti nella Sezione **??**.

6.5 Esercizi commentati sulle funzioni

Nel seguito esaminiamo alcuni esercizi risolti di programmi che usano funzioni che utilizzano le diverse modalità per il passaggio dei parametri e per la restituzione del risultato. Alcune delle funzioni proposte sono corrette ed altre, per meglio raggiungere lo scopo, sono appositamente errate.

Esercizio Risolto 6.11 *Consideriamo il seguente programma. Innanzitutto la funzione. Essa riceve per valore un* `int *` *che restituisce con il* `return x;` *e questo è coerente con il tipo dichiarato del risultato* `int *`*. Inoltre l'assegnazione* `*x=5` *è corretta rispetto ai tipi perchè* `x` *ha tipo* `int *` *e quindi* `*x=5;` *assegna un intero alla variabile puntata da* `x`*. Quindi la funzione* `f` *sembra corretta. Esaminiamo ora il main che invoca la funzione. Il parametro attuale dell'invocazione è* `&y`*, cioè l'L-valore di* `y` *che è intera e quindi il valore del parametro attuale ha tipo* `int *`*, cioè lo stesso tipo del parametro formale. Inoltre l'invocazione* `f(&y)` *viene dereferenziata e anche questo è corretto in quanto la funzione restituisce un puntatore. Quindi anche il main è corretto. Vediamo cosa stampa il programma. La funzione* `f` *riceve l'L-valore di* `y` *che diventa l'R-valore di* `x` *e quindi* `*x=5;` *assegna 5 alla variabile* `y` *puntata da* `x`*. La funzione* `f` *restituisce il puntatore a* `y` *e quindi *f(&y)=25; assegna 25 a* `y`*. Quindi il programma stampa 25.*

```
int * f(int * x) {*x=5; return x;}

main()
{
  int y=1;
  *f(&y)=25;
  cout<<y<<endl;
}
```

Esercizio Risolto 6.12

```
int **f(int * x){*x=5; return &x;}
main()
{
  int y=1;
  **f(&y)=25;
  cout<<y<<endl;
}
```

La funzione `f` *riceve per valore un* `int*` *che diventa R-valore di* `x`*. Usa *x e fin qui va tutto bene e poi restituisce* `&x`*, cioè un valore di tipo* `int **`*. Quindi è coerente col tipo del risultato dichiarato. L'invocazione del main ha come parametro attuale* `&y` *che è un* `int*`*, quindi uguale al tipo del parametro formale. Inoltre l'invocazione restituisce un* `int **` *e quindi è corretto dereferenziare 2 volte questo valore come fa il main. L'assegnazione di 25 ad una variabile intera è anch'essa corretta. Tutto ok? Purtroppo, no. Questo programma nasconde un errore molto grave che non è segnalato dal compilatore e che raramente viene segnalato anche a run time. L'errore è quello che nella Sezione 6.4 abbiamo chiamato un dangling pointer. Infatti* `f` *restituisce* `&x` *che è l'indirizzo del parametro formale* `x` *della funzione, cioè l'indirizzo di una variabile che viene deallocata quando la funzione esegue il* `return`*. Quindi quando il main dereferenzia una volta il valore restituito dalla funzione, accede una locazione RAM deallocata (quella della variabile locale* `x`*). Potrebbe essere che nel frattempo questa locazione sia già usata per qualche altro scopo ed allora probabilmente (ma non sempre) la seconda dereferenziazione operata dal main (che tratta il contenuto della locazione come un indirizzo) produrrà un errore di segmentation fault a run time, oppure la locazione contiene ancora l'L-valore di* `y` *e quindi la seconda dereferenziazione ci porta a* `y` *a cui assegneremmo il valore 25. In questo caso, tutto sembrerebbe perfetto, mentre invece il programma ha un errore logico dovuto al dangling pointer. Quindi in generale avremmo un programma che per certe esecuzioni sembra corretto e per altre esibisce comportamenti inattesi. Ecco perchè questi errori possono essere difficili da trovare.*

Esercizio Risolto 6.13

```
int & f(int * & x){*x=5; return *x;}
main()
{
  int y=1; int *z=&y;
  f(z)=25;
  cout<<y<<endl;
}
```

In questa funzione `f` *riceve un parametro per riferimento di tipo* `int *`. *Il parametro attuale dell'invocazione* `z` *è correttamente una variabile di tipo* `int *`. *Nella funzione il puntatore* `x` *viene dereferenziato e poi viene restituito l'oggetto puntato da* `x`. *Visto il tipo di* `x`, *l'oggetto puntato è una variabile intera e quindi ritornare* `*x` *è coerente con il tipo* `int &` *del risultato. Visto che la funzione restituisce una variabile, è corretto che l'invocazione sia alla sinistra dell'assegnazione. Il riferimento restituito da* `f` *non è un dangling pointer, infatti viene restituito l'oggetto puntato da* `x` *che è alias di* `z`. *Quindi* `f` *restituisce la variabile* `y` *del main. Quindi y prende il valore 25.*

6.6 Esercizi proposti

Esercizio 6.14 *Dato il seguente programma:*

```
int k=5, *z=&k;
*f(&z)=k+5;
cout<<*z <<endl;
```

Scrivere una funzione f(..) tale che, invocata dal programma precedente, produca la stampa di 10.

Esercizio 6.15 *Il seguente programma è corretto? Se si, dire cosa stampa, se no, spiegare l'errore.*

```
char * C(char x, char &y) {x='b'; y=x; return &x;}
main(){
  char A[] = {'a','b','c'}, *p;
  p=C(A[2],A[0]);
  cout << A[0] << A[1]<< A[2] << endl;
}
```

Esercizio 6.16 *Il seguente programma è corretto? Se si dire cosa stampa, se no spiegare l'errore.*

```
char & C(char &x, char &y) {y=x; return x;}
main(){
char A[] = {'a','b','c'};
C(A[0],A[1]) = A[2];
cout << A[0] << A[1] << A[2] <<endl; }
```

Per tutti gli esercizi che seguono, la soluzione deve contenere pre- e post-condizione ed i principali cicli iterativi devono avere il loro invariante.

Esercizio 6.17 *Scrivere una funzione che riceve come parametri un array di* `int` *e il numero di elementi da considerare e cerca nell'array un elemento di valore 0. Se lo trova restituisce* `true` *e la posizione in cui l'ha trovato. Altrimenti, restituisce* `false`.

Esercizio 6.18 *Scrivere una funzione che riceve come parametri un array* `int [][5]` *ed il numero di righe* `limite_1` *e restituire il riferimento all'elemento minimo dell'array. Modificare questa funzione in modo che restituisca il puntatore all'elemento minimo della matrice. Pensate se conviene restituire questo puntatore per riferimento o per valore.*

Esercizio 6.19 *Scrivere un programma che legge da input standard i valori per inizializzare una matrice* `int [4][5]` *e moltiplica per 2 il valore dell'elemento minimo della matrice (mantenendolo al suo posto nella matrice) servendosi della prima funzione richiesta nell'esercizio precedente.*

Esercizio 6.20 *Scrivere una funzione che riceve come parametro un array ad una dimensione di* `char` *con il numero dei suoi elementi* `limite_1` *e che restituisce il valore minimo ed il valore massimo dell'array. Cercate di calcolare entrambi i valori con un unico passaggio sull'array.*

Esercizio 6.21 *Scrivere una funzione capace di ricevere un'array* `int []`*, il numero di elementi dell'array ed un intero* `n` *e che cerca tra gli elementi dell'array se c'è un valore uguale ad* `n` *e in caso ci sia, lo elimina spostando gli elementi che seguono l'elemento eliminato di una posizione verso sinistra. Nel caso nell'array ci fossero più occorrenze del valore di* `n` *va eliminata quella più a sinistra. La funzione deve restituire il numero di elementi che restano nell'array.*

Esercizio 6.22 *Modificare l'esercizio precedente in modo che vengano eliminate tutte le occorrenze di* `n` *nell'array. Gli elementi restanti dell'array (se ne restano) vanno compattati verso sinistra. Di nuovo la funzione deve restituire il numero dei valori che restano nell'array.*

Esercizio 6.23 *Scrivere una funzione che riceve come parametri un array di* `char` *e il numero dei suoi elementi* `limite_1` *e cerca in posizioni contigue dell'array la sequenza di caratteri* `'w','h','y'`. *Se la trova restituisce* `true` *e la posizione del primo carattere* `'w'`. *Altrimenti restituisce* `false`. *Conviene usare un array di* `char` *per contenere* `''why''`.

Esercizio 6.24 *Si vuole una funzione simile a quella dell'esercizio precedente, ma che cerca la sequenza di caratteri* `'w','h','y'` *nell'array anche in posizioni non contigue. Quindi si richiede che, se la posizione in cui (eventualmente) viene trovato il* `'w'` *è* `k`*, allora quella dell'* `'h'` *(se c'è) deve essere maggiore di* `k` *e lo stesso per lo* `'y'` *finale.*

Esercizio 6.25 *Si richiede di scrivere una funzione che riceve attraverso i parametri formali un array* `int [][20]`*, un intero* `n` *ed un intero* `ele` *che, a differenza degli esercizi precedenti, non rappresenta il numero di elementi della prima dimensione dell'array (che abbiamo sempre indicato con* `limite_1` *negli esercizi precedenti), ma rappresenta invece il numero totale degli elementi dell'array che vanno considerati. Per esempio se* `ele=43` *allora, l'array avrebbe le prime due righe piene (con 20 elementi) e la terza riga con solo 3 elementi. La funzione richiesta deve determinare se c'è una riga dell'array (considerando anche l'eventuale riga parziale, come la terza riga nell'esempio precedente) che contiene esattamente 2 occorrenze di* `n`*. In caso ci sia una tale riga la funzione deve restituire l'indice della riga. Altrimenti restituisce -1.*

Esercizio 6.26 *Questo esercizio serve a illustare una tecnica che è spesso utile nel caso si debba scrivere una funzione che manipoli array a molte dimensioni. Abbiamo visto in Sezione 6.2 che il parametro formale capace di accogliere un array a più dimensioni, passato alla funzione, deve avere tutte le dimensioni, successive alla prima, completamente specificate. Questo ovviamente specializza la funzione a lavorare solo con array che differiscono al massimo per la prima dimensione e spesso non si vuole accettare questa limitazione. Un modo di evitarla è quella di passare alla funzione il puntatore al primo elemento dell'array e poi di passare come ulteriori parametri i limiti delle sue dimensioni. Seguendo questa idea, una funzione in grado di accogliere array di interi a 4 dimensioni, possiede i seguenti parametri:* `f(int *M, int limite_1, int limite_2, int limite_3, int limite_4)` *e l'invocazione per passarle l'array* `int K[5][8][10][3]`*, è la seguente:* `f(***K,5,8,10,3)`.

Si chiede di scrivere una funzione che segue la tecnica appena illustrata per accogliere array di interi a 3 dimensioni e che ricevendo, oltre ai parametri che realizzano la tecnica, anche 3 parametri interi `a, b` *e* `c`*, restituisca per riferimento l'elemento* `M[a][b][c]` *dell'array passatole, dopo aver controllato che i tre interi* `a, b` *e* `c` *rispettano i limiti dell'array. In caso di indici sbagliati, la funzione deve sollevare un'opportuna eccezione che deve essere gestita in un* `main` *che invoca la funzione. In sostanza la funzione richiesta implementa il* subscripting*, con in più il controllo che l'elemento cercato sia effettivamente un elemento dell'array.*

Penso che questo esercizio non vada inserito

Esercizio 6.27 *Ancora una variante dei due esercizi precedenti: la funzione richiesta determina quante volte nell'array compaia la sequenza* `'w','h','y'` *in posizioni anche non contigue (vedi esercizio precedente) e con la restrizione che ogni elemento dell'array possa fare parte al massimo di una delle sequenze trovate. Quindi, se per esempio l'array contiene* `'w','w','h','i','h','y'`*, la risposta deve essere 1 e non 2. Mentre se contiene:*
`'w','a','h','w','y','y','h','l','y','w','i','h','e'`*, la risposta è 2.*

Capitolo 7

Estensioni del C++

In questo Capitolo presenteremo alcuni importanti nuovi concetti e costrutti del linguaggio C++. Essi ci aiuteranno ad affrontare in modo appropriato gli esercizi futuri.

7.1 Costanti

A volte conviene dare un nome a valori costanti che utilizziamo nei nostri programmi. Questo serve ad aumentarne la leggibilità. A questo scopo il C++ permette di premettere a qualsiasi dichiarazione la specifica `const` che esplicita appunto che si sta dichiarando una costante, cioè un nome il cui R-valore non cambia durante tutta l'esecuzione del programma. Per esempio la seguente è una dichiarazione della costante pi_greco: `const double pi_greco=3.14;`. Nel caso delle costanti è **sempre** necessario specificare il loro valore al momento della dichiarazione. Le dimenticanze vengono immediatamente rilevate dal compilatore con opportuni messaggi di errore.

È possibile anche definire puntatori a costanti. Ecco un esempio: `const int y=3; const int *p=&y;`. È importante osservare esattamente a cosa si applicano i `const`: il primo dichiara che `y` è costante, mentre il secondo dichiara che `p` punta ad una costante, ma `p` non è costante, cioè il suo R-valore può cambiare. Questo esempio ci serve da spunto per un paio di osservazioni utili. La prima è che il secondo `const` è necessario, cioè `const int y=3; int *p=&y;` verrebbe scartato dal compilatore come errato. Il motivo è che se `p` fosse un semplice `int *` si potrebbe modificare l'oggetto puntato da `p` che invece è la costante `y`. Al contrario, la seguente dichiarazione `int y=3; const int *p=&y;` verrebbe accettata dal compilatore. Questo comportamento si spiega con il seguente ragionamento: la specifica `const` protegge l'oggetto a cui si applica nel senso che con questa specifica l'oggetto non può essere modificato. Quindi un `const` non può venire cancellato (come in `const int y=3; int *p=&y;`), mentre è sempre possibile aggiungere un `const`, come in `int y=3; const int *p=&y;`. Questa situazione concede di modificare l'R-valore di `y` e anche quello

di `p`, ma non è ammesso modificare il valore dell'oggetto puntato da `p`, quindi, per esempio `(*p)++;` sarebbe segnalato come errore dal compilatore.

Naturalmente possiamo definire anche puntatori costanti. Questo lo si ottiene per esempio nel modo seguente: `int x; int * const p= &x;`. L'R-valore di `p` (che è l'L-valore di `x`) è costante, ma l'oggetto puntato non è costante. Quindi `(*p)++;` è corretto, mentre `p++;` è un errore. Si noti che essendo `p` costante dobbiamo inizializzarlo al momento della dichiarazione (con `&x` nell'esempio). La dichiarazione `const int * const p=&x;` dichiara un puntatore costante che punta ad un oggetto costante.

A volte è necessario "proteggere" dei parametri attuali passati per riferimento ad una funzione. Per esempio, se dobbiamo passare un array `char X[10];` ad una funzione, dobbiamo farlo passando per valore il puntatore al primo elemento dell'array e quindi consentiamo alla funzione di modificare l'array. Se volessimo invece impedire che la funzione cambi l'array, potremmo specificare che il corrispondente parametro formale ha il seguente tipo: `const char * A` oppure `const char A[]`, ovviamente se la funzione cercasse di modificare l'array con `*(A+i)=exp;` oppure `A[i]=exp;` il compilatore darebbe un errore. In certi casi può avere senso specificare `const` anche per un parametro formale passato per valore (anche non puntatore) in modo da rendere esplicito che la funzione non modifica quel parametro.

Quando una funzione ha un parametro formale dichiarato come `const T & X`, per un tipo `T` qualsiasi, il C++ consente di invocarla con un valore `V` di tipo `T` come corrispondente parametro attuale (anzichè una variabile di tipo `T`). Infatti in questo caso il compilatore C++ crea una variabile temporanea di cui `X` diventa alias e quindi tutto avviene con un'unica copia del valore `V`. Questo metodo è preferibile al passaggio per valore, che sarebbe la maniera "giusta" di passare un valore `V`, in quanto col passaggio per valore andrebbero costruite due copie di `V` e questo sarebbe un fatto negativo qualora `V` occupasse molto spazio nella memoria. In realtà questo risparmio di memoria diventa rilevante solo nel caso in cui si voglia passare alle funzioni degli oggetti (nel senso di istanze di classi) di grandi dimensioni e questo esce dai limiti del presente testo.

7.2 Tipi definiti dall'utente

Il C++ (come tutti i linguaggi di programmazione moderni) offre la possibilità al programmatore di definire dei tipi di dato che si adattano al problema in esame.

Per prima cosa dobbiamo essere precisi su cosa si intende con *definire nuovi tipi*. I tipi che introduciamo possono essere semplicemente nomi nuovi di tipi già esistenti oppure effettivamente tipi che prima della definizione non c'erano. Nel primo caso di parla di tipi **trasparenti**, nel secondo di tipi **opachi**. Per definire tipi trasparenti il C++ ha il comando `typedef` il cui uso viene illustrato dal seguente esempio. `typedef char stringa [10];` in cui viene introdotto il nuovo tipo trasparente `stringa` come un altro nome per il tipo pre-esistente array di 10

caratteri. Se successivamente dichiariamo, `stringa X[20];`, `X` ha tipo `char [20][10]`, cioè un array di 20 elementi `stringa`, cioè 20 array di 10 `char`.

L'introduzione di tipi trasparenti dovrebbe normalmente avere lo scopo di rendere il programma più leggibile. Purtroppo, nella mia personale esperienza, viene spesso raggiunto lo scopo diametralmente opposto. Quindi il mio consiglio è di evitare l'uso di `typedef`. Molto più importante è invece il ruolo dei tipi opachi del C++. Nel seguito, vedremo solo i tipi opachi più semplici, cioè i tipi enumerazione e le strutture. La parte orientata agli oggetti del C++ è quella più ricca rispetto ai tipi definiti dall'utente, ma essa esula dal programma di questo corso.

I **tipi enumerazione** consentono di introdurre nel programma delle costanti allo scopo di renderlo più facile da leggere e capire. Per esempio se il programma deve manipolare i giorni della settimana, è possibile rappresentare i giorni con degli interi, per esempio da 0 a 6, ma in questo caso potrebbe essere facile successivamente avere dubbi, per esempio, se 0 rappresenta domenica o lunedi. Evidentemente questi dubbi non ci sarebbero se potessimo usare direttamente le costanti lunedi, martedi, mercoledi, fino a domenica. Questo è possibile attraverso la seguente dichiarazione di tipo: `enum Giorni {lunedi, martedi, mercoledi, giovedi, venerdi, sabato, domenica};`. Questa dichiarazione introduce un nuovo tipo `Giorni` dando esplicitamente i valori del tipo. Questi valori sono `lunedi, martedi, mercoledi, giovedi, venerdi, sabato` e `domenica` che sono quindi costanti (anche dette **token**) utilizzabili nel programma. Dopo questa dichiarazione potremo definire variabili di tipo `Giorni` che potranno assumere come R-valore uno dei tokens elencati per il tipo `Giorni`. Per esempio, la seguente può essere una dichiarazione con inizializzazione corretta: `Giorni g=lunedi;`. È importante realizzare che i token non sono stringhe, ma costanti. In realtà i token sono realizzati dal compilatore C++ con interi 0,1,2,... ed in una lista di token il primo token viene associato all'intero 0, il secondo all'1 e così via. Si può intervenire su quest'associazione per esempio come segue: {lunedi =10, martedi=1, mercoledi=2, giovedi=9, venerdi=3, sabato=8, domenica=4};. Che i token sono realizzati con interi lo si riscontra per esempio quando si stampa un token. Infatti un comando `cout << lunedi <<endl;` produce "solo" la stampa dell'intero corrispondente al token `lunedi`. Si deve anche osservare che non è possibile usare l'operatore `>>` per leggere l'R-valore di una variabile di tipo enumerazione. In realtà, non è difficile capire perchè questo fatto succede. Gli operatori di lettura e scrittura funzionano per i tipi predefiniti, ma sarebbe decisamente troppo aspettarsi che senza il nostro intervento essi si estendessero per funzionare anche con valori di nuovi tipi definiti da noi. Anticipiamo che è d'altra parte possibile intervenire su molti operatori, tra cui quelli di i/o, per estenderli ai tipi che definiamo. Per quanto riguarda le conversioni, da quanto detto in precedenza, è ovvio che un valore enumerazione è convertibile in un intero, ma purtroppo la conversione inversa non funziona e anche gli operatori aritmetici non sono applicabili ai valori enumerazione. Insomma i tipi enumerazione hanno dei limiti d'uso importanti ed essi servono unicamente per rendere i programmi più leggibili.

I tipi **struttura** permettono di definire collezioni di valori di tipi anche diversi. Per esempio `struct S{int k; char c; double d;};` introduce un nuovo tipo `S` che consiste di tre **campi**. Per esempio, la seguente dichiarazione `S x;` introduce una variabile `x` che possiede 3 campi, indicati rispettivamente con `x.k, x.c` e `x.d` il cui tipo è specificato nella dichiarazione di `S` e cioè, rispettivamente, `int,char` e `double`. L'inizializzazione dei campi di `x` può essere fatta o esplicitamente con assegnazioni come le seguenti, `x.k=1; x.c='a'; x.d=1.35;` oppure specificando nel tipo `S` anche una funzione chiamata **costruttore del tipo S** che avrà la seguente forma, `S(int w1, char w2, double w3) {k=w1; c=w2; d=w3;};`. Si osservi che questa funzione ha lo stesso nome del tipo appunto perchè essa serve a costruire una struttura del tipo `S` e questo giustifica anche l'assenza del tipo del valore ritornato dalla funzione in quanto esso è ovviamente `S`. Un esempio di utilizzo del costruttore è la seguente dichiarazione: `S y(1,'a',1.35);` che assegna a `y` un valore di tipo `S` i cui 3 campi hanno i valori `1, 'a'` e `1.35`, rispettivamente. La maniera più semplice per fare l'output di una variabile di tipo strutturato è di stampare ogni campo separatamente. Naturalmente una struttura può contenere un campo C di un (altro) tipo strutturato ed in questo caso si dovrebbe stampare i campi contenuti in C separatamente e così via in caso di ulteriori innestamenti. Esiste un'altra maniera di stampare i tipi strutturati che consiste nello sfruttare il sovraccaricamento per introdurre una definizione della funzione di output `<<` per il tipo strutturato che ci interessa.

7.3 Nuovi comandi di controllo

Descriviamo ora brevemente alcuni comandi di controllo del C++ che a volte tornano utili. Nei primi linguaggi di programmazione degli anni 50-60 era molto usata l'operazione di salto incondizionato `GOTO`. Si è dimostrato che l'uso di questa operazione tende a produrre codice complesso e quindi nei linguaggi moderni essa è stata evitata o almeno viene usata molto raramente. Dagli anni 70 i linguaggi di programmazione sono detti strutturati a blocchi nel senso che un programma è costituito da una serie di blocchi eventualmente innestati che contengono dichiarazioni di variabili che sono visibili solo in quel blocco. Questa struttura a blocchi rende inaccettabile un salto dell'esecuzione dall'esterno all'interno di un blocco senza passare per l'inizio del blocco che produce l'allocazione delle variabili dichiarate nel blocco. Quindi le istruzioni di salto che si trovano nei linguaggi moderni rispettano la struttura a blocchi. Le istruzioni di salto viste finora si trovano nel comando condizionale (dopo il ramo `then` si deve saltare il ramo `else`), nei comandi iterativi (while e for) dalla fine del ciclo si ritorna all'inizio, i `return` delle funzioni e le `throw` delle eccezioni. Questi ultimi 2 salti sono quelli più complicati da realizzare nella pratica. La loro gestione è molto interessante, ma purtroppo trascende dai limiti di questo corso.

Ora introduciamo 2 istruzioni di salto che rispettano la struttura a blocchi. Premettiamo che queste istruzioni vanno comunque usate il meno possibile e solo se

il codice ne guadagna in semplicità , cosa che generalmente non succede.

