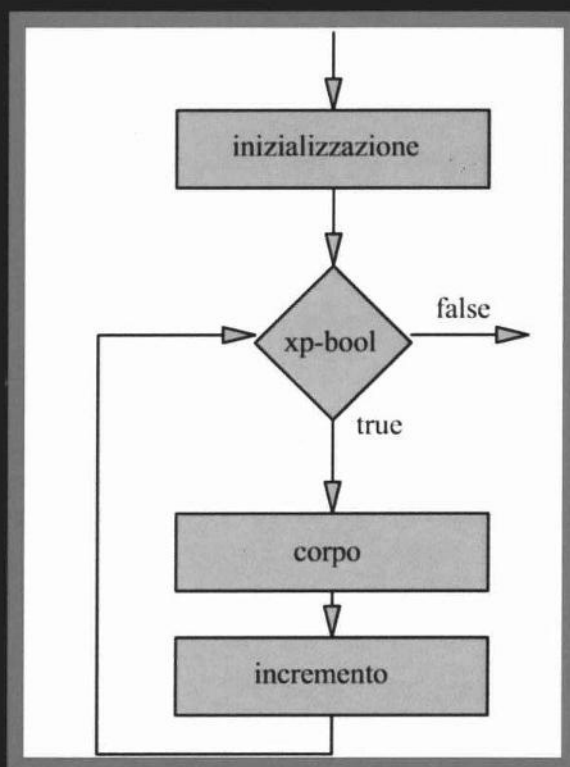


Gilberto Filè

Programmazione consapevole (semplice è bello)

quarta edizione



EDIZIONI LIBRERIA PROGETTO PADOVA

Gilberto Filè

Programmazione consapevole (semplice è bello)

2015-2016

quarta edizione



EDIZIONI LIBRERIA PROGETTO PADOVA

Indice

1	Introduzione	1
1.1	Struttura di un computer	2
1.2	Come scrivere ed eseguire un programma	3
1.3	Notazione	6
2	Tipi predefiniti e variabili	7
2.1	Tipi predefiniti	7
2.2	Variabili e dichiarazioni	10
2.3	Espressioni e Conversioni	11
2.4	A cosa servono i tipi	13
3	Istruzioni di base	15
3.1	Input/Output	15
3.1.1	I file	16
3.1.2	Collegamento tra programma e file	17
3.1.3	Operazioni di i/o	18
3.2	Assegnazione	21
3.3	Condizionale	22
3.4	Cicli while	22
3.5	Esempi	23
3.6	Sulla visibilità delle variabili	30
4	La Correttezza dei Programmi	35
4.1	Regola di prova del while	37
4.2	Altri esempi di prove di correttezza	42
4.2.1	Esempio di gestione delle eccezioni	42
5	Puntatori, array ed aritmetica dei puntatori	55
5.1	Puntatori e riferimenti	55
5.2	Array e for	59
5.3	Il tipo degli array	65
5.4	L'aritmetica dei puntatori	67
5.5	Stringhe alla C e array di caratteri	69

6	Alla Ricerca dell'Invariante Perduto	71
7	Funzioni	79
7.1	Funzioni	79
7.2	Passaggio dei parametri	83
7.3	Funzioni ed arrays	89
7.4	Variabili globali	95
7.5	Funzioni e valori restituiti	96
7.6	Esercizi commentati sulle funzioni	100
7.7	Esercizi proposti	102
8	Esercizi sulle funzioni	107
8.1	Problemi (1) e (2)	107
8.1.1	Soluzione del Problema (1)	108
8.1.2	Soluzione del Problema (2)	112
8.2	Problemi (3) e (4)	115
8.2.1	Soluzione del Problema (3)	116
8.2.2	Soluzione del Problema (4)	118
8.3	Problema (5)	120
8.3.1	Soluzione del Problema (5)	120
9	Estensioni del C++	125
9.1	Costanti	125
9.2	Tipi definiti dall'utente	126
9.3	Nuovi comandi di controllo	129
9.4	Conversioni ed operatori di cast	132
9.5	Gestione delle eccezioni	136
9.6	Contenitori del C++	139
9.7	Compilazione separata e namespace	143
9.8	L'overloading delle funzioni	147
9.9	Makefile	152
10	Ricorsione	155
10.1	Programmazione Ricorsiva	155
10.2	Esempi di Ricorsione	158
10.2.1	Le prove induttive	160
11	Gestione dinamica dei dati e liste	171
11.1	Gestione dinamica dei dati	171
11.2	Liste concatenate	173
11.3	Ricorsione ed iterazione	182

12 Alberi Binari	185
12.1 Nozioni di base sugli alberi binari	185
12.2 Esercizi sugli alberi binari	188
12.3 Alberi binari di ricerca	201
12.4 Un esercizio da esame	204

Prefazione

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

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

Capitolo 1

Introduzione

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

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

Il linguaggio di programmazione adottato nel testo è il C++ senza le classi e gli oggetti. Non si tratta del C, infatti anche senza le classi e gli oggetti, il C++ ha alcune caratteristiche che lo rendono più espressivo del suo predecessore C, come per esempio il passaggio dei parametri per riferimento, e la gestione delle eccezioni. Per non ripetere continuamente *il C++ senza le classi e gli oggetti*, nel seguito

chiameremo questo linguaggio semplicemente C++. Per quanto riguarda caratteristiche del C++ che non hanno trovato posto in questo libro, esistono molti manuali di riferimento del linguaggio, alcuni dei quali sono disponibili gratuitamente su internet. Anche il sito (www.cplusplus.com) è ricco di informazioni.

Il libro è organizzato in 12 Capitoli. I Capitoli 2, 3, 5, 7, e 9 trattano il linguaggio di programmazione. Ovviamente nei primi Capitoli sono illustrate le cose più semplici come i tipi predefiniti e le istruzioni di base, per passare successivamente ai tipi definiti dall'utente e ai contenitori, per finire con il sovraccaricamento e i namespace. Questi Capitoli sono intercalati da Capitoli con molti esempi ognuno corredato dalla relativa dimostrazione di correttezza. Il Capitolo 4 introduce le prime dimostrazioni di correttezza, ma il Capitolo 6 riveste un ruolo particolarmente importante in quanto in esso viene formulata una semplice *ricetta*, detta *ricetta di indicizzazione*, che è molto utile in tutte le dimostrazioni successive. La ricorsione è introdotta nel Capitolo 10 e approfondita nei successivi Capitoli 11 e 12 che illustrano le liste concatenate e gli alberi binari. Questi Capitoli contengono molti esempi di funzioni ricorsive la cui correttezza è dimostrata con l'induzione.

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

1.1 Struttura di un computer

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

Un computer contiene un'unità di calcolo ed una memoria. L'unità di calcolo è detta **Central Processing Unit** (CPU) e consiste di registri capaci di contenere i valori da manipolare durante l'elaborazione e da circuiti in grado di effettuare operazioni (per esempio, la somma o la sottrazione) sui valori contenuti nei registri e anche di trasferire valori dalla memoria ai registri e viceversa. Le operazioni che questi circuiti sono in grado di effettuare sono dette **operazioni macchina**. La memoria è costituita da una sequenza di byte, ciascuno composto da 8 bit e identificato da un indirizzo. Gli indirizzi dei byte della memoria iniziano da 0 e crescono fino alla dimensione totale della memoria stessa. Un valore, per esempio un valore intero, può occupare più byte contigui e in questo caso diremo che il valore risiede nella memoria all'indirizzo del primo di questi byte. Abbiamo già detto che la

CPU è in grado di eseguire operazioni che trasportano valori dalla memoria RAM ad un registro della CPU e viceversa. Queste operazioni richiedono di specificare l'indirizzo del primo byte da cui si preleva (o in cui viene copiato) il valore trasportato. Dato che tramite gli indirizzi è possibile accedere ad un qualsiasi byte della memoria, questa memoria è chiamata **Random Access Memory (RAM)**, cioè Memoria ad Accesso Casuale. Per quanto riguarda l'esecuzione dei programmi da parte di un computer, essa avviene in linea di massima nel modo seguente: il programma da eseguire ed i valori che esso manipola durante l'esecuzione, si trovano entrambi nella RAM (ma in parti diverse). Uno dei registri della CPU, detto **Program Counter (PC)** contiene in ogni momento l'indirizzo della prossima istruzione del programma da eseguire e la CPU *riconosce* quell'istruzione, la esegue e passa alla istruzione successiva incrementando il PC. L'esecuzione di un'istruzione in generale modifica i valori dei dati del programma.

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

1.2 Come scrivere ed eseguire un programma

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

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

Per eseguire i programmi presentati in questo libro, basterà avere sul proprio computer un compilatore C++ possibilmente conforme al C++ standard. Consigliamo il compilatore GNU C++ 4.1 (e qualunque versione successiva). Si tratta di software libero che è facile trovare e scaricare dalla rete sul proprio computer (gcc.gnu.org). Questo compilatore è realizzato per funzionare con il sistema operativo Linux, ma per i sistemi operativi Microsoft è possibile scaricare l'ambiente **Cygwin** che simula un ambiente Linux e possiede al suo interno un compilatore C++ della GNU. Anche Cygwin è software libero (Cygwin.com). Nel seguito as-

¹In 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.

sumeremo che i nostri lettori siano in grado di aprire una finestra di comando (detta **shell**), da cui potranno spostarsi nelle diverse cartelle del file system che contengono i loro programmi e da cui potranno lanciare i comandi di compilazione e di esecuzione dei programmi, che descriveremo dopo il seguente esempio.

Esempio 1.1

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

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

Nonostante la sua semplicità, ci sono parecchie cose da spiegare nel programma dell'Esempio 1.1:

1. la prima istruzione `#include<iostream>` non è un'istruzione C++ bensì un comando, rivolto al compilatore, che richiede l'inclusione della classe di

input/output `iostream` nel namespace `std`. La successiva istruzione `using namespace std;` permette al nostro programma di accedere a tutto quello che è definito nel namespace `std` e quindi anche alla classe `iostream`. Usare questa classe è necessario perché essa contiene le definizioni delle operazioni di input e di output che usiamo nel nostro programma e quindi se la classe `iostream` non fosse presente le operazioni di input e di output risulterebbero indefinite. Tutto questo sembrerà certamente *ostrogoto* al neofita. Per il momento si tratta di usare queste istruzioni senza capirle. La successiva trattazione dovrebbe renderle più comprensibili.

2. `main()` è il nome di una **funzione** e quello che segue, racchiuso tra parentesi graffe, è il **corpo** della funzione, cioè le istruzioni che la compongono. Nel seguito una sequenza di istruzioni racchiuse tra graffe viene chiamato un **blocco**. In inglese `main` significa *principale*. Ogni programma deve contenere esattamente una funzione `main` e l'esecuzione di ogni programma inizia sempre dalla prima istruzione della funzione `main`. Le parentesi tonde `()` che seguono il nome `main` indicano che si tratta di una funzione e che (in questo caso) non ha argomenti. In realtà il C++ consente di passare parametri al `main`, ma è decisamente troppo presto per questo genere di dettagli tecnici. Rinviamo il lettore inguaribilmente curioso al sito www.cplusplus.com per maggiori informazioni su questo.
3. Nel C++ l'input e l'output sono visti come sequenze di byte (come la memoria RAM). In inglese sequenza si dice "stream" ed infatti i nomi legati all'input ed all'output sono variabili di tipo `stream`. L'operazione di output più comune è `<<`. Il dispositivo standard di output è il video ed il nome dello stream che gli viene associato nella libreria `iostream` è `cout`. Per l'input, l'operazione base è `>>`, il dispositivo standard di input è la tastiera ed il nome dello stream associato è `cin`.
4. Nel programma che stiamo considerando usiamo sia l'operazione di scrittura `<<` che quella di lettura `>>`. Queste operazioni hanno entrambe 2 argomenti: lo stream `cout` o `cin` ed il valore da mandare in output oppure, rispettivamente, la variabile a cui assegnare il valore letto da `cin`. Un esempio di un comando di output è la seconda riga del corpo del `main`: questa istruzione (quando viene eseguita) produce la stampa su video del valore stringa di caratteri `"inserire 2 interi"`. Nella terza riga del `main` troviamo invece un esempio di un'istruzione di lettura. Questa istruzione va vista in realtà come `(cin >> x) >> y;` in cui la prima parte `(cin >> x)` legge il prossimo valore dallo stream di input `cin`, lo assegna alla variabile `x` e restituisce `cin` in modo da consentire la seconda lettura, `(cin >> y)` che assegna il prossimo valore letto a `y` e restituisce `cin` che però non serve a nulla dato che le letture sono finite. Per finire, la quarta istruzione del corpo del `main` è un output che stampa di seguito le strin-

ghe ``valore di  $x$ `` e ``valore di  $y$ ``, ciascuna seguita dal corrispondente valore (appena letto) di x e di y .

1.3 Notazione

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

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

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

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

Avremo bisogno di considerare insiemi S di oggetti e di affermare che tutti gli elementi di S soddisfano una certa proprietà $P(\cdot)$. Questa affermazione è descritta dalla seguente formula: $\forall a \in S, P(a)$. Invece $\exists a \in S, P(a)$ significa che esiste (almeno) un $a \in S$ tale che valga $P(a)$. Vale la pena di osservare che qualora $S = \emptyset$, allora $\forall a \in S, P(a)$ è vero perché richiedere che per tutti gli elementi di S (cioè nessuno) valga $P(\cdot)$ è banalmente vero. Al contrario, se $S = \emptyset$, allora $\exists a \in S, P(a)$ è, altrettanto banalmente, falso perché essa richiede che ci sia qualche elemento $a \in S$ per cui $P(a)$ sia vero.

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

Capitolo 2

Tipi predefiniti e variabili

Come (quasi) tutti i linguaggi di programmazione, anche il C++ 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. Introduciamo anche le **variabili**, che in C++ devono sempre avere un tipo, ed inizieremo ad illustrare la nozione di **visibilità o scope** delle variabili stesse. Chiuderemo il capitolo illustrando l'importanza che i tipi hanno per l'individuazione degli errori che sono spesso presenti nei programmi che vengono scritti.

2.1 Tipi predefiniti

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

- Per il tipo intero, l'intervallo dei valori rappresentabili con 4 byte (dal compilatore GNU 4.1) è, $-2^{31}, \dots, 2^{31} - 1$; 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

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

di maggiore $>$, il test di maggiore o uguale $>=$, eccetera. L'ambiente C++ fornisce 2 costanti `INT_MAX` e `INT_MIN` che hanno come valore, rispettivamente, il massimo ed il minimo intero rappresentabile. Useremo spesso queste costanti per inizializzare variabili intere con un valore che sia sicuramente il più grande o il più piccolo possibile. Visto che gli interi rappresentati nel computer sono finiti, è naturale chiedersi cosa succede se sommando o moltiplicando 2 interi si esce dall'intervallo dei valori rappresentabili. In questo caso diremo che si è verificato un errore di **overflow** ed il risultato dell'operazione è privo di senso. Purtroppo l'overflow non viene, in generale, segnalato automaticamente dal computer.