Il comando `break` serve a uscire immediatamente da un costrutto iterativo (e da un comando di `switch` che vedremo tra breve) indipendentemente dalla condizione d'uscita del costrutto stesso. Il comando `continue`, usato anch'esso all'interno del corpo di un costrutto iterativo, causa invece il salto alla fine del corpo del costrutto stesso. Nel caso si usi il `continue` all'interno di un `for` si deve fare attenzione al fatto che non viene saltato l'incremento che viene eseguito dopo il corpo e prima d el test d'uscita. Vediamo i 2 costrutti all'opera con un semplice esempio. Cominciamo col `break`

```
int x=0,;
for(int i=0; i<10;i++)
  {
    cout << x << ' ' << i <<endl;
    break;
    x++;
  }
}
```

Questo programma stampa semplicemente 0 0 dopo di che l'esecuzione del `break` porta l'esecuzione a uscire dal ciclo. Se al posto de `break` inserissimo `continue` il programma stamperebbe: 0 0, 0 1, 0 2,...., 0 9, mostrando come il `continue` faccia saltare `x++`, ma non l'incremento del `for`: `i++`. È opportuno chiedersi come si comporti il `break` nel caso venga eseguito in un ciclo contenuto dentro un altro ciclo. Lo illustriamo con il seguente esempio.

```
for(int j=0; j<10; j++)
    {
      int x=0;
      for (int i=0; i<10; i++)
        {
          cout << x<< ' '<< i<< ' ' << j << endl;
          break;
          x++;
        }
     }

stampa: 0 0 0, 0 0 1, ...., 0 0 9
```

Dalla stampa effettuata da questo programma si evince che l'esecuzione del `break` fa uscire solo dal ciclo interno.

A volte si deve riconoscere uno tra un insieme finito (e generalmente piccolo) di valori. In casi come questi l'uso del normale condizionale rischia di rendere

il codice di difficile lettura a causa dell'innestamento dei condizionali. Il C++ ci offre un condizionale a molte via che si chiama `switch`. Esso prende la seguente forma.

```
enum colore{bianco, giallo, rosso, blu, verde} X=giallo;
  // comandi......in cui si cambia il valore di X.......
switch(X)
 {
  case bianco: X=giallo; break;
  case giallo: X=rosso; break;
  case rosso: X=blu; break;
  default: X=bianco;
  }
```

La semantica dello `switch` è trasparente: si valuta l'espressione `X` e se il suo valore è `bianco` allora si sceglie il primo caso e si eseguono le istruzioni previste per esso, cioè `X=giallo; break;`, se il valore è `giallo` si sceglie il secondo caso e così via. Se nessuno dei valori considerati dopo la key word `case:` è uguale al valore di `X` allora viene scelto il caso `default:` eseguendo i comandi previsti in questo caso. In realtà è scelta del programmatore se inserire o meno il caso `default`, ma è buona norma inserirlo.

Ci sono 2 cose importanti da sapere sullo `switch`. Per prima cosa l'espressione che regola lo `switch` (`X` nell'esempio precedente) deve avere un tipo discreto, cioè `int, char, boolean, enum` (e naturalmente short int). Non può avere tipo `double` o `struct`. Inoltre le istruzioni associate ad ogni `case` devono terminare con un `break` perchè altrimenti, una volta che un caso si applica verrebbero eseguiti tutti i comandi che seguono anche se i relativi `case` test fossero falsi. Se nell'esempio precedente non ci fossero i `break` il valore di `X` alla fine del comando sarebbe sempre `bianco`. Infine, può essere utile anche sapere che alcuni casi si possono "mettere insieme" come indicato nel seguito.

```
switch(X)
{
  case bianco: X=giallo; break;
  case giallo: X=rosso; break;
  case rosso: case verde: X=blu; break;
  default: X=bianco;
}
```

7.4 Conversioni ed operatori di cast

Per prima cosa dobbiamo capire che nei programmi è spesso necessario fare calcoli che mescolano valori di tipi diversi. Per esempio, un programma che traduce le lire in euro deve dividere un intero (che rappresenta le lire) per il `double` `1936.27`

(il valore di 1 euro in lire). Quindi praticamente ogni linguaggio di programmazione deve consentire operazioni con tipi misti. Nel seguito cercheremo di spiegare cosa succede nel C++ quando una tale espressione viene valutata.

Nella valutazione di un'espressione con valori di tipi diversi, i valori vengono resi uguali con opportune conversioni automatiche in modo che ogni operazione sia applicata a valori dello stesso tipo. Per quanto riguarda l'esempio precedente, il C++ converte automaticamente il valore intero in un `double` e poi esegue la divisione tra 2 valori double ottenendo quindi un `double` come risultato. La conversione del valore intero è necessaria perché l'hardware del computer riesce ad eseguire operazioni (come per esempio la divisione) solo tra valori di tipo uguale. Quindi nel nostro esempio sarebbe possibile converire l'intero in un `double` oppure quest'ultimo in un intero. Abbiamo detto che il C++ segue la prima strada, ma perchè? Il motivo è che questa conversione è sicura (safe), cioè qualsiasi valore intero può venire rappresentato con un `double` in modo tale da non perdere informazione sul valore. Questo per il semplice motivo che in un `double` la mantissa occupa più dei 32 bit di un intero. Quello della salvaguardia dell'informazione è il principio che guida tutte le conversioni che in C++ avvengono automaticamente durante la valutazione di un'espressione. Le conversioni sicure del linguaggio C++ sono le seguenti:

1. da `short int` a `int` e da `int` a `long int`
2. da `float` a `double` e da `double` a `long double`;
3. da `int` a `double`;
4. da `int` a `float`;
5. da `char` e `bool` a `int`;
6. da tipo `enumerazione` a `int`;

Per la verità, queste conversioni sono tutte sicure meno che quella da `int` a `float`. Infatti se cercassimo di rappresentare in floating point con 23 bit per la mantissa un valore intero la cui rappresentazione binaria avesse distanza massima tra i suoi bit a 1 maggiore di 24, la rappresentazione dell'intero sarebbe inesatta. In pratica se consideriamo l'intero $2^{24}+1$ e lo assegnamo ad un `float`, e poi riconvertiamo il valore `float` in intero, quello che otteniamo è un intero diverso da $2^{24}+1$. Si osservi che questo non succede per potenze del 2 inferiori a 24 e che inoltre non succede se invece di convertire l'intero in un `float` lo convertissimo in un `double` che usa un numero maggiore di bit per la mantissa. Comunque la conversione `int` $\rightarrow$ `float` è , in generale, meno rovinosa di quella inversa, che perde tutti gli eventuali decimali, e questo spiega la scelta del C++.

È importante sapere che se scriviamo un'espressione aritmetica che coinvolge anche solo valori di tipo `char`, essi devono in ogni caso essere convertiti in `int`

per poter valutare l'espressione, perché la CPU possiede solo operazioni aritmetiche tra interi cioè su valori che occupano 4 byte e non un solo byte. Lo stesso vale per espressioni con valoro di tipo `boolean`, `short int` ed enumerazione.

Da quanto detto sulle conversioni automatiche operate dal C++ è facile dedurre che in questo linguaggio ogni espressione che contiene valori di tipo predefinito ed operazioni applicabili a questi tipi (come per esempio le operazioni aritmetiche) possiede sempre un valore. Il tipo di questo valore sarà il tipo massimo tra i tipi delle componenti dell'espressione rispetto all'ordine totale tra i tipi: {`char, bool, enum` } → {`short int` } → {`int` } → { `float` } → { `double` }

In un'assegnazione C++, $\texttt{T}_x$ `x=exp;` il valore V prodotto dalla valutazione dell'espressione `exp` avrà un certo tipo T che potrà in generale essere diverso dal tipo $\texttt{T}_x$ di `x`. In questo caso il valore V dovrà essere convertito in un valore di tipo $\texttt{T}_x$. Ovviamente in questo caso non ci sono scelte di conversioni più o meno sicure che si possono fare: al contrario c'è una sola conversione che **deve** essere fatta; da T a $\texttt{T}_x$. Se questa conversione ricade tra quelle sicure elencate prima, essa viene inserita nel codice oggetto prodotto dal compilatore, senza che il programmatore sia neppure avvertito della cosa. Se invece la conversione va nella direzione opposta ad una conversione sicura, il compilatore la inserirà ugualmente, ma emetterà un `warning` (ammonimento) per avvertire del fatto che il programma richiede una conversione non sicura.

C'è una conversione che (forse inaspettatamente) non è consentita ed è quella tra `int` e tipo enumerazione. Il programma che la richieda causerà un messaggio d'errore del compilatore. Altri errori in compilazione possono venire causati da conversioni che riguardano tipi struttura diversi.

Riflettiamo ora su cosa significa convertire un valore di un tipo T in un'altro tipo T'. Facciamo 2 esempi. Prima di tutto consideriamo una conversione sicura: `char` → `int`. Un valore di tipo `char` è un piccolo intero tra -128 e 127, quindi viene trasformato in un intero dello stesso valore che occupa 4 byte. Il segno viene conservato. Una conversione non sicura è, per esempio, quella inversa `int` → `char`. Dell'intero viene semplicemente conservato il byte meno significativo.

A volte le conversioni automatiche non ci stanno bene e vogliamo "forzare" conversioni diverse. Per esempio, considerate l'operazione di calcolare un valore medio: `double m = v / n;` con `v` ed `n` interi. Ovviamente la divisione, avendo operandi interi, ha risultato intero che viene convertito automaticamente in `double` per venire assegnato a `m`. Visto che `m` è `double`, chi ha scritto l'assegnazione intende calcolare la media con i decimali, ma questo assegnazione invece li perde. Rimediare è facile. Dobbiamo convertire almeno uno dei 2 operandi della divisione in un `double`. A questo scopo il C (e di conseguenza il C++) ci fornisce l'operatore di **cast** che nel nostro esempio verrebbe usato nella maniera seguente: `double x= (double) v / n;`. Nell'esempio l'R-valore di `v` viene trasformato in un equivalente `double` e quindi anche `n` viene convertito in un `double` e viene usata la divisione tra `double` come desiderato. I cast C come questo sono richieste di conversione cui il compilatore "deve" obbedire. Visto che le conversioni (non sicure) sono operazioni delicate che spesso possono introdurre perdita

di informazione nei programmi, al posto del cast di C che impone al compilatore la conversione, il C++ offre 4 diversi operatori di cast ognuno dei quali si adatta ad un particolare tipo di conversione. È importante capire qual'è il vantaggio di avere 4 cast anzichè uno solo. Questo è un punto che nuovamente rivela il forte cambiamento rispetto all'utilità dei tipi nei linguaggi di programmazione che si è verificato dagli anni 60 ai 90. Dovendo il programmatore inserire il cast appropriato per la conversione che sta chiedendo di operare (pena un errore di tipo in compilazione), egli è forzato a capire esattamente la conversione stessa. Inoltre questi cast hanno anche la funzione di commenti che "spiegano" quale conversione viene richiesta. I cast del C++ sono i seguenti:

- `static_cast`: serve per richiedere conversioni sicure e più spesso per chiedere conversioni che vanno nella direzione opposta delle conversioni sicure. Per esempio se vogliamo trasformare un `double x` in un `int y` possiamo scrivere `y= static_cast<int>(x);`, con questa istruzione l'R-valore di `x` viene convertito in un "equivalente"valore `int` che diventa l'R-valore di `y`. L'aggettivo equivalente è tra virgolette ad evidenziare la possibile perdita di informazione causata dalla conversione.

- `const_cast`: serve a trasformare un puntatore ad una costante di tipo T in un puntatore ad un tipo T non costante. Un esempio d'uso è il seguente:

  ```
  const int x=100, * p=&x;
  int * q=const_cast<int *>(p);
  (*q)++;
  cout<< x << *q << *p << endl;
  ```

  Questo è un programma corretto ed ha un comportamento sorprendente: il comando di stampa stampa 100 101 101. Sembrerebbe che ci siano 2 x: quella costante e quella puntata da p e da q. Ma non è così. Il punto è che il compilatore sostituisce nel codice sorgente ogni occorrenza del nome di una costante con il suo valore. Questo è possibile visto che le costanti vanno inizializzate al momento della loro dichiarazione e l'espressione usata per l'inizializzazione è composta solo da costanti (possono compaire anche altre costanti introdotte prima). Quindi nell'esempio precedente $cout << x << \ldots$ il compilatore produce istruzioni che stampano 100 e il successivo cambiamento in 101 non ha effetto su questa stampa. Deve essere chiaro che esiste un solo x il cui L-valore non cambia ed è semplicemente $\&x$. Si deve osservare che il `const_cast` non trasforma alcun valore. Solo consente di usare un puntatore in modo diverso rispetto all'originale, ma il valore del puntatore resta lo stesso.

- `reinterpret_cast`: serve a cambiare un qualunque puntatore ad oggetti di un tipo ad un puntatore ad oggetti di un altro tipo. Deve essere chiaro che questa conversione non produce, al contrario di quelle prodotte con

`static_cast`, alcuna trasformazione di un valore da una certa rappresentazione ad un'altra (per esempio da floating point a intero in complemento a 2). Qui il valore del puntatore a cui si applica questo cast non cambia. Semplicemente l'oggetto a cui il puntatore punta, dopo il cast, viene considerato di un tipo diverso da quello originale. Quindi per esempio se abbiamo `char *x; double *y;... x=reinterpret_cast<char*>(y);` il primo byte puntato da `y` (cioè, probabilmente, il primo degli 8 byte che servono per rappresentare un valore `double`) viene interpretato attraverso `x` come contenente un `char`. Per inciso, in questo modo è facile andare a "leggere" byte a byte come sono rappresentati valori di ogni tipo, anche di tipo struttura. Va da se che questi cast sono estremamente pericolosi e, se fatti con poca attenzione, possono causare errori difficile da individuare.

- L'ultimo cast che il C++ ci offre è il `dynamic_cast`. Il suo scopo è impossibile da capire con quello che attualmente conosciamo del C++. Nei programmi C++ che usano la parte orientata agli oggetti, durante l'esecuzione dei programmi, l'R-valore a run-time delle variabili può avere un tipo diverso da quello dichiarato per esse nel testo del programma sorgente (chiamato **tipo statico**). Il `dynamic_cast` permette di conoscere il **tipo dinamico** (detto anche tipo a run-time) delle variabili ed a scegliere le azioni da fare in base a questo tipo. Il suo uso verrà spiegato nel dettaglio in corsi maggiormente avanzati di C++.

C'è un'altra situazione in cui le conversioni automatiche entrano in gioco ed è nell'invocazione delle funzioni con il passaggio dei parametri per valore. Trattando di questo punto, nella Sezione 6.2, abbiamo scritto che l'invocazione di una funzione con il passaggio per valore deve avere parametri attuali che concordano con quelli formali sia per numero che per tipo. Questa affermazione è purtroppo spesso violata in vari modi. Per quanto riguarda violazioni sul numero dei parametri attuali, abbiamo visto nella Sezione 6.2, che il C++ permette di scrivere funzioni in cui alcuni parametri (passati per valore) hanno valori di default. Un esempio aiuterà a ricordare:

```
int f(char x, int y=0)
{ return x+y;}

main()
{cout << f( 'a',2)<< ' '<< f('c')<<endl;}
```

Entrambe le invocazioni presenti nel `main` fanno riferimento alla funzione `f`. La prima provvede un valore per entrambi i parametri formali e restituisce il valore `97+2` (97 è il codice ASCII di `'a'`). La seconda invocazione ha un solo parametro attuale `'c'` che viene associato al parametro formale `x`, quindi il parametro formale `y` assume il valore di default `0`. Il valore restituito da questa invocazione è ancora `99`. È importante ricordare che gli argomenti formali con valori di default

devono sempre essere quelli più a destra in modo da avere una sola associazione possibile tra i parametri attuali presenti ed i formali.

Oltre ai parametri di default, il C++ permette che con il passaggio per valore i parametri attuali di un'invocazione abbiano tipo diverso dai corrispondenti formali. In questo caso, quello che succede è semplice: il C++ applica le conversioni automatiche appena viste, per convertire i valori dei parametri attuali nei tipi dei corrispondenti parametri formali. Si consideri l'esempio:

```
int g(char x)
{......}
............
cout<<g(45.67); // accettata, con un semplice warning !
// il compilatore converte  double --> char !
```

Insomma vengono applicate le stesse conversioni automatiche (con o senza warning) che vengono usate per l'assegnazione. Quindi sui tipi base (non puntatori), l'unica che da errore è la conversione `int --> enum`. Al contrario nessuna conversione è consentita tra puntatori diversi (a parte l'aggiunta di `const`). Per "forzare" queste conversioni si deve usare l'operatore `reinterpret_cast` visto prima.

Questo fenomeno delle conversioni applicate automaticamente per "far funzionare" le invocazioni, si lega nel C++ all'overloading delle funzioni, facendo spesso nascere problemi non banali. La Sezione **??** tratta l'overloading in dettaglio.

Le possibili discrepanze tra parametri attuali e corrispondenti parametri formali si possono verificare esclusivamente nel caso di passaggio dei parametri per valore. Quando i parametri sono passati per riferimento, il parametro attuale deve essere (quasi) sempre una variabile dello stesso tipo del corrispondente parametro formale. L'unica eccezione è quella già spiegata nella Sezione 7.1 ed è quella in cui il tipo del parametro formale sia `const T &` e il corrispondente parametro attuale sia un'espressione il cui valore sia di tipo `T`.

7.5 Gestione delle eccezioni

Durante l'esecuzione di un programma possono verificarsi delle situazioni inattese, queste situazioni vengono chiamate **eccezioni**. Nel passato il programmatore trattava le eccezioni inserendo degli appropriati input nel suo programma. Non sempre questa soluzione è soddisfacente. Consideriamo per esempio che l'eccezione avvenga durante l'esecuzione di una funzione `F`. Potremmo volere interrompere subito l'esecuzione di `F`, tornare in qualche punto speciale del programma nel quale eseguire qualche operazione volta a risolvere il problema creatosi, e continuare l'esecuzione da lì. Riuscire a compiere queste operazioni non sarebbe semplice usando i meccanismi normali dell'invocazione delle funzioni. Si tratterebbe di prevedere che certe funzioni possano ritornare in modo "normale", ma anche in

modo “anormale” compiendo cose diverse nei 2 casi. I linguaggi di programmazione moderni come il C++ offrono dei comandi che permettono di fare queste operazioni in modo semplice. In pratica questi comandi permettono di definire dei punti (a piacimento) del programma in cui l’esecuzione salta e nei quali vengono eseguite le operazioni che gestiscono le eccezioni. Quando un’eccezione si verifica, viene eseguita l’operazione che “solleva”(`throw`) quell’eccezione ed il controllo automaticamente passa al punto del programma in cui quel tipo di eccezione viene gestita. Ci fossero più punti possibili per la gestione di un’eccezione, verrebbe usato quello più vicino al punto in cui l’eccezione si è verificata (rispetto alla sequenza delle funzioni in esecuzione).

Vediamo in dettaglio. I comandi per la gestione delle eccezioni nel C++ sono i seguenti tre:

1. **try**: serve a individuare un blocco di una funzione che contiene istruzioni che possono causare il sollevamento di eccezioni. Alla fine del blocco `try` ci sono istruzioni `catch`, descritte dopo, che gestiscono le eccezioni sollevate.

2. **throw(T)**: serve a sollevare (lanciare) un’eccezione che consiste di un valore di un tipo `T` qualsiasi. L’eccezione che lancia un valore di tipo `T` viene gestita dal più vicino `catch(T x)`, come spiegato nel punto successivo.

3. **catch(T e)**: introduce un gestore d’eccezioni di tipo `T` che in pratica si comporta come una funzione con parametro formale `e` di tipo `T`. Ogni istruzione di `catch` è seguita da un blocco che viene eseguito quando il `catch(T e)` “cattura” un’eccezione (naturalmente di tipo `T`). In questo blocco si trovano i comandi che “gestiscono” l’eccezione e che, in generale, usano il parametro formale `e`.

Vediamo un esempio.

```
#include<iostream>
using namespace std;
int g(int x)
{
  if(x<0)
    throw(true);
  else
    return x+2;
}

bool f(int x)
{
  if(x<0)
    throw(1);
  else
```

```
    if(g(x-1)>0)
      return true;
    else
      return false;
}


main()
try{
  if(f(0)) cout<<"SI"<<endl;
  else
    cout<<"NO"<<endl;
}
catch(int x){cout<<"eccezione intera="<<x<<endl;}
catch(bool x) {cout<<"eccezione booleana="<<x<<endl;}
```

Il `try` racchiude l'intero corpo del `main` il che significa che qualsiasi eccezione venga sollevata durante l'esecuzione del programma dovrebbe venire gestita da qualche `catch` presente nel programma. Se questo non fosse vero il programma sarebbe sbagliato, come vedremo dopo. In questo programma la funzione `g` solleva un'eccezione lanciando il valore `true`. A questo punto il controllo passa immediatamente al `catch` che aspetta un valore `bool` e che stampa quindi "eccezione booleana" seguita da 1 (per `true`).

Restano vari aspetti da chiarire. Che succede se viene sollevata un'eccezione di un tipo `T` che nessun `catch` aspetta? Per prima cosa entrano in gioco le conversioni automatiche del C++ (vedi Sezione 7.4). Se neppure con le conversioni si trova un match allora semplicemente il controllo passa al sistema operativo. Insomma il programma è sbagliato e termina con un messaggio di terminazione non normale. Cerchiamo anche di chiarire quale `catch` viene eseguito quando ce n'è più di uno compatibile con l'eccezione sollevata. A questo fine, nell'esempio precedente modifichiamo la funzione `f` come segue:

```
bool f(int x)
{
  if(x<0)
    throw(1);
  else
    try{
    if(g(x-1)>0)
      return true;
    else
      return false;
    }
  catch(bool x) {cout<<" nuovo catch bool"<<x<<endl;}
```

```
  return false;
}
```

Ovviamente il `catch` introdotto in `f` è più vicino al punto in cui l'eccezione booleana viene sollevata (cioè `g`) e quindi verrà scelta al posto di quella del `main`. Una seconda cosa importante illustrata da questo esempio è che dopo l'esecuzione del corpo del `catch` l'esecuzione continua normalmente, cioè esegue il `return false;` e quindi il controllo torna al `main` col valore `false` e stampa `NO`. Insomma dopo un `throw` ed il corrispondente `catch` l'esecuzione continua normalmente.

Un ultimo aspetto importante è che si possono sollevare eccezioni di ogni tipo e non solo di uno dei tipi predefiniti. Per esempio si possono definire strutture "ad hoc" per le eccezioni che contengono campi adatti a contenere informazioni relative all'eccezione da gestire (per esempio stringhe con messaggi diagnostici).

7.6 Contenitori del C++

In questa Sezione descriveremo brevemente alcuni **contenitori** del C++ che risultano utili per scrivere programmi corretti. Queste strutture dati sono create utilizzando la parte orientata agli oggetti del C++, quindi, visto che questo testo non tratta questa parte del C++, le descriveremo quanto basta per imparare a servircene senza portare a fondo la trattazione. Il C++ definisce classi, dette contenitori, che servono a gestire dati per mezzo di apposite funzioni, dette **metodi**. Le classi che esamineremo sono la **string**, la **vector**, e la **map**. Le ultime due classi sono definite nella **Standard Template Library (STL)**. La STL contiene molte altre classi. Informazioni più dettagliate si posso trovare in testi sul C++ e anche in rete sul sito www.cplusplus.com.

La classe `string` serve a realizzare sequenze di caratteri, mentre la classe `vector` serve a realizzare sequenze ordinate di elementi di un tipo qualsiasi. Quindi sia `string` che `vector` ri-definiscono cose che sono già presenti nel C (e quindi nel C++), ma queste ri-definizioni portano dei vantaggi rilevanti rispetto all'esistente. Le nuove classi sono capaci di accomodare dinamicamente un numero variabile di elementi. Quindi queste strutture si allungano e si accorciano a seconda del bisogno ed in modo trasparente all'utilizzatore. Inoltre esse rendono impossibile commettere errori tipici dell'uso degli array tradizionali, quali accedere elementi fuori dai limiti dell'array. Infine tutte queste classi offrono `iteratori` che permettono di percorrerle in modo uniforme. Mentre `string` è specializzato a gestire stringhe di caratteri, `vector` permette si definire vettori di elementi di tipi qualsiasi ed è detta perciò una classe `template`. Il terzo contenitore che introdurremo è il `map` è anch'esso un template e permette di definire tabelle **associative**, cioè insiemi di coppie di valori di tipo diverso in cui una delle due componenti è la **chiave** attraverso la quale è possibile accedere in modo efficientealla seconda componente associata a quella chiave.

Iniziamo dalla classe `string`. Innanzitutto, per usare questo tipo, i programmi devono contenere all'inizio la direttiva `#include<string>` di inclusione della relativa libreria. La dichiarazione di una stringa vuota è semplicemente: `string X;`, essendo vuota, `X.size()==0`. Il metodo `size()` restituisce sempre il numero di caratteri della stringa. Tutti i metodi di una classe e quindi di ogni oggetto di quella classe vengono invocati nello stesso modo in cui viene invocato il metodo `size()` e cioè con l'operatore di selezione . (punto). Si osservi che l'operatore di selezione è lo stesso che abbiamo già incontrato all'inizio del Capitolo per la selezione dei campi di una struttura in Sezione 7.2. Le classi del C++ sono infatti un'importantissima estensione delle strutture che erano già presenti nel C.

Data una stringa `X`, possiamo dichiararne un'altra con lo stesso valore con, `string Y(X);`. Una stringa (come ogni contenitore) viene acceduta tramite **iteratori** (che funzionano come i puntatori). Un iteratore per un valore `string` viene dichiarato da: `string::iterator p;`. Se `S` è una stringa, un iteratore che punta al suo primo elemento è `p= S.begin();` mentre un iteratore alla fine della stringa (all'elemento successivo all'ultimo) è `p= S.end();`. `p++;` passa all'elemento successivo se c'è e segnala errore altrimenti. Per esempio il seguente è un ciclo che percorre una stringa dall'inizio alla fine stampandone un carattere alla volta in colonna:

```
string X=''pippo'';
for(string::iterator p=X.begin(); p!= X.end(); p++)
cout<<*p<<endl;
```

In questo esempio vediamo che possiamo inizializzare la variabile `X` di tipo `string` con una stringa C. Si ricordi che in questo caso il carattere `'\0'` non compare in `X`. Per appendere una porzione di stringa (`string` o stringa alla C) ad una `string` `X` possiamo eseguire: `string X=''pippo'', Y=''topolino''; X.append(Y, 0,3); X.append(''topolino'',3,5);`. Il primo `append` appende i primi 3 caratteri "top" di "topolino", mentre il secondo `append` si occupa dei restanti 5 caratteri, cioè "olino". Quindi dopo il primo `append`, `X=pippotop` e dopo avere eseguito entrambi gli `append` `X=pippotopolino`. L'operatore `+` è un'altro modo per appendere una stringa (o un carattere) ad un'altra. Per esempio, `string X=''pippo'', Y=''topolino''; X=X+''topolino''; X=X+Y;`. Appende 2 volte "topolino" a `X`. Si osservi anche:

```
string X=''pippo'', Y=''topolino'';
for(int i=0;i<Y.size();i++)
X+=Y[i];
```

in cui percorriamo `Y` col subscripting e appendiamo a `X` un carattere alla volta. Questo esempio mostra la coesistenza nel C++ di 2 modi diversi di accedere alle stringhe: il subscripting degli array (eredità del C) e gli iteratori tipici dei contenitori. Questo fatto lo vedremo anche negli altri contenitori.

Se X e Y sono stringhe come negli esempi precedenti, allora `X.insert(2,Y);` inserisce la stringa Y dentro alla X da prima del carattere in posizione 2 (partendo da 0). Il risultato è la stringa: `''pitopolinoppo''`. Con `X.erase(2,8);` si riottiene `''pippo''` come valore di X.

È a volte utile anche cercare una stringa in un'altra, in generale questo problema si chiama **pattern matching**. Il metodo che esegue questa azione è naturalmente `find` che si usa come segue: `string S=''topolino'', Y=''po''; cout<<S.find(Y)<<endl;`. Questo programma stampa 2 che è la posizione in cui inizia il match. È possibile anche iniziare il match in una posizione qualsiasi con:

```
string S=''topolino'', Y=''po''; cout<<S.find(Y,3)<<endl;
```

che ovviamente non troverebbe il match e quindi stamperebbe un particolare campo `string::npos` della classe `string` che rappresenta la massima lunghezza delle stringhe rappresentabili. Non importa quale sia il valore di `npos`. La cosa importante è che usando il nome `string::npos` si può testare se il find ha avuto successo o no.

Sul tipo `string` sono definite inoltre molte operazioni tra cui, l'assegnazione, l'uguaglianza, i confronti, il subscripting e l'input/output. Si deve fare attenzione ad usare il subscripting per modificare una stringa. Per esempio:

```
string X=''pippo''; X[0]='b'; X[5]='z'; X[7]='w'; cout<<X;
```

stamperebbe `''bippo''`. Quindi mentre la prima assegnazione funziona come atteso, le successive 2 che si riferiscono a elementi non esistenti della stringa, non hanno effetto e non producono neppure un messaggio d'errore.

Veniamo ora al contenitore `vector`. Per usarlo dobbiamo inserire il comando `#include<vector>` nel nostro programma. Per dichiarare un `vector` d'interi dovremo scrivere nel programma: `vector<int> A;` Da questo dovrebbe essere chiaro che non si specifica il numero d'elementi del vettore A (esattamente come per le stringhe). Gli oggetti istanze della classe `vector` sono "allungabili" in modo assolutamente trasparente al programmatore che li usa. Per la nozione di array "allungabili", vedi Sezione **??**. Se invece preferiamo specificare un numero iniziale di elementi del vettore, possiamo farlo con la dichiarazione: `vector<int> A(100);`. Con `vector<int> A(100,1);` specifichiamo che tutti i cento elementi devono essere inizializzati a 1. Con `vector<int> B(A);` dichiariamo un nuovo vettore B che è una copia di A (ma è indipendente da esso). Chiariamo il ruolo del tipo `<int>` nella precedente dichiarazione. Come già anticipato, la classe `vector` è una classe **template**, cioè la classe ha un parametro di tipo che per ottenere una classe vera deve essere istanziato con un tipo concreto. Nell'esempio precedente lo abbiamo istanziato con `int`. Naturalmente possiamo dichiarare variabili di tipo `vector<T>` per ogni tipo T predefinito e definito dall'utente (quindi anche strutture, enumerazioni, e contenitori).

I metodi della classe `vector<T>` sono simili a quelli visti prima per `string`. `A.size();` restituisce il numero d'elementi di A,

```
vector<int>::iterator p=A.begin();
```

è l'iteratore che punta al primo elemento di `A` e `A.end()` punta all'elemento successivo all'ultimo. Nello stesso modo di prima possiamo percorrere un `vector` usando gli iteratori. I vettori hanno anche metodi `erase` e `insert` per cancellare parte del vettore ed inserirvi nuovi elementi. Vediamo alcuni semplici esempi:

```
vector<char> C(6, 'a'); C[2]='b';C[4]='c';
vector<char>::iterator p=C.begin()+2, q=p+2;
C.erase(p,q);
for(int i=0; i<C.size();i++)
cout<<C[i]<<endl;}
```

Alla fine di questo programma, il vettore `C` ha i seguenti 4 elementi: `aaca`. Potremmo inserire nuovamente gli elementi tolti da `C` aggiungendo al precedente programma le seguenti 2 istruzioni:

```
p=C.insert(p,'a');
p=C.insert(p,'b');
```

Per finire, il metodo `push_back` serve ad appendere un nuovo elemento alla fine dell'array. Esso viene usato nel modo seguente: `C.push_back('z');` aggiunge un elemento di valore 'z' in fondo all'array `C`. Sono definite le operazioni di confronto tra `vector` dello stesso tipo.

La classe template `map` ha 2 parametri di tipo, cioè `map<KT,VT>` in cui `KT` è il tipo della chiave e `VT` il tipo della seconda componente (valore). Per esempio, `map<char,string> M;`. Per inserire una nuova coppia dentro `M`, usiamo il metodo `M.insert(pair<char,string>('a',''aiuola''));`. Per cercare la coppia associata ad una chiave usiamo: `map<char,string>::iterator p= M.find('a');` Se la chiave è presente allora `p` è l'iteratore che punta alla coppia con quella chiave. Altrimenti riceve `M.end()`. Se `p` punta ad una coppia esistente in `M`, allora `(*p).first` è il primo elemento della coppia puntata da `p` (cioè la chiave), mentre `(*p).second` è la seconda componente. Con `M.erase(p);` eliminiamo la coppia puntata da `p`. I metodi `size(), begin()` e `end()` si applicano anche sui `map`.

Capitolo 8

Ricorsione

La ricorsione è una tecnica di programmazione che permette in molti casi di scrivere soluzioni eleganti, concise e corrette. In questo capitolo introduciamo la ricorsione applicandola a problemi via via più complicati. La ricorsione mostra appieno i suoi benefici quando si affrontano problemi che riguardano strutture ricorsive come le liste concatenate e gli alberi binari. Tipicamente le liste e gli alberi binari servono per modellare situazioni dinamiche in cui la struttura deve potersi modificare per riflettere modifiche della situazione che rappresenta. Questo comportamento è reso possibile da un apposito comando del C++ che permette, durante l'esecuzione di un programma, di creare strutture dati la cui esistenza sfugge alle regole delle variabili automatiche viste finora. Queste strutture dinamiche restano a disposizione finchè il programma stesso non le distrugge con un'istruzione apposita. Questa gestione è infatti chiamata gestione dinamica dei dati.

8.1 Programmazione Ricorsiva

Una funzione è **ricorsiva** se contiene nel suo corpo almeno un'invocazione di se stessa. In questo caso si parla di **ricorsione diretta**. Delle funzioni sono dette **mutuamente ricorsive** se una di esse ne chiama un'altra e così via finchè non si torna ad invocare di nuovo la prima. Il fatto che una funzione invochi se stessa (direttamente o indirettamente) non cambia in alcun modo il meccanismo generale di gestione delle invocazioni delle funzioni. Ogni invocazione di una funzione causa l'allocazione sulla memoria a stack dello spazio necessario a contenere le variabili locali ed i parametri formali della funzione e dell'indirizzo di ritorno. Nel caso di una sequenza di invocazioni di una stessa funzione ricorsiva, succede esattamente quello che succederebbe per una sequenza di invocazioni di funzioni diverse, con la sola eccezione del fatto che l'indirizzo di ritorno di queste invocazioni saranno tutti indirizzi di istruzioni che appartengono alla stessa funzione ricorsiva. Si deve evitare quindi di pensare che, nel caso di funzioni ricorsive, possano essere contemporaneamente attive varie "copie" della stessa funzione. Come per le funzioni non ricorsive ci sarà sempre una sola invocazione attiva (l'ultima effettuata)

mentre le altre (che l'hanno causata) saranno in attesa di ripartire quando l'invocazione che hanno effettuato eseguirà il suo `return`. Nonostante che l'intuizione delle copie multiple della funzione ricorsiva sia falsa, in alcuni casi la useremo per semplificare la spiegazione.

Ogni funzione ricorsiva deve possedere almeno un **caso base**, cioè una condizione che, qualora soddisfatta, causa la terminazione dell'esecuzione del corpo della funzione senza che venga eseguita alcuna ulteriore invocazione ricorsiva. Il caso base è una condizione sui valori dei parametri della funzione. Ovviamente, se non ci fosse almeno un caso base, avremmo una sequenza infinita di invocazioni della funzione ricorsiva, cioè un programma che non termina che è ovviamente un errore.

Molti esercizi che coinvolgono soluzioni ricorsive sono illustrati nella Sezione 8.2 e seguenti. Tra ricorsione ed iterazione c'è una strettissima relazione: il calcolo eseguito da un qualsiasi programma che usa una delle 2 può essere eseguito anche da un programma che usa l'altra. Nella Sezione 8.6 si discute, con un semplice esempio, del fatto che ogni funzione ricorsiva può venire simulata da una funzione iterativa. Discutiamo qui la direzione più semplice, cioè che ogni ciclo iterativo può venire simulato da una funzione ricorsiva. Di nuovo affrontiamo la questione con un esempio. Si consideri il seguente ciclo `while`:

```
int x=10, y=0, z=1;
while(x>y)
{z=z+x; x--;}
cout<<z<<endl;
```

esso calcola un valore per `z` e lo stampa. Il valore calcolato dal ciclo è `1+10+9+8+...+1=56`. Vediamo come fare lo stesso calcolo con una funzione ricorsiva.

```
int simula_while(int x, int y, int z)
{
  if(x<=y) return z;  // caso base
    else
     return x+simula_while(x-1,y,z);
}

// invocazione
int x=10, y=0, z=1;
cout<<simula_while(x,y,z)<<endl;
```

Per prima cosa, cerchiamo di capire perchè `simula_while` fa la stessa cosa del ciclo `while`. Il ciclo `while` somma ripetutamente `x` a `z` e contemporaneamente diminuisce `x` di 1 ad ogni iterazione. La funzione `simula_while` invoca se stessa con parametro attuale `x-1`. Quindi avremo una sequenza di 11 invocazioni ricorsive. Il valore del parametro formale `x` è 10 per la prima invocazione,

diventa 9 per la seconda e per l'ultima è 0 e a questo punto si applica il caso base `x <=y` e la funzione ritorna `z` che è 1. Ma dove si ritorna ? Si ritorna all'esecuzione precedente della funzione `simula_while`, cioè quella con `x=1` ed esattamente si ritorna ad eseguire `x+1` dove l'1 è il valore restituito dall'invocazione di `simula_while` con `x=0`, quindi quella con `x=1`, ritorna 2 all'invocazione precedente con `x=2` che a sua volta ritorna 2+2 all'invocazione precedente,cioè con `x=3` che ritorna 3+4 e così via. Dovrebbe essere chiaro che il valore restituito dalla prima invocazione con `x=10` al chiamante è `10+9+8+...+1+1=56`. Si osservi che la funzione `simula_while` usa la negazione del test del `while` come caso base. Questo è dovuto al fatto che se il test è vero il ciclo deve continuare, mentre se il caso base è vero la ricorsione deve terminare. Si osservi inoltre che l'espressione `x+simula_while(x-1,y,z)` della funzione simula le assegnazioni del corpo del `while`. In particolare, il passaggio del valore `x-1` simula `x--` dato che ogni invocazione ha la sua copia del parametro formale `x`, mentre nel ciclo c'è una sola variabile `x`.

8.2 Esempi di Ricorsione

In questa breve sezione introduciamo degli esempi di funzioni ricorsive.

Esercizio Risolto 8.1 *Scrivere una funzione ricorsiva* `void reverse()` *che legge da* `cin` *una sequenza di caratteri che termina con* `';'` *e la stampa a video (';' compreso) in ordine inverso.*

```
void reverse()
{
 char x ;
 cin>>x;
 if (x!=';')
    reverse();
 cout << x ;
}
```

Un esercizio simile è il seguente. Si tratta di scrivere una funzione ricorsiva `int conta()` *che legge da* `cin` *una sequenza di caratteri (che termina con ';') e li conta (senza considerare il ';' finale) restituendo il risultato.*

```
int conta()
{
  char x;
  cin>> x;
  if (x==';') {return 0;}
  else
   return 1 + conta();
}
```

Esercizio 8.2 *Ecco due variazioni della funzione* `conta` *dell'esercizio precedente:*

- *Modificare* `conta` *in modo che conti anche il ';' finale.*
- *Si scriva una funzione ricorsiva* `int conta(char ch)` *che riceve come parametro* `ch` *un carattere, legge da* `cin` *una sequenza di caratteri (che termina con ';') e restituisce quante volte il carattere* `ch` *vi appare.*

Esercizio Risolto 8.3 *Si vuole realizzare una funzione ricorsiva* `int ins(char * A, int p, int limite)` *che riceve come parametri un array* `A` *di* `limite` *posizioni di tipo* `char` *e un indice* `p` *che è inizialmente 0 e che cresce di 1 ad ogni ricorsione per contare i caratteri letti. Infatti, la funzione deve leggere da* `cin` *una sequenza di caratteri (che termina sempre con ';') e deve inserirli in posizioni contigue di* `A` *fino a che c'è posto oppure fino a che non viene trovata la sentinella ';' che va inserita in* `A` *anch'essa. La funzione deve restituire il numero di caratteri inseriti in* `A`*.*

```
int ins(char *A, int p, int limite)
{
 if(p==limite) // finito il posto in A, ritorno 0 caratteri
   return 0;

 char x;
 cin >> x;
 if (x==';') //c'e' posto in A, ma leggo ';'
             //quindi la ricorsione
             //finisce e ritorno 1 (char inserito)
   {A[p]=x; return 1;}
 else  // inserisco il carattere e ricorro con p+1
       //per cercare di inserire il prossimo carattere
       //se ci sarà ancora posto in A
  {
   A[p]=x;
   return 1+ins(A,p+1,limite);
  }
}
```

Cerchiamo di scrivere la pre- e la post-condizione per `ins`*:*

- **PRE** =**(**`cin` **contiene** $a_1 \dots a_k$';', $k \geq 0$, $0 \leq$ `p` $\leq$ `limite`**)**;
- **POST** =**(se** $k \geq$ `limite-p` **allora** `A[p..limite-1]`=$a_1 \dots a_{\texttt{limite}-\texttt{p}}$ **e ritorno** = `limite-p`**, altrimenti** `A[p..(p+k)]`=$a_1 \dots a_k$';' **e ritorno** = $k+1$**)**.

In questa funzione ricorsiva ci sono due casi base: il primo si verifica quando la condizione `p==limite` *diventa vera e corrisponde al completo riempimento di* `A` *(senza aver letto ';'). Mentre il secondo caso base corrisponde a leggere ';'. Nel primo caso base viene restituito 0. Questa azione soddisfa* **POST***?* `k` $\geq$ `limite-p==0` *e quindi* `A[p..limite-1]` *è vuoto e la condizione è trivialmente soddisfatta, visto che riguarda una porzione vuota, cioè inesistente, di array. Inoltre viene restituito 0 che è uguale a* `limite-p` *come richiesto. Nel secondo caso base,* $k = 0$ *mentre* `limite-p > 0` *e quindi deve essere verificata la seconda parte di* **POST** *ed infatti così è, visto che* `A[p..p+k]=A[p]=';'` *e che viene restituito 1 che è il numero di caratteri letti, cioè la sentinella ';'. Resta da considerare il caso in cui viene effettuata l'invocazione ricorsiva, cioè il caso in cui il carattere letto non è la sentinella. In questo caso sappiamo che* `p < limite` *quindi* `p+1` $\leq$ `limite`*, e viene letto un carattere da* `cin` *che è diverso da ';' e quindi l'invocazione ricorsiva viene fatta in uno stato che soddisfa* **PRE** *(la sentinella è ancora in* `cin`*) e possiamo quindi assumere che al ritorno valga* **POST** *e quindi che in* `A[p+1....]` *siano stati inseriti i caratteri giusti e che il loro numero sia il valore ritornato dall'invocazione ricorsiva. Visto che prima dell'invocazione abbiamo eseguito* `A[p]=x`*, siamo anche certi di aver messo in* `A[p....]` *i caratteri giusti e che il numero di questi caratteri sia 1 in più rispetto al numero di quelli inseriti dall'invocazione ricorsiva. Ecco spiegato il* `return 1+ins(A,p+1,limite);`.

In questa dimostrazione facciamo uso del **principio di induzione**. *Riassumiamo i passi effettuati. Per prima cosa è stato dimostrato che se vale* **PRE** *allora vale* **POST** *nei due casi base e successivamente abbiamo dimostrato che se dopo il ritorno dell'invocazione ricorsiva contenuto nella funzione* `ins`*, vale* **POST***, allora anche alla fine della funzione* `ins` *vale* **POST**. *Il seguente argomento dovrebbe aiutare a capire che questi passi dimostrano che qualsiasi invocazione della funzione* `ins` *venga fatta a partire da una situazione del calcolo che soddisfa* **PRE***, al ritorno da questa invocazione sarà stato raggiunto uno stato del calcolo che soddisfa* **POST**. *Se l'invocazione soddisfa subito uno dei 2 casi base, allora abbiamo dimostrato che al ritorno vale* **POST**. *Se invece viene eseguita una invocazione ricorsiva che soddisfa uno dei 2 casi base, allora dalla prova precedente abbiamo che vale* **POST**. *Quindi questo caso è corretto. Ma allora è anche corretto il caso di 2 invocazioni ricorsive seguite da un caso base e poi anche quello con 3 invocazioni ricorsive seguite da un caso base e così via per ogni numero di invocazioni ricorsive.*

Esercizio 8.4 *Scrivere una funzione ricorsiva* `int trova_pos(char * A, int limite, char ch)` *che, dato l'array* `A` *di* `limite` *caratteri, cerca* `ch` *in* `A` *e restituisce la prima posizione di* `A` *(partendo da 0) che contiene il valore* `ch` *e se non esiste una tale posizione, allora la funzione restituisce -1. Naturalmente si deve aver cura di non accedere ad elementi di* `A` *che non esistono. Questo errore viene chiamato* uscire dal limite dell'array.

8.3 Gestione dinamica dei dati

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 stack descritta nella Sezione 2.3. Ci sono casi in cui è utile avere memoria per contenere valori la cui allocazione e deallocazione sfugga alle regole delle variabili automatiche. In questo caso parliamo di gestione dinamica dei dati e anche della memoria. Il C++ fornisce operazioni per realizzarla. L'operazione di allocazione della memoria è la funzione `new` che viene invocata con `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 finché 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 lo stack 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 cui 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.

```
struct S{int x,y; char a,b;};
S* p=new S[10];
if(p==0) throw(..errore..) // errore d'allocazione, gestiscilo
p[0].x=10;
p[2].a='h';
```

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 `[i]` per ottenere l'i-esimo elemento di tipo `S` dell'array. Infine l'operatore di selezione `.` ci permette di accedere ai campi di questo elemento.

È estremamente importante capire che con la `new` si alloca spazio sullo heap che resta a disposizione del programma finchè il programma stesso non lo dealloca, ma l'accesso a quest'area di memoria avviene sempre attraverso un puntatore `p` che è una variabile automatica e che quindi è attiva solo nel blocco in cui è dichiarata. È giusto chiedersi come sia possibile accedere alla memoria puntata da `p` fuori dal blocco di `p`. La risposta è semplice: basta assegnare l'R-valore di `p` a un'altra variabile la cui visibilità si estende oltre quella di `p`. Per esempio, se `p` ha come visibilità il corpo di una funzione, allora è sufficiente che la funzione restituisca

il valore di `p` al suo chiamante. Il seguente frammento di programma estende l'esempio precedente in modo da mostrare il fenomeno appena descritto.

```
struct S{int x,y; char a,b;};
S * F()
{
S* p=new S[10];
if(p==0) throw(..errore..); // errore d'allocazione, gestiscilo
p[0].x=10;
p[2].a='h';
return p;
}
```

Quando lo spazio allocato sullo heap non serve più, esso può venire deallocato con la funzione di sistema `delete`. Si consideri il puntatore `p` dell'esempio precedente, allora, `delete [] p;` dealloca l'array. Le parentesi `[ ]` servono ad indicare che si vuole deallocare un array e questo vale per ogni array, indipendentemenete dalle sue dimensioni. Per esempio se abbiamo la richiesta d'allocazione seguente:

```
int P [][5][10][15]=new int[20][5][10][15];
```

la deallocazione di questo array si richiede semplicemente con: `delete [] P;`. Ovviamente si deve deallocare un array di 20 elementi di tipo `int[5][10][20]`. Non c'è niente di concettualmente diverso rispetto a deallocare un array di 20 interi. Quando si vuole invece deallocare lo spazio per un singolo valore allora si invoca `delete` semplicemente. Vediamo un esempio:

```
int * p=new int(10);
// ....usa *p....
delete p;  // dealloca *p
```

Nell'allocazione precedente viene anche inizializzato con 10 lo spazio per un intero che viene allocato. La possibilità di allocare memoria dinamicamente permette di costruire programmi che sono in grado di adattarsi alle circostanze. Per esempio se una struttura dati si rivela durante l'esecuzione, troppo piccola (o troppo grande) rispetto alle reali necessità, il programma può allocarne una più grande (più piccola), ricopiare in essa il contenuto della struttura precedente e deallocare quest'ultima. Vediamo un esempio semplice.

```
char  A[][10]=new char[5][10];
//........si usa A che si rivela troppo piccola
char B[][10]=new char[10][10]; // alloco matrice con + righe
if(B==0) throw(...errore..);
for(int i=0;i<5;i++) // ricopiamo A in B
```

```
  for(int j=0; j<10; j++)
     B[i][j]=A[i][j];
delete[] A; // dealloco la vecchia matrice
A=B; // adesso la nuova matrice si chiama A
//.....posso andare avanti come se niente fosse successo
```

8.4 Liste concatenate

La ricorsione è una tecnica di programmazione particolarmente adatta a risolvere problemi che concernono dati che sono organizzati in strutture dati ricorsive. La prima di queste strutture dati ricorsive è la **lista concatenata**. Una lista concatenata è:

- **Caso base:** una lista vuota;
- **Caso ricorsivo:** un nodo concatenato al resto della lista concatenata (che è una lista concatenata).