- Per i reali il C++ offre 2 tipi: `double` e `float`. Il tipo `double` che occupa 8 byte offre maggiore precisione del `float` per cui si usano solo 4 byte. Una costante reale come 12.3 viene considerata dal compilatore di tipo `double`. È necessario aggiungere `f` alla fine per renderla `float`: 12.3f. La scrittura 2.3e4 indica il numero `double` $2.3 \cdot 10^4$.
- Le operazioni aritmetiche e quelle di confronto si possono applicare sia a valori `int`, che `float`, che `double`. Chiariamo immediatamente che non si tratta delle stesse operazioni, ma solo di simboli uguali per indicare operazioni diverse. La diversità è causata dalla diversa rappresentazione interna degli interi (complemento a 2) rispetto a quella dei reali (floating point con 4 o 8 byte). Questo fenomeno di indicare con lo stesso simbolo operazioni simili concettualmente, ma che sono diverse a causa del tipo degli operandi, si chiama **sovraccaricamento o overloading**. Vedremo nel seguito del capitolo come il compilatore decide qual'è l'operazione che deve venire effettivamente usata a fronte di un'operazione sovraccaricata nel programma che sta traducendo.
- L'insieme di valori del tipo `char` è costituito da 256 caratteri. Il compilatore C++ 4.1 della GNU usa un byte per rappresentare questi 256 valori ed infatti con otto bit si rappresentano 256 valori diversi che visti come interi in

complemento a 2 sono gli interi da -128 a 127. Quindi ogni carattere è rappresentato nella RAM da uno di questi interi. I caratteri con codifica da 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, come per esempio in, 'v', '?', e '!', cf. la Tabella 2.1. Ci sono caratteri che non sono visibili sullo schermo, ma hanno la funzione di spostare il cursore, per esempio, il carattere '\n' serve per spostare il cursore all'inizio della prossima riga del video, mentre '\t' muove il cursore al prossimo punto di tabulazione. Con la crescente importanza di paesi che adottano alfabeti diversi dal nostro, sono nati tipi in grado di rappresentare insiemi di caratteri più ampi di 256. A questo scopo il C++ prevede il tipo `wchar_t` i cui valori occupano 2 o 4 byte.

- Il tipo booleano, denotato `bool`, consiste solamente di 2 valori, `true` e `false`. Ai valori di tipo `bool` si applicano gli operatori booleani che sono, la congiunzione (AND), rappresentata con `&&`, la disgiunzione (OR), rappresentata con `||` e la negazione (NOT), rappresentata con `!`. I valori booleani sono rappresentati internamente al computer con gli interi 0 (`false`) e 1 (`true`). Il fatto che i valori del tipo `bool` siano rappresentati 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, il fatto che i valori booleani siano rappresentati internamente con interi, permette di applicare ai valori booleani tutte le operazioni applicabili agli interi. Per esempio, `true - true == false` è un'espressione corretta del C++ ed ha valore `true` (con rappresentazione interna 1). Visto che i valori booleani sono in realtà interi, anche la seguente espressione, `2 && 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 7.3.

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

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

2.2 Variabili e dichiarazioni

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

Le variabili vengono introdotte nei programmi C++ attraverso le operazioni di **dichiarazione** che specificano il tipo delle variabili dichiarate. Per esempio, la dichiarazione `int x, pippo;` specifica che `x` e `pippo` sono variabili di tipo `int` (o semplicemente sono variabili intere). In generale, se il programma è corretto, una tale dichiarazione garantisce che i valori che, durante l'esecuzione del programma, verranno associati alle due variabili `x` e `pippo` saranno di tipo `int`. Al momento della dichiarazione di una variabile, è anche possibile (non necessario) assegnare un valore alla variabile stessa, come per esempio in: `int x=0,`

`pippo=-1;` In questo caso parliamo di **inizializzazione** delle variabili. Una variabile dichiarata, ma non inizializzata, è **indefinita**. Esempi di dichiarazioni di variabili di tipo `char` sono: `char y1, z;` e `char y1='a', z;` in cui `y1` viene inizializzata al carattere 'a'. I nomi delle variabili devono in ogni caso essere diversi dalle parole chiave (keyword) del C++ come, per esempio, `int` e `char` e altre che saranno introdotte nel seguito.

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

2.3 Espressioni e Conversioni

Un'espressione è composta nel caso più semplice da costanti, per esempio, `2*(5+24)`, il cui valore è 58. Espressioni possono contenere variabili, per esempio, `x*(pippo+24)`, il valore di questa espressione lo si calcola sostituendo ad ogni variabile il suo R-valore. Supponiamo che `x` abbia R-valore 5 e `pippo` abbia R-valore 2, in questo caso l'espressione ha valore 130. Sottolineiamo un fatto semplice, ma importante: le espressioni nei programmi vengono sempre valutate e questo procedimento (se ha successo) produce il **valore** dell'espressione. È possibile che la valutazione di un'espressione non abbia successo? La risposta è affermativa e il fallimento della valutazione può avvenire per vari motivi. La ragione più semplice è che l'espressione richieda di eseguire operazioni indefinite come la divisione per 0. Questo errore produce generalmente una terminazione anormale del programma in esecuzione. Un altro possibile motivo che può impedire 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 perché le operazioni macchina si applicano su valori dello stesso tipo. Per esempio, i computer possono eseguire l'operazione di somma di 2 operandi interi e anche la somma di 2 operandi reali, ma non la somma tra un intero ed un reale. Quindi cosa fa il compilatore se deve tradurre un'espressione come, $2 + 3.14$? Ovviamente ci sono 2 possibilità: o trasformare l'intero 2 in un reale (scrivendo 2 in forma floating point) oppure trasformare il reale 3.14 in un intero, per esempio troncando la parte decimale riducendo 3.14 a 3. Il compilatore C++ sceglie sempre la prima possibilità. Il motivo è semplice e logico: un intero occupa meno byte (4) di un double (8 byte) e quindi la conversione dell'intero in double non comporta mai una perdita di informazione: ogni intero tra -2^{31} e $2^{31} - 1$ ha una rappresentazione precisa in floating point con 8 byte. 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 che soddisfano questo principio, vengono chiamate **promozioni**. In Sezione 9.4 si possono trovare maggiori dettagli sulle promozioni e in generale su tutti i tipi di conversione e su cosa comporti in pratica convertire un valore da un tipo ad un altro tipo.

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

Per le espressioni booleane che contengono gli operatori logici $\&\&$ e $||$, il C++ usa la cosiddetta valutazione **abbreviata** (o **shortcut**). Consideriamo l'espressione esaminata prima: $(x >= 'a') \&\& (x <= 'z')$. La valutazione procede da sinistra a destra, ma se la condizione di sinistra $x >= 'a'$ è falsa allora non viene valutata la condizione di destra. Infatti l'intera espressione è falsa, indipendentemente

dal valore della condizione di destra. Nel caso di un'espressione che contiene l'operatore `||`, come per esempio, `(x < 'a') || (x > 'z')`, se la condizione di sinistra è vera non viene valutata quella di destra perché l'intera espressione è vera indipendentemente dal valore della condizione di destra.

2.4 A cosa servono i tipi

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

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

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

assegnano ad una variabile di un certo tipo degli R-valori di tipo diverso (e incompatibile) senza che queste violazioni vengano segnalate né durante la compilazione né durante l'esecuzione.

Capitolo 3

Istruzioni di base

Questo Capitolo è dedicato alla descrizione delle istruzioni di base del C e C++. Chiariamo subito che queste istruzioni si trovano in qualsiasi linguaggio di programmazione della grande famiglia dei linguaggi imperativi¹, come per esempio Fortran, Pascal, ma anche quelli orientati agli oggetti come C++, Java eccetera. Tratteremo in primo luogo le istruzioni di input e di output e successivamente esamineremo l'assegnazione, il comando condizionale ed il comando iterativo `while`.

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

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

3.1 Input/Output

I programmi devono poter scambiare informazioni con l'esterno ed a questo servono le operazioni di lettura e stampa, dette di input/output e abbreviate in i/o. Ci limiteremo a spiegare le operazioni più semplici che poi useremo sempre nel seguito del testo. Operazioni di i/o maggiormente sofisticate sono descritte in

¹Imperativi sono detti quei linguaggi di programmazione che possiedono un'istruzione di assegnazione, vedi la Sezione 3.2.

www.cplusplus.com. Un programma comunica con l'esterno per mezzo di dispositivi molto diversi: in un computer moderno, l'input standard è la tastiera, l'output standard è lo schermo, ma un programma potrebbe voler scrivere o leggere da un dispositivo USB o da CD o DVD o semplicemente da un file nella memoria del computer o anche un file che risiede nella memoria di qualche altro computer raggiungibile 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 un **file**. Vediamo innanzitutto cos'è un file, successivamente ci occuperemo di come un programma acceda ad un file e delle principali operazioni di i/o disponibili.

3.1.1 I file

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

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

Facciamo un semplice esempio per chiarire la differenza tra file di testo e file binari. Consideriamo il valore intero 8. In un file di testo il valore 8 viene rappresentato da un solo byte che contiene la codifica ASCII estesa del carattere '8', cioè 56 (per la precisione il valore binario 00111000), mentre in un file binario il valore 8 è rappresentato da 4 byte come ogni valore intero, cf. Tabella 2.1: i primi 3 composti da tutti 0, mentre il quarto conterrebbe: 00001000, cioè la codifica binaria di 8. Il valore -8 verrebbe rappresentato su un file di testo da 2 byte (uno che contiene la codifica ASCII estesa di '-' e l'altro quella di '8'), mentre in un file binario verrebbe rappresentato ancora da 4 byte che rappresentano in binario il valore 4294967288, che è la rappresentazione in complemento a 2 dell'intero -8, cioè $2^{32} - 8$.

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

3.1.2 Collegamento tra programma e file

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

- i) all'inizio del programma va inserita la direttiva `#include<fstream>`,
- ii) inserendo, per esempio nel `main`, la dichiarazione,

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

si crea lo *stream* `XX` che viene associato al file `pippo`, il quale viene simultaneamente **aperto** per eseguire input, cioè i suoi dati sono ora a disposizione del nostro programma per operazioni sequenziali di lettura;
- iii) per aprire il file `minni` in output, cioè per scriverci sopra, si deve inserire nel programma la dichiarazione: `ofstream YY('minni');` che apre il file `minni`.

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

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

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

3.1.3 Operazioni di i/o

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

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

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

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

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

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

' ' '3' '4', allora verrebbe calcolata la rappresentazione floating point su 8 byte di 12 ed assegnata come R-valore a x .

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

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

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

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

Sull'input/output ci sarebbero molte altre cose da dire. Per esempio gli stream di input ci offrono anche altre operazioni di input come la `getline` che legge i

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

3.2 Assegnazione

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

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

Il C++ consente di scrivere assegnazioni in forme abbreviate come, `potenza* = 2;` e `esponente++;`. Esse corrispondono alle seguenti assegnazioni normali:
`potenza* = 2;` $\Rightarrow$ `potenza = potenza*2;`
`esponente++;` $\Rightarrow$ `esponente = esponente + 1;`

Lo stile abbreviato di queste assegnazioni proviene dal C ed aveva originalmente il duplice scopo di permettere di scrivere codice conciso e anche di ottenere codice oggetto più efficiente. In realtà i compilatori moderni non hanno bisogno delle forme abbreviate per ottimizzare il codice che producono e quindi la forma abbreviata è ora motivata solo dalla ricerca della concisione del codice e forse dall'abitudine. Nel seguito useremo spesso gli operatori `++` e `--` sia in forma postfissa che prefissa, come per esempio in `esponente++;` e `++esponente;`. Entrambi questi comandi effettuano l'assegnazione `esponente = esponente + 1;` e non c'è alcuna differenza tra i 2 comandi se essi sono usati da soli, mentre c'è differenza

se essi vengono usati all'interno di un'espressione più complessa. Consideriamo un semplice esempio: confrontiamo l'effetto di eseguire `int x=1; x=(x++)*2;` con quello di eseguire `int x=1; x=(++x)*2;`. Nel primo caso, dopo l'esecuzione, il valore di `x` è 3, mentre nel secondo caso il valore di `x` diventa 4. Il motivo è che nel primo caso l'incremento richiesto da `x++`, viene eseguito dopo l'assegnazione `x = 1*2` e quindi il valore finale di `x` è 3, mentre nel secondo caso l'incremento richiesto da `++x` viene fatto immediatamente e quindi l'assegnazione diventa `x = 2*2`.

3.3 Condizionale

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

3.4 Cicli while

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

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

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

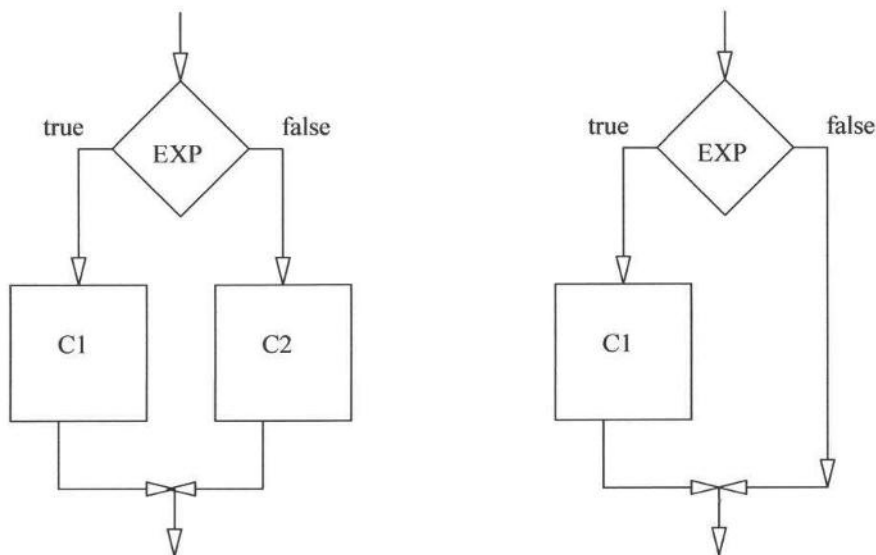


Figura 3.1: schema a blocchi dell'if-then-else e dell'if-then

che segue immediatamente il corpo del while. Lo stato del calcolo alla fine del ciclo sarà S'' .

Può capitare che EXP non diventi mai false. In questo caso (a meno di istruzioni di salto contenute nel corpo) l'esecuzione non esce mai dal 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.