Questa definizione segue la stessa struttura delle funzioni ricorsive: abbiamo il caso base e il caso ricorsivo. Questo spiega perchè la lista concatenata è una struttura ricorsiva. Le funzioni ricorsive che si applicheranno alle liste concatenate, seguiranno questo stesso pattern: il caso base delle funzioni coinciderà (in generale) con l'invocazione su una lista vuota, mentre il caso ricorsivo si occuperà delle liste non vuote.

Usando la gestione dinamica dei dati, vista nella Sezione 8.3, costruiremo liste concatenate che possono crescere e ridursi a seconda della necessità. Innanzitutto spieghiamo come realizzare le liste concatenate in C++. La seguente struttura realizza un **nodo** di una lista concatenata: `struct nodo{char info; nodo * next;}; .` Il campo `info` è l'informazione contenuta nel nodo. Chiaramente possiamo definire nodi con informazioni di qualsiasi tipo (anche altre strutture). Per i nostri esempi consideriamo nodi con informazione di tipo carattere. La cosa più interessante della struttura `nodo` è il campo `next`. Esso ha tipo `nodo *` e serve a puntare il prossimo nodo della lista. L'ultimo nodo di una lista concatenata deve avere sempre campo `next` con R-valore uguale a 0 (o equivalentemente, uguale a `NULL`) in modo da segnalare appunto la fine della lista. Dotiamo la struttura `nodo` di un costruttore che ci permetterà di scrivere programmi più concisi. Il costruttore è: `nodo::nodo(char a=0, nodo* b=0)info=a; next=b;`. I valori di default per tutti e 2 i parametri, permettono di usare il costruttore con 0, 1 e 2 parametri attuali.

Le liste sono strutture dati dinamiche in quanto possiamo allocare nuovi nodi con la `new` ed aggiungerli alla nostra lista e possiamo anche togliere nodi dalla nostra lista e deallocarli con `delete`. Vediamo il seguente semplice esempio in cui si costruisce una lista di 3 nodi che contengono rispettivamente nel campo `info` i caratteri `'a', 'g'`, e `'y'` e poi eliminiamo il nodo centrale della lista che resta quindi composta solo dal primo e dal terzo nodo.

```
nodo *terzo=new nodo('y',0), *secondo=new nodo('g',terzo),
  * primo=new nodo('a',secondo);
secondo=terzo=0; // tengo solo il puntatore al primo nodo
nodo * t= primo->next; // mi ricordo il puntatore al secondo nodo
primo->next=t->next;// collego il primo al terzo
delete t; // dealloco il secondo nodo
```

Il caso base della definizione di una lista concatenata è la lista vuota, cioè senza nodi. Una tale lista è rappresentata semplicemnete da una variabile `nodo *` che abbia valore 0 e che quindi non punti a nessun nodo. Nel seguito consideriamo alcune operazioni sulle liste concatenate e mostriamo vari modi di realizzarle attraverso funzioni ricorsive e iterative. Negli esercizi che seguono assumiamo il tipo `struct nodo{int info; nodo* next; };` con costruttore `nodo::nodo(int a=0, nodo* b=0){info=a; next=b;}`.

Esercizio Risolto 8.5 *Vogliamo costruire una lista concatenata che contenga* `k` *nodi ognuno con campo informativo uguale a 0. La soluzione che proponiamo è ricorsiva su* `k`*, con caso base* `k=0` *in cui lalista da costruire è vuota e quindi la funzione deve semplicemente restituire 0 che rappresenta la lista vuota. Nel caso ricorsivo invece dovremo costruire un nodo e poi concatenarlo con il resto della lista prodotta dall'invocazione ricorsiva.*

- **PRE** =**(**`k`$\geq$ **0)***;*
- **POST** =**(restituisce una lista di** `k` **nodi con campo** `info` **uguale a 0)**.

```
nodo * build(int k)
{
 if(k==0)
    return 0;
 return new nodo(0,build(k-1));
}
```

Il caso base è semplice: se `k==0` *allora basta restituire 0 che rappresenta la lista vuota. Nel caso ricorsivo, bisogna capire che significa* `return new nodo(0,build(k-1))`*. Il risultato dell'invocazione ricorsiva viene usato come secondo parametro nell'invocazione del costruttore di* `nodo`*. Visto che in questo caso vale* `k >0` *e quindi* `k-1` $\geq$ `0`*, allora* **PRE** *vale quando l'invocazione ricorsiva viene eseguita, quindi possiamo assumere che valga* **POST** *quando essa termina (stiamo usando il principio di induzione introdotto nell'Esercizio 8.3) e cioè che il valore che essa restituisce è un puntatore ad una lista di* `k-1` *nodi (con campo* `info` *a 0). Il costruttore di* `nodo` *viene invocato in* `new nodo(0,build(k-1))` *in modo tale che il puntatore restituito da* `build(k-1)` *diventi il valore del campo* `next` *di questo nodo. Quindi la* `new` *restituisce il puntatore ad un nodo (con campo* `info`

uguale a 0) che punta ad una lista di `k-1` *nodi (con campo* `info` *uguale a 0) e quindi la funzione restituisce un puntatore ad una lista di* `k` *nodi (con campo* `info` *uguale a 0). Da questo segue che, alla fine dell'invocazione della funzione* `build(k)`, *vale* **POST**.

Esercizio 8.6 *1. Modificare la funzione dell'Esercizio 8.5 in modo che i* `k` *nodi che vengono costruiti, anzichè avere campi* `info` *uguali a 0, contengano valori* $1, \ldots, k$ *con 1 nel primo nodo e* k *nell'ultimo. Si possono aggiungere parametri alla funzione nell'Esercizio 8.5.*

2. Scrivere una funzione ricorsiva che riceve il puntatore al una lista concatenata e restituisce il numero dei nodi della lista stessa. Questa quantità si chiama la **lunghezza** *della lista. Scrivere anche* **PRE** *e* **POST** *della funzione.*

3. Scrivere una funzione iterativa che calcoli la lunghezza di una lista concatenata.

4. Scrivere una funzione ricorsiva che dato il puntatore ad una lista ed un intero `x`*, restituisce il numero di nodi della lista il cui campo* `info` *sia uguale a* `x`.

5. Scrivere una funzione che, dato il puntatore ad una lista concatenata ed un intero `k`*, dealloca i primi nodi della lista fino a deallocarne* `k` *oppure fino a che i nodi finiscano. La funzione deve restituire il puntatore alla lista che resta di quella originale dopo le deallocazioni. Scrivere* **PRE** *e* **POST** *della funzione.*

Esercizio Risolto 8.7 *Consideriamo ora il problema di inserire un nuovo nodo in una lista concatenata. Prima consideriamo di inserirlo al primo posto, successivamente lo inseriremo in fondo alla lista. Visto che l'inizio della lista è immediatamente raggiungibile, l'operazione di inserimento all'inizio non necessita di ricorsione (né di iterazione):*

```
nodo * ins_inizio(nodo* n, int x)
{
   return new nodo(x,n);
}
```

Si osservi che questa operazione cambia sempre il puntatore alla lista. Il nuovo puntatore è restituito col `return`. *Lo stesso effetto può essere ottenuto col passaggio per riferimento:*

```
void ins_inizio_rif(nodo* &n, int x)
{
   n=new nodo(x,n);
}
```

Consideriamo ora di voler aggiungere un nodo con campo `info` *uguale a* `x` *in fondo alla lista. Dobbiamo quindi scorrere la lista fino ad arrivare in fondo, cioè all'ultimo nodo della lista, e dobbiamo cambiare il suo campo* `next` *(che ha valore 0) in modo che punti al nuovo nodo (con campo* `info` *uguale a* `x`*).*

```
void ins_fine(nodo* n, int x)
{
   if(n->next==0)
      n->next= new nodo(x,n);
   else
      ins_fine(n->next,x);
}
```

Ogni invocazione di questa funzione deve essere tale che la lista puntata dal parametro attuale `n` *deve avere lunghezza almeno 1. D'altra parte, la funzione è pensata in modo da fermarsi all'ultimo nodo della lista e quindi è ovvio che la lista debba contenere almeno un nodo. Si osservi che questa precondizione va garantita assolutamente visto che, qualora la funzione venisse invocata con* `n==0`*, il test* `n->next == 0` *causerebbe una violazione di accesso della memoria (detto segmentation fault). Quindi la funzione* `ins_fine` *può venire usata solo con liste non vuote e il caso della lista vuota andrebbe controllato ed escluso prima dell'invocazione. Ovviamente questo è possibile:* `if(n==0) n=new(x,0); else ins_fine(n,x);`*, ma sarebbe più semplice non doversene preoccupare. È possibile scrivere una funzione che inserisca un nodo in fondo ad una lista qualsiasi (vuota o no)? La risposta è si e nel seguito vedremo 2 modi diversi di farlo.*

```
nodo* ins_fine_val(nodo* n, int x)
{
  if(!n)
    return new(x,0);
  n->next=ins_fine_val(n->next,x);
  return n;
}
```

Si osservi il test `!n` *che è equivalente, ma più conciso, di* `n==0`*. La precondizione di questa funzione è* **PRE** =**(la lista puntata da** `n` **ha lunghezza maggiore o uguale a 0)**. *Quindi non abbiamo più la limitazione di* `ins_fine`. **POST** =**(restituisce la lista iniziale con un nuovo nodo con campo** `info== x` **aggiunto alla fine)**. *Il caso base è la lista vuota e, in questo caso, la funzione restituisce il puntatore al nuovo nodo. Ovviamente questo garantisce* **POST**. *Nel caso ricorsivo,* **PRE** *vale prima dell'invocazione ricorsiva e quindi vale* **POST** *dopo l'invocazione, da cui abbiamo che la funzione soddisfa* **POST** *alla fine. Infatti in questo caso, la lista a cui aggiungere un nodo inizia con* `n` *e la risposta desiderata è proprio* `n` *che punta alla lista prodotta dall'invocazione ricorsiva. Quindi la funzione è corretta per ogni lista. Però* `ins_fine_val` *ha un difetto: compie delle operazioni inutili. Si consideri, infatti, un qualsiasi nodo che non sia l'ultimo della lista*

originale. Allora, dopo la fine della funzione, il valore del campo `next` *di questo nodo non sarà cambiato rispetto al suo valore prima dell'esecuzione della funzione anche se per esso la funzione ha eseguito:* `n→ next=ins_fine_val(n→ next,x);` *che risulta quindi inutile. L'unico campo* `next` *che viene modificato è quello dell'ultimo nodo della lista originale che dopo l'esecuzione della funzione punta al nuovo nodo. Il problema è che la funzione non può distinguere l'ultimo nodo da quelli intermedi, altrimenti ritorneremmo al problema visto con* `ins_fine`*, e quindi non c'è altro che eseguire* `n→ next=ins_fine_val(n→ next,x);` *per ogni nodo della lista originale.*

Riassumendo, il problema è che, dovendo accettare anche la lista vuota il suo caso base deve essere appunto `n==0`*. Quando questa condizione è verificata, allora dobbiamo creare il nuovo nodo e dovremmo inserire il puntatore a questo nodo nel campo* `next` *dell'ultimo nodo della lista originale. In realtà esattamente questo campo* `next` *è il parametro attuale dell'invocazione finale della funzione. Visto che il parametro è passato per valore, viene passato solo l'R-valore 0 di questo campo* `next`*, ma se passassimo il parametro per riferimento, allora l'invocazione finale avrebbe l'R-valore del campo* `next` *per fare il test* `n==0`*, ma anche il suo L-valore per assegnargli il nuovo nodo. La funzione che usa quest'idea è la seguente:*

```
void ins_fine_rif(nodo*& n, int x)
{
  if(!n)
    n=new nodo(x,0);
  else
    ins_fine_rif(n->next,x);
}
```

Questa funzione segue la stessa idea usata per `ins_fine`*, ma sfrutta il passaggio per riferimento per fare tutte le operazioni nel caso base, i.e.,* `n=new nodo(x,0);` *e quindi non c'è più niente da fare nel caso ricorsivo: basta fare l'invocazione ricorsiva per percorrere la lista fino ad arrivare al caso base e al ritorno della ricorsine non c'è più nulla da fare (visto che tutto è fatto nel caso base). Si osservi anche che, se la lista iniziale è vuota e quindi se il puntatore alla lista è, per esempio,* `I`*, allora* `I==0` *e la prima invocazione ricorsiva è* `ins_fine_rif(I,x);`*, allora essa raggiunge subito il caso base eseguendo* `n=new nodo(x,0);` *e visto che* `n` *è alias di* `I` *(a causa del passaggio del parametro per riferimento), quest'ultima punterà al nuovo nodo al ritorno dall'invocazione.*

Esercizio 8.8 *Nei seguenti esercizi consideriamo varie operazioni sulle liste.*

1. *Scrivere una funzione ricorsiva* `nodo * append(nodo* p, nodo* Q)` *che riceve (per valore) due liste concatenate (possibilmente vuote)* `P` *e* `Q` *e*

restituisce col `return` *una lista che è ottenuta appendendo* `Q` *alla fine di* `P`. *Nessun nuovo nodo deve essere creato e nessuno deve essere distrutto. Semplicemente le due liste date vanno unite per farne una sola.*

2. *Scrivere una funzione ricorsiva che esegua la stessa operazione del punto precedente, ma con prototipo* `void append(nodo* & P, nodo* Q)` *e con l'idea che restituisca la nuova lista attraverso il parametro* `nodo* & P`, *passato per riferimento.*

3. *Scrivere una funzione ricorsiva* `nodo* alt_mix(nodo* P, nodo* Q)`, *che riceva per valore due liste (possibilmente vuote)* `P` *e* `Q` *e restituisca col* `return` *una lista composta da un nodo di* `P`, *seguito da un nodo di* `Q`, *poi di nuovo un nodo di* `P` *e così via. Se una delle 2 liste finisce prima dell'altra, i nodi di quella rimasta verrano appesi alla fine della lista da costruire.*

4. *Scrivere una funzione ricorsiva che compia l'operazione descritta nel punto precedente, ma restituendo il puntatore alla nuova lista attraverso il parametro* `P` *che deve essere passato per riferimento.*

5. *Scrivere una funzione ricorsiva che, data per riferimento una lista* `P` *ed un intero* `k`, *restituisca 2 liste: quella composta dai primi* `k` *nodi di* `P` *e poi quella composta dai restanti nodi di* `P` *(dal* `(k+1)`*-esimo in poi). Se* `P` *contiene un numero di nodi non maggiore di* `k`, *allora la prima lista conterrà tutti i nodi di* `P` *e la seconda sarà vuota. La seconda lista va ritornata col* `return`, *mentre la prima va restituita attraverso il parametro* `P` *che va passato per riferimento.*

Esercizio 8.9 *I seguenti esercizi considerano varie operazioni in cui un nodo viene eliminato da una lista. Per tutti gli esercizi si può assumere la precondizione che la lista* `Q` *non sia vuota.*

1. *Scrivere una funzione* `nodo* pop(nodo* Q)` *che elimini il primo nodo della lista data* `Q` *e restituisca col* `return` *quello che resta della lista. Il nodo eliminato va deallocato con* `delete`.

2. *Scrivere una funzione* `int pop(nodo*& Q)` *che elimina il primo nodo di* `Q`, *restituisca la nuova lista attraverso il parametro passato per riferimento, e restitisca anche il campo* `info` *del nodo eliminato.*

3. *Scrivere una funzione ricorsiva* `nodo * del_end(nodo* Q)` *che elimini l'ultimo nodo della lista* `Q` *e restituisca col* `return` *la lista che resta.*

4. *Scrivere una funzione ricorsiva* `int del_end_rif(nodo * & Q)` *che elimini l'ultimo nodo della lista e ritorni la lista rimanente attraverso il parametro e il campo informativo del nodo eliminato col* `return`.

Esercizio 8.10 *Consideriamo ora operazioni su liste ordinate, cioè tali che il campo informativo dei suoi nodi non decresca percorrendo la lista dall'inizio alla fine.*

1. *Scrivere una funzione ricorsiva* `nodo* ins_ord(nodo* Q, int x)` *che inserisca un nuovo nodo con campo* `info` *uguale a* `x` *nella lista ordinata* `Q` *in modo da produrre una nuova lista ordinata che andrà restituita con il* `return`.

2. *Scrivere una funzione* `void ins_ord_rif(nodo *&Q, int x)` *simile alla precedente, ma restituisca la nuova lista attraverso il parametro passato per riferimento.*

3. *Scrivere una funzione* `nodo * del_ord(nodo * Q, int x)` *che elimini dalla lista ordinata* `Q` *tutti i nodi (ce potrebbero essere 0, 1, e anche più di 1) che contengono campo informativo uguale a* `x`. *la nuova lista va restituita col* `return`.

4. *Scrivere una funzione ricorsiva* `void del_ord(nodo * & Q, int x)` *che elimini tutti i nodi con campo informativo uguale a* `x` *dalla lista ordinata* `Q`, *restituisca la nuova lista attraverso il parametro passato per riferimento.*

5. *scrivere una funzione iterativa che esegua l'operazione di eliminazione da una lista ordinata di tutti i nodi con un dato campo informativo. La funzione deve restituire al chiamante la nuova lista col* `return`.

8.5 Alberi Binari

La definizione di albero binario è ricorsiva. Un albero binario è:

- **Caso base:** l'albero vuoto, senza nodi;
- **Caso ricorsivo:** un nodo che possiede due (sotto)alberi, quello sinistro e quello destro.

Per descrivere le relazioni che intercorrono tra i nodi vicini in un albero si usano le relazioni familiari (come fosse un albero genealogico). Quindi parliamo di **radice** di un albero, dei **figli** di un nodo, di **figlio destro** e **figlio sinistro**, e di **discendenti** di un nodo. I nodi che hanno almeno un figlio sono detti **interni**, mentre **foglie** sono i nodi senza figli. Dato un qualsiasi albero `T` ed un nodo `n` di `T`, l'albero **radicato in** `n` è il `sottoalbero` di `T` che contiene i successori di `n` compreso `n` stesso. Un **cammino** in un albero è una sequenza di archi che collegano un nodo a qualche suo discendente. La lunghezza di un cammino è il numero di archi che lo compongono. Il cammino si dice `vuoto` se collega un nodo con se stesso ed è quindi composto da zero archi. L'**altezza** di un albero è la lunghezza massima tra i cammini presenti nell'albero. La **profondità** di un nodo

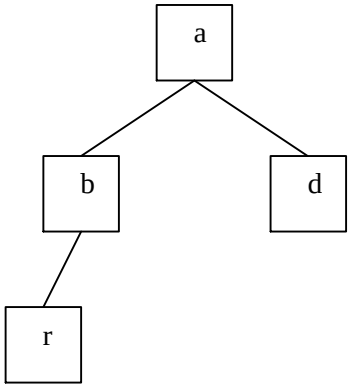

Figura 8.1: L'albero binario con 4 nodi costruito da BUILD_TREE

è la lunghezza del cammino che lo collega alla radice dell'albero e la sua **altezza** è l'altezza dell'albero radicato in quel nodo.

In C++ un nodo di un albero binario viene modellato da una struttura con un campo `info` di tipo qualsiasi (come per le liste concatenate) e due campi di tipo `nodo *`, il campo `left` che punta al sotto-albero sinistro e `right` che punta a quello destro:

```
struct nodo {char info; nodo* left, *right;};
```

Dotiamo la struttura `nodo` del seguente costruttore: `nodo::nodo(char a=0, nodo* b=0, nodo*c=0){info=a; left=b; righr=c;}`. Si noti che il campo `info` dei nodi è di tipo `char`, mentre per i nodi delle liste è `int`. Questa differenza ha il solo scopo di sottolineare che il tipo del campo informativo dei nodi può variare a piacimento. Un albero binario vuoto è semplicemente una variabile, per esempio `nodo* r`, che abbia valore 0. Nel seguito studieremo vari esercizi che concernono gli alberi binari. Iniziamo con alcune istruzioni che costruiscono un albero binario con 4 nodi che è raffigurato nella Figura 8.1.

```
{ // il frammento di programma BUILD_TREE
 nodo *quarto=new nodo('r',0,0);
 *terzo=new nodo('d',0,0);
 *secondo=new nodo('b',quarto,0);
 *primo=new nodo('a',secondo,terzo);
}
```

Nella Figura 8.1 ogni nodo è rappresentato con un quadrato che contiene il carattere del suo campo informativo. I puntatori diversi da 0 sono rappresentati con segmenti che congiungono il nodo col figlio sinistro o destro in questione. I puntatori a 0 come quelli delle foglie ed il puntatore `right` del nodo che contiene `b`, non corrispondono a nulla nella rappresentazione grafica dell'albero. La variabile `nodo * primo` di `BUILD_TREE` punta alla radice dell'albero di Figura

8.1. Inoltre, `terzo==primo->right` e `quarto==primo->left->left`. L'albero di Figura 8.1 può essere rappresentato in maniera lineare con la seguente espressione: `a(b(r(0,0),0),d(0,0))`, in cui `0` indica un puntatore uguale a 0 (equivalente a un albero vuoto).

Vediamo ora una funzione ricorsiva che, dato il puntatore alla radice di un albero binario, lo distrugge completamente deallocando tutti i suoi nodi. La ricorsione rende il compito particolarmente semplice. La cosa principale da capire è che, prima di deallocare un nodo, si devono deallocare i suoi sotto-alberi. Questo è realizzato nella seguente funzione posizionando `delete x` dopo le invocazioni ricorsive sui figli di `x`. Se avessimo inserito `delete x` prima delle invocazioni ricorsive, allora avremmo commesso un errore eseguendo `x->left` dato che questa espressione richiederebbe la dereferenziazione di un puntatore deallocato e che quindi punta ad un'area di memoria (che risiede sullo heap) che non è più *proprietà* del programma. Ovviamente lo stesso varrebbe per `x->right`.

```
void dealloca_albero(nodo * x)
{
if(x) //l'albero non e' vuoto
  {
  dealloca_albero(x->left);
  dealloca_albero(x->right);
  delete x;
  }
}
```

Gli alberi binari sono strutture dati utili per molti scopi. Nel seguito svilupperemo alcune funzioni ricorsive e iterative che eseguono operazioni interessanti sugli alberi binari.

Esercizio Risolto 8.11 *La prima funzione calcola il numero di nodi di un albero radicato nel nodo puntato da* `n`.

```
int conta_nodi(nodo * n)
{
 if(n)
  return 1+conta_nodi(n->next)+conta_nodi(n->right);
 else
  return 0;
}
```

Dimostrare la correttezza di `conta_nodi` *è molto facile.* **PRE** =**(**`n` **punta ad un albero binario, possibilmente vuoto)**. **POST** =**(la funzione restituisce il numero dei nodi nell'albero puntato da** `n`**)**.

Come sempre per dimostrare la correttezza di una funzione ricorsiva usiamo il principio di induzione introdotto nell'Esercizio 8.3. Il caso base dell'induzione è l'albero vuoto, cioè il caso `!n` *e in questo caso la funzione* `conta_nodi` *restituisce 0 che è, ovviamente, il numero di nodi di un albero vuoto. Per il passo induttivo* `n` *è diverso da 0 e quindi, dalla* **PRE***, punta ad un albero non vuoto e quindi* `n->left` *e* `n->right` *puntano ad alberi (eventualmente vuoti, ma sempre alberi). Quindi per entrambe le invocazioni ricorsive, vale la postcondizione* **POST** *e cioè esse restituiscono il numero di nodi del sotto-albero sinistro e destro del nodo* `n` *e quindi, sommando 1 (per contare* `n`*) alla somma dei valori restituiti dalle 2 invocazioni, si ottiene il numero di nodi dell'albero puntato da* `n`*. Questo dimostra che il* `return` *restituisce il valore richiesto e quindi la funzione soddisfa la* **POST** *anche nel caso induttivo.*

Esercizio Risolto 8.12 *Vogliamo ora realizzare l'operazione di ricerca di un nodo di un albero che contenga un campo informativo di un valore dato. Potrebbero esserci più nodi nell'albero che soddisfano la condizione, ma si chiede di trovarne uno qualsiasi e di restituire un puntatore a questo nodo. Il caso base è naturalmente che la ricerca venga fatta su un albero vuoto e che quindi ovviamente fallisca. Visto che la funzione deve restituire un valore* `nodo*`*, il fallimento corrisponde in modo naturale a restituire il valore 0 (che contraddistingue un puntatore indefinito, cf. Sezione 5.1). Quando la ricerca si applica ad un albero non vuoto dobbiamo ragionare nel modo seguente: l'albero ha la radice, quindi dobbiamo controllare innanzitutto se questa radice soddisfa la proprietà desiderata. Se è così allora la ricorsione termina con la restituzione del puntatore al nodo radice. Se non è così, allora dovremo cercare il nodo nei sotto-alberi della radice, non importa in quale ordine. Per esempio possiamo cercare il nodo prima nel sotto-albero sinistro. Se questa invocazione ricorsiva ha successo, allora inutile cercare nel sotto-albero destro. Altrimenti, dovremo cercare anche nel sotto-albero destro e a questo punto, qualsiasi sia il risultato di quest'ultima ricerca, esso sarà il risultato finale della funzione. Vediamo di concretizzare questi ragionamenti in una funzione.*

```
nodo * ricerca(nodo* n, char x)
{// caso base:
 if(!n)
   return 0;
 //prova n
 if(n->info==x)
   return n;
 nodo* z=ricerca(n->left,x);
 if(z)
   return z;
 else
   return ricerca(n->right,x);
}
```

La precondizione della funzione è semplice: **PRE** =**(**`n` **punta ad un albero, possibilmente vuoto)**. *La postcondizione è,* **POST** =**(se la funzione restituisce un valore diverso da 0, allora il valore punta ad un nodo** `N` **dell'albero radicato in** `n`**, tale che** `N->info==x`**, altrimenti nessun nodo esiste nell'albero radicato in** `n` **che soddisfa questa condizione)**. *La* **POST** *è lunga, ma semplice da capire. Come sempre seguiamo il principio di induzione. Il caso base è ovvio. Nel caso induttivo* `n` *punta ad un nodo e quindi è vero che il test* `n->info==x` *non da errore ed è anche vero che se restituisce* `true`*, allora la funzione soddifa* **POST**. *Se invece* `n` *non è il nodo cercato, il fatto che* `n` *sia comunque un nodo, implica che le due invocazioni ricorsive sui sotto-alberi soddisfano* **PRE**. *Quindi* **POST** *vale per* `nodo* z=ricerca(n->left,x);` *quindi se* `z` *non è 0, il nodo puntato da* `z` *è quello che cerchiamo e quindi restituirlo soddisfa* **POST**. *Se invece* `z==0` *allora viene fatta la seconda invocazone ricorsiva e visto che sappiamo a questo punto che* `n` *ed il suo sotto-albero sinistro non contengono nodi con campo informativo uguale a* `x`*, allora, la risposta di* `ricerca(n->right,x);`*, che, per ipotesi induttiva, è corretta per il sotto-albero desto di* `n`*, è anche corretta per l'intero albero radicato in* `n`. *Quindi alla fine vale* **POST**.

Esercizio Risolto 8.13 *Consideriamo ora un'operazione nuova sugli alberi binari. Quella di aggiungere un nodo nuovo ad un albero. Va chiarito subito che un nuovo nodo va inserito sulla frontiera dell'albero, cioè esso sarà sempre inserito come foglia dell'albero. L'albero iniziale potrebe essere vuoto. Quindi successive invocazioni di questa funzione possono costruire alberi binari partendo da un albero vuoto. Per fare in modo che l'albero ottenuto in questo modo sia bilanciato, facciamo in modo che il nodo venga aggiunto sempre nella parte dell'albero in cui ci sono meno nodi. Per ottenere questo effetto, la nostra funzione quando si trova nel nodo* `n` *e deve decidere se inserire il nuovo nodo nel sotto-albero sinistro oppure in quello destro di* `n`*, conta prima i nodi di questi 2 sotto-alberi e inserisce il nuovo nodo in quello dei 2 che ha meno nodi. Se i 2 sotto-alberi hanno lo stesso numero di nodi, inserisce il nuovo nodo nel sotto-albero sinistro.*

```
void inserisci(nodo * & n, char x)
{
 if(!n)
  n=new(x,0,0);
 else
  {
   if(conta_nodi(n->left)<=conta_nodi(n->right))
     inserisci(n->left,x);
   else
     inserisci(n->right,x);
  }
}
```

La precondizione della funzione è uguale a quelle già viste negli esercizi precedenti: **PRE** =**(**`n` **punta ad un albero, possibilmente vuoto)**. *La postcondizione è più difficile da specificare perchè la funzione esegue una sequenza di scelte non semplice da descrivere. Consideriamo un albero binario* `T`*, diremo che l'albero binario* `T'` *è quello* **che vogliamo produrre da** `T` *se valgono le seguenti condizioni:*

(a) se `T` *è l'albero vuoto, allora* `T` *consiste di un solo nodo con campo informativo uguale a* `x`*;*

(b) se `T` *non è vuoto, allora si può ottenere* `T'` *da* `T` *aggiungendogli una nuova foglia* `f` *con campo informativo* `x` *tale che questa foglia soddisfi la seguente condizione: per ogni nodo* `m` *sul cammino tra* `f` *e la radice di* `T'`*, se* `f` *si trova nel sotto-albero destro di* `m`*, allora il numero di nodi di questo sotto-albero è minore o al massimo uguale di quello del sotto-albero sinistro di* `m`*. Altrimenti* `f` *è nel sotto-albero sinistro di* `m`*.*

Ora possiamo scrivere la postcondizione di `inserisci`*.* **POST** =**(alla fine della funzione l'albero puntato da** `n` **è quello che vogliamo produrre da quello puntato da** `n` **all'inizio della funzione)**. *È facile vedere che nel caso base, cioè quando vale* `!n`*, vale* **POST***:* `n` *punta al nuovo nodo con campo informativo* `x`*. Inoltre, va osservato che, visto che* `n` *è passato per riferimento, il chiamante riceve un puntatore al nuovo albero.*

Nel caso induttivo, possiamo contare sulla correttezza della funzione `conta_nodi`*, che abbiamo dimostrato nell'Esercizio 8.11. Dalla* **PRE** *di* `inserisci` *e dal fatto che* `n` *punta ad un nodo, segue che* `n->left` *e* `n->right` *puntano ad alberi e quindi la* **PRE** *di* `conta_nodi` *è verificata prima delle 2 invocazioni e quindi queste restituiscono gli interi corretti. Da questo segue che l'invocazione ricorsiva a* `inserisci` *verrà sempre fatta al sotto-albero di* `n` *con numero di nodi non maggiore dell'altro sotto-albero e in caso di parità tra i due sotto-alberi, l'invocazione ricorsiva verrà fatta sul sotto-albero sinistro. Di nuovo la* **PRE** *di questa invocazione ricorsiva è verificato e quindi, per ipotesi induttiva, anche la sua* **POST** *è verificata. Quindi l'invocazione inserisce nel campo* `n->left` *o* `n->right` *il puntatore ad un albero che è quello che vogliamo produrre da quello all'inizio dell'invocazione. Infine, da quanto asserito su come la funzione decide se fare l'invocazione ricorsiva su* `n->left` *o su* `n->right`*, segue che anche l'albero puntato da* `n` *dopo il ritorno dell'invocazione ricorsiva, è quello che vogliamo produrre da quello puntato da* `n` *all'inizio della funzione. Quindi vale la* **POST** *alla fine della funzione* `inserisci`*.*

È interessante osservare la similarità tra questa funzione e la funzione `ins_fine_rif` *dell'Esercizio 8.7. In entrambe le funzioni è fondamentale il passaggio del nodo corrente* `n` *per riferimento. Anche in questo caso, nel caso base,* `!n`*, il parametro* `n` *è un alias del campo* `left` *o* `right` *del nodo padre di* `n` *e quindi l'assegnazione* `n=new nodo(x,0,0)` *attacca il nuovo nodo al padre. Si osservi che, grazie*

al passaggio per riferimento, la funzione è corretta anche nel caso venga invocata inizialmente su un albero vuoto.

Esercizio 8.14 *Scrivere una funzione ricorsiva che esegua l'operazione già discussa nell'Esercizio precedente, ma che, a differenza della funzione illustrata in quell'Esercizio, abbia il seguente prototipo:* `nodo* inserisci_val(nodo* n, char x)`. *La funzione richiesta dovrebbe seguire la tecnica già vista nella funzione* `ins_fine_val` *dell'Esercizio 8.7.*

Percorrere un albero (binario) significa visitare ogni suo nodo. Fondamentalmente esistono 2 maniere di percorrere un albero: **in larghezza** e **in profondità** (breath-first e depth-first). Il percorso in larghezza visita la radice, poi i suoi figli, poi i figli dei figli della radice e così via. Insomma i nodi sono visitati per livello (o profondità): livello 0, livello 1, livello 2, eccetera. Ovviamente si può scorrere ogni livello da sinistra a destra o da destra a sinistra. Il percorso in profondità significa invece che si scende lungo qualche cammino fin dove possibile e poi si risale e così via. Il percorso in profondità può seguire tre diversi ordini: **l'ordine infisso**, **l'ordine prefisso**, e **l'ordine postfisso**. L'ordine prefisso significa che ogni nodo viene considerato prima che vengano considerati i suoi sotto-alberi e per questi, si visita prima il sinistro e poi il destro. L'ordine infisso significa che viene considerato prima il sotto-albero sinistro di ogni nodo, poi si considera il nodo stesso e per ultimo si considera il suo sotto-albero destro. Infine l'ordine postfisso significa che si visitano il sotto-albero sinistro e destro di ogni nodo, prima del nodo stesso.

Il seguente programma segue il percorso in profondità secondo l'ordine infisso per stampare i campi `info` di tutti i nodi di un albero:

```
void infix_stampa(nodo * n)
{
 if(n)
 {
   innfix_stampa(n->left);
   cout<<n->info<<' ';
   infix_stampa(n->right);
 }
}
```

Esercizio 8.15 *Scrivere funzioni ricorsive che attraversano l'albero puntato da* `n` *in profondità secondo l'ordine pre e postfisso per stampare i campi* `info` *di tutti i suoi nodi.*

Esercizio Risolto 8.16 *Vediamo ora una funzione che calcola l'altezza di un albero. Come spiegato all'inizio di questa Sezione, l'altezza di un albero è la lunghezza massima dei suoi cammini. Si deve fare attenzione al caso base. Potremmo essere tentati di fissare a 0 l'altezza di un albero vuoto, ma questo sarebbe un errore*

infatti 0 è l'altezza di un albero che consiste della sola radice. Quindi dobbiamo attribuire un valore diverso all'altezza dell'albero vuoto. Ovviamente si tratta di un valore convenzionale. L'albero vuoto non ha nodi e quindi la nozione di altezza non si applica a tale albero. Potremmo quindi decidere che l'altezza di un albero vuoto è un qualsiasi valore negativo. È necessario prendere un valore negativo in quanto quelli non negativi possono essere l'altezza di un albero. Scegliamo di fissare a -1 l'altezza di un albero vuoto. Il valore -1 è coerente con la seguente maniera di calcolare l'altezza di un albero:

- *se l'albero è vuoto allora l'altezza è -1*
- *altrimenti l'albero ha una radice* `n` *e la sua altezza è l'altezza maggiore tra quella del sotto-albero sinistro e quella del sotto-albero destro di* `n` *a cui si somma 1.*

Per capire che aver fissato a -1 l'altezza dell'albero vuoto fa funzionare lo schema induttivo appena descritto, si consideri un albero che consiste di un solo nodo `n`*. Allora lo schema ci dice che dobbiamo calcolare l'altezza maggiore tra quella dei 2 sotto-alberi di* `n`*. Essi sono entrambi vuoti e quindi l'altezza maggiore dei due è -1 da cui otteniamo 0 sommando 1 e 0 è proprio l'altezza del nostro albero con un solo nodo. La funzione seguente realizza questo schema induttivo.*

```
int altezza(nodo * n)
{
 if(!n) // albero vuoto
  return -1;
 else
  {
   int h1=0, h2=0
   h1=altezza(n->left)+1; // calcoliamo l'altezza del sotto-albero
                          // sinistro e di quello destro
   h2=altezza(n->right)+1;
   if(h1>h2)
    return h1;
   else
    return h2;
 }
}
```

Dimostrare la correttezza di questa funzione è facile. **PRE** =**(**`n` **punta ad un albero binario, possibilmente vuoto)** *e* **POST** =**(la funzione restituisce l'altezza dell'albero puntato da** `n`**)**.

- **Caso base:** *se* `n` *è* `0`*, allora l'altezza è -1 che è corretto per la convenzione che abbiamo fissato che l'altezza di un albero vuoto è -1;*

- **Caso induttivo:** *vale che* `n` *punti ad un nodo e da questo segue che le 2 invocazioni ricorsive hanno* **PRE** *vera e quindi, per ipotesi indiuttiva, esse restituiscono l'altezza dei sotto-alberi sinistro e destro. Allora vale* **POST** *alla fine della funzione, perchè l'altezza di un albero con radice* `n` *è proprio la maggiore tra le altezze dei due sotto-alberi di* `n` *più 1 e questo è esattamente il valore restituito dalla nostra funzione.*

Esercizio 8.17 *Per entrambi gli esercizi che seguono si deve scrivere pre e post-condizione e dimostrare la correttezza.*

1. *Se per l'altezza di un albero vuoto adottassimo un valore convenzionale diverso da -1, per esempio, se adottassimo -2, come dovremmo riscrivere la funzione* `altezza` *descritta nell'Esercizio 8.16?*

2. *Scrivere una funzione* `void scrivi_alt(nodo * n)` *che inserisca nel campo* `info` *di ogni nodo* `m` *dell'albero radicato in* `n` *(comprendendo anche* `n`*) il valore dell'altezza dell'albero radicato in* `m`*.*

Esercizio Risolto 8.18 *In questo esercizio assumiamo di considerare alberi con nodi il cui campo informativo abbia tipo intero. Vogliamo realizzare ora una funzione che inserisca nel campo* `info` *di ogni nodo* `n` *di un albero binario dato, la lunghezza minima dei cammini tra* `n` *ed una foglia. Come nell'Esercizio 8.16, dobbiamo decidere cosa fare per un albero vuoto. Ovviamente quando non ci sono nodi la funzione non deve fare nulla. Quando invece abbiamo un nodo* `n` *il calcolo del valore da inserire in* `n->info` *sarà il seguente: si prende il minimo valore tra* `n->left->info` *e* `n->right->info` *e gli si somma 1. Naturalmente però si dovrà considerare anche i casi che* `n` *abbia un solo figlio e anche il caso che non abbia alcun figlio. Vediamo la funzione che segue questo schema:*

```
void dist_min_foglia(nodo *n)
{
  if(!n)
    return;
  dist_min_foglia(n->left);
  dist_min_foglia(n->right);
  if(!n->left && !n->right) //  n e' una foglia
     n->info=0;
  if(n->left && !n->right)
     n->info=n->left->info+1;
  if(n->right && !n->left)
     n->info= n->right->info+1;
  if(n->left && n->right)
     {
       if(n->left->info <=n->right->info)
          n->info=n->left->info+1;
```

```
       else
          n->info=n->right->info+1;
     }

}
```

L'ultimo comando condizionale potrebbe venire evitato inserendo degli else, ma il programma presentato ci sembra più semplice e quindi preferibile. Il codice di questa funzione appare di una complessità sproporzionata al suo semplice compito.È quindi naturale cercare qualche altra idea che possa semplificare la funzione. L'idea è semplice e l'abbiamo già usata nell'Esercizio 8.16: si tratta di riuscire a trattare il caso dell'albero vuoto come quello dell'albero non vuoto, cioè fornendo un risultato convenzionale che vada d'accordo *con i valori restituiti dagli alberi non vuoti. La scelta di restituire -1 nel caso dell'albero vuoto (come nell'Esercizio 8.16) non funziona perchè nell'attuale problema dobbiamo scegliere il minore risultato tra i due sotto-alberi e non il maggiore dei due, come nel caso dell'altezza. Possiamo invece restituire* `INT_MAX`*, cf. Sezione 2.1, essendo sicuri che questo valore non potrà essere minore di qualsiasi valore restituito da un'invocazione ricorsiva su un sotto-albero non vuoto (al più sarà uguale, ma è una possibilità molto remota). Ecco la funzione che otteniamo seguendo questa idea.*

```
int dist_min_foglia_2(nodo * n)
{
 if(!n)
   return INT_MAX;
 if(!n->left && !n->right) // n e' una foglia
     return n->info=0;
 int h_l=dist_min_foglia_2(n->left);
 int h_r=dist_min_foglia_2(n->right);
 if(h_l<=h_r)
   return n->info=h_l+1;
 else
   return n->info=h_r+1;
}
```

Si noti che per equiparare l'albero vuoto a quelli non vuoti, è stato necessario richiedere che la funzione restituisca col `return` *il valore che calcola, oltre ad inserirlo in* `n->info` *(cosa che sarebbe impossibile per l'albero vuoto). Per capire bene cosa succede con l'esecuzione di, per esempio,* `return n->info=h_r+1;` *si deve capire che* `n->info=h_r+1;` *in C++ è un'espressione che, oltre a eseguire l'assegnazione, ha un valore e che questo valore è quello della parte destra dell'assegnazione, cioè di* `h_r+1` *nel nostro esempio. Quindi il* `return` *restituisce questo valore al chiamante. Il lettore potrebbe (e dovrebbe!) chiedersi se il test che distingue il caso che* `n` *sia una foglia, sia veramente necessario e perchè. La*

risposta è affermativa e la spiegazione viene data attraverso la dimostrazione di correttezza della funzione `dist_min_foglia_2` *che segue.*

PRE =(`n` **punta ad un albero binario, possibilmente vuoto)**, **POST** =**(se** `n` **punta all'albero vuoto, allora restituisce** `INT_MAX`**, altrimenti, alla fine dell'esecuzione ogni nodo dell'albero puntato da** `n` **ha nel suo campo** `info` **il valore corretto e viene restituito** `n->info`**)**. *Il caso* `!n` *è ovvio. Anche nel caso in cui* `n` *è una foglia la* **POST** *è soddisfatta. Se* `n` *non è foglia, allora significa che* `n` *ha almeno un figlio e quindi almeno una delle due invocazioni ricorsive restituisce un valore che fa riferimento ad un sotto-albero di* `n` *(anche se questo valore è* `INT_MAX`*). Quindi il calcolo che viene eseguito dalla funzione, cioè scegliere il minore tra* `h_l` *e* `h_r` *e inserirlo in* `n->info` *dopo aver sommato 1 e restituirlo, è il calcolo corretto che soddisfa* **POST**.

Esercizio 8.19 *Per i seguenti esercizi scrivere pre e postcondizione e fare la dimostrazione di correttezza.*

1. *Usare le idee discusse nell'Esercizio 8.18 per realizzare una funzione ricorsiva che inserisca in ogni nodo* `n` *di un albero dato sia la distanza minima di* `n` *da una foglia (come nell'Esercizio 8.18), sia la distanza massima di* `n` *da una foglia (cioè l'altezza dell'albero radicato in* `n`*). Naturalmente si dovrà modificare la struttura* `nodo` *di un albero binario in modo da accomodare 2 campi interi* `info1` *ed* `info2`.

2. *Usare le idee discusse nell'Ersercizio 8.18 per costruire una funzione ricorsiva che inserisce nel campo* `info` *di ogni nodo* `n` *di un albero binario il numero dei nodi del sotto-albero radicato in* `n` *che hanno 1 solo figlio.*

3. *Un'estensione dell'esercizio precedente: vogliamo una funzione che inserisca in ogni nodo* `n` *sia il numero dei nodi del sotto-albero radicato in* `n` *con il solo figlio sinistro che il numero dei nodi con il solo figlio destro. Di nuovo i nodi dovranno avere 2 campi* `info1` *e* `info2` *per accogliere questi due valori.*

4. *Scrivere una funzione ricorsiva che riceva il puntatore ad un albero e inserisca nel campo* `info` *di ogni nodo* `n` *il numero dei nodi con 1 solo figlio nel sottoalbero radicato in* `n`.

Esercizio Risolto 8.20 *Vediamo ora il problema di trovare in un albero binario il nodo con campo* `info` *massimo che nel seguito chiameremo semplicemente il nodo massimo. Nel caso ci fossero più nodi con campo informativo uguale tra loro e maggiore o uguale a quello di tutti gli altri nodi, il nodo massimo è semplicemente uno di questi nodi. Studieremo 2 funzioni che eseguono questo calcolo. La prima è particolarmente breve, ma non facile da capire. Calcola il puntatore al nodo con campo* `info` *massimo in un parametro formale* `nodo *& M`*, cioè passato per riferimento. In ogni momento questo parametro* `M` *punta al nodo massimo trovato fino a quel momento. La funzione* `max_rif(n,M)` *confronta* `M->info`

con `n->info` *modificando il massimo* `M` *se* `n` *è un nodo maggiore del massimo trovato fino a quel momento. Dopo di che invoca semplicemente se stessa sui figli di* `n`. *Le invocazioni con* `n=0` *non modificano* `M`. *L'invocazione iniziale e':* `nodo * M=0; max_rif(root, M);`.

```
void max_rif(nodo *n, nodo *& M)
{
  if(n)
   {
    if(!M)
       M=n;
    else
       if(n->info > M->info)
         M=n;// trovato un nodo piu' grande, cambio il massimo M
    max_rif(n->left,M);
    max_rif(n->right,M);
   }
}
```

Dimostrare la correttezza di `max_rif` *ci pone davanti ad un ostacolo nuovo: il parametro* `M` *punta ad un nodo che potrebbe non essere nell'albero radicato in* `n`. *Come riusciamo a caratterizzare precisamente le proprietà che questo* `M` *soddisfa all'inizio della funzione, cioè nella* **PRE** *e alla fine della funzione, cioè nella* **POST***? Innanzitutto, chiamiamo* `T` *l'albero intero di cui vogliamo calcolare il nodo massimo. Dato un qualsiasi nodo* `n` *di* `T`, *i nodi di* `T` *che* **precedono** `n`, *sono quelli che vengono percorsi prima di arrivare ad* `n`, *percorrendo* `T` *in ordine prefisso, come fa la funzione* `max_rif`. *Allora, la precondizione di* `max_rif` *è:* **PRE** =**(**`n` **punta ad un sotto-albero, possibilmente vuoto, di** `T` **e** `M` **punta al nodo massimo tra quelli che precedono** `n`**)***; e la postcondizione è:* **POST** =**(**`M` **punta al nodo massimo tra quelli che precedono** `n` **e i discendenti di** `n`**)**. *Il caso base è che* `n` *punti all'albero vuoto e questo, assieme a* **PRE** *implica* **POST** *visto che non ci sono nuovi nodi da considerare per stabilire il massimo. Consideriamo il caso induttivo, cioè che* `n` *punti ad un nodo, allora si considera il caso particolare che* `M==0`, *cioè che nessun nodo sia ancora stato considerato e quindi* `M` *è ancora uguale a 0 ad indicare questa situazione iniziale. Questo significa che* `n` *è la radice di* `T`. *Essendo* `n` *il primo nodo esaminato è corretto che esso diventi il massimo (come viene fatto da* `max_rif`*). Quindi, la prima invocazione ricorsiva viene eseguita con* **PRE** *vera e quindi, quando essa termina, vale la sua* **POST** , *da cui segue che vale la* **PRE** *per la seconda invocazione ricorsiva e quindi vale la sua* **POST** , *da cui segue che vale* **POST** *per l'intera funzione che quindi trova il nodo massimo di* `T`. *Per il caso in cui* `M !=0` *vale lo stesso ragionamento. Si noti che i nodi che precedono* `n->left` *sono quelli che precedono* `n` *più* `n` *stesso (e questi nodi sono considerati prima della prima invocazione ricorsiva), mentre i nodi che precedono* `n->right` *sono quelli che precedono* `n`, `n` *stesso e quelli*

dell'albero radicato in `n->left` *(e di nuovo sono tutti considerati prima della seconda invocazione ricorsiva).*

La seconda funzione per calcolare il nodo massimo `max2` *è parecchio più complicata della precedente, ma vale la pena di studiarla comunque. Essa è una funzione ricorsiva classica, nel senso che l'invocazione di* `max2(n)` *restituisce (con un* `return`*) il nodo massimo dell'albero radicato in* `n`*. La funzione ha senso solo se* `n` *non è 0. Per poter eseguire questa operazione, la funzione deve fare le chiamate ricorsive sui figli di* `n`*, ottenere i loro massimi e confrontarli con* `n` *per calcolare il massimo di tutto l'albero radicato in* `n`*. È importante fare attenzione ad eseguire le chiamate ricorsive solo quando i figli di* `n` *non sono 0. La funzione è organizzata in modo da evitare di distinguere in modo esplicito i diversi casi ed in questo modo guadagnamo in concisione (che, come di consueto, paghiamo con minore semplicità).*

```
nodo * max2(nodo *n) // n e' sempre != 0
{
 nodo * primo, *k;
 primo=n;
 if(n->left)
  {
   k= max2(n->left);
   if(k->info> n->info)
     primo=k;
  }
 if(n->right)
  {
   k= max2(n->right);
   if(k->info> primo->info)
     primo=k;
  }
 return primo;
}
```

Lasciamo come esercizio al lettore il compito di scrivere la pre-e la postcondizionedi questa funzione e di dimostrare per induzione la sua correttezza.