3.5 Esempi

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

Esercizio Risolto 3.2 *Dopo aver letto 2 valori dallo stream di input cin ed averli assegnati alle variabili x e y , vogliamo sommare i due valori ed assegnare questo valore ad una nuova variabile intera SOM e per ultimo, usando l'istruzione condizionale, se il valore di SOM è maggiore di 0 vogliamo assegnare il valore di SOM ad y , altrimenti, (cioè se $SOM \leq 0$), vogliamo assegnare SOM a x . Il programma che compie queste azioni segue. Esso ci mostra alcune cose interessanti. Per esempio che le dichiarazioni possono essere in qualsiasi punto di un blocco, cf. la dichiarazione di SOM . Inoltre il programma illustra l'uso del condizionale che ci*

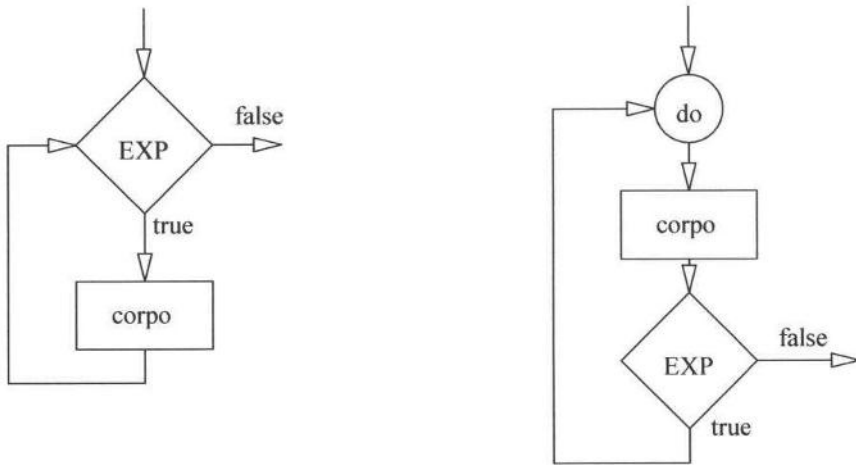


Figura 3.2: schema a blocchi del while e del do-while

consente di fare cose diverse a seconda dello stato del calcolo in cui ci troviamo. Per esempio assumiamo che i valori letti per x e y siano 2 e -4. Allora SOM assume il valore $2-4=-2$ e quindi la condizione $SOM>0$ in questo stato del calcolo è falsa e quindi viene eseguito il ramo else del condizionale cioè l'assegnazione $x=SOM$. Per cui lo stato finale S è come segue: $S(x)=-2$, $S(y)=-4$ e $S(SOM)=-2$. Se invece di 2 e -4 dallo stream di input vengono letti i valori 5 e -4 le cose sarebbero andate diversamente? In che modo?

```

#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;
}

```

Esercizio Risolto 3.3 Consideriamo ora di voler leggere degli interi dall'input standard fino a che non si legge il valore 0 e dei valori letti si vuole calcolare

quantità sono positivi e quanti negativi (quindi lo 0 non viene contato). Benché l'esercizio sia semplice proponiamo due diverse soluzioni, una che usa un ciclo while ed una seconda con un do-while, vedi Figura 3.2. La prima soluzione è la seguente:

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

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

```
#include<iostream>
using namespace std;
main()
{
    int pos=0, neg=0, x;
    do
    {
        cin>>x;
        if(x < 0)
            neg=neg+1;
        else
            if(x>0)
```

```

        pos=pos+1;
    }
    while(x!=0);
    cout<<"neg="<< neg <<" pos="<<pos<<endl;
}

```

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

Esercizio Risolto 3.4 Si vuole realizzare un programma che legge 10 interi dall'input standard, memorizza il minimo e il massimo tra i valori letti e alla fine li stampa sull'output standard. Il programma deve avere un ciclo `while` che esegue 10 volte e che dopo aver letto il prossimo intero lo confronta con il massimo ed il minimo tra i valori letti in precedenza e li modifica a seconda dei casi. Chiamiamo le due variabili i cui R-valori sono, rispettivamente, il massimo ed il minimo tra i valori letti in precedenza, max e min . Sembra semplice, ma c'è un problema. Alla prima esecuzione del `while`, non abbiamo ancora letto alcun valore e quindi che valore dovranno avere max e min ? Questo problema si presenterà spesso in altri esercizi proposti nel seguito. Lo affronteremo in modo completo subito. Possiamo assegnare a max e min dei valori ad hoc che fanno funzionare le cose, oppure leggere il primo valore (sappiamo che c'è visto che sono previste 10 letture) e assegnarlo sia a max che a min e dopo leggere gli altri 9 valori. Pensiamo prima alla soluzione con i valori ad hoc. Per max abbiamo bisogno di un valore che sarà certamente minore o uguale di ogni valore intero che potremo leggere. Questo valore è ovviamente `INT_MIN`: il minimo intero rappresentabile. Analogamente, per min , il valore giusto è `INT_MAX`. Vediamo un programma che segue questa idea.

```

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

```

```

        max=x;
    else
        if (min>x)
            min=x;
        i=i+1;
    }
    cout<<"valore massimo è "<< max
        <<" il valore minimo è "<< min;
}

```

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

```

// SOLUZIONE 1
#include<iostream>
using namespace std;
main()
{
    int x, i=0, max=INT_MIN, min=INT_MAX;
    while(i<10)
    {
        cout << "inserire il prossimo intero ";
        cin >>x;
        if(x > max)
        {
            max=x;
            if(i==0)
                min=x;
        }
        else
            if(min>x)
            {
                min=x;
                if(i==0) //($) questo commento e' usato nel
                    max=x; //($) primo esercizio del prossimo Capitolo
            }
    }
}

```

```

    i=i+1;
}
cout<<"valore massimo è "<< max
    <<" il valore minimo è "<< min;
}

```

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

```

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

```

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

Esercizio Risolto 3.5 *Questo esercizio ha lo scopo di illustrare e confrontare alcune operazioni di i/o viste nella Sezione 3.1.3. L'Esempio è strettamente collegato ai due esercizi seguenti che andrebbero infatti considerati immediatamente dopo questo. Si tratta di scrivere un programma che apre come input un file di testo pip-pip.cpp e legge il suo contenuto carattere per carattere ricopiando i caratteri letti*

sul file di output standard (il video, associato allo stream `cout`). La coppia di operazioni di lettura e stampa è ripetuta grazie all'uso del comando iterativo `while` la cui condizione di permanenza è `! PP.eof()`, cioè le due operazioni sono ripetute fino a che l'end of file di `PP` non è raggiunto.

```
#include<iostream>
using namespace std;
#include<fstream>
main()
{
    ifstream PP("pippo.cpp");
    if(PP !=0) // se l'apertura ha avuto successo
    {
        while (! PP.eof())
        {
            char c;
            c=PP.get();
            cout << c;
        }
        PP.close();
    }
    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, carattere per carattere, il contenuto del file e stampa sul video ciascun carattere immediatamente dopo averlo letto. Cambiando opportunamente il nome `pippo.cpp` è possibile usare questo programma per copiare a video ogni file (di testo).

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

Esercizio 3.7 Si chiede di modificare il programma dell'Esercizio precedente, sostituendo l'operazione di lettura con la seguente: `PP >> c;`. Sia ancora `pippo.cpp` il file che contiene il programma modificato. L'esecuzione di questo nuovo

programma stampa sul video tutti i caratteri del file pippo.cpp omettendo gli spazi e gli accapo. In sostanza il contenuto del file viene scritto tutto di seguito. Questo strano comportamento è dovuto a quanto detto in precedenza sull'operazione >>: essa non legge i caratteri di separazione (spazio e accapo), ma li usa solo per separare le stringhe di caratteri buoni. Invece l'operazione get legge tutti i caratteri, anche quelli di separazione. Va notato anche che ora nessun carattere strano conclude la stampa. Il motivo è sempre lo stesso: l'operazione >> salta i separatori e quindi, dopo la lettura dell'ultimo carattere buono, salta qualunque carattere di separazione lo segua e attiva subito l'end of file.

3.6 Sulla visibilità delle variabili

È utile capire cosa succede alle variabili ed ai loro valori quando un programma viene eseguito. Abbiamo già visto che il programma sorgente viene tradotto dal compilatore in un equivalente programma oggetto che consiste di istruzioni macchina. Quando il programma oggetto viene eseguito, esso deve risiedere nella memoria RAM del computer ed un'altra area della RAM contiene i valori delle variabili del programma. Eseguire un programma significa che in ogni momento una particolare istruzione del programma oggetto viene eseguita e che un particolare sottoinsieme delle variabili del programma sono **attive**. Quindi la zona di memoria RAM che contiene i valori delle variabili del programma non è statica, ma al contrario essa cambia sia nella sua dimensione sia nei valori che contiene perché in ogni momento cambia l'insieme delle variabili attive. Cerchiamo di capire quali sono le variabili attive in ogni momento dell'esecuzione e come vengono gestite. In generale, un programma è costituito da blocchi annidati gli uni negli altri, dove ogni blocco può contenere dichiarazioni di variabili (oltre a istruzioni). Ecco un main che consiste di 3 blocchi annidati con dichiarazioni e operazioni di output:

```
main()
// (*)
{
    int x=0; //... (0)
    {
        int y=1; //... (1)
        {
            int z=2; //... (2)
        }
        cout<<x<<y<<endl; // (1')
    }
    cout<<x<<endl; // (0')
}
// (*')
```

La pila dei dati cresce e diminuisce dinamicamente

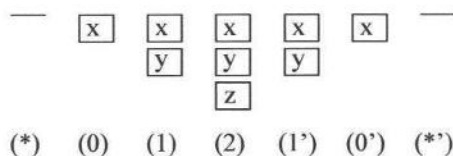


Figura 3.3: Allocations e deallocations della memoria durante l'esecuzione di un programma

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

Da questa descrizione dovrebbe essere facile vedere che la gestione dei dati manipolati da un programma segue un andamento a pila: quando l'esecuzione entra in un blocco, le variabili dichiarate nel blocco vengono allocate sulla cima della pila e vengono deallocate dalla cima della pila quando l'esecuzione esce dal blocco stesso. La Figura 3.3 illustra questa gestione a pila. Visto che l'allocazione e la deallocazione delle variabili dipende solamente dal blocco in cui le variabili sono dichiarate, esse sono dette **automatiche** per sottolineare la loro differenza con variabili i cui valori sono gestiti in un modo diverso che verrà illustrato in Sezione 11.1. Una variabile automatica è attiva nel periodo che va dalla sua allocazione alla sua deallocazione. La **visibilità** di una variabile è sempre il blocco in cui è dichiarata, dalla sua dichiarazione in poi. Va ricordato che la pila delle variabili attive è realizzata in una parte della memoria RAM.

Nell'esempio di Figura 3.3, per semplicità, abbiamo considerato una sola dichiarazione per blocco. Il caso di più dichiarazioni in un blocco viene gestito in modo analogo: lo spazio necessario per tutte le variabili dichiarate viene allocato



Figura 3.4: il fenomeno dell'oscuramento delle variabili

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

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

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

Il corpo di un `while` o il ramo `then` o `else` di un condizionale costituiscono un blocco e naturalmente questi blocchi al loro interno ne possono contenere altri

con livello di annidamento arbitrario. Inoltre ogni blocco può contenere dichiarazioni. Ogni dichiarazione appartiene ad un solo blocco e bisogna essere precisi nel definire quale sia. Il blocco di appartenenza di una dichiarazione D è il blocco che contiene D ed è tale che non ci sia alcun altro blocco, annidato in esso, che contenga D (attenzione: proprio D, non un'altra dichiarazione uguale a D). Quindi, delle dichiarazioni che appartengono ad un blocco, non fanno parte quelle che compaiono nei blocchi annidati.

La visibilità o usabilità di una variabile inizia dal punto del programma immediatamente successivo alla sua dichiarazione e si estende fino alla fine del blocco cui la dichiarazione appartiene. Eventuali blocchi annidati sono inclusi nella visibilità. Quindi le variabili dichiarate in un certo blocco sono, in linea di principio, visibili anche nei blocchi annidati.

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

Concludiamo la Sezione con un'osservazione importante sulle dichiarazioni che si trovano nel blocco costituito dal corpo di un comando iterativo. Ad ogni iterazione, l'esecuzione entra nel blocco, producendo l'allocazione delle variabili dichiarate al suo interno, giunge alla fine del blocco, producendo la deallocazione delle stesse variabili e così via per ogni iterazione. Quindi il risultato del seguente programma è la stampa di 10 per 10 volte: ad ogni iterazione si alloca una nuova x con R-valore 10 e la successiva somma non si trasmette da un'iterazione all'altra.