Esercizio 8.21 *Risolvere i seguenti esercizi con funzioni ricorsive di cui è richiesta anche la dimostrazione di correttezza.*

1. *Il secondo nodo di un albero è il nodo con il campo* `info` *immediatamente più piccolo del massimo. Si scriva una funzione che restituisca il puntatore al secondo nodo di un albero. Ai fini di inizializzare le variabili, conviene supporre che l'albero iniziale abbia almeno 2 nodi. La funzione segue lo schema del calcolo del secondo di un array visto nella Sezione* **??**.

2. *Scrivere una funzione che restituisca il puntatore del nodo di un albero che sia il minimo nodo con campo* `info` *maggiore di quello della radice dell'albero.*

Esercizio 8.22 *Consideriamo ora il seguente problema: dato un albero binario, allocare un array di interi che abbia un numero di elementi uguale al numero di nodi dell'albero e poi inserire in ogni elemento dell'array i campi info di tutti i nodi dell'albero. Procediamo nella maniera seguente. Per prima cosa contiamo i nodi dell'albero e se sono più di 0 allora allochiamo un array di dimensione uguale a questo numero. Dopo di che percorriamo l'albero ed inseriamo il puntatore al nodo corrente in un successivo elemento dell'array. Il punto essenziale della soluzione è l'utilizzo del parametro* `x` *di tipo* `int * &` *che ad ogni istante punta al prossimo elemento libero dell'array* `A`*. Quando un nuovo elemento* $info$ *viene inserito, allora* `x` *viene incrementato di 1, cioè punta al prossimo elemento libero. Osserva che la seconda chiamata a* `fai_ric` *riceve il valore di* `x` *dalla prima chiamata ricorsiva che lo avrà incrementato di tante posizioni quanti sono gli inserimenti nell'array che questa chiamata ha effettuato (osserva che il numero di questi inserimenti è uguale al numero di nodi del sotto albero sinistro di* `n`*).*

```
int conta(nodo *n)
{
 if(n)
  return 1+conta(n->left)+conta(n->right);
 else
  return 0;
}

 void fai_ric(nodo * n, int* &x)  // se n=0 non fa nulla
{
 if(n)
  {
   *x=n->info;
   x=x+1; // passa al prossimo elemento dell'array
   fai_ric(n->left,x);
   fai_ric(n->right,x);
  }
}

int * fai(nodo * n)
{
  int nodi=conta(n);
  int* A=new int[nodi];
  int * m=A; // altrimenti perderemmo il valore di A
  fai_ric(n,m);
```

```
  return A; // restituisce l'array riempito, ha senso visto
            // che e' allocato dinamicamente

}
```

Esercizio 8.23 *1. Si modifichi l'esercizio precedente in modo che l'array abbia elementi di tipo* `nodo*` *anzichè* `int` *ed ogni elemento dell'array deve contenere un puntatore ad un diverso nodo dell'albero.*

2. Si chiede di scrivere un programma che faccia lo stesso calcolo del programma dell'esercizio 8.22 e che inoltre segua lo stesso schema con le due funzioni `fai` *e* `fai_ric`*, ma passando il parametro* `x` *alla funzione* `fai_ric` *per valore anzichè per riferimento.*

8.6 Ricorsione ed iterazione

Un punto importante della ricorsione è che ogni funzione ricorsiva può essere simulata da una iterativa, ma quest'ultima deve spesso costruire una struttura dati che simula la pila delle chiamate ricorsive ed è quindi parecchio più complicata. In realtà ci sono 2 tipi di ricorsione: uno semplice da simulare iterativamente ed uno no. Quello semplice viene chiamato **recursione di coda** (tail recursion in inglese). Vediamo un esempio di tail recursion. Come esempio di tail recursion useremo una funzione che stampa una lista dal primo all'ultimo elemento. La funzione ricorsiva è la seguente:

```
void stampa(nodo *n)
{
 if(n)
  {
   cout<<n->info<<' ';
   stampa(n->next); // (*)
  }
}
```

La ricorsione usata in questa funzione è terminale perchè al ritorno dalla chiamata ricorsiva marcata (*) la funzione non deve più fare nulla, cioè deve solo ritornare al chiamante. Questa proprietà rende facile trasformarla in iterativa nel modo seguente.

```
void stampa_iter(nodo * n)
{
 while(n)
  {
    cout<<n->info<<' ';
    n=n->next;
```

```
  }
}
```

Si noti che questa funzione percorre la lista dall'inizio alla fine (come quella ricorsiva) ma, al contrario di quella ricorsiva, non ritorna indietro all'inizio come fa questa. D'altra parte essendo la ricorsione terminale, tornare indietro non serve a niente. Quindi evitarlo fa addirittura guadagnare in efficienza. È facile però trovare un esempio in cui la trasformazione non è così facile. La seguente funzione stampa una lista in ordine inverso (dalla fine all'inizio) ed essa non è tail recursive: dopo la chiamata (*) c'è la stampa da fare.

```
void stampa_inv(nodo *n)
{
 if(n)
  {
    stampa(n->next); // (*)
    cout<<n->info<<' ';
  }
}
```

Nonostante diventi più difficile, è possibile trasformare anche funzioni non tail recursive in iterative. In realtà questa trasformazione si può sempre fare. La tecnica generale è di costruire esplicitamente la pila delle chiamate ricorsive in modo da poter anche simulare il ritorno indietro della ricorsione (che non serve a nulla nel caso di tail recursion). Nel caso specifico questa tecnica si può applicare facilmente: basta ricordarsi i campi `info` da stampare in un array e poi stamparli in ordine inverso. L'array simula la pila delle chiamate ricorsive.

```
void stampa_iter_inv(nodo * n)
{
 int nodi=conta(n);
 int *A=new int [nodi];
 int pos=0;
 while(n)
  { A[pos]=n->info; pos++; n=n->next;}
 pos--;
 while(pos>=0)
 { cout<<A[pos]<<' ';pos--;}
}
```

In questa funzione la scansione della lista serve solo a costruire A. Un secondo ciclo di stampa che scandisce A dalla fine all'inizio simula il ritorno indietro automatico delle invocazioni ricorsive.

Capitolo 9

Approfondimenti

9.1 La visibilità delle variabili

In C++ una sequenza di comandi racchiusa tra parentesi graffe è un **blocco**. Naturalmente un blocco può contenerne un altro e così via. In questi casi parliamo di **annidamento (nesting)** di blocchi. Un blocco può contenere dichiarazioni di variabili. Il C++ ha una regola molto semplice che le dichiarazioni devono rispettare, che si chiama **regola della dichiarazione singola** e che legge come segue: in uno stesso blocco non ci possono essere dichiarazioni multiple della stessa variabile (anche con tipi diversi). Liberiamo il campo da possibili incomprensioni La regola della dichiarazione singola consente che una variabile dichiarata in un blocco B venga ridichiarata in un blocco annidato dentro a B. Quello che è invece vietato è la ridichiarazione in un blocco allo stesso livello di annidamento.

Le variabili dichiarate nei blocchi si chiamano **automatiche** in quanto la loro gestione, cioè allocazione di memoria e deallocazione, avviene automaticamente durante l'esecuzione del programma. Nella Sezione 2.3 abbiamo descritto la gestione di queste variabili. La riassumiamo di seguito. Quando l'esecuzione *entra* in un blocco, in cima alla pila dei dati viene allocato uno spazio di memoria capace di contenere gli R-valori di tutte le variabili dichiarate nel blocco. Questo spazio viene deallocato dalla pila quando l'esecuzione esce dal blocco stesso. Si ricordi che l'invocazione di una funzione equivale a entrare nel blocco che contiene il corpo della funzione e lo spazio di memoria allocato sulla pila in questo caso, è sufficiente a contenere gli R-valori dei parametri formali e delle variabili locali della funzione. Quindi ogni variabile è *utilizzabile* solo all'interno del blocco in cui è dichiarata (e nei blocchi annidati in esso) e a partire dal punto del blocco in cui compare la dichiarazione. Questa porzione di blocco viene detta lo **scope** oppure il **campo di visibilità** della variabile dichiarata.

Consideriamo ora i vari blocchi che possono esistere in un programma C++. Abbiamo già detto che ogni volta che una funzione viene invocata le sue variabili locali ed i suoi parametri formali vengono allocati sullo stack. A differenza delle variabili normali, i parametri ricevono anche dei valori iniziali che sono quelli dei

corrispondenti parametri attuali dell'invocazione. È importante capire che non c'è alcuna relazione tra gli L-valori dei parametri formali e delle variabili locali di una funzione in 2 invocazioni diverse della funzione stessa. Ogni invocazione della funzione usa le sue variabili e queste cessano di esistere quando l'esecuzione della funzione termina. Cioè quando il flusso di esecuzione ritorna al chiamante e quindi esce dallo scope delle sue variabili locali e parametri formali.

Anche i comandi iterativi `while` e `for` definiscono un blocco che contiene il corpo del comando e che quindi può contenere dichiarazioni. Ogni iterazione del comando corrisponde ad una nuova entrata nel blocco e quindi le variabili definite nel blocco sono allocate sulla pila, mentre alla fine dell'iterazione le variabili sono deallocate. Il lettore deve guardarsi dal prendere le precedenti affermazioni troppo alla lettera. Infatti la gestione della memoria che viene effettivamente realizzata durante l'esecuzione dei programma, obbedisce a considerazioni di efficienza che spesso impongono gestioni semplificate. Per esempio, alla fine di un'iterazione, probabilmente la memoria utilizzata per le variabili del corpo dell'iterazione non viene deallocata, ma semplicemente essa viene *riciclata* per la prossima iterazione. Nella Sezione 5.2, abbiamo già descritto il comportamento del `for`. Esaminiamo anche il comportamento del `while` con il seguente esempio.

```
int t=0;
while(t<10)
{int x=1; x++; cout<<x<<endl; t++;}
```

Questo programma stampa 10 volte 2 (mentre `t` passa da 0 a 10).

Ad ogni esecuzione del corpo viene "creata" una nuova variabile `x` che viene inizializzata a 1, a cui si aggiunge 1 ottenendo 2 che viene stampato. Inoltre se dopo il corpo del `while` aggiungessimo l'assegnazione `t=x+1;`, il compilatore ci avvertirebbe che `x` non esiste in questo punto del programma: è fuori dal suo scope.

Nei rami `then` ed `else` di un comando `if` si possono inserire dichiarazioni per le quali valgono le cose appena dette.

Oltre ai blocchi che formano il corpo dei diversi comandi del C++, è possibile introdurre blocchi a piacimento semplicemente racchiudendo dei comandi tra parentesi graffe. Nella Sezione 2.3 abbiamo discusso quello che succede quando si inseriscono dichirazioni di variabili con uguale nome in blocchi annidati. In ogni blocco è visibile solo la variabile corrispondente all'ultima dichiarazione effettuata. Questo fenomeno di *oscuramento* delle dichiarazioni precedenti all'ultima è realizzato in modo semplice gestendo i dati con una pila nella maniera descritta in Sezione 2.3.

Oltre ai blocchi che si trovano all'interno delle funzioni, esiste un blocco implicito che contiene tutti gli altri blocchi e che chiameremo blocco **globale**. Il blocco locale consiste del file che contiene il nostro programma.È cioè il blocco che contiene tutte le nostre funzioni e le dichiarazioni globali (di variabili e di tipo) che sono dichiarate all'esterno da tutte le funzioni.

Le variabili dichiarate nel blocco globale del programma all'esterno da ogni funzione, si chiamano **globali** ed hanno come scope il blocco globale (dal punto in cui sono dichiarate in poi) e sono quindi utilizzabili in tutte le funzioni che sono contenute nel loro scope. In Sezione 6.3 trovate alcune osservazioni interessanti che riguardano l'uso di variabili globali all'interno delle funzioni. In Sezione 9.2 viene spiegato cosa succede del cosidetto blocco globale quando il programma è distribuito in diversi file.

Tutto quanto detto nella presente Sezione sulle variabili si applica anche ai tipi definiti dall'utente introdotti nei programmi. Le dichiarazioni di tipo vengono spesso inserite nel blocco globale in modo che i tipi definiti siano visibili in tutte le funzioni, ma nulla vieta di inserirle in blocchi più interni. In quest'ultimo caso esse avranno uno scope più limitato esattamente come le variabili. Le dichiarazioni di tipo devono aderire alla regola della dichiarazione singola (come le variabili) e presentano anche il fenomeno della "copertura" tra dichiarazioni ripetute in blocchi innestati, che abbiamo esaminato per le variabili. La regola della dichiarazione singola per i tipi vale per ogni file. Nel caso di programmi contenuti su file diversi è consentito ed a volte è necessario che la stessa dichiarazione di tipo sia ripetuta in ogni file. La gsetione di programmi su file diversi verrà spiegata nella Sezione 9.2.

9.2 Compilazione separata e namespace

Finora abbiamo considerato che i nostri programmi fossero contenuti in un unico file. Per applicazioni industriali questa ipotesi non è realistica, infatti i programmi di grosse dimensioni sono realizzati contemporaneamente da diverse equipe di sviluppatori ed ogni equipe deve comunque poter compilare la propria parte indipendentemente dal resto, per poter controllare almeno alcuni dei possibili errori.

Quindi in generale un programma C++ è suddiviso in più file (che chiameremo **unità di compilazione**) ed il compilatore è capace di compilare una singola unità producendo un file compilato. Il comando da dare per ottenere la compilazione di un'unità di nome `U1` è `g++ -c U1.cpp` che, se la compilazione ha successo, produce il file compilato in `U1.o`. Un'apposita applicazione chiamata **linker** successivamente collega i diversi file compilati in un unico programma oggetto eseguibile. Se abbiamo 2 unità compilate `U1.o` e `U2.o` esse vengono linkate con il comando `g++ U1.o U2.o`. Il modulo oggetto prodotto viene messo nel file `a.exe (a.out)`.

Quando il programma viene diviso in varie unità, possono diventare necessari molti comandi per compilare certe unità e per linkarle tra loro. Inoltre, se solo certe unità vengono modificate, rispetto all'ultima compilazione, facilmente finiremmo per ricompilare unità che non sono state modificate facendo lavoro inutile. Unix offre il comando `Makefile` che semplifica questo tipo di problemi. Questo comando è descritto brevemente nell'appendice (A).

Torniamo al nostro programma composto da varie unità. Ogni unità di compilazione, cioè ogni file, contiene, in generale, delle definizioni di funzioni, di tipi e di variabili globali. Chiaramente le diverse unità che compongono un programma non sono indipendenti tra di loro, ma alcune unità usano delle funzioni/tipi/variabili globali definiti in altre unità. È necessario quindi esportare dichiarazioni da un'unità ad un'altra. Visto che in C++ l'uso di ogni oggetto (funzione/tipo/variabile globale). Questo viene fatto inserendo nell'unità che importa delle opportune dichiarazioni (attenzione, non le definizioni, ma solo delle dichiarazioni) degli oggetti importati. Nel seguito descriveremo esattamente la forma di queste dichiarazioni, che è differente a seconda degli oggetti importati.

(a) Per quanto riguarda le funzioni la questione è semplice. Se l'unità `U` vuole usare una funzione la cui definizione risiede su su un'altra unità, allora è sufficiente inserire in `U`, prima della sua invocazione, il prototipo della funzione invocata. Non ci sono limitazioni sul numero di prototipi di una stessa funzione che possono essere contenuti in un programma.

(b) La regola della dichiarazione singola (vedi Sezione 9.1) impedisce dichiarazioni multiple di una variabile in uno stesso blocco. Questa regola si applica anche alle variabili globali. Quindi se in 2 unità di compilazione diverse, appaiono 2 dichiarazioni globali di una stessa variabile, il linker dà un errore in quanto la regola della dichiarazione singola è violata. Questo ci insegna che tutte le unità di compilazione che formano un programma, hanno un unico spazio delle variabili globali. La soluzione a questo problema è di avere in una sola unità di compilazione la dichiarazione normale della variabile globale, diciamo `int Globale;` e di inserire nelle altre unità di compilazione che vogliono usare la variabile globale `Globale`, la seguente dichiarazione, `external int Globale;`. La keyword `external` specifica al compilatore che la variabile `Globale` è dichiarata in un'altra unità con tipo `int`. In pratica, qual'è la differenza tra una dichiarazione normale di variabile globale ed una con `external`? La differenza sta nell'effetto che esse hanno sul compilatore. In corrispondenza di una dichiarazione normale della variabile globale `int Globale`, il compilatore genera istruzioni che riservano, nello stack dei dati, lo spazio RAM per contenere l'R-valore di questa variabile. Invece, quando il compilatore incontra la dichiarazione `external int Globale;`[1], esso la usa come semplice informazione sul tipo di `Globale` in modo da poter eseguire il normale controllo di tipi. Se nelle diverse unità che costituiscono un programma, ci fossero delle dichiarazioni di `Globale` con `external` e nessuna dichiarazione normale di `Globale`, allora il linker fallirebbe con un opportuno messaggio d'errore. Per quanto detto finora, una dichiarazione `external int Globale=2;` in una unità di compilazione `U` non ha senso. Infatti,

[1] Si osservi che le 2 dichiarazioni devono coincidere per il tipo della variabile dichiarata, `int` nell'esempio, altrimenti il linker darà errore.

visto che il compilatore che compila `U` indipendentemente dalle altre unità, ignora l'L-valore di `Globale` non può inizializzare la variabile globale nel modo richiesto. In realtà, il compilatore in questo caso non segnala alcun errore e l'`external` viene semplicemente ignorato e la dichiarazione viene considerata normale, cioè essa alloca memoria per `Globale` (che conterrà 2). Quindi, se esistesse un'altra unità che riservasse anch'essa memoria per la variabile globale `Globale`, allora il linker, assemblando le due unità, segnalerebbe una violazione alla regola della dichiarazione singola.

(c) Per le dichiarazioni di tipo la regola è molto semplice. Se un tipo serve in diverse unità di compilazione, allora basta ripetere la dichiarazione del tipo in ciascuna unità. Le dichiarazioni ripetute devono essere assolutamente identiche, sia nel nome del tipo che nella sua struttura e nel nome dei suoi campi. Quindi, per i programmi su più file, il C++ consente, nel caso dei tipi, di violare la regola della dichiarazione singola, purchè tutte le dichiarazioni siano identiche. Si osservi che questa *liberalità* del C++ è motivata dal fatto che le dichiarazioni di tipo non richiedono allocazione di memoria e quindi non causano i problemi visti nel punto (b) per le dichiarazioni di variabili.

Da quanto visto finora, le diverse unità che compongonoun programma devono spesso includere dichiarazioni di oggetti definiti su altre unità. Le dichiarazioni devono essere correte rispetto alle definizioni. Incongruità tra dichirazioni definizioni risulterebbe in errori segnalati dal linker nel caso più fortunato ed in un comportamento anomalo del programma nel caso peggiore. Per cercare di evitare incoerenze conviene rendere esplicite le dichiarazioni importate da ciascuna unità. A questo fine si usano dei file particolari, detti **header** che sono contrassegnati dall'estensione `.h`. La tecnica è semplice. L'equipe che sviluppa una unità di compilazione, definisce un file header con le definizioni contenute in questa unità che possono servire alle altre unità. Ogni unità include gli header delle altre che contengono le dichiarazioni da importare. Il contenuto di questi header definisce **l'interfaccia** delle diverse unità. Leinterfaccia formano una specie di accordo tra le diverse equipe che permette a ciascuna di esse di sviluppare la propria unità, contando sul fatto che le altre unità siano sviluppate in modo coerente rispetto alle intefaccia stabilite. L'inclusione di un file header di nome `equiv.h` viene realizzata con l'istruzione `#include ''equiv.h''`.

Il fatto che un programma di grandi dimensioni sia composto da diverse unità riflette spesso l'organizzazione del lavoro di sviluppo e questo, non necessariamente coincide con l'organizzazione logica del programma stesso. È ovvio però che l'organizzazione logica esige dal linguaggio di programmazione degli strumenti appositi per venire espressa nel programma realizzato. Il C++ offre questi strumenti attraverso la nozione di **namespace**. Un `namespace` è un insieme di oggetti che logicamente sono collegati tra loro nel senso che assolvono un particolare compito all'interno dell'intero programma. La dichiarazione di un `namespace` ha la seguente forma:

```
namespace nome {...oggetti del namespace...};
```

È possibile avere definizioni di oggetti che appartengono allo stesso namespace anche su unità diverse. Questo fatto viene usato per mostrare ai diversi utenti del `namespace` delle interfacce diverse. Inoltre, per rendere maggiormente leggibile la definizione di un namespace, conviene inserire all'interno della definizione del namespace solo i prototipi delle funzioni del namespace. Le definizioni complete delle funzioni possono essere date successivamente specificando la loro appartenenza al `namespace` attraverso il nome qualificato (cioè specificando con l'operatore di scoping "::" il namespace cui appartiene la funzione). Come esempio consideriamo il `namespace pippo` che contiene la funzione `f`:

```
namespace pippo{
.....
double f(int, double); // solo prototipo
.....
};

double pippo::f( int x, double y) // definizione esterna
{......corpo.....}
```

Per assicurare la back-compatibility con i programmi C, ogni programma ha di default (senza bisogno di dichiararlo) il namespace **globale** che deve sempre contenere il `main`. Tutti gli oggetti che non vengano esplicitamente inseriti in un particolare namespace (tramite l'apposita dichiarazione), sono nel namespace globale che coincide quindi con il blocco globale introdotto in Sezione 6.3. Un namespace è un dominio di oggetti (variabili, tipi e funzioni) che interagiscono tra loro secondo le regole illustrate (per il namespace globale) nella Sezione 9.1. Di default, namespace diversi non possono cooperare, ma il C++ offre delle istruzioni particolare che permettono di istaurare una comunicazione tra namespace diversi. In primo luogo si possono usare i nomi qualificati. Per esempio se la funzione `pippo` è contenuta nel `namespace pluto`, allora un altro `namespace` può invocarla nel modo seguente: `pluto::pippo(...);`. I nomi qualificati possono appesantire il codice e quindi il C++ offre delle direttive per "rendere visibile" tutto un `namespace` o solo parti di esso in un'altro `namespace`. Una tale direttiva l'abbiamo inserita in tutti i programmi visti finora: `using namespace std;`, che appunto permette al nostro namespace globale di "vedere" gli oggetti definiti nel namespace `std` che sono quelli della *Standard Template Library (STL)*. Questa direttiva è grossolana: potremmo essere interessati a "vedere" (e usare) solo alcuni degli oggetti definiti in un namespace e non tutti! Possiamo quindi essere più selettivi con direttive come: `using std::cin;` in cui specifichiamo appunto che ci interessa usare solo `cin` del namespace `std` e non il resto.

Ci pare importante sottolineare nuovamente che i namespaces che costituiscono un programma riflettono la sua organizzazione logica, mentre le unità di compilazione riflettono maggiormente l'organizzazione del lavoro volto a produrre il

programma. Ovviamente è auspicabile che queste 2 organizzazioni non siano in contrasto, ma che al contrario siano coerenti tra loro.

9.3 L'overloading delle funzioni

L'overloading (sovraccaricamento) delle funzioni è un aspetto molto rilevante del C++ che consiste nel permettere di definire nei programmi varie funzioni con lo stesso nome, ma con lista di parametri formali diverse. Per esempio, il C++ ci permette di definire: `void f(int x);` e `void f(double x);`. Poi se nel nostro programma invochiamo `f(5);` il compilatore sceglierà automaticamente la prima funzione mentre se invochiamo `f(3.5);` sceglierà la seconda. Ovviamente il compilatore basa la scelta sul tipo del parametro attuale delle invocazioni. Questo fenomeno avviene costantemente nel C++ e generalmente lo si prende per scontato. Per esempio, ci pare naturale poter scrivere i comandi di output; `cout<<''ecco un comando'';` e `cout<<5;` e diamo per scontato che il primo stampi la stringa ed il secondo il numero 5. Vale la pena però di chiedersi come mai questo succeda. In realtà la prima richiesta di stampa viene soddisfatta dall'operatore `<<(const ostream &, const char *);`, mentre la seconda viene soddisfatta dall'operatore `<<(const ostream &, const int );`. Abbiamo già visto che il compilatore sceglie la versione della funzione da usare in base alla corrispondenza tra i tipi dei parametri attuali e i tipi di quelli formali. Nel caso che ci sia perfetta corrispondenza non ci sono problemi, ma se ci sono differenze tra questi tipi, è necessario fare ricorso alle conversioni automatiche e quindi il compilatore può facilmente trovare il caso in cui applicando conversioni diverse, una certa invocazione potrebbe venire soddisfatta da 2 o anche più funzioni sovraccaricate.