```
int i=0;
while(i<10)
{
    int x=10;
    cout<<x<<endl;
    x=x+10;
```

```
    i++;  
}
```

Capitolo 4

La Correttezza dei Programmi

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

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

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

- **PRE** = (`cin` contiene l'intero 0 preceduto da n interi diversi da 0, con $n \geq 0$)

- **POST** = ($\text{neg} + \text{pos} = n$ e neg è il numero degli interi negativi che precedono 0 su cin , mentre pos è il numero degli interi positivi che precedono 0)

La *precondizione* asserisce che su cin c'è certamente uno 0 e che esso è preceduto da zero o più interi diversi da 0. Questo è importante perché ci assicura che il programma terminerà sempre. La *postcondizione* invece asserisce che il programma deve calcolare in neg il numero dei negativi e in pos quello dei positivi che precedono lo 0. Ovviamente i nomi pos e neg sono arbitrari. Per dimostrare che la prima soluzione dell'Esercizio Risolto 3.3), quella che usa il `while`, è corretta, dobbiamo dimostrare che il suo ciclo `while`, iterazione dopo iterazione arriva ad attribuire a neg e pos il valore giusto rispetto ai valori letti da cin . Possiamo dire che ogni volta che stiamo per ripetere il ciclo `while`, avremo letto $\text{neg} + \text{pos} + 1$ interi da cin e i primi $\text{neg} + \text{pos}$ sono diversi da 0 e ce ne sono neg negativi e pos positivi, e inoltre, se x è 0 allora $\text{neg} + \text{pos} = n$, mentre se è diverso da 0, allora $\text{neg} + \text{pos} < n$. Cerchiamo di scrivere tutto questo in un'asserzione più sintetica e precisa:

$$R = \left\{ \begin{array}{l} (1) (\text{letti } \text{neg} + \text{pos} + 1 \text{ valori da } \text{cin} \text{ (neg e pos non negativi)}) \\ (2) (\text{i primi } \text{neg} + \text{pos} \text{ valori letti sono tutti diversi da 0} \\ \quad \text{e ce ne sono neg negativi e pos positivi}) \\ (3) (x = 0 \Rightarrow \text{neg} + \text{pos} = n) \\ (4) (x \neq 0 \Rightarrow \text{neg} + \text{pos} < n) \end{array} \right.$$

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

- 1) la prima volta che l'esecuzione arriva al ciclo (cioè prima di valutare il test di permanenza del ciclo), **R** vale: abbiamo letto un solo valore e infatti $\text{neg} + \text{pos} = 0$ (vale (1) di **R**) e se $x = 0$, allora n è 0 e quindi $\text{neg} + \text{pos} = 0$ (e quindi vale (3) di **R**), mentre se $x \neq 0$, allora $n > 0$ (e quindi vale (4) di **R**). Il punto (2) di **R** vale in modo banale.
- 2) Se **R** vale all'inizio del ciclo e il test di permanenza è vero allora dopo aver eseguito il corpo del ciclo una volta di più, vale ancora **R**. Vista l'ipotesi che il test di permanenza è vero, x non è 0 e quindi è un valore che precede 0 per cui il test aumenta correttamente o neg o pos ((2) di **R**). Poi viene letto un nuovo valore, ma visto che $\text{neg} + \text{pos}$ è aumentata di 1, $\text{neg} + \text{pos} + 1$ è ancora uguale al numero di letture (punto (1) di **R**). (3) e (4) di **R** valgono banalmente. Questo significa che **R** vale ogni volta che l'esecuzione arriva all'inizio del ciclo. Per questo motivo **R** è detto **invariante del ciclo**.
- 3) Se all'inizio del ciclo vale **R** e il test di permanenza è falso, allora vale **POST**. Che il test di permanenza sia falso significa che $x = 0$ e in questo caso i punti (2) e (3) di **R** implicano **POST**.

I tre punti appena visti formano lo schema che useremo sempre per dimostrare la correttezza dei cicli. Chiaramente il cuore della dimostrazione è l'invariante **R** che

descrive quello che il ciclo calcola dopo ogni numero di iterazioni. Ma possiamo seguire questo schema anche per dimostrare la correttezza della seconda soluzione dell'Esercizio Risolto 3.3, quella che usa il `do-while`? **PRE** e **POST** servono anche per il secondo programma, visto che descrivono il problema da risolvere e non il particolare programma che lo risolve. Certo si può osservare che **POST** usa le variabili `neg` e `pos`, ma si tratta solo di nomi per le quantità che si chiede di calcolare. Sarebbe facile riscrivere la postcondizione con altri nomi o anche senza nomi del tutto. Per gestire il `do-while` abbiamo bisogno di trovare un invariante che sia vero ogni volta che l'esecuzione arriva a valutare il test di permanenza. Possiamo usare anche qui **R**? Se ci pensiamo un po' ci accorgiamo che ora la situazione del calcolo quando arriva a valutare il test di permanenza è diversa rispetto a prima: ora l'ultimo intero letto, che è in `x`, è già stato testato e se è diverso da 0, `neg` o `pos` sono già state correttamente aumentate. L'invariante deve tenerne conto e sarà quindi come segue:

$$\mathbf{R'} = \begin{cases} (3) (x=0 \Rightarrow \text{neg}+\text{pos}=n) \text{ e letti } n+1 \text{ interi, di cui neg sono i negativi} \\ \quad \text{e pos i positivi} \\ (4) (x \neq 0 \Rightarrow \text{letti } \text{neg}+\text{pos} \leq n, \text{ di cui neg sono i negativi e pos i positivi}) \end{cases}$$

Abbiamo lasciato i numeri (3) e (4) per collegarli ai punti corrispondenti di **R**. Ma **R'** è ancora invariante? Ovviamente il corpo del ciclo viene eseguito se $x \neq 0$, e quindi vale il caso (4). Si osservi in particolare che nel caso (4) è necessario inserire che $\text{neg}+\text{pos} \leq n$. Il minore stretto sarebbe proprio sbagliato. Viene fatta una nuova lettura e se viene letto 0 allora l'esecuzione del condizionale non cambierebbe `neg` e `pos` e quindi dopo varrebbe (3), mentre se viene letto un valore diverso da 0, il condizionale aggiungerà 1 a quello giusto tra `neg` e `pos` e quindi dopo varrà (4). Insomma **R'** è invariante. Il punto (3) di **R'** vale quando la condizione di permanenza è falsa, cioè quando $x=0$ e (3) implica direttamente **POST**. Dimostrare che **R'** vale la prima volta che si arriva a valutare il test di permanenza nel `do-while` è facile, ma si deve osservare che la prima volta che il `do-while` viene eseguito, `pos` e `neg` sono entrambe 0 e che nel ciclo si esegue la prima lettura.

4.1 Regola di prova del while

Vogliamo generalizzare quanto è stato descritto nell'Esercizio Risolto 4.1 in modo da avere una regola da applicare a cicli `while` e anche `do-while` qualsiasi. La regola di verifica del `while` è la seguente: se **P** e **Q** sono asserzioni, allora per dimostrare che vale $\mathbf{P} <\text{while}(\mathbf{B}) \mathbf{C}> \mathbf{Q}$ è necessario dimostrare i seguenti 3 punti:

- (condizione iniziale) $\mathbf{P} \Rightarrow \mathbf{R}$
- (invarianza) $\mathbf{R} \ \&\& \ \mathbf{B} <\mathbf{C}> \mathbf{R}$
- (condizione d'uscita) $\mathbf{R} \ \&\& \ !\mathbf{B} \Rightarrow \mathbf{Q}$

Chiariamo bene cosa si dimostra con i 3 punti appena enunciati. Si dimostra che il ciclo `<while (B) C>` è tale che, se la sua esecuzione inizia in uno stato del calcolo che soddisfa la preconditione **P** e, se la sua esecuzione termina, esso termina in uno stato del calcolo che soddisfa la postcondizione **Q**. Uno stato del calcolo **soddisfa** un'asserzione quando l'asserzione è vera nella situazione delle variabili descritta dallo stato. Due cose vanno notate:

- Con i 3 punti appena enunciati non viene dimostrata la terminazione del ciclo sotto esame. Il ciclo potrebbe essere corretto (rispetto a pre- e postcondizioni date), ma potrebbe non terminare mai. La terminazione dei cicli e dei programmi è importante e va dimostrata con una prova ad hoc che non segue lo schema dei 3 punti. I programmi che studieremo in questo corso sono nella grande maggioranza terminanti (ed è molto facile dimostrarlo). Un programma non terminante (ma corretto) è descritto nell'Esercizio Risolto 4.9. Generalmente in questo testo ci concentreremo sulla correttezza rispetto ad una pre- e postcondizione. Senza la terminazione. Questa correttezza viene chiamata **parziale** mentre, quando è dimostrata anche la terminazione, allora si parla di correttezza **totale**. Per i due programmi dell'Esercizio Risolto 4.1 è molto facile dimostrare la terminazione: la preconditione garantisce che `cin` contiene 0 e quindi, prima o poi, `x` assumerà il valore 0 e quindi il ciclo terminerà.
- Come appena osservato nel punto precedente, lo stato del calcolo che useremo nelle dimostrazioni dei programmi estende in modo facilmente comprensibile lo stato del calcolo introdotto all'inizio del Capitolo 3. Lo stato del calcolo che usiamo nelle prove parla dell'R-valore delle variabili, ma parla anche dello stato degli stream di i/o. In questo senso, l'Esercizio Risolto 4.1 usa asserzioni che specificano cose sul contenuto di `cin` e quindi anche lo stato del calcolo deve specificare il contenuto di `cin` per poter soddisfare tali asserzioni.

Lo schema di dimostrazione con i 3 punti appena enunciato serve anche per i cicli `do-while`. L'unica cosa da tenere a mente è che l'invariante è ora associato con l'ultima istruzione del ciclo. Quindi nella dimostrazione della condizione iniziale si deve considerare la prima esecuzione del corpo. Altrimenti la prova è organizzata esattamente come quella del `while`.

Esercizio Risolto 4.2 *Riconsideriamo l'Esercizio 3.4. Per prima cosa specifichiamo la **precondizione** (PRE) e la **postcondizione** (POST). Esse sono come segue:*

- **PRE** = (`cin` contiene almeno 10 interi)
- **POST** = (`cout` contiene il massimo e il minimo dei primi 10 valori di `cin`)

Se *SOL* è una qualsiasi soluzione del problema in questione, cercheremo di dimostrare che **PRE** <*SOL*> **POST**, che a parole significa che, se prima dell'esecuzione di *SOL* vale **PRE**, allora dopo l'esecuzione di *SOL* (se termina) vale **POST**.

Consideriamo ora la prima soluzione (si ricordi che è sbagliata) esaminata nell'Esercizio 3.4. La ripetiamo qui di seguito per chiarezza:

```
main()
{
    int x, i=0, max=INT_MIN, min=INT_MAX;
    while(i<10)
    {
        cout << "inserire il prossimo intero ";
        cin >> x;
        if (x > max)
            max=x;
        else
            if (min>x)
                min=x;
        i=i+1;
    }
    cout<<"valore massimo è "<< max
        <<" il valore minimo è "<< min;
}
```

L'intuizione del programma è che ad ogni esecuzione del *while*, *max* e *min* sono il massimo ed il minimo degli *i* valori letti fino a quel momento. Questa idea è formalizzata dal seguente invariante:

R=($0 \leq i \leq 10$, letti i primi *i* valori da *cin*, *max* e *min* sono il massimo e il minimo dei valori letti).

Purtroppo **R** non è invariante in quanto esso è falso la prima volta che si arriva al *while*. Infatti, in quel caso non si è letto niente, ma *max* e *min* sono, rispettivamente, *INT_MIN* e *INT_MAX*. Per tenere conto del caso di *i*=0, si potrebbe cambiare **R** come segue:

R'=($0 \leq i \leq 10$, letti i primi *i* valori da *cin*, se *i* = 0, allora *max*=*INT_MIN* e *min*=*INT_MAX*, se *i* > 0, allora *max* e *min* sono il massimo e il minimo dei valori letti)

Purtroppo però anche **R'** non è invariante perché sarebbe falso dopo la prima lettura, perché se il valore letto in *x* è maggiore di *INT_MIN* (cosa molto probabile), allora *max* assume il valore *x* che è effettivamente il massimo dei valori letti, come richiede **R'**, ma *min* continua ad avere valore *INT_MAX* e quindi **R'** è falsa. In un caso però **R'** funziona: se venisse letto il valore *INT_MIN* allora, $x > \text{max}$ sarebbe falso e $\text{min} > x$ sarebbe vero e quindi **R'** varrebbe visto che *max* e *min*

sarebbero entrambe uguali a `INT_MIN` cioè entrambi uguali all'unico valore letto. In tutti gli altri casi però R' sarebbe falso. In effetti, il problema non è R' , ma il programma e le osservazioni appena fatte rivelano con chiarezza l'errore del programma che era già stato discusso nell'Esercizio Risolto 3.4. Vale la pena di notare che l'errore deriva da 2 intuizioni entrambe da sole ragionevoli, ma che, messe assieme, producono un programma sbagliato:

- (i) *inizializzare max e min con, rispettivamente, `INT_MIN` e `INT_MAX`, è sensato in quanto qualsiasi valore letto sarà (molto probabilmente) maggiore di `INT_MIN` (e minore di `INT_MAX`) e quindi max (min) assumeranno il valore letto e, se anche il valore letto fosse uguale a `INT_MIN` (`INT_MAX`), max (min) avrebbe il valore giusto senza bisogno di cambiarlo.*
- (ii) *quando si legge un nuovo valore che risulta maggiore di max, si deve cambiare max, ma è inutile confrontare il valore letto con min perché normalmente dovrebbe essere minore o uguale a max. È proprio qui che risiede l'incompatibilità tra le due intuizioni. L'inizializzazione di max e min rende falso che $\text{max} \geq \text{min}$.*

Passiamo ora alla prima soluzione corretta data nell'Esercizio 3.4, quella che inizia col commento *SOLUZIONE 1*. L'asserzione R' è invariante del ciclo di questo programma. Infatti, è ovvio che R' valga la prima volta che l'esecuzione arriva al test e la seconda volta (cioè quando $i=1$) R' vale, sia nel caso che il primo valore letto sia maggiore di `INT_MIN` (ramo *then*), sia che esso sia uguale a `INT_MIN` (ramo *else*), dato che in entrambi i casi max e min hanno entrambi il valore dell'unico elemento letto e quindi vale che $\text{max} \geq \text{min}$. Dal precedente punto (ii) è facile vedere che R' resta vero dopo ogni successiva lettura (cioè con $i > 1$). Queste osservazioni mostrano che effettivamente le istruzioni di *SOLUZIONE 1* marcate con (\$) sono inutili. Infatti queste operazioni possono venire eseguite solo con $i=0$ e quando viene letto come primo valore `INT_MIN`. In questo caso $\text{max}=\text{INT_MIN}$ e quindi l'assegnazione marcata con (\$) è inutile.

Visto che R' è invariante, esso è vero ad ogni ripetizione del ciclo e quindi anche l'ultima volta, cioè quando $i=10$ e l'esecuzione esce dal ciclo. A questo punto R' garantisce che la successiva stampa di max e min stampa su `cout` i 2 valori richiesti, e quindi alla fine del programma vale **POST**.

Veniamo ora alla seconda soluzione corretta presentata nell'Esercizio 3.4, quella con commento *SOLUZIONE 2*. Questo programma entra nel ciclo `while` solo dopo una lettura e dopo aver assegnato a max e a min il valore letto. Quindi R è quasi invariante per questo ciclo. L'unica cosa da cambiare è che l'intervallo di variabilità per i va cambiato in $[1..10]$ per tenere conto della prima lettura fatta fuori dal ciclo. Quindi l'invariante per questo ciclo è:

$R''=(1 \leq i \leq 10, \text{ letti i primi } i \text{ valori da } \text{cin}, \text{ max e min sono il massimo e il minimo dei valori letti})$.

Anche in questo caso R'' con $i=10$ dimostra che vale **POST**.

Nel seguito risolveremo altri esercizi mostrando formalmente che le soluzioni che proponiamo sono corrette.

Esercizio Risolto 4.3 *Vogliamo calcolare il minimo esponente intero tale che 2 elevato a quell'esponente sia maggiore o uguale ad un intero positivo letto da cin. La prima cosa da fare è formalizzare la pre- e la postcondizione del programma da realizzare. Se non sappiamo cosa vogliamo fare, come potremmo mai farlo? Le due asserzioni sono le seguenti:*

PRE = (cin contiene un intero maggiore di 0)

POST = (esponente è il minimo intero tale che $2^{\text{esponente}} \geq$ dell'intero letto da cin), asserisce che alla fine del calcolo la variabile esponente ha il valore che vogliamo calcolare.

Vogliamo quindi realizzare un programma SOL per cui valga **PRE** <SOL> **POST**. L'idea è relativamente semplice: si tratta di leggere il valore positivo da cin in una variabile del programma, diciamo X, poi è necessario provare esponenti crescenti e per ciascuno di essi si deve calcolare la corrispondente potenza del 2 fermandosi non appena questa superi o eguagli X. Si tratta quindi di calcolare potenze crescenti del 2. Per farlo useremo una variabile potenza il cui valore iniziale deve essere ovviamente $1 = 2^0$. La variabile esponente conterrà l'esponente raggiunto nel calcolo della potenza e quindi il suo valore iniziale sarà 0. Quindi SOL inizierà con le seguenti istruzioni:

```
{
1. int X, potenza=1, esponente=0;
2. cout<<'Digita un valore positivo: ';
3. cin>>X;
...
}
```

Dopo l'istruzione 3 va inserito un ciclo while che calcoli le potenze successive e con condizione di permanenza che indichi che potenza non ha ancora raggiunto X. L'invariante di un tale ciclo è il seguente:

R = ($2^{\text{esponente}-1} < X$, $\text{potenza} = 2^{\text{esponente}}$). La seconda parte di **R** dice che stiamo calcolando la prossima potenza, mentre la prima parte dice che le precedenti potenze erano più piccole di X. Il programma completo è il seguente:

```
#include<iostream>
using namespace 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++;
}
}

```

Dato **PRE** e le inizializzazioni di *potenza* e *esponente*, è facile vedere che la prima volta che l'esecuzione raggiunge l'istruzione 4 vale l'invariante **R**: 2^{-1} è certamente minore di *X* che è positivo a causa di **PRE** e inoltre, $1 = 2^0$. Quindi vale la condizione iniziale della Sezione 4.1. Passiamo all'invarianza. Chiaramente, se la condizione di permanenza $X > \text{potenza}$ è vera, il ciclo calcola la prossima potenza e aumenta di 1 l'esponente e questo, al ritorno all'istruzione 4, mantiene vero $\text{potenza} = 2^{\text{esponente}}$. La prima parte di **R** segue dal fatto che la condizione di permanenza era vera al giro precedente e cioè quando *esponente* valeva 1 meno di quanto valga ora (il corpo del ciclo ha appena aumentato *esponente* di 1).

Veniamo ora alla condizione d'uscita, vedi Sezione 4.1. L'uscita dal ciclo si ha quando la condizione di permanenza diventa falsa: $X \leq \text{potenza}$, inoltre la prima parte di **R** garantisce che questa è la più piccola potenza che supera o eguaglia *X*, quindi alla fine del programma vale **POST**.

Abbiamo quindi dimostrato **PRE** <SOL> **POST**, cioè la correttezza parziale di *SOL* (rispetto a **PRE** e **POST**), ma non sappiamo se *SOL* termina sempre o no. Per farlo dobbiamo ragionare sul fatto che *esponente* inizia da 0 e cresce e quindi prima o poi *potenza* deve raggiungere *X*.

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 delle eccezioni del C++. Si tratta di una tecnica molto utile che conviene imparare subito. Maggiori elementi sulla tecnica verranno forniti nella Sezione 9.5.

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 cosiddette 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` 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" abbia 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 viene fatta in Sezione 9.5. 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 4.4 *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 $c_1 \dots c_k$ sono caratteri numerici,*

allora $\text{NUM}(c_1 \dots c_k)$ indica il valore intero rappresentato da $c_1 \dots c_k$. Per convenzione se $k = 0$, $\text{NUM}() = 0$. Innanzitutto scriviamo la pre- e la postcondizione del programma.

PRE = (input contiene i caratteri $c_1, \dots, c_k, 'a', k \geq 0, \forall j \in [1..k], c_j$ è un carattere numerico);

POST = (scrive su output $\text{NUM}(c_1, \dots, c_k)$).

Per capire come organizzare il programma, immaginiamo di aver letto 3 caratteri numerici c_1, c_2, c_3 , di aver già calcolato $\text{NUM}(c_1, c_2, c_3)$, e di leggere il prossimo carattere c_4 . Si possono verificare 2 casi: se c_4 è numerico allora dobbiamo calcolare $\text{NUM}(c_1, c_2, c_3, c_4)$ e continuare la lettura dei prossimi caratteri. Se c_4 è 'a', allora il programma deve fermarsi e $\text{NUM}(c_1, c_2, c_3)$ è il valore cercato. Quindi dobbiamo realizzare un ciclo che abbia il seguente invariante: **R**=(sono stati letti $c_1 \dots c_n$ caratteri, $\text{num} = \text{NUM}(c_1 \dots c_{n-1})$, $q = c_n$), che prescrive di calcolare in una variabile num l'intero corrispondente ai caratteri letti meno l'ultimo che è conservato in q. La condizione di permanenza del ciclo testa se $q \neq 'a'$. In questo caso cambia num per tenere conto del carattere numerico in q e legge in q il prossimo carattere che ci deve essere perché 'a' non è stato ancora letto. Se la condizione di permanenza non è verificata e quindi $q = 'a'$, allora num contiene già il valore da calcolare e quindi è giusto uscire dal ciclo. Questo è esattamente quello che succede nel seguente programma. Si noti che la prima lettura viene necessariamente fatta prima del ciclo.

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

Consideriamo ora le 3 parti della regola del while, cf. Sezione 4.1. La condizione iniziale è facile: **R** diventa **R**=(è stato letto c_1 , $\text{num} = \text{NUM}()$, $q = c_1$) che è vera visto che $\text{num} = 0$ e $\text{NUM}() = 0$. L'invarianza di **R** la si dimostra come segue. Se il corpo viene rieseguito, significa che il test di permanenza è vero, cioè c_n è numerico e va quindi aggiunto alla sequenza di caratteri letti finora, i.e., $c_1 \dots c_{n-1}$. Aggiungere c_n alla destra della sequenza $c_1 \dots c_{n-1}$ equivale a moltiplicare per 10 $\text{NUM}(c_1, \dots, c_{n-1})$ e ad aggiungere al risultato l'intero rappresentato da c_n , cioè $q - '0'$. Questo è esattamente quello che fa la prima istruzione del corpo del while. Le altre istruzioni, leggono il prossimo carattere c_{n+1} e poi rimettono a posto l'indice con $n++$. Quindi alla fine dell'iterazione vale ancora **R**. La condizione d'uscita, **R** && ($q = 'a'$) $\Rightarrow$ **POST**, segue immediatamente dalla **PRE**.

*Per ottenere un programma completo che risolve l'esercizio 4.4 basta inserire la sequenza di istruzioni data prima nel blocco `try` dello schema di `main` descritto in Sezione 4.2.1. Si noti che la variabile `n` non serve nel programma, ma serve a rendere più comprensibile l'invariante **R**.*

Esercizio 4.5 *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”. Si richiede di scrivere un programma che risolva il problema proposto e di fornire anche le asserzioni necessarie a dimostrare la sua correttezza.*

La soluzione dell'esercizio 4.5 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 4.6 *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. Per eseguire il calcolo appena descritto, è utile l'operazione di modulo che in C++ si indica con il simbolo `%`. Per esempio, `num%2` indica il modulo di `num` rispetto a 2 e cioè il resto della divisione di `num` per 2. Oltre a realizzare un programma che risolva il problema proposto, si chiede di fornire la pre- e la post-condizione del programma e di dimostrare la correttezza del programma rispetto a queste asserzioni.*

Continuiamo con altri esercizi di difficoltà crescente.

Esercizio Risolto 4.7 *Vogliamo scrivere un programma che legga dall'input standard dei caratteri e man mano che li legge li scriva 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, per esempio, 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 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 $m_1, \dots, m_k$, 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 $m_1, \dots, m_k$). Se $m_1, \dots, m_k$ è ben-formata allora esiste sempre un prefisso di lunghezza minima $m_1, \dots, m_j$, con $j \leq k$, che soddisfa esattamente una delle 3 condizioni. Questo fatto è dovuto alla proprietà che nessuna delle condizioni (a), (b) e (c) è contenuta in una 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. Nel seguito, per brevità, chiameremo il prefisso minimo che soddisfa esattamente una delle 3 condizioni, semplicemente **prefisso minimo**. Quindi il nostro programma deve leggere (e stampare) il prefisso minimo della sequenza di caratteri presente su cin. **POST**=(cout contiene il prefisso minimo $m_1, \dots, m_j$ di $m_1, \dots, m_k$, seguito da "caso(a)", "caso(b)", "caso(c)", a seconda di quale dei 3 casi è soddisfatto da $m_1, \dots, m_j$).

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 perché è più semplice. Quindi useremo `int conta=0, contab=0, contac=0`; L'invariante di questo ciclo potrebbe essere: **R**=(letti $m_1, \dots, m_h$, con le occorrenze di 'a'=conta, quelle di 'b'=contab e quelle di 'c'=contac). Il programma segue:

```
main()
{
    int conta=0, contabb=0, contac=0;
    while(!(conta==2) && !(contab==2) &&
           !(conta==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 dimostrare che:*

R && (contaa=2 || contab=2 || (contaa=1 && contab=1 && contac≥1))
 ⇒ **POST**.

R ci dice che se $m_1, \dots, m_h$ è stato letto, allora se $\text{contaa}=2$ essa contiene 2 'a' e lo stesso per le altre 2 condizioni, ma non dimostra che $m_1, \dots, m_h$ deve essere il prefisso minimo che soddisfa una sola delle 3 condizioni. Potrebbe per esempio essere vero che $m_1, \dots, m_h$ 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≥1))**.

Inizialmente sono tutte false e quindi l'asserzione è vera. Per la prova dell'invarianza di **R**, 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 $m_1, \dots, m_k$, come vuole **POST**.

Esercizio Risolto 4.8 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 programma 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'esercizio¹: se su "input" si trova [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], 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], su "output" si deve scrivere [1 2 3 4 5 6 7 8 9 0] perché il secondo 0 è in undicesima posizione e quindi la sentinella non viene trovata. Iniziamo definendo precisamente la pre- e la postcondizione del programma richiesto che, nel seguito, chiameremo SOL.

PRE=(input contiene $b_1, \dots, b_k$, con $k \geq 2$ e ($k \geq 10 \parallel b_{k-1} = b_k = 0$);

Data una qualsiasi sequenza di interi, $s=b_1, \dots, b_k$, che soddisfa **PRE**, $\text{Prefix}(s)$ indica $b_1, \dots, b_{10}$ se $k \geq 10$ e $b_1, \dots, b_{10}$ non contiene due 0 contigui, altrimenti, se li contiene, diciamo in posizione j e $j+1$, con $1 \leq j < 10$, allora $\text{Prefix}(s)=b_1, \dots, b_{j-1}$. Si osservi che è possibile che $j=1$, nel qual caso $\text{Prefix}(s)$ è la sequenza vuota di valori. Con questa notazione possiamo esprimere la postcondizione del programma che è:

POST=(output= $\text{Prefix}(b_1, \dots, b_k)$).

Una soluzione del problema usa due variabili booleane, una (uno0) per indicare che alla lettura precedente è stato letto 0 e la seconda (due0) per indicare che è stato letto anche un secondo 0 di seguito al primo (insomma che la sentinella è stata letta). L'invariante del ciclo di lettura, che come al solito chiamiamo **R**, consiste di varie formule che essenzialmente esprimono lo scopo di queste due variabili booleane. Prima di ogni iterazione del ciclo, tutte le condizioni devono essere verificate:

1. ($0 < n \leq 10$), specifica che il programma deve al più fare 10 letture;
2. letti $b_1, \dots, b_n$;
3. ($!\text{uno0} \ \&\& \ !\text{due0}$) $\Rightarrow$ (output= $b_1, \dots, b_n$);
4. $\text{due0} \Rightarrow (b_{n-1} = b_n = 0 \ \&\& \ \text{output}=b_1, \dots, b_{n-2})$;
5. ($\text{uno0} \ \&\& \ !\text{due0}$) $\Rightarrow (b_n = 0 \ \&\& \ \text{output}=b_1, \dots, b_{n-1})$;

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 della lettura successiva. Il ciclo per cui **R** è invariante deve semplicemente fare quello che **R** specifica:

```
{
  int X, n=0;
```

¹Negli esempi, per semplicità, si usano solo interi di 1 cifra, ma i valori possono avere un qualsiasi numero di cifre.

```

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<<' ';
  }
}
}

```

Si osservi la condizione di permanenza del ciclo: correttamente essa specifica che il ciclo va eseguito solo se sono state fatte meno di 10 letture e la sentinella non è stata trovata. Non è difficile dimostrare che **R** è effettivamente invariante di questo ciclo. 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 $!(\text{!due0} \ \&\& \ n<10) = (\text{due0} \ || \ n \geq 10)$ 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): assieme a **R** implica che n=10, da cui segue che sono stati letti 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 4.9 *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 X_1 il valore double (minore di 1) da trasformare e data una sequenza $s = b_1, \dots, b_k$ di caratteri 0/1, sia $VAL(s) = b_1 * 2^{-1} + \dots + b_k * 2^{-k}$. Vogliamo quindi un programma che da X_1 produca una sequenza s di caratteri 0/1 tale che $X_1 = VAL(s)$. Questa relazione ci suggerisce la maniera di procedere per ottenere s : visto che $X_1 = VAL(s)$, allora $2 * X_1 = 2 * VAL(s)$ e ovviamente, moltiplicando per 2 il numero binario $0.s = 0.b_1 \dots b_k$, si ottiene $b_1.b_2 \dots b_k$ e l'uguaglianza di questo valore con $2 * X_1$ ci permette di conoscere b_1 ; se $2 * X_1 \geq 1$ allora $b_1 = 1$, altrimenti $b_1 = 0$. Possiamo ripetere lo stesso trucco per conoscere b_2 ? Certo! Il prossimo valore X_2 , da considerare a questo fine, è come segue: se $X_1 * 2 < 1$ allora $X_2 = X_1 * 2$ e altrimenti, $X_2 = X_1 * 2 - 1$. Questo garantisce che X_2 sia uguale a $VAL(b_2 \dots b_k)$ e

quindi possiamo di nuovo moltiplicarlo per 2 per determinare il valore del secondo bit b_2 . Ovviamente possiamo continuare nello stesso modo per $X_1, X_2, X_3, \dots$, ma dopo n passi, qual'è la relazione tra X_1, X_n e la sequenza di 0/1 $b_1, \dots, b_{n-1}$ prodotta negli $n-1$ passi che portano da X_1 a X_n ? La relazione è la seguente: $X_1 = X_n * 2^{-(n-1)} + VAL(b_1, \dots, b_{n-1})$. Intuitivamente, X_n rappresenta la parte di X_1 che resta da tradurre, ma le $n-1$ moltiplicazioni per 2 fatte negli $n-1$ passi, devono essere eliminate per ottenere veramente quello che resta da tradurre di X_1 e questo compito è realizzato dal fattore $2^{-(n-1)}$. Sulla base di questa spiegazione è 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 **output** è vuoto, allora $VAL(output) = 0$. 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 4.4, 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 di verifica della Sezione 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 $R' = (|output| = n - 2, X_1 - (X * 2^{-1}) * 2^{-(n-2)} = VAL(output)) = (X_1 - (X * 2^{-(n-1)}) = VAL(output))$ e l'istruzione condizionale che segue può togliere $1 * 2^{-(n-1)}$ da $X * 2^{-(n-1)}$, ma in quel caso stampa '1' e quindi $VAL(output \cdot 1) = VAL(output) + 1 * 2^{-(n-1)}$. 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 che l'aggiunta di '0' o '1' ad 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 sì, ma non si deve dimenticare che abbiamo dimostrato la correttezza parziale del programma, cioè che qualora il programma termini, allora vale la postcondizione. Ma il nostro programma termina sempre? 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, eseguiamo 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 morde 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 anziché 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 $0 \leq X * 2^{-10} < 2^{-10}$, 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** = $(X_1 - X \cdot 2^{-10} = \text{VAL}(\text{output}))$, e **R** = $(0 < n \leq 11, |\text{output}| = n - 1, X_1 - X \cdot 2^{-(n-1)} = \text{VAL}(\text{output}))$. 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.

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.

```
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;
    cout<<" l'L-valore di X e' : "<< &X <<endl;
}
```


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 3 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;
- l'ultima istruzione di scrittura stampa l'L-valore di X , da cui si può constatare che X è una variabile indipendente da q che risiede in una zona di memoria diversa da quella di q .

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 ovvia, ma 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 dichiarazione di un puntatore: se T è un qualunque tipo, allora `T*` Z ; dichiara che Z è un puntatore ad una variabile 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: `char * Z, *P`; . In questo modo è possibile dichiarare variabili 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, una variabile diventa attiva immediatamente dopo essere stata dichiarata. Quindi nella dichiarazione di p possiamo usare `&a`, ma non viceversa.

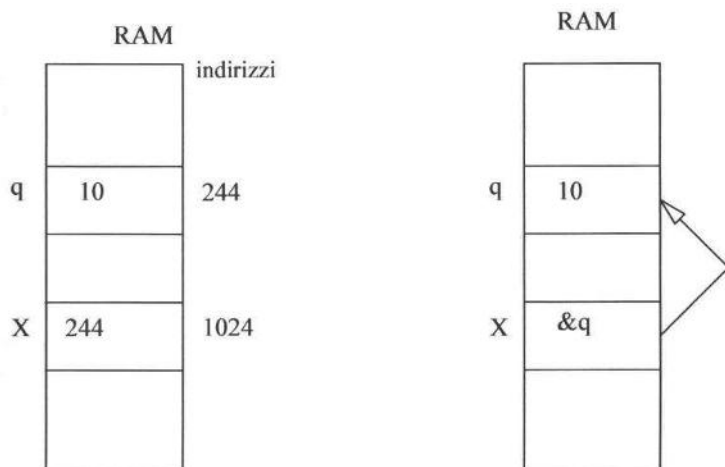


Figura 5.1: 2 modi di mostrare che X punta a q

- Nel caso di un puntatore Z ad una variabile di tipo T, l'espressione *Z indica la variabile puntata da Z. In questo caso, l'operatore asterisco applicato ad un puntatore restituisce la variabile puntata dal puntatore. Questa operazione è detta **dereferenziazione** 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 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 a delle posizioni casuali della RAM. Questo può causare un errore a run-time del tipo *segmentation fault* (violazione di accesso alla memoria), 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). Nel Capitolo 7 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 **referimento**. Intuitivamente, una variabile referimento non è che un nuovo nome per una variabile (normale) già esistente. Un esempio di dichiarazione di un variabile referimento ad intero R è il seguente: `int X; ... int &R = X;`. Dopo la seconda dichiarazione R e X sono **alias**, cioè esse hanno lo stesso L- e quindi lo stesso 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.

```
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 può 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 referimento è un errore che verrebbe segnalato dal compilatore:

```
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 referimento. La loro utilità diventerà chiara nel Capitolo 7 dove introdurremo le funzioni e più precisamente nella Sezione 7.2 che descrive il passaggio dei parametri per riferimento.