Per risolvere casi come questo, il C++ usa un algoritmo preciso che si chiama algoritmo di **risoluzione** dell'overloading. L'idea di fondo dell'algoritmo è quella di ordinare i diversi tipi di conversioni in funzione del grado di conversione dei valori dei parametri attuali che esse richiedono. Insomma le conversioni vengono catalogate per *pericolosità* e viene scelta dal compilatore la funzione che causa le conversioni strettamente meno pericolose di quelle che le altre funzioni richiederebbero. Si osservi la coerenza tra questo principio e quello che guida le conversioni automatiche trattate nella Sezione 7.4.

Vediamo come sono ordinate per pericolosità le conversioni. Nel seguito, per semplicità, consideriamo funzioni con 1 solo parametro ed elenchiamo i diversi tipi di conversione per pericolosità crescente.

1. tra parametro attuale e formale c'è perfetta uguaglianza oppure è necessaria una conversione triviale come tra `T&` e `T`, tra `int[]` e `int *`, e tra `T` e `const T`;

2. per convertire il tipo del parametro attuale in quello del parametro formale è sufficiente una promozione. Si ricordi che le promozioni sono: {`char,`

`bool, enum` } $\rightarrow$ {`short int` } $\rightarrow$ {`int` } $\rightarrow$ { `float` } $\rightarrow$ { `double` }, vedi Sezione 7.4;

3. per convertire il tipo del parametro attuale in quello del parametro formale è necessaria una conversione che va in senso opposto alle promozioni ricordate nel punto precedente, cioè, per esempio, `int` $\rightarrow$ `char` e `double` $\rightarrow$ `int`;

4. per convertire il tipo del parametro attuale in quello del parametro formale è necessaria una conversione definita dall'utente (questa possibilità riguarda soprattutto le classi del C++ che verrà studiata in testi più avanzati di questo).

L'algoritmo di overloading resolution, attribuisce ad ogni funzione che potrebbe soddisfare un'invocazione il livello di pericolosità della conversione necessaria a trasformare il parametro attuale in quello formale della funzione considerata. Se si trova una funzione che ha pericolosità strettamente minore di quella delle altre pretendenti, allora il compilatore la sceglie. Nel caso di funzioni con più di un parametro formale, l'algoritmo di risoluzione dell'overloading deve calcolare per ogni funzione possibile il grado di pericolosità della conversione necessaria per ciascun suo parametro (con la classificazione appena illustrata). L'algoritmo sceglierà quella funzione `F` tale che soddisfi le seguenti 2 condizioni: (i) essa è, tra le possibili, la funzione che ha il grado minimo per ogni parametro e, (ii) se confrontata con ogni altra possibile funzione `F'`, è possibile trovare un parametro tale che il grado di pericolosità che `F` possiede per quel parametro è minore di quello di `F'` per lo stesso parametro. Se una tale `F` esiste, l'algoritmo di risoluzione la sceglie per soddisfare l'invocazione in esame, altrimenti segnala un errore dichiarando che il programma contiene un'ambiguità.

Vediamo alcuni esempi tratti dal libro `The C++ Programming Language` di B. Stroustrup. Terza Ed. Addison Wesley, 1997. B. Stroustrup è l'inventore del C++.

```
void print(int); // (1)
void print(const char *); // (2)
void print(double);  // (3)
void print(long);  // (4)
void print(char);  // (5)

void f(char c, int i, short int s, float f)
{
 print(c);   // invoca (5) senza conversioni,
       //livello di pericolosita' (1)
 print(i);   // invoca (1) senza conversioni,
       //livello (1)
 print(s);   // invoca (1) con promozione
       //short int --> int, livello (2)
```

```
 print(f);   // invoca (3) con promozione
      //float --> double livello (2)
 print('a'); // invoca (5) senza conversioni,
      //livello (1)
 print(49);  // invoca (1) senza conversioni,
      //livello (1)
 print(''a'')// invoca (2) senza conversioni,
      //livello (1)
}
```

Per un esempio di conversione con livello di pericolosità (3), basta riflettere sulla ragione per cui `print(49);` non invoca la funzione (5): la conversione richiesta da questa funzione è `int --> char` che ha livello (3). Visto che la funzione (1) richiede una conversione di livello (1) viene preferita alla (5). Non ci fosse (1), avremmo un'ambiguità tra le funzioni (3) e (4) che entrambe richiedono conversioni di livello (2).

Vediamo un altro esempio per mostrare conversioni di livello (4). A questo fine, introduciamo un tipo struttura con un costruttore, come abbiamo già visto nella Sezione 7.2, e anche con degli operatori di conversione che specificano come un valore del tipo struttura può all'occorrenza essere convertito in un valore di qualche altro tipo. Vediamo l'intero programma che usa questa struttura.

```
struct T{int a; char b; double d;
  T(int a1,char a2, double a3){a=a1; b=a2; d=a3;} // costruttore di T
  operator int(){return a;}   // operatore di conversione  T --> int
};
void print(double a)
{cout<<a<<endl;}

void print(int s)
{cout<<s<<endl;}

main()
{
  T a(12,'c',3.1);
  print(a);   // usa print(int) con conversione definita dall'utente,
      // cioe' livello (4), stampa 12
}
```

È facile modificare la struttura `T` in modo che l'invocazione contenuta nel `main` dell'esempio precedente trovi un'ambiguità: basta inserire nel tipo `T` un secondo operatore di conversione da `T --> double`, cioè

```
 operator double(){return d;}
```

In questo caso il compilatore non può decidere quale `print` invocare, visto che entrambe richiedono una conversione di livello (4).

Per completezza notiamo che nella parte orientata agli oggetti del C++ (non coperta in questo testo), una delle nozioni principali è quella di **ereditarietà** che permette di definire strutture che ereditano campi e funzioni da altre strutture. Le prime vengono chiamate **sottoclassi** e le seconde **classi base**. Generalmente, una relazione di ereditarietà tra una sottoclasse `A` ed una classe base `B` definisce una conversione (di livello (4)) tra oggetti di tipo `A` in oggetti di tipo `B`.

Nell'illustrazione dell'algoritmo di risoluzione dell'overloading fatta finora, non abbiamo considerato la presenza di parametri di default. In realtà questi parametri non entrano in gioco. L'algoritmo di risoluzione considera solo i parametri attuali presenti nell'invocazione considerata. Quindi se tra le funzioni considerate dalla risoluzione ce n'è una con parametri di default aggiuntivi rispetto a quelli attuali, questi non contribuiscono alla scelta dell'algoritmo. Chiariamo con un esempio.

```
void f(int x, int y=10)  // funzione (1)
{
  cout <<"con default"<<endl;
}
void f(long int x)   // funzione (2)
{
  cout<<"senza default"<<endl;
}
main()
{
  f(10);
}
```

L'algoritmo di risoluzione considera entrambe le funzioni (1) e (2). La prima ha un match perfetto (livello (1)) sul primo argomento. Il secondo argomento non viene considerato visto che non c'è nell'invocazione e che questo è consentito dalla funzione (1) che prevede un valore di default per questo parametro. La funzione (2) richiede una conversione di livello (2) sul primo argomento. Viene scelta la funzione (1). Insomma l'argomento di default non entra in gioco.

C'è un'ultima considerazione da fare riguardo al tipo ritornato dalle funzioni. Da quanto detto finora dovrebbe essere chiaro che il tipo ritornato dalle funzioni sovraccaricate viene semplicemente ignorato dall'algoritmo di risoluzione dell'overloading. Il motivo è semplice: l'algoritmo di risoluzione appena presentato è tale che considera solo l'invocazione e non il contesto in cui essa si trova. Se tenessimo conto anche del tipo del valore ritornato dalla funzione, il contesto entrerebbe in gioco complicando le cose e causando scelte incoerenti. Illustriamo questa affermazione con il seguente esempio, di nuovo tratto dal libro di Stroustrup. Con la risoluzione dell'overloading vista prima vengono fatte le scelte indicate nell'esempio.

```
float sqrt(float);  //(1)
double sqrt(double); //(2)

void f(double da,float fla)
{
 float fl=sqrt(da);//invoca (2)
 double d=sqrt(da); // invoca (2)
 fl=sqrt(fla); // invoca (1)
 d=sqrt(fla); // invoca (1)
}
```

Se facessimo dipendere la scelta della funzione dal contesto in cui si trova l'invocazione, troveremmo un'ambiguità per esempio nell'invocazione, `float fl=sqrt(da);` e quindi daremmo un errore per un'invocazione che nel comando seguente verrebbe trattata con successo. Se invece, per soddisfare `float fl=sqrt(da);` usassimo la funzione (1), sarebbe comunque strano visto che per l'invocazione uguale del comando seguente usiamo (2).

Capitolo 1

La Correttezza dei Programmi

In questo capitolo mostreremo come sia possibile dimostrare in modo convincente che i nostri programmi sono corretti, cioè fanno effettivamente quello per cui li abbiamo scritti. Iniziamo con un esempio. Successivamente descriveremo una regola generale per trattare la correttezza del `while` e la useremo per dimostrare la correttezza di alcuni semplici programmi.

Il lettore si chiederà quale sia il vantaggio di fare prove di correttezza dei programmi. Per rispondere conviene partire dall'osservazione di cosa succede quando le prove non vengono fatte. In questo caso il programmatore realizza il suo programma seguendo un'intuizione che in generale non è precisa e poi controlla che il programma sia corretto eseguendo alcuni test con input diversi. Questa maniera di procedere, molto comune, ha una debolezza. Essa non sviluppa in modo chiaro l'intuizione alla base del programma e solo questo chiarimento può far capire se l'intuizione è corretta oppure sbagliata. Invitiamo il lettore a riconoscere nella trattazione che segue, i momenti in cui l'idea intuitiva trova la sua formalizzazione precisa.

Esercizio Risolto 1.1 *Vogliamo calcolare il minimo esponente intero tale che 2 elevato a quell'esponente sia maggiore o uguale ad un intero positivo letto dallo stream di input. Un programma che risolve questo problema segue. Nel resto dell'esempio lo chiameremo programma 1.1.*

```
#include<iostream>
using namepace std;
main()
{
 1. int X, potenza=1, esponente=0;
 2. cout<<''Digita un valore positivo:'';
 3. cin>>X;
 4. while(X>potenza)
  {
```

```
 5. potenza*=2;
 6. esponente++;
  }
 7. cout << ''l'esponente e':''<< esponente << endl;
           // endl stampa un accapo
}
```

Questo programma sembra ragionevole, si basa sull'intuizione che dobbiamo provare successive potenze del 2 fino a che non troviamo quella che uguaglia o supera `X`*. In questo modo dovremmo determinare l'esponente cercato. Ma è veramente corretto? E poi che significa in realtà che sia corretto? Cerchiamo innanzitutto di rispondere a questa seconda domanda. Poi vedremo com'è possibile dimostrare che il programma è veramente corretto. Il primo passo è quello di specificare una coppia di asserzioni: la* **Precondizione** *(***PRE***) e la* **Postcondizione** *(***POST***) del nostro programma. In generale, un'asserzione relativa ad un programma è una qualunque affermazione sulle relazioni che legano i valori delle variabili del programma. Un'asserzione viene legata ad un punto del programma e l'idea è che essa deve essere vera ogni volta che l'esecuzione raggiunge quel punto del programma. Ricordando la nozione di stato del calcolo, data nell'introduzione di questo Capitolo, un'asserzione caratterizza un insieme di stati del calcolo: gli stati nei quali le variabili del programma hanno valori che rendono vera l'asserzione.*

La precondizione **PRE** *descrive la situazione a partire dalla quale il programma viene eseguito ed è legata al punto iniziale del programma. La postcondizione* **POST** *descrive la situazione dopo che il programma termina ed è legata al punto finale del programma. Quindi la postcondizione descrive quello che vogliamo che il nostro programma calcoli e la dimostrazione della correttezza del programma consiste nel dimostrare che ogni volta che il programma termina, lo stato del calcolo raggiunto renderà vera la postcondizione. Per il nostro esempio, la precondizione* **PRE** *e la postcondizione* **POST** *sono le seguenti:*
PRE = **(lo** `stream cin` **contiene un intero maggiore di 0)** *asserisce che solo valori interi positivi verrano letti ed assegnati a* `X`*;*
POST = **(**`esponente` **è il minimo intero tale che** $2^{\texttt{esponente}} \geq \texttt{X}$**)** *asserisce che alla fine del calcolo la variabile* `esponente` *ha il valore che vogliamo calcolare.*

Si osservi che **PRE** *considera lo stream* `cin` *e non le variabili del programma, ma visto che il contenuto di* `cin` *viene letto in* `X`*, si tratta di un'estensione ovvia. Per riportarci al caso normale basterebbe eliminare l'input ed assumere che* `X>0`.

Vogliamo dimostrare che qualora l'esecuzione del Programma 1.1 inizi da uno stato del calcolo che soddisfa **PRE** *, allora, se l'esecuzione termina, essa terminerà in uno stato del calcolo che soddisfa* **POST**. *Questa proprietà si esprime con la formula:* **PRE** `<Programma 1.1>` **POST**

Un passo importante per riuscire a fare questa dimostrazione è di spezzare il programma in pezzi tali che la loro dimostrazione sia semplice. Nel nostro esempio avremo 2 pezzi: la parte di inizializzazione, costituita dalle istruzioni 1, 2 e 3 ed il ciclo `while`*, costituito dalle istruzione 4, 5, e 6. L'istruzione 7 è di output,*

quindi non calcola nulla e viene ignorata dalla dimostrazione. Chiamiamo i 2 pezzi di programma `INIZ` *e* `WHILE`*. Dovremo dimostrare che:* **PRE** `INIZ` **P** *e* **P** `WHILE` **POST**, *dove* **P** *è un'opportuna asserzione. È semplice capire che la composizione delle 2 dimostrazioni prova che:* **PRE** `INIZ WHILE` **POST** *e quindi che* **PRE** `<Programma 1.1>` **POST**

Trattare i cicli è sempre la parte più dura, quindi partiamo da li. La dimostrazione di correttezza dei cicli si fonda sull'idea di **invariante**. *Un invariante di un ciclo* `while` *è un'asserzione che è vera ogni volta che l'esecuzione è sul punto di iniziare ad eseguire il ciclo. L'invariante di* `WHILE` *formalizza l'intuizione precedente. La prima volta che si esegue il ciclo è un caso speciale che consideriamo dopo, ma consideriamo la seconda, o la terza, o la n-esima volta. Se siamo all'inizio ciclo per l'n-esima volta (n>0) significa che c'è stata un'esecuzione precedente e che in quell'occasione* `potenza` *ha un valore* **v** *che è minore di quello di* `X`*. Altrimenti l'esecuzione sarebbe uscita dal ciclo e non saremmo all'n-esimo inizio ciclo. Avendo eseguito il ciclo una volta di più sappiamo che ora* `potenza` *ha valore* **v*2**, *insomma è la prossima potenza del 2. Questo è esattamente quanto scriviamo nell'invariante:*

R=`(X>0, potenza=`$2^{\texttt{esponente}}$`, X>`$2^{\texttt{esponente}-1}$`)`

Mentre il ragionamento che ci ha portato all'invariante **R** *è chiaro per le esecuzioni del ciclo dalla seconda volta in poi, un dubbio nasce pensando alla prima esecuzione del ciclo. In effetti per trattare anche questo caso, dobbiamo garantire che la parte* `INIZ` *del programma rende vera* **R**. *Questa è proprio la dimostrazione del primo pezzo del programma che avevamo trascurato per occuparci del ciclo. Si tratta di dimostrare quindi:* **PRE** `INIZ` **R**. *Da* **PRE** *e dalla istruzione di lettura 3, segue che* `(X>0)`. *Le dichiarazioni con inizializzazione della riga 1 rendono vero che* `potenza=1 =` $2^{\texttt{esponente}=0}$. *Inoltre, visto che* `(X>0)`*, segue che* `X>`2^{-1}`=0,5`. *Quindi vale* **PRE** `INIZ` **R** *che dimostra che la prima volta che si arriva al* `WHILE` *vale* **R**. *Per concludere la dimostrazione restano 2 cose da fare:*

- *Dimostrare che* **R** *è effettivamente invariante del nostro ciclo. Questo consiste nel dimostrare che se la condizione del* `WHILE` *è vera e si esegue il corpo del* `WHILE` *una volta, si arriva in uno stato che soddisfa nuovamente* **R**. *Questo significa dimostrare la seguente proprietà :*

  R **&&** `(X >` $2^{\texttt{esponente}}$`)` `<potenza*=2; esponente++;>` **R**

  Per questo basta osservare che **R** **&&** `(X >` $2^{\texttt{esponente}}$`)` *implica* `(X >` $2^{\texttt{esponente}}$`, potenza =` $2^{\texttt{esponente}}$`)` *e quindi dopo l'esecuzione di* `potenza*=2;` *vale*

  `(X >` $2^{\texttt{esponente}}$`, potenza =` $2^{\texttt{esponente}+1}$`)` *e dopo l'esecuzione di* `esponente++;` *vale di nuovo,*

  `(X >` $2^{\texttt{esponente}-1}$`, potenza =` $2^{\texttt{esponente}}$`)` = **R**.

- *Per finire si deve mostrare che all'uscita dal ciclo (se viene mai raggiunta), vale* **POST**. *All'uscita dal ciclo l'espressione booleana che regola il ciclo*

deve essere falsa, quindi vale: **R &&** `!(X > potenza)` *che implica* `(X > 2^(esponente-1)) && (X <= 2^esponente)` *da cui discende immediatamente la postcondizione:*
POST = **(**`esponente` **è il minimo intero tale che** $2^{\texttt{esponente}} \geq$ `X`**)**.

1.1 Regole di prova

Generalizziamo la dimostrazione appena discussa per avere una regola da applicare a cicli `while` qualsiasi.

In generale la regola di verifica del `while` è la seguente: per dimostrare che vale **P** `<while(B) C>` **Q** è necessario fare le seguenti 3 dimostrazioni:

1. **P** $\Rightarrow$ **R** (condizione iniziale);
2. **R** **&&** `B <C>` **R** (invarianza);
3. **R** **&&** `!B` $\Rightarrow$ **Q** (condizione d'uscita).

Il lettore attento avrà notato che nella precedente dimostrazione di correttezza abbiamo dimostrato che il programma è corretto *se termina* e che non abbiamo dimostrato che esso termina effettivamente. Tecnicamente abbiamo dimostrato la correttezza **parziale** del programma. Per essere veramente certi che il nostro programma sia completamente corretto dovremmo dimostrare anche che esso termina e solo a fronte di tale dimostrazione, potremmo asserire la correttezza **totale** del programma. Generalmente la prova di terminazione è indipendente da quella della correttezza parziale. Per il Programma 1.1 provare la terminazione è facile: la nonterminazione può dipendere solo dal ciclo `while` e la **PRE** ci assicura che `X` è positivo, grande a piacere, ma finito e quindi, calcolando potenze successive del 2, certamente questo valore prima o poi verrà uguagliato o superato e l'esecuzione uscirà dal ciclo. Purtroppo non sempre le prove di terminazione sono così semplici !

1.2 Altri esempi di prove di correttezza

In questa Sezione affronteremo alcuni problemi non triviali utilizzando gli elementi di C++ introdotti finora. Gli esercizi riguardano il passaggio dalla rappresentazione esterna a quella interna di interi e di reali. Prima di affrontare questi esercizi vogliamo però introdurre alcuni elementi della gestione degli eccezioni del C++. Si tratta di una tecnica molto utile che conviene imparare subito. Maggiori elementi sulla tecnica verranno forniti nella Sezione **??**.

1.2.1 Esempio di gestione delle eccezioni

Spesso i programmi usano file per leggere il loro input. Nella Sezione **??** abbiamo visto che l'apertura di un file può fallire (quando il file non è nel posto indicato dal

nome usato nella dichiarazione che richiede l'apertura). In casi come questo è utile avere un modo di gestire in modo chiaro e semplice i casi di errore (le cosidette eccezioni) e il C++ ce lo offre (così come molti altri linguaggi tra cui Java). Il frammento che segue mostra come questo possa venire fatto usando le apposite istruzioni C++ `try, throw` e `catch` e che spiegheremo dopo il programma.

```
#include<iostream>
#include<fstream>
using namespace std;
main()
 try // apertura file
    {
      ifstream IN("input");
      if(!IN)
         throw(0);
      ofstream OUT("output");
      if(!OUT)
         throw(1);

       ..... resto del main

      IN.close(); OUT.close(); // chiusura dei file
    }
     catch(int x){
      if(x==0)
          cout<<"problemi con il file di input"<<endl;}
      else
           cout<<"problemi con il file di output"<<endl;}
```

Il corpo nel `main` è costituito da un'istruzione `try` che indica che nel blocco che segue può venire sollevata un'eccezione ed infatti in questo blocco sono presenti istruzioni `throw( )` che, qualora eseguite, producono un salto all'istruzione `catch( )` in cui viene gestita l'eccezione occorsa. Nel frammento, viene controllato che l'apertura dei due file "input" e "output" ha successo ed in caso contrario viene eseguito `throw(0)` e `throw(1)`, rispettivamente. Entrambe queste istruzioni, qualora eseguite, comportano un salto alla `catch` in fondo al `main` che stampa un opportuno messaggio. Dopo la `catch` l'esecuzione raggiunge la fine del `main` e quindi termina. Quello appena illustrato è un esempio molto semplice di gestione delle eccezioni. Una trattazione più approfondita di questo argomento è in Sezione **??**. Il precedente frammento di programma ci mostra anche che, una volta usati i file, è necessario chiuderli con le istruzioni `IN.close();` e `OUT.close();`.

Esercizio Risolto 1.2 *Il primo esercizio che vogliamo risolvere chiede di realizzare un programma capace di eseguire la seguente sequenza di operazioni: legge*

dal file "input" una sequenza di caratteri numerici tra '0' e '9' che termina con il carattere 'a' e converte questa sequenza nel corrispondente valore intero. Esempio, se legge '2' '5' '0' '1' 'a' deve costruire il valore intero 2501. Questo valore va stampato sul file "output". Per esprimere precisamente le asserzioni di questo programma usiamo la seguente notazione. Se $c_1...c_n$ *sono caratteri numerici, allora* `NUM(`$c_1...c_n$`)` *indica il valore intero rappresentato da* $c_1...c_n$*. Innanzitutto scriviamo la pre- e la postcondizione del programma.*

PRE =**(**`IN` **contiene i caratteri** $c_1, \ldots, c_k, 'a'$**,** $\mathbf{k \geq 0}$**,** $\forall$ **j**$\in$**[1,k],** c_j **è numerico)***;*

POST =**(calcola NUM(**$c_1, \ldots, c_k$**))**.

Per capire come organizzare il programma, immaginiamo di aver letto 3 caratteri numerici c_1, c_2, c_3*, di aver già calcolato* `NUM(`c_1, c_2, c_3`)`*, e di leggere il prossimo carattere* c_4*. Si possono verificare 2 casi: se* c_4 *è numerico allora dobbiamo calcolare* `NUM(`c_1, c_2, c_3, c_4`)` *e continuare la lettura dei prossimi caratteri. Se* c_4 *è 'a', allora il programma deve fermarsi e* `NUM(`c_1, c_2, c_3`)` *è il valore cercato. Quindi dobbiamo trovarci sempre nella situazione di avere il* prossimo *carattere da sistemare. Questo è esattamente quello che succede nel seguente programma:*

```
char q;
int num=0, n=1;
IN >> q;
while(q!='a')
{
 num=num*10+(q-'0');
 IN >> q;
 n++
}
OUT<<num<<endl;
```

L'invariante del ciclo è il seguente: **R=(sono stati letti** $c_1...c_n$ **caratteri, num=NUM(**$c_1...c_{n-1}$**),** **q=**c_n**)**. *L'invariante può sembrare inutilmente complicato, ma è necessario escludere l'ultimo carattere letto* $q = c_n$ *da* `NUM` *perchè l'invariante deve essere vero prima che il test del* `while` *sia eseguito e quindi* c_n *potrebbe essere sia un nuovo carattere numerico (che farebbe continuare il ciclo), sia 'a' (che farebbe terminare il programma). Che* **R && (q=='a')**$\Rightarrow$ **POST***, è ovvio dalla* **PRE***. Per ottenere un programma completo che risolve l'esercizio 1.2 basta inserire questo ciclo nel blocco* `try` *dello schema di* `main` *dato in precedenza. Si noti che la variabile* `n` *non serve nel programma, ma serve a rendere più comprensibile* **R**.