Esercizio 5.1 Scrivere un programma che definisce 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. L'array `pippo` ha una dimensione e 10 elementi, mentre la matrice A ha due dimensioni, la prima con limite 20 e la seconda con limite 100. Si possono definire array con qualsiasi numero di dimensioni, per esempio con 4 dimensioni come in `char K[3][5][10][20];`. I limiti delle 4 dimensioni di K sono 3, 5, 10 e 20. Il C++ Standard richiede che nella dichiarazione di un array i limiti siano specificati da (espressioni) 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 11.1. 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`

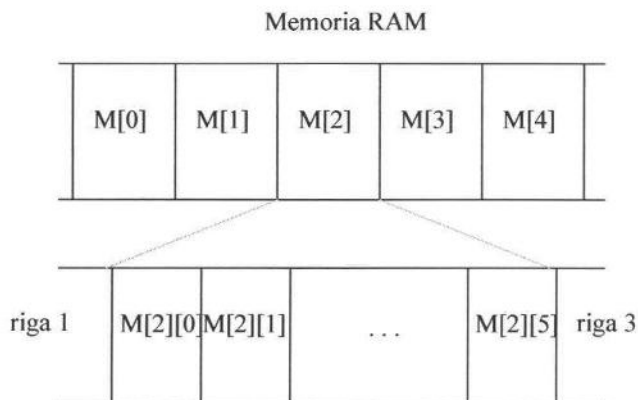


Figura 5.2: l'allocazione delle matrici nella memoria nel C++

$A[10]$; 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 byte 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 può essere più appropriato convertire i valori esadecimali nel tipo `long int` che dovrebbe consentire di rappresentare valori interi più grandi di `INT_MAX`. Anche nel caso degli 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 alloggiato 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 gli elementi di un array al momento della dichiarazione dell'array stesso.

Esercizio Risolto 5.3 *Il primo programma che consideriamo determina la posizione del massimo in un array di interi ad una dimensione con limite qualsiasi. Nel seguito viene usata la seguente dichiarazione con inizializzazione di un array: `int A[]={10,0,5,-12,45,-9,23};`. Attraverso questa dichiarazione A risulta avere limite 7 perché 7 sono i valori specificati tra parentesi graffe e ovviamente `A[0]=10,...,A[6]=23`. Se avessimo inserito in modo esplicito il limite degli elementi dell'array, come in, `int A[20]={10,0,5,-12,45,-9,23};`, allora gli ultimi 13 elementi di A verrebbero inizializzati a 0. Quindi, `int A[20]={};` inizializza tutti i 20 elementi a 0. Nel seguito consideriamo:*

`int A[]={10,0,5,-12,45,-9,23};`

Vogliamo realizzare un programma che sia corretto rispetto alle seguenti pre- e postcondizione.

PRE=(`A[0..6]` è definito);

POST=(`A[pos_max]` è il valore massimo in `A[0..6]`);

La postcondizione chiede che il programma da fare usi una variabile `pos_max` in cui conservare la posizione del massimo elemento di A. Inizializzeremo `pos_max` a 0, infatti è ovvio che `A[0]` è il massimo valore della porzione di A che consiste solo di `A[0]`. Poi dovremo inserire un ciclo che, ad ogni iterazione considera un nuovo elemento di A, calcolando dove è il massimo fino a quel punto. L'invariante di un tale ciclo while è:

R=($1 \leq i \leq 7$, `A[pos_max]` è il massimo in `A[0..i-1]`).

*Si osservi che **R** è praticamente **POST** in cui, al posto dell'indice 6 dell'ultimo elemento dell'array, è stato sostituito `i-1`. Questo fatto è logico dato che il ciclo calcola il massimo in prefissi crescenti di A. Quando il prefisso sarà l'intero array, il ciclo dovrà terminare e il massimo calcolato è quello dell'intero array. Questa relazione tra invariante e postcondizione verrà studiata in maggiore dettaglio nel Capitolo 6. Il ciclo che mantiene l'invariante **R** è facile, basta confrontare `A[i]` con il massimo trovato fino a quel momento, cioè `A[pos_max]` e, a seconda del risultato del confronto, decidere se cambiare `pos_max` o no. Inoltre la condizione di permanenza del ciclo è che `i < 7`, cioè di non aver ancora considerato tutti gli elementi di A. Questo è ovvio visto che, senza guardare tutti gli elementi, non si può calcolare il massimo. La soluzione completa segue. Si osservi che **PRE** è vera dopo la dichiarazione con inizializzazione di A.*

```
main()
{
    int A[]={10,0,5,-12,45,-9,23}; // A ha 7 elementi
    //PRE
    int i=1, pos_max=0;
    while(i<7) // R
    {
        if(A[i]> A[pos_max])
            pos_max=i;
        i++;
    }
```

```

    }
    cout<<' 'Il massimo e': '<<A[pos_max]<<' ' in posizione:'
    <<pos_max<<endl;
//POST
}

```

È semplice mostrare che **R** è invariante e che vale la condizione d'uscita. Per la condizione iniziale, basta osservare che con $i=1$, **R** diventa: $(1 \leq i \leq 7, A[0]$ è il massimo in $A[0..0])$, che è ovviamente vero.

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'Esercizio 5.3 nel modo seguente:

Esempio 5.5

```

main()
{
    int A[]={10,0,5,-12,45,-9,23};
    int pos_max=0;
    for(int i=1;i<7;i++)
        if(A[i]> A[pos_max])
            pos_max=i;
    cout<<' 'Il massimo e': '<<A[pos_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 3.6, sappiamo che ogni dichiarazione è visibile solo entro il blocco a cui appartiene. È importante qui renderci conto che il `for` introduce un nuovo blocco contenuto nel corpo del `main` e che *i* è visibile solo in questo blocco. Quindi, se dopo il corpo del `for` introducessimo l'istruzione: `cout<<i<<endl;` il compilatore ci darebbe un errore perché la variabile *i* è sconosciuta all'esterno del `for`. Studiamo in dettaglio i blocchi creati dal `for` usando il seguente esempio:

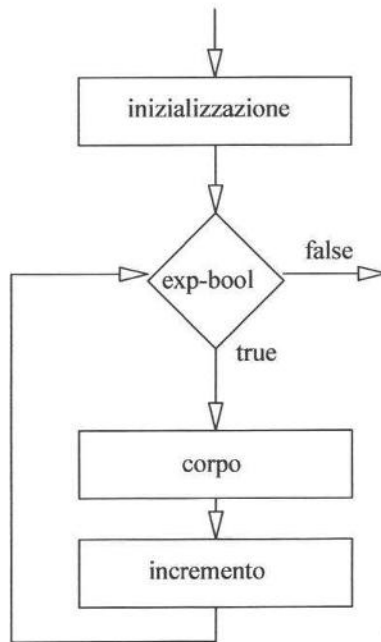


Figura 5.3: il costrutto iterativo for