Esercizio 1.3 *Si tratta di leggere dal file "input" una sequenza di caratteri da '0' a '9' terminata dal carattere 'a', ma questa volta la sequenza va interpretata come un valore decimale. Quindi se si legge '3' '2' '8' 'a' va costruito il valore* `double` *.328. Questo valore va stampato sul file "output". Scrivere un programma che risolva il problema proposto e fornire anche le asserzioni necessarie a dimostrare la sua correttezza.*

La soluzione dell'esercizio 1.3 è molto simile a quella appena discussa. I caratteri letti vanno trasformati in interi come prima, ma il loro valore deve essere diviso per 10^{-1}, 10^{-2}, 10^{-3}, ... È facile calcolare le necessarie potenze decrescenti del 10 nel ciclo di lettura. Un altro esercizio interessante è il seguente.

Esercizio 1.4 *Si scriva un programma che legga con* `>>` *dal file "input" un valore intero (direttamente in una variabile dichiarata intera) e calcoli la rappresentazione binaria (interna) di questo valore stampandola sul file "output". Per esempio se da "input" si legge il valore 30, allora la sua rappresentazione binaria è 11110 e si chiede di stampare la sequenza di caratteri numerici '0', '1','1','1','1', cioè la sequenza dei bit a rovescio, cioè dal bit meno significativo a quello più significativo. Questa richiesta serve a mantenere l'esercizio semplice, infatti la sequenza richiesta viene calcolata dividendo ripetutamente il valore intero per 2 e prendendo il resto. È utile per fare queste operazioni l'operazione di modulo che in C++ si indica con il simbolo* `%` *e quindi,* `num%2` *calcola il modulo di* `num` *rispetto a 2 e cioè il resto della divisione di* `num` *per 2. Scrivere un programma che risolva il problema proposto e fornire anche le asserzioni necessarie a dimostrare la sua correttezza.*

Continuiamo con altri esercizi di difficoltà crescente.

Esercizio Risolto 1.5 *Vogliamo scrivere un programma che legga dall'input standard dei caratteri e man mano che li legge li scrive sull'output standard. Il programma deve continuare a leggere e stampare caratteri per tutto il tempo in cui nessuna delle seguenti 3 condizioni è vera (cioè la lettura termina quando una delle 3 condizioni diventa vera):*

a) ha letto 2 caratteri 'a' (anche non di seguito);

b) ha letto 2 caratteri 'b' (anche non di seguito);

c) ha letto un carattere 'a', un 'b' ed un 'c' (anche non di seguito e non necessariamente nell'ordine specificato, quindi il caso si applica anche se è stato letto prima 'c', poi 'a' e poi 'b').

Non appena una di queste condizioni è vera, il programma deve terminare stampando la corrispondente stringa caso a, caso b o caso c a seconda del caso che è diventato vero. Il programma deve sempre stampare tutti i caratteri che legge dall'input standard. Cerchiamo in primo luogo di stabilire la pre- e soprattutto la postcondizione. A prima vista l'esercizio sembra banale, ma quando ci si accinge a scrivere la sua pre- e postcondizione, ci si accorge con sorpresa che non lo è poi tanto. Ovviamente vogliamo che il compito del programma termini sempre e questo richiede che l'input letto da `cin` *garantisca che una delle tre condizioni (a), (b) o (c) diventi vera prima o poi. Possiamo procedere come segue. Data una qualsiasi sequenza di caratteri $m_1, \ldots, m_k$, diciamo che è* **ben-formata** *se contiene o (almeno) 2 'a', o (almeno) 2 'b', o (almeno) 1 'a' , 1 'b' ed 1 'c'. Quindi la*

precondizione può essere: **PRE** =**(**`cin` **contiene una sequenza ben-formata di caratteri** $m_1, \ldots, m_k$**)**. *Se* $m_1, \ldots, m_k$ *è ben-formata allora essa ammette sempre un prefisso di lunghezza minima* $m_1, \ldots, m_j$,$j \leq k$ *che soddisfa esattamente una delle 3 condizioni. Questo fatto è dovuto alla proprietà che nessuna delle condizioni (a), (b) e (c) è contenuta in un a delle altre: in una situazione in cui nessuna delle 3 proprietà è vera, la lettura di un 'a' può soddisfare la condizione (a) o la (c), ma non entrambe (se soddisfa (c) allora è il primo 'a' e quindi (a) è falsa), se viene letto un 'b' si ragiona in modo simile e con un 'c' è ovvio che solo (c) può diventare vero. Quindi il nostro programma deve leggere (e stampare) il prefisso di lunghezza minima della sequenza letta da* `cin`. **POST** =**(**`cout` **contiene il prefisso minimo** $m_1, dots, m_j$ **di** $m_1, \ldots, m_k$**, seguito da "caso(a)", "caso(b)", "caso(c)", a seconda di quale dei 3 casi è soddisfatto da** $m_1, \ldots, m_j$**)**.

Il programma che cerchiamo deve contenere un ciclo che esegue la lettura e la stampa di un carattere e deve contare le occorrenze di alcuni caratteri (cioè, di 'a', 'b' e 'c'). Per contare le occorrenze di questi caratteri è conveniente usare variabili intere piuttosto che booleane. Non per risparmiare variabili, ma perchè è più semplice. Quindi useremo int contaa=0, contab=0, contac=0; L'invariante di questo ciclo potrebbe essere: **R**=**(letti** $m_1, \ldots, m_h$**, con le occorrenze di 'a'=**`contaa`**, quelle di 'b'=**`contab` **e quelle di 'c'** = `contac`**)**. *Il programma segue:*

```
main()
{
  int contaa=0, contabb=0, contac=0;
  while(!(contaa==2) && !(contab==2) &&
       !(contaa==1 && contab==1 && contac>=1))
    {
      char q;
      cin >> q;
      cout<< q;
      if(q=='a')
         contaa++;
      else
         if(q=='b')
    contab++;
  else
    if(q=='c')
      contac++;
    }
  if(contaa==2)
    cout << endl<< "caso (a)"<<endl;
  else
    if(contab==2)
      cout<<endl << "caso (b)"<<endl;
```

```
    else
      cout<<endl<<"caso (c)"<<endl;
}
```

Il caso iniziale del ciclo e il fatto che **R** *sia invariante sono facili. Invece il caso di uscita dal ciclo non lo è: vorremmo dmostrare che*

R *&&* **(**`contaa==2` || `contab==2` || **(**`contaa==1 && contab==1 && contac`$\geq$**1)) ⇒POST**.

R *ci dice che se* $m_1 \ldots, m_h$ *è stato letto, allora se* `contaa==2` *essa contiene 2 'a' e lo stesso per le altre 2 condizioni, ma non dimostra che* $m_1, \ldots, m_h$ *deve essere il prefisso minimo che soddisfa una sola delle 3 condizioni. Potrebbe per esempio essere vero che* $m_1, \ldots, m_h$ *contenga sia 2 'a' che 2 'b'. Per dimostrare che questo non può succedere, dobbiamo aggiungere ad* **R** *la seguente asserzione:* **(al più vale una di queste condizioni (**`contaa==2`**),(**`contab==2`**),(**`contaa==1 && contab==1 && contac`$\geq$**1))**.

Inizialmente sono tutte false e quindi l'asserzione è vera. Per il caso invariante, se si esegue il ciclo vale il test del ciclo e quindi sono tutte e 3 false e perciò (come spiegato prima) la lettura di un 'a' o di un 'b' o di un 'c' ne può al massimo rendere vera una! Quindi se si esce dal ciclo, esattamente una delle 3 condizioni è vera e quindi il programma legge (e stampa) il prefisso minimo di $m_1, \ldots, m_k$*, come vuole* **POST**.

Esercizio Risolto 1.6 *Quando si leggono valori da un file, una* **sentinella** *è un valore convenzionale che segnala che l'input è finito. A volte è possibile usare l'end of file come sentinella (e abbiamo visto esempi di questo nel Capitolo precedente), ma ora vogliamo invece usare come sentinella una coppia di 0 contigui. Ecco il problema. Vogliamo scrivere un programmma che legga dei valori interi dal file "input" rispettando il seguente vincolo: deve leggere al più 10 valori, ma deve fermarsi qualora legga la sentinella composta da due 0 contigui. Per ogni lettura del valore* `v`*, il programma deve stampare su "output"* `v`*, solo a condizione che esso non appartenga alla sentinella. In nessun caso il programma deve eseguire più di 10 letture. Qualche esempio di input output chiarirà l'esercizio: se su "input" si trova [0 1 0 1 0 1 0 1 0 1 0 1] (nota che sono 12 valori) il programma deve scrivere su "output" [0 1 0 1 0 1 0 1 0 1], cioè i primi 10 valori letti. Si noti che in questo caso la sentinella non viene trovata in quanto non ci sono due 0 contigui. Se su "input" si trova [0 0 1 2 3], su "output" non si deve scrivere nulla. Se su "input" c'è [1 2 3 4 5 6 7 8 9 0 0 0 0], su "output" si deve scrivere [1 2 3 4 5 6 7 8 9 0] perchè il secondo 0 è in undicesima posizione e quindi la sentinella non viene trovata.*

Ecco il primo tentativo. Usa due variabili booleane, una per indicare che alla lettura precedente è stato letto 0 e la seconda per indicare che è stato letto anche un secondo 0 di seguito al primo (insomma che la sentinella è stata letta). Il programma contiene un errore banale, ma non troppo e lo scopriremo grazie alla prova di correttezza.

```
{
  int X, n=0;
  bool uno0=false, due0=false;
  while (!due0  && n<10)
  {
   IN>>X;
   n++;
   if(X==0)
      if(uno0)
         due0=true;
      else
        uno0=true;
    else
    {
    if(uno0)
        {
         OUT << 0 <<' ';
         uno0=false;
        }
    OUT<<X<<' ';
    }

  }
}
```

La precondizione è **PRE** (input=$b_1, \ldots, b_k$, k > 9). *Dato s=*$b_1, \ldots, b_k$, *con* $k > 9$, Prefix(s) *indica* $b_1, \ldots, b_{10}$ *se questa sequenza non contiene due 0 contigui, oppure se li contiene, diciamo in posizione* j *e* $j+1$, *con* $0 < j < 10$, *allora* Prefix(s)=$b_1, \ldots, b_{j-1}$. *Adesso possiamo esprimere la postcondizione del programma che è* **POST** =(output=Prefix($b_1, \ldots, b_k$)). *L'invariante del ciclo, che come al solito chiamiamo* **R**, *consiste di varie formule che devono essere sempre tutte verificate all'inizio del ciclo:*

1. (0<=n<=10);
2. letti $b_1, \ldots, b_n$;
3. (!uno0 && !due0)$\Rightarrow$ (output=$b_1, \ldots, b_n$);
4. due0 $\Rightarrow$ ($b_{n-1} = b_n = 0$ && output=$b_1, \ldots, b_{n-2}$);
5. (uno0 && !due0) $\Rightarrow$ ($b_n = 0$ && output=$b_1, \ldots, b_{n-1}$).

R *specifica il ruolo dei 2 booleani. Se sono entrambi falsi, formula (3), allora non ci sono allarmi e scriviamo in output tutto quello che abbiamo letto. Se* due0 *è vera, formula (4), allora non abbiamo scritto le ultime due letture visto che sono*

due 0. Se solo `uno0` *è vero, formula (5), allora non abbiamo scritto l'ultima lettura (che è 0) in attesa di quella successiva. Non è difficile dimostrare che questa asserzione è effettivamente invariante. Si deve ragionare per casi:*

(i) Se leggiamo un valore non zero, allora dobbiamo scriverlo in output, ma prima di farlo dobbiamo scrivere anche l'eventuale 0 che era in attesa e `uno0` *ci dice se è il caso e ovviamente dobbiamo mettere il booleano a* `false`*;*

(ii) Se leggiamo zero, allora di nuovo `uno0` *ci dice se abbiamo trovato la sentinella (*`due0` *deve diventare* `true` *e non si scrive nulla), oppure è il primo zero e quindi* `uno0=true` *senza scriverlo.*

Per concludere la prova dobbiamo dimostrare che **R** *assieme alla negazione del test del ciclo, cioè* `!(!due0 && n<10)=(due0 || n>= 10)` *implicano la postcondizione. Visto che ci sono 2 condizioni di uscita dal ciclo, dobbiamo considerarle separatamente:*

(`due0`*): in questo caso l'invariante ci assicura che abbiamo stampato la parte dell'input che precede la sentinella;*

(`n>=10`*): usando* **R** *(è importante ricordare che vale anche* **R** *!),* `n=10`*, quindi abbiamo letto 10 valori. Dobbiamo considerare le diverse situazioni possibili.*

- *se* `!uno0 && !due0`*, allora* **R** *ci garantisce che la postcondizione è verificata;*
- *se* `due0` *è vero, allora siamo nel caso precedente già trattato;*
- *se* `uno0 && ! due0`*, allora l'ultima lettura ha letto uno zero e quindi non l'abbiamo scritto in attesa della prossima lettura che non c'è visto che siamo a* `n=10`*, quindi dovremmo stampare questo 0, ma il programma non lo fa e quindi la postcondizione non è verificata in questo caso! Per mettere le cose a posto, basta aggiungere la seguente istruzione all'uscita del ciclo:*

```
if(uno0 && !due0) OUT<< 0 <<' ';
```

Concludiamo il Capitolo con un Esercizio non banale.

Esercizio Risolto 1.7 *Questo esercizio, oltre a fare esperienza nelle prove di correttezza, serve a farci toccare con mano due cose importanti, già dette in precedenza, ma rimaste per il momento tra le eventualità teoriche. Scriveremo un programma e ne dimostreremo la correttezza parziale per poi accorgerci del fatto che (a volte) esso non termina. In sostanza il programma soddisfa la correttezza parziale, ma non quella totale. Dovremo modificare il programma per garantirne la terminazione. Inoltre, questo esempio mostra che la rappresentazione dei*

reali in un computer non è precisa e che questa imprecisione può dare risultati completamente diversi dall'attesa.

Il problema che affrontiamo è il seguente. Abbiamo un numero reale, per esempio `0.5` *e vogliamo metterlo in forma binaria, cioè come* `0.1`*. Un altro esempio è* `0.25` *che si traduce in* `0.01`*. Queste traduzioni si basano sul fatto che in un numero binario con decimali, il primo bit alla destra del punto decimale vale* $2^{-1} = 0.5$*, il secondo bit vale* $2^{-2} = 0.25$ *e così di seguito. Vogliamo realizzare un programma che trasformi in binario un qualsiasi valore* `double` *minore di 1. Precisamente il programma che vogliamo, a fronte di* `0.5` *deve produrre il carattere '1' come output, mentre a fronte di* `0.25` *deve produrre i 2 caratteri* `'0' '1'` *in questo ordine. Vediamo di essere precisi sullo scopo del programma che vogliamo costruire. Nel seguito sia* $\mathtt{X}_1$ *il valore* `double` *(minore di 1) da trasformare e data una sequenza* $s = b_1, \ldots, b_k$ *di caratteri 0/1, sia* $VAL(s) = b_1 * 2^{-1} + \cdots + b_k * 2^{-k}$*. Vogliamo quindi un programma che da* $\mathtt{X}_1$ *produca una sequenza* s *di caratteri 0/1 tale che* $\mathtt{X}_1 = Val(s)$*. Questa relazione ci suggerisce la maniera di procedere per ottenere* s*: visto che* $\mathtt{X}_1 = Val(s)$*, allora* $\mathtt{2*X}_1 = \mathtt{2*}Val(s)$ *e ovviamente moltiplicando per 2 il numero binario* $0.s = 0.b_1 \ldots b_k$*, si ottiene* $b_1.b_2 \ldots b_k$ *e l'uguaglianza di questo valore con* `2*X` *ci permette di conoscere* b_1*; se* $\mathtt{2*X}_1 \geq 1$ *allora* $b_1 = 1$*, altrimenti* $b_1 = 0$*. Possiamo ripetere lo stesso trucco per conoscere* b_2*? Certo! Il nuovo* $\mathtt{X}_2$ *da considerare è* `if (`$\mathtt{X}_1$`*2 <1)` $\mathtt{X}_1$`*2 else` $\mathtt{X}_1$`*2-1`*. $\mathtt{X}_2$ *è uguale a* $VAL(b_2 \ldots b_k)$ *e quindi possiamo di nuovo moltiplicarlo per 2 per scoprire* b_2*. Ovviamente possiamo continuare nello stesso modo per* $\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3 \ldots$*, ma qual'è la relazione tra* $\mathtt{X}_1$*,* $\mathtt{X}_n$ *e la sequenza di 0/1* $b_1, \ldots, b_{n-1}$ *che viene prodotta negli n-1 passi che portano da* $\mathtt{X}_1$ *a* $\mathtt{X}_n$*? La relazione è la seguente:* $\mathtt{X}_1 = \mathtt{X}_n * 2^{-(n-1)} + VAL(b_1, \ldots, b_{n-1})$*. Intuitivamente,* $\mathtt{X}_n$ *rappresenta la parte di* $\mathtt{X}_1$ *che resta da tradurre, ma le* $n - 1$ *moltiplicazioni per 2 fatte negli* $n - 1$ *passi devono essere eliminate per avere veramente quello che resta da tradurre di* $\mathtt{X}_1$*. Sulla base di questa spiegazione sarà facile scrivere il programma desiderato, corredato delle asserzioni che ne provano la correttezza.*

```
//PRE
double X_1=X;
int n=1;
while(X>0) //R
{
  n++;
  X=X*2;
  if(X < 1)
     cout<<'0'<<' ';
  else
     {
       cout<<'1'<<' ';
       X=X-1;
```

```
      }
 }
//POST
```

Si noti che nel programma n serve solo per scrivere l'invariante, mentre X_1 *serve a mantenere il valore iniziale di* X*. Con queste 2 variabili la pre-, la postcondizione e l'invariante del programma sono i seguenti:* **PRE** =**(0** $<$ X $<$ **1)**, **POST** =**(**X_1=VAL**(output))**, *e* **R**=**(**|**output**|=n-1**,** X_1-X$*2^{-(n-1)}$=VAL**(output))**. *In queste formule output indica la sequenza di caratteri 0/1 stampati dal programma nel momento in cui l'esecuzione raggiunge il punto associato a ciascuna formula. Per convenzione, se con c'è nessun output,* $VAL(output) = 0$*. Con* $|output|$ *indichiamo il numero di caratteri di* $output$*. L'invariante* **R** *esprime il fatto che l'output prodotto è solo una traduzione parziale di* X_1 *e quello che resta da rappresentare è* $X*2^{-(n-1)}$*.*

Come per il programma 1.2, anche questo va inserito in uno schema di $main$ *simile a quello descritto in precedenza e che si occupa di aprire il file di input segnalando l'eventuale mancata apertura. Dovrebbe essere semplice trovare il* $main$ *appropriato per questo nuovo esempio.*

Usiamo la regola generale 1.1 per dimostrare la correttezza del nostro programma. Innanzitutto, la prima volta che l'esecuzione arriva al ciclo while*,* X_1=X *mentre* n=1 *e, visto che nessun output è stato ancora eseguito,* VAL*(output)=0. Quindi* **R** *è verificata. Assumiamo poi che* **R** *sia verificata e che valga* X > 0*, allora viene eseguito il corpo del ciclo. Le prime 2 istruzioni del corpo trasformano* **R** *in* **R'**=**(**|**ouput**|=n-2**,** X_1 **−(**$X*2^{-1}$**)** $*2^{-(n-2)}$=VAL**(output))**= **(**X_1**−(**$X*2^{-(n-1)}$**)**=VAL**(output))** *e l'istruzione condizionale che segue può togliere* $1 * 2^{-(n-1)}$ *a* $X*2^{-(n-1)}$*, ma in quel caso stampa '1' e quindi* VAL*(output· 1)=*VAL*(ouput)+*$1 * 2^{-(n-1)}$*. Si noti che usiamo la condizione* $|output| = n - 2$ *di* **R'** *da cui segue che l'1 che viene stampato è in posizione* $(n-1)$ *dell'output e quindi vale* $1*2^{-(n-1)}$*. Se invece viene stampato '0', niente succede a* X *e correttamente* VAL*(output· 0)=*VAL*(output). Si osservi anche che l'aggiunta di '0' o '1' all'output, ripristina la condizione* $|output| = n - 1$ *e quindi alla fine dell'esecuzione del corpo del ciclo* **R** *vale ancora. Resta da considerare il caso di uscita dal ciclo, ma è il caso più facile. Infatti l'esecuzione abbandona il ciclo se* X = 0 *e quindi* **R** *diventa* X_1=VAL*(output) che è esattamente la postcondizione. Quindi il programma è corretto! Certamente si, ma non si deve dimenticare che abbiamo dimostrato la correttezza parziale del programma, cioè che quando il programma arriva alla fine allora vale la postcondizione. Ma arriva sempre alla fine il nostro programma ? La risposta è no. Ci sono molti valori iniziali di* X *per cui il programma non termina. Per esempio se applichiamo la procedura appena descritta al valore* X=0.4 *ci accorgiamo facilmente che la procedura entra in un ciclo producendo una sequenza infinita di decimali. D'altra parte, se invece di fare i conti a mano, eseguissimo questo programma su un computer, con sorpresa potremmo constatare che il programma termina sempre producendo una sequenza (finita) di decimali diversa da quella prodotta a mano. Come spiegare questo strano fenomeno? La spiegazione è facile.*

Si tratta del cane che si mangia la coda. Chiaramente `0.4` *necessita di un numero infinito di cifre decimali per essere rappresentato in binario e questo è impossibile su un computer. Quindi il nostro programma esegue i suoi calcoli non su* `0.4`*, ma sulla sua rappresentazione interna finita e quindi la sequenza prodotta è quella inesatta e finita che rappresenta* `0.4` *nel nostro computer. Da cui ovviamente segue che il programma termina sempre. A scopo didattico, ignoriamo questo fatto e immaginiamo, nel seguito dell'esercizio, di fare i conti a mano.*

Da quanto osservato prima, il nostro programma è quindi un esempio in cui la correttezza parziale può venire dimostrata senza avere la terminazione, senza avere cioè la correttezza totale. Per rendere il programma totalmente corretto dobbiamo fissare un grado di precisione accettabile per la traduzione che stiamo facendo, cioè anzichè continuare il ciclo fintanto che `X > 0`*, possiamo consentire un piccolo errore, per esempio potremmo inserire il seguente* $\mathtt{X}*2^{-(n-1)} > 2^{-10}$ *come test del ciclo. Questo è equivalente a richiedere che la sequenza di caratteri stampati sia al più di 10 caratteri. Infatti dopo 10 esecuzioni del ciclo* n *diventa 11 e quindi l'errore è* $\mathtt{0<=X}*2^{-10} < 2^{-10}$*, visto che il valore di* `X` *è minore di 1. Visto che è più semplice contare il numero di iterazioni del ciclo rispetto a calcolare potenze negative del 2, adottiamo questa maniera di procedere. Il nuovo programma diventa il seguente.*

```
 //PRE
 double X_1=X;
 int n=1;
 while(X>0 && n<11) //R
{
  n++;
  X=X*2;
  if(X < 1)
     cout<<'0'<<' ';
  else
     {
       cout<<'1'<<' ';
       X=X-1;
     }
}
//POST
```

Le asserzioni di questo nuovo programma diventano le seguenti: **PRE** =**(0** $<$ `X` $<$ **1)**, **POST** =**(**$\mathtt{X}_1-\mathtt{X}*2^{-10}=VAL$**(output))**, *e* **R**=**(0**$<$`n`$\leq$**11,** |**output**|=`n-1`**,** $\mathtt{X}_1-\mathtt{X}*2^{-(n-1)}=VAL$**(output))**. *Vale la pena di osservare che la postcondizione esprime il fatto che la traduzione ammette un possibile errore. La terminazione del ciclo è ovvia visto che* `n` *aumenta di 1 ad ogni iterazione. Che il nuovo invariante dimostri la correttezza parziale viene lasciato come esercizio.*