```

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 !!!
}

```

Questo programma rispetta la regola della dichiarazione unica, cf. Sezione 3.6. Quindi tutte le dichiarazioni di `i` presenti in esso sono in blocchi diversi (e annidati uno dentro l'altro) e quindi dichiarano variabili diverse. Da questo 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 perché le 3 dichiarazioni non entrano in conflitto. Dai commenti a lato delle istruzioni di stampa del corpo del `for`, è facile capire che la prima stampa contenuta nel corpo del `for`, stampa la `i` dell'istestazione 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'istestazione ha effetto.

Dalla Figura 5.3 è chiaro che c'è una stretta relazione tra `for` e `while`. Questa relazione ci consente di usare la regola di verifica del `while`, cf. Sezione 4.1, anche per dimostrare la correttezza del `for`. Ci sono solo 2 cose da ricordare e la Figura 5.3 le mostra entrambe in maniera molto chiara: supponiamo di voler dimostrare $P \langle \text{for}(\text{iniz}; \text{test}; \text{incr}) \ C \rangle Q$,

- (i) La parte di inizializzazione viene eseguita solo la prima volta che l'esecuzione arriva al `for`, quindi questa parte entra solo nella dimostrazione della condizione iniziale: $P \langle \text{iniz} \rangle R$, in cui R è l'invariante del ciclo;
- (ii) L'incremento fa in realtà parte del corpo del `for` e viene eseguito prima di tornare a valutare il test di permanenza nel ciclo, quindi la dimostrazione di invarianza diventa: $R \ \&\& \ (\text{test}) \ \langle C; \text{incr} \rangle \ R$.

Esercizio 5.6 Scrivere un programma che dichiara un array `C` di 10 char e legge 10 caratteri dall'input standard negli elementi di `C`. Dopo di che esamina `C` per vedere se tra i suoi elementi è presente il carattere 'a'. Se il carattere 'a' c'è, allora il programma assegna ad una variabile booleana `apres` il valore `true` e ad un'apposita variabile intera `pos` assegna la posizione in `C` in cui è stato trovato. Altrimenti il programma assegna a `apres` il valore `false`. Si chiede di fare due versioni del programma, una con un ciclo `for` ed una con un ciclo `while`. Per la versione che usa il `while`, scrivere l'asserzione che rappresenta quanto si vuole che sia vero dopo il ciclo, poi scrivere l'invariante del ciclo ed infine dimostrare la correttezza del programma. Successivamente utilizzare le osservazioni fatte immediatamente prima di questo esercizio, per trasformare la prova di correttezza del ciclo `while` in una prova di correttezza del ciclo `for`. Se i due cicli sono sufficientemente simili, la trasformazione è immediata.

Esercizio 5.7 Modificare l'esercizio precedente nei seguenti modi:

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

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'.

Per ogni esercizio scrivere la postcondizione del programma e l'invariante del ciclo che costituisce il programma stesso.

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 l'eventuale -1, qualora esso sia stato letto. Per prima cosa si cerchi di scrivere pre- e post-condizione del programma. Si consiglia di scrivere una pre-condizione che garantisca che il programma termini sempre le letture in modo regolare, cioè o per aver letto 20 valori oppure, prima di riempire l'array, per aver letto la sentinella -1.

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 A1 ed A2 hanno in comune. Per evitare equivoci, chiariamo l'esercizio con un esempio in cui per semplicità consideriamo gli array di 6 elementi: sia $A1 = [2, 1, 2, 3, 1, 4]$ e $A2 = [0, 2, 3, 1, 0, 1]$. A1 e A2 hanno tre valori in comune: 1, 2 e 3.

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 in (quasi) tutte le circostanze il C++ considera il tipo `int[15]`, equivalente a `int[ ]` che mostra che il limite 15 dell'array può (generalmente) venire ignorato. Questo fatto implica che tutti gli array ad 1 dimensione di interi hanno lo stesso tipo, `int[ ]`, e questa cosa è molto comoda per passare array alle funzioni in quanto permette che un'unica funzione possa compiere una data operazione su array con limiti (cioè numero di elementi) diversi. Per esempio, una stessa funzione può ordinare array con 10, 100, 1000,... elementi, cfr. Sezione 7.3.

In C++ il tipo `int[ ]` è molto vicino al tipo puntatore `int*`. Per vederlo basta eseguire le seguenti istruzioni: `int A[10]; cout << A << &A[0];`. L'output stampa 2 indirizzi (interi esadecimali) identici. Quindi, l'R-valore di A è l'indirizzo del primo elemento `A[0]`. La seguente dichiarazione è quindi corretta: `int* a=A;`. Va osservato che c'è un'importante differenza tra A e a: A è costante, mentre a non lo è. Quindi l'operazione `A++;` causerebbe un errore in fase di compilazione, mentre `a++;` sarebbe perfettamente valida. La ragione per richiedere che il nome degli array, come A, siano imm modificabili è che essi sono la

strada per accedere agli elementi degli array e quindi modificandoli rischieremmo di pregiudicare questo accesso. Si potrebbe allora pensare che gli array abbiano tipo `T *const`, dove `T` è un tipo qualsiasi, che siano cioè dei puntatori costanti, ma in realtà ci sono contesti in cui si osserva che il limite dell'array è importante. Per esempio se `int A [10]` allora `sizeof(A)` dà risultato 40 (byte) e non 4 (byte) che sarebbe la dimensione di un valore `int *` (e anche `int * const`).

Che `int [ ]` sia il tipo di tutti gli array ad una dimensione di interi non si estende agli array a più di una dimensione. Infatti il tipo di `double K[10][12][20]` **non** è `double [ ] [ ] [ ]`, ma `double [ ] [12][20]`. Possiamo "dimenticare" il primo limite, ma non gli altri. Questo è vero per qualsiasi numero di dimensioni. Il motivo è legato al problema di passare gli array alle funzioni e verrà spiegato nella Sezione 7.3. Un'altra maniera assolutamente equivalente di scrivere in C++ il tipo `double [ ] [12][20]` è, `double (*) [12][20]`. Questa scrittura rivela con maggiore forza che si tratta di un tipo puntatore e, a questo proposito, di nuovo, eseguire `cout << K << &K[0][0][0]`; mostra che l'R-valore di `K` è l'indirizzo del primo elemento dell'array (primo inteso relativamente all'ordine in cui gli elementi degli array sono immagazzinati nella memoria). Queste osservazioni mostrano che due puntatori di tipo diverso (`K` e `&K[0][0][0]`) possono avere lo stesso R-Valore. Ovviamente `&K[0][0][0]` ha tipo `int*`.

Visto che `K` è un puntatore, lo possiamo dereferenziare, applicando `*` come in `*K`. Cosa otteniamo? Se dereferenziamo un puntatore, otteniamo sempre l'oggetto puntato da questo puntatore. Nel presente caso, `K` ha tipo `double (*) [12][20]` cioè puntatore ad un array, `double [12][20]` e qual'è il tipo di un tale array? E' `double [ ] [20]` o equivalentemente `double (*) [20]`. Quindi questo è il tipo di `*K`. E quello di `**K`? Visto che ogni volta si perde la prima dimensione, il tipo sarà `double (*)`, ma, visto che la stella non è seguita da parentesi quadrate, è semplicemente `double*`. Riassumendo, l'array `double K[10][12][20]` ha 3 dimensioni ed è quindi costituito da 10 strati ognuno costituito da 12 righe ciascuna di 20 elementi. Quindi il nome `K` punta all'inizio del primo strato dell'array, mentre `*K` punta alla prima riga del primo strato e `**K` punta al primo elemento della prima riga del primo strato. Finalmente `***K` avrà tipo `double` in quanto si tratterà del primo elemento della prima riga del primo strato. Infatti `***K` è completamente equivalente a `K[0][0][0]`. Va anche osservato che l'istruzione `cout << K << *K << **K`; stampa a video 3 volte lo stesso valore (esadecimale). Di nuovo abbiamo un esempio di puntatori di tipo diverso che puntano allo stesso indirizzo di memoria.

Chiariamo ora la relazione tra la dereferenziazione degli array e la notazione con le parentesi quadrate, detta **subscripting**. Usiamo ancora la dichiarazione `double K[10][12][20]`. Abbiamo spiegato prima il significato di `*K`. Esso è equivalente a `K[0]`. In modo simile, `**K` è equivalente a `K[0][0]`. Il lettore si chiederà se abbia senso scrivere per esempio `K[2]` o `K[-1]`. Se `K[0]` ha tipo `double (*) [20]` ed ha come valore l'indirizzo del primo elemento della prima riga del primo strato, allora `K[2]` ha lo stesso tipo `double (*) [20]` mentre ha come valore l'indirizzo del primo elemento della prima riga dello strato di indi-

ce 2, cioè del terzo strato. Ma $K[-1]$? Questo cosa potrà mai significare? Beh, in C++ è un'espressione perfettamente valida, ed ha ancora tipo `double (*) [20]` e come valore l'indirizzo del primo elemento della prima riga dello strato di indice -1, cioè lo strato che precede il primo strato di K . Pensate che questo sia senza senso visto che non esiste alcuno strato che precede il primo? E' vero! Ma comunque, se ci fosse, il suo primo elemento avrebbe un certo indirizzo e quindi è possibile calcolarlo e questo è il valore di $K[-1]$. La filosofia del C++ (ereditata dal C) è che il programmatore sa quello che fa e se vuole calcolare il punto d'inizio dello strato -1 di K , allora questo va calcolato. Nella prossima Sezione approfondiremo questi problemi arrivando a calcolare con precisione il valore di $K[-1]$ o di $K[-1][1]$ rispetto a quello di K .

Concludiamo la sezione osservando che, come già notato per gli array unidimensionali, anche quelli a più di una dimensione non hanno in tutti i contesti lo stesso tipo. Consideriamo ancora la dichiarazione, `double K[10][12][20]`. Se invociamo `sizeof(K)` otteniamo $10 \times 12 \times 20 \times 8$ (assumiamo che i valori di tipo `double` occupino 8 byte), quindi il tipo di K in questa occasione pare essere `double[10][12][20]` (e non `double(*)[12][20]` che *dimentica* il limite 10 della prima dimensione). Anche l'espressione `&K` è corretta ed ha lo stesso R-valore di K (cioè l'indirizzo del primo elemento dell'array), ma ha il tipo `double(*)[10][12][20]`. Quindi `double(*Q)[10][12][20] = &K` è corretto e Q è un array a 4 dimensioni che contiene K come primo elemento a 3 dimensioni. Questo fenomeno avviene anche con gli array ad una dimensione. Sia `double A[20]`, allora `&A` ha tipo `double(*)[20]` e quindi in questo contesto per A serve il tipo `double A[20]`. Infine si osservi che `&K` ha lo stesso R-valore di K e `&A` ha lo stesso R-valore di A . Questo dimostra che gli array non hanno L-valore il che è completamente coerente col fatto che il loro valore non può in nessun modo venire modificato.

5.4 L'aritmetica dei puntatori

La nozione di aritmetica dei puntatori è tipica del C e del C++ ed è riassunta nella seguente frase: data la dichiarazione `T* p`; del puntatore p , dove T indica un tipo qualsiasi, l'espressione $p+1$ ha come R-valore $p + \text{sizeof}(T)$ e, in generale, $p+k$ ha R-valore uguale a $p + \text{sizeof}(T) * k$. In modo simile $p-k$ ha R-valore uguale a $p - \text{sizeof}(T) * k$. L'idea dietro a questo fenomeno è che 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, basta scrivere $p+1$, $p+k$ e anche $p-k$ per ottenere, rispettivamente, il puntatore all'elemento successivo a quello puntato da p , il puntatore all'elemento che segue di k posizioni quello puntato da p e infine il puntatore all'elemento che precede di k posizioni quello puntato da p . Da quanto già visto nella Sezione 5.3 non dovrebbe sorprendere che risultati simili si ottengano usando il subscripting: $p[1]$, $p[k]$ e $p[-k]$. Va però subito specificato che $p[k]$ è equivalente a $*(p+k)$, quindi la notazione col

subscripting esegue sempre una dereferenziazione dopo aver applicato l'aritmetica dei puntatori. Quindi il tipo di $p[k]$ è T , mentre il tipo di $p+k$ è $T*$, come quello di p .

Come osservato alla fine della Sezione 5.3, questo meccanismo funziona sempre, cioè indipendentemente da se, immediatamente dopo o prima dell'oggetto puntato da p , ci siano altri valori dello stesso tipo T o no. Tutto quello che serve è un puntatore a qualcosa, tutto funzionerebbe anche se fosse un puntatore indefinito, cioè contenente un valore arbitrario. È responsabilità del programmatore usare l'aritmetica dei puntatori in modo corretto. Il compilatore non fa alcun controllo sul fatto che gli indirizzi calcolati attraverso l'aritmetica dei puntatori contengano valori appropriati. D'altronde, sarebbe impossibile farlo in generale! Il lettore deve però capire che produrre puntatori *sballati* (che puntano cioè ad una zona di memoria che non contiene un valore sensato) non è di per se un errore. L'errore (segnalato o no) si produce solo quando il puntatore sballato viene dereferenziato. E' come giocare con le armi da fuoco. Finché non si preme il grilletto non succede nulla. Certo è meglio non giocarci!

Quindi per calcolare il valore di $p+k$ rispetto a quello di p è necessario sapere il tipo T puntato da p in modo da calcolare $\text{sizeof}(T) * k$. Tutto è semplice quando consideriamo puntatori a tipi predefiniti. Sappiamo che in generale $\text{sizeof}(\text{int})$ è 4 byte e $\text{sizeof}(\text{double})$ è 8 byte. Quindi se p ha tipo $\text{double}*$, allora $p-3$ ha valore $p-3*8$. Il caso in cui p punta ad un elemento di un array di double non aggiunge nulla. Le cose si complicano invece quando p punta ad un array. In questo caso è necessario capire esattamente il tipo dell'array puntato in modo da calcolare la sua dimensione.

Facciamo un esempio. Consideriamo la dichiarazione, $\text{double } p[8][10][8];$. Nella Sezione 5.3 abbiamo visto che il tipo di p è $\text{double } (*)[10][8]$ e quindi $p+2$ ha valore uguale a $p+2*(10*8)*8$. Questo è completamente coerente con l'enunciato iniziale di cos'è l'aritmetica dei puntatori: p punta ad un array $\text{double}[10][8]$ e quindi un tale array consta di $10*8$ elementi double che occupano 8 byte ciascuno. Quindi $p+2$ ha come valore la somma del valore di p e di $2*(10*8)*8$. Ovviamente un simile calcolo viene fatto automaticamente per qualsiasi intero venga sommato o sottratto a p .

Abbiamo appena mostrato che avere il tipo preciso del puntatore p è essenziale per prevedere l'effetto dell'aritmetica dei puntatori applicata a p . Si possono scrivere espressioni complicate che inframmezzano l'aritmetica dei puntatori e la dereferenziazione che cambia il tipo del puntatore a cui viene applicata.

Per non farsi confondere dalla dereferenziazione, come visto nella Sezione 5.3, la maniera di ragionare è la seguente: **se dereferenziamo un puntatore ad un oggetto otteniamo l'oggetto puntato**. Applicando questo ragionamento al nostro esempio precedente, visto che p e quindi anche $p+2$ hanno tipo $\text{double } (*)[10][8]$, il tipo di $*(p+2)$ è quello di un array $\text{double}[10][8]$ e cioè è $\text{double}(*)[8]$. Quindi l'oggetto puntato da $*(p+2)$ ha dimensione $8*8$ byte. Da questo segue che il valore di $*(p+2)-3$ è uguale a $p+2*10*8*8-3*8*8$.

Cioè il +2 che si applica a `p` fa saltare 2 strati 10×8 di `double` e quindi $2 \times 10 \times 8 \times 8$. Mentre il -3 che si applica a `* (p+2)` fa saltare 3 array di 8 `double` e quindi $-3 \times 8 \times 8$.

Continuando l'esempio, potremmo considerare il tipo di `* (* (p+2) -3)`. Applichiamo di nuovo il ragionamento di prima. Visto che il tipo di `* (p+2) -3` è `double (*) [8]`, dereferenziandolo, otteniamo un array `double[8]` il cui tipo è `double[]` o `double *`. Quindi, la differenza tra `* (p+2) -3` e `* (* (p+2) -3) +4` è solo 4×8 , cioè il puntatore si sposta di 4 `double`.

Va sottolineato che le espressioni esaminate in precedenza, come per esempio `* (p+2)`, sono appunto espressioni e quindi non hanno un L-valore, ma hanno un tipo ed un R-valore il quale può venire assegnato ad una variabile del tipo coerente. Per esempio, visto che `* (p+2)` ha tipo `double (*) [8]`, la seguente dichiarazione è corretta: `double(*q)[8]=* (p+2) ;`.

Concludiamo la Sezione chiarendo la relazione tra subscripting da una parte e aritmetica dei puntatori e dereferenziazione dall'altra. A questo fine, basteranno alcuni esempi. Sia ancora `p` di tipo `double (*) [10] [8]`, l'espressione `p[2]` è equivalente a `* (p+2)`, l'espressione `p[-2] [-3]` è equivalente a `* (* (p-2) -3)` e infine, `(*p[-1])[2]` è equivalente a `* (* (*p) -1) +2`. Al lettore di determinare esattamente cosa rappresentino queste ultime due espressioni equivalenti rispetto al puntatore iniziale `p`.

5.5 Stringhe alla C e array di caratteri

Le stringhe alla C, introdotte nella Sezione 2.1, sono delle sequenze costanti di caratteri racchiuse tra doppi apici. Esse servono nei programmi per produrre in modo semplice dei messaggi di output e hanno delle caratteristiche che dipendono proprio da questo loro uso particolare. Una caratteristica delle stringhe alla C è che esse terminano sempre con un carattere *invisibile*. Si tratta del carattere nullo (che ha codice ASCII 0 e si scrive `'\0'`). Quindi, se nel nostro programma scriviamo la costante `"pippo"` in realtà essa è `"pippo\0"`. Questo carattere terminale ha la funzione di sentinella di fine stringa. Serve per esempio nel caso dell'istruzione `cout<<"pippo";` in cui si produce l'output della sequenza di caratteri `pippo` e la sentinella serve a segnalare la fine della stringa, in modo da fermare la stampa. 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 terminale. La descrizione di questa libreria è disponibile all'indirizzo <http://www.cplusplus.com/doc/>.

Gli array di caratteri sono simili alle stringhe alla C, ma i loro elementi non sono costanti (a meno di richiederlo esplicitamente nella dichiarazione) e non possiedono sentinelle alla fine (a meno di inserirla esplicitamente). Questa vicinanza ci permette di usare stringhe alla C per inizializzare gli array di caratteri. Per esempio nel modo seguente: `char A[] = "pippo";`. I caratteri di `"pippo"` (compresa la sentinella terminale) sono semplicemente copiati negli elementi di `A`, cioè `A[0] == 'p'`, `A[1] == 'i'` e così via. Deve essere chiaro che la dichiarazio-

ne `char A[] = "pippo"`; crea un nuovo array `A` di 6 caratteri (attenti alla sentinella) nel quale vengono copiati i caratteri della stringa `"pippo"`, ma è comunque indipendente dalla stringa alla `C` `"pippo"`, cioè `A` e `"pippo"` non condividono alcun elemento di memoria.

Dopo quanto detto sulla stampa delle stringhe alla `C`, non dovrebbe sorprendere che l'esecuzione di `cout<<A`; produca la stampa della stringa `"pippo"`. Ovviamente è possibile dichiarare e inizializzare array di caratteri anche senza usare le stringhe alla `C`. Per esempio nel modo seguente: `char A[]={'p','i','p','p','o'}`; . Cosa succederebbe se con questo array `A` si eseguisse `cout<<A`? Verrebbero stampati tutti i caratteri da `A[0]` fino alla prima sentinella che segue in memoria `A[0]`. Ovviamente, visto che la sentinella non è stata inserita in `A`, la stampa potrebbe continuare oltre l'ultimo elemento di `A` invadendo una zona di memoria che non dovrebbe essere acceduta. Questo potrebbe portare ad un errore a run-time. Il lettore è sollecitato ad eseguire le istruzioni appena discusse per osservare questo fenomeno con i propri occhi.

È importante notare che solo per gli array di caratteri vale che la stampa del nome di un array causi la stampa di tutto l'array (e anche oltre!). 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, cosa che può causare un errore run-time o meno.

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 poiché il C++ Standard non permette l'assegnazione tra array. Neppure tra array di caratteri.

Capitolo 6

Alla Ricerca dell'Invariante Perduto

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

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

Dato un array $C[r][k]$ sia R una sua riga e K una sua colonna. Possiamo vedere R e K come insiemi e chiederci se R è contenuto in K , oppure possiamo

vedere R e K come multi-insiemi e chiederci se R è m -contenuto in K . Per essere certi che la differenza sia chiara diamo il seguente esempio:

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

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

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

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

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

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

¹Con la notazione $C[\dots][j]$, nel seguito indicheremo la colonna j di C .

La regola di verifica del `while` (vedi Sezione 4.1) dice che la prova di correttezza si articola in 3 parti. Concentriamoci sulla parte (3), che considera l'uscita dal ciclo. La regola dice che in questo caso dobbiamo dimostrare che: $R1 \ \&\& \ ! (i < 6) \Rightarrow \text{POST}$. Da $(0 \leq i \leq 6)$, contenuta in $R1$, assieme a $! (i < 6) \equiv (i \geq 6)$, segue che $i = 6$ e quindi, da $R1$, abbiamo che, $(\forall a \in [0..5] \ \& \ \forall b \in [0..4], C[a][b] \text{ è definito})$ che è uguale alla nostra POST e quindi l'implicazione è verificata! In effetti, applicando la ricetta di indicizzazione, abbiamo costruito $R1$ da POST proprio per fare in modo che esse coincidessero quando tutte le righe fossero riempite!

Il nostro programma continua con

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

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

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

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

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

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

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

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

POST1 = (B è calcolata bene) che si può scrivere in modo preciso come segue, $(\forall a \in [0..5] \text{ e } \forall b \in [0..4], B[a][b] \text{ sse } C[a] \text{ è contenuta in } C[...][b])$.

Visto che B viene riempita per righe, servono anche qui 2 cicli for annidati, il più esterno che scandisce le righe di B e quello interno che scandisce le colonne. Assumiamo che gli indici dei due cicli siano i e j, rispettivamente. Allora l'invariante del ciclo più esterno (secondo la ricetta delineata prima) lo ricaviamo modificando **POST1** nel modo seguente: **R1** = $(\forall a \in [0..i-1] \text{ e } \forall b \in [0..4], B[a][b] \text{ sse } C[a] \text{ è contenuta in } C[...][b]) \text{ \&\& } (0 \leq i \leq 6)$ e quello del for interno avrà invariante: **R2** = $(\forall b \in [0..j-1], B[i][b] \text{ sse } C[i] \text{ è contenuta in } C[...][b]) \text{ \&\& } (0 \leq j \leq 5)$ e alla fine di questo ciclo dovrà valere l'asserzione, **POST2** = (ho messo a posto la riga i) che è espressa in maniera precisa nel modo seguente, $(\forall b \in [0..4], B[i][b] \text{ sse } C[i] \text{ è contenuta in } C[...][b])$. Si osservi che **POST2** è ottenuta da **R2** con la solita ricetta, ma applicata nella direzione opposta rispetto al solito. Quindi il programma che riempie B comincia così:

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

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

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

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

Da **POST4** otteniamo nel solito modo **R4** invariante dell'ultimo (quarto) ciclo: **R4** = $(\text{trovato sse } \exists a \in [0..z-1], \text{ tale che } C[i][k] = C[a][j]) \text{ \&\& } (0 \leq z \leq 6)$

la relazione tra **R4** e **POST4** è quella della nostra ricetta e quindi garantisce che, all'uscita del ciclo, allorché vale ($z==6$), sia vera **POST4**. Quindi il ciclo più annidato è come segue:

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

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

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

Mettiamo ora assieme i quattro cicli annidati:

```
for(int i=0; i<6; i++)
    for(int j=0; j<5; j++)
    {
        bool OK=true;
        for(int k=0; k<5; k++)
        {
            bool trovato=false;
            for(int z=0; z<6; z++)
                if (C[i][k]==C[z][j])
                    trovato=true;
            if(!trovato) //POST4 ci dice che è la cosa giusta
                OK=false;
        }
        B[i][j]=OK; //POST3 ci dice che è la cosa giusta
    }
}
//POST1 = (per ogni a in [0..5] e per ogni b in [0..4], B[a][b] sse
// C[a] è contenuta in C[..][b])
```

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

```

for(int k=0; k<5 && OK; k++) //R3
{
    bool trovato=false;
    for(int z=0; z<6 && !trovato ;z++) //R4
        if (C[i][k]==C[z][j])
            trovato=true;
    if(!trovato)
        OK=false;
}

```

Gli invarianti trovati prima continuano a valere. Le dimostrazioni dell'inizializzazione e invarianza di questi 2 cicli non cambiano, ma la condizione d'uscita dal ciclo va riconsiderata visto che ora si può uscire da questi due cicli anche con $k < 5$ e $z < 6$. Consideriamo quindi la dimostrazione del nuovo caso d'uscita: **R3** && !OK $\Rightarrow$ **POST3** (si ricordi che fondamentalmente **POST3** asserisce che (OK va bene)):

R3 && !OK $\Rightarrow$ (false sse $\forall b \in [0..k-1], \exists a \in [0..5], C[i][b] = C[a][j]$)
che è equivalente ad asserire che:

(*) ($\exists b \in [0..k-1]$ tale che $\neg \exists a \in [0..5]$ tale che $C[i][b] = C[a][j]$)

e questo implica che **POST3** è vera, visto che **POST3** con OK uguale a false è (false sse $\forall b \in [0..4] \exists a \in [0..5], C[i][b] = C[a][j]$), e la parte destra di questa equivalenza è falsa a causa di (*).

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

```

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

```

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

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

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

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

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

```
bool T[6]={};
bool OK=true;
for(int k=0; k<5 && OK; k++) //R3'
{
    OK=false;
    for(int z=0; z<6 && !OK ;z++)//R4'
        if(!T[z] && C[i][k]==C[z][j])
            {OK=true; T[z]=true;}

    //POST4' }
// POST3'
```

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

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

R3' =

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

- $(0 \leq k \leq 5)$.

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

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

POST4' consiste di 2 parti:

- $(OK \Rightarrow \exists a \in [0..5] \text{ tale che } (C[i][k] = C[a][j] \ \&\& T[a] = \text{true} \ \&\& T[a] \neq \hat{T}[a] \ \&\& \forall b \in [0..5] \text{ con } b \neq a \ T[a] = \hat{T}[a]))$;
- $!OK \Rightarrow \nexists a \in [0..5] \text{ tale che } (C[i][k] = C[a][j] \ \&\& \hat{T}[a] = \text{false})$.

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

Capitolo 7

Funzioni

7.1 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:

```
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 è detta quindi la funzione **chiamata**. Al termine della sua esecuzione, la funzione chiamata restituisce al chiamante (eseguendo l'istruzione `return`) un valore booleano, chiamiamolo `V`, che è il valore dell'invocazione. A questo punto l'esecuzione continua con l'esecuzione del comando condizionale `if (V) ...` del chiamante. Insomma, l'invocazione `maggiore(A[i],max)` è un'espressione il cui valore è il valore booleano `V` restituito dalla funzione chiamata.

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 cosiddetti **parametri formali**. Invece i parametri tra parentesi usati nell'invocazione della funzione sono detti **parametri attuali**. Ovviamente il numero dei parametri formali deve essere lo stesso di quello dei parametri attuali. 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 9.4.

Nell'esempio precedente la funzione `maggiore` è invocata con 2 parametri attuali: `A[i]` e `max`. Quest'invocazione fa sì 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` distinta dal corrispondente parametro attuale `A[i]`. Dopo l'invocazione `maggiore(A[i],max)`, il parametro formale primo riceve come R-valore l'R-valore del parametro attuale `A[i]`, ma

primo ha L-valore diverso da quello di `A[i]`. Lo stesso vale per il parametro formale secondo ed il corrispondente parametro attuale `max`.

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 del chiamante immediatamente successiva all'invocazione della funzione che esegue il `return`. Esistono anche funzioni che non restituiscono alcun risultato, per esempio, funzioni che eseguono delle stampe. Tali funzioni dichiarano nell'intestazione di restituire un risultato di tipo `void`, cf. Sezione 2.1. Una funzione che restituisce `void`, termina o quando viene eseguito un `return`; (che non specifica alcun valore da ritornare) oppure quando l'esecuzione arriva semplicemente alla fine del corpo della funzione. In ogni caso, al termine dell'esecuzione di una funzione, l'esecuzione salta all'istruzione del chiamante che segue immediatamente l'invocazione della funzione.

Nell'esempio appena visto abbiamo inserito la definizione della funzione maggiore prima del `main` per un motivo preciso: visto che il `main` invoca maggiore, questa funzione deve essere nota al compilatore quando esso arriva a compilare l'invocazione, in modo che il compilatore possa verificare la concordanza tra l'invocazione e la definizione della funzione. In realtà, non è necessario inserire l'intera definizione della funzione prima della sua invocazione, ma è sufficiente inserire il **prototipo** che nel caso di maggiore sarebbe: `bool maggiore(int, int);`. Il prototipo di una funzione fornisce al compilatore informazioni sul tipo del risultato della funzione e anche sul tipo dei parametri della funzione. Inserire nel prototipo nomi per i parametri formali, come per esempio, `bool maggiore(int a, int b);` è possibile, ma inutile. Una volta che le invocazioni di una funzione sono precedute dal prototipo della funzione, la definizione completa della funzione può anche seguire l'invocazione nello stesso file oppure essere addirittura su un altro file, cf. Sezione 9.7. Ovviamente però la definizione della funzione ci deve essere, infatti se il compilatore non la trova, terminerà la compilazione con un messaggio di errore che segnala che il programma invoca una funzione indefinita. In un programma il prototipo di una funzione può venire ripetuto un numero arbitrario di volte, ma ci deve essere sempre **una sola** definizione della funzione stessa! Va inoltre osservato che in C e in C++ le definizioni delle funzioni non possono essere annidate. Cioè ogni funzione è potenzialmente visibile ed utilizzabile da ogni altra funzione, a condizione che ogni invocazione sia preceduta da un prototipo della funzione invocata.

Abbiamo già detto che una funzione viene invocata, esegue e poi ritorna al punto immediatamente successivo all'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 finché la funzione non è invocata sarebbe inutile assegnare spazio RAM alle sue variabili. Quindi le variabili di una funzione

vengono allocate in cima alla pila 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 3.6. In particolare, quando la funzione termina, le sue variabili vengono deallocate dalla pila dei dati 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 *next* l'istruzione immediatamente successiva a questa invocazione. Al momento dell'invocazione di una funzione, assieme alle variabili della funzione invocata, vengono allocate nella pila dei dati varie altre informazioni utili per la corretta esecuzione della funzione stessa. Tra queste informazioni, c'è anche l'indirizzo RAM in cui si trova l'istruzione *next* che è detto **indirizzo di ritorno** dell'invocazione. Questo indirizzo è necessario per realizzare l'operazione di ritorno della funzione che è infatti un salto all'istruzione *next* (cioè all'indirizzo RAM in cui si trova l'istruzione *next*). 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 7.1 *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'istestazione della funzione. Ogni volta che la funzione `divisore` viene invocata, verranno 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`;

7.2 Passaggio dei parametri

Le due funzioni presentate nella Sezione 7.1 ricevono dal chiamante dei valori che elaborano per calcolare il risultato che restituiscono al chiamante con un `return`. Quindi i parametri attuali di una invocazione sono delle espressioni i cui valori diventano gli R-valori dei corrispondenti parametri formali della funzione invocata, per cui la terminologia di passaggio dei parametri per valore è completamente appropriata. Per esempio la funzione `divisore` dell'Esempio 7.1, potrebbe essere invocata nel seguente modo: `int x=235, y=divisore(12*x);`. Collegata al passaggio dei parametri per valore è 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+1);` 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. Nello specificare 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 suo risultato. 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 della funzione invocata sulle variabili del chiamante si dicono **side-effect**. Essi si possono ottenere in due modi.

- passando per valore puntatori a variabili anziché 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 L-valori). Quindi, ogni cambiamento che la funzione invocata effettua su un parametro formale passato per riferimento si riflette automaticamente sul corrispondente parametro attuale.

Illustriamo questi nuovi concetti con qualche esempio:

Esempio 7.2 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, come mostra la funzione che segue che prova inutilmente ad eseguire questo compito.

```
#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

La pila dei dati nella RAM

x	10	10	10
y	20	20	20
primo	10	20	dealloc
secondo	20	10	dealloc
T	indef	10	dealloc

Figura 7.1: Da sinistra a destra: la pila dei dati quando `scambia` viene invocata, alla fine della sua esecuzione e dopo il ritorno

funzione, i valori di `x` ed `y` sono gli stessi di prima dell'invocazione. La situazione della RAM è rappresentata in Figura 7.1.

Vediamo ora come ottenere l'effetto desiderato passando per valore, come parametri attuali, gli L-valori delle variabili da scambiare.

```
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);        // 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` ed hanno quindi tipo `int*` e questi valori vengono assegnati a `primo` e `secondo`, rispettivamente.*

Lo pila dei dati nella RAM

x	10	302	20	20
y	20	306	10	10
primo	302	310	302	dealloc
secondo	306	314	306	dealloc
T	indef	318	10	dealloc

Figura 7.2: Il comportamento di `scambia` quando riceve per valore puntatori alle variabili da scambiare

- la funzione `scambia` dereferenzia `primo` e custodisce il suo R- valore nella variabile locale `T`. L'R-valore di `T` è quindi uguale a quello di `x`, cioè 10. La dereferenziazione è necessaria anche per le operazioni successive di scambio e di stampa. Se si stampasse per esempio `primo` anziché `*primo`, otterremmo l'L-valore di `x` anziché 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;
*secondo=*primo;
```

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

L'evoluzione della RAM quando questa funzione `scambia` viene invocata, è rappresentata in Figura 7.2.

Consideriamo ora la seconda possibile soluzione 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 riferiment
{
    int T=primo;
    primo=secondo;
    secondo=T;
    cout<<primo<<secondo<<endl;
```

La pila dei dati nella RAM

primo=x	10	20	20
secondo=y	20	10	10
T	indef	10	dealloc

Figura 7.3: Comportamento di scambio quando riceve le variabili da scambiare per riferimento

```

}
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 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 7.3 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 scambio o no? Una spiegazione è data alla fine di questa Sezione.

Esempio 7.3 Realizziamo ora una funzione F per compiere il seguente side-effect. Supponiamo di avere una funzione nella quale sono dichiarate due variabili x ed y di tipo int e vogliamo che l'invocazione di F causi il fatto che la variabile x assuma 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. Realizziamo F nelle due maniere appena viste: prima passando per valore i puntatori alle due variabili e poi passando le variabili per riferimento.

```

void F(int* px, int* py)
{
    if (*py > *px)
    {

```



```

        int t=*px;
        *px=*py;
        *py=t;
    }
}
// invocazione: F(&x,&y);

void F(int & a, int & b)
{
    if (b>a)
    {
        int t=a;
        a=b;
        b=t;
    }
}
// invocazione: F(x,y);

```

Così come nel passaggio per valore il parametro attuale può essere un'espressione qualsiasi, purché fornisca un R-valore del tipo giusto, nel caso di passaggio per riferimento, il parametro attuale può essere un'espressione qualsiasi che fornisca un L-valore del tipo giusto. Generalmente il parametro formale è semplicemente una variabile che diventerà un alias del corrispondente parametro formale, ma si possono anche usare espressioni più complicate. Per esempio se abbiamo una funzione `F(int & x)` e nel chiamante abbiamo una variabile `int y;`, allora l'invocazione `F(y)` è corretta, ma anche l'invocazione `F(*(&y))` sarebbe corretta ed equivalente alla precedente (anche se meno sensata). Un altro esempio di invocazione corretta (di nuovo non molto sensata, ma è *solo* un esempio) è il seguente: `int y, *p=&y, **q=&p; F(**q);`. Si osservi che, ogni volta che si dereferenzia un puntatore, si ottiene una variabile, cioè un L-valore. Questo è ovvio perché un puntatore punta ad un indirizzo di memoria e quindi l'oggetto puntato ha sempre un L-valore (e quindi anche un R-valore).

Visto che ogni parametro passato per riferimento crea una variabile "in comune" tra chiamante e chiamato, vedi Figura 7.3, 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 9.2 vedremo che questa limitazione è meno grave di quanto ci possa sembrare ora (potremo definire strutture che racchiudono un numero arbitrario di informazioni in un unico valore), 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)`. È naturale chiedersi perché nel C++ siano stati aggiunti i riferimenti, che sono realizzati con puntatori nascosti, quando il linguaggio già disponeva di puntatori espliciti (ereditati dal C). La risposta è già nei 20 e più anni che separano la nascita del C++ da quella del C. Il C è nato nei tardi anni 60 per realizzare il sistema operativo Unix e quello scopo richiedeva un linguaggio di programmazione molto efficiente e che consentisse la massima libertà d'utilizzo, in modo che programmatori molto esperti potessero facilmente eseguire cose complicate di cui erano perfettamente consapevoli. Dagli anni 60 agli 80, in cui è stato definito il linguaggio C++, la visione di quello che deve essere un buon linguaggio di programmazione 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 linguaggio abbia componenti poco chiare o che dipendono dal computer su cui i programmi sono eseguiti. 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 preso maggiormente 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.

7.3 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 considereremo anche quelli a più dimensioni.

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` con limite qualsiasi. Per questo motivo, spesso, accanto al parametro formale che accoglie l'array si prevede un parametro intero che accoglie il limite 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 valida potrebbe essere, `f(K, 100)`, in cui entrambi i parametri sono passati per valore. Si osservi che nell'esempio viene passato 100, anziché il vero limite di `K` che sarebbe 1000, per mostrare che non è sempre necessario passare il limite vero, ma che si è liberi di passare il numero di elementi più adatto all'operazione che la funzione invocata deve eseguire. Deve essere chiaro comunque che quando la funzione `f` riceve l'R-valore di `K`, essa ha tutto quello che le serve per accedere a qualsiasi elemento dell'array `K`. Il secondo parametro 100, in nessun modo, limita la libertà di accedere elementi di `K` oltre la posizione 100 (e comunque anche di uscire erroneamente dai limiti di `K`).

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 7.4 *Nella seguente funzione si sfrutta il fatto che i caratteri sono in effetti rappresentati da interi (il codice ASCII esteso) 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 è il limite di elementi che vanno considerati. In questo modo la funzione può lavorare su array di limite 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) e poi dereferenzia il puntatore ottenuto, vedi Sezione 5.4.*

*Naturalmente, ogni funzione deve assolvere un particolare compito che, in generale consiste nel 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 preconditione **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> 0 elementi definiti);
- **POST** = (il valore di max è il valore massimo in C[0..limite-1]);

*L'invariante del ciclo for è: **R**=($1 \leq i \leq \text{limite}$, max è il massimo in C[0..i-1]).*

*Si osservi che **R** è ottenuto da **POST** applicando la ricetta di indicizzazione del Capitolo 6. Questo fatto rende "automatica" la verifica della condizione finale del ciclo. La condizione iniziale è immediata perché se i=1 allora il massimo in C[0..0] è semplicemente C[0] che è il valore di max. L'invarianza di **R** è conseguenza del fatto che il corpo considera C[i] ed eventualmente cambia max se C[i] si rivela il massimo in C[0..i]. L'incremento i++ rende di nuovo vero **R**. 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 limite 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 limite qualsiasi. La libertà sul limite è solo per la prima dimensione, mentre per ciascuna delle successive dimensioni il limite deve essere dato. Il motivo di questo vincolo, che spesso si rivela piuttosto fastidioso, non è probabilmente quello della sicurezza, cioè che in questo modo il codice può venire controllato in modo più stringente (si ricordi 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 perché. 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 un puntatore al primo elemento di `A` (indipendentemente dal tipo 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 $i * (20 * 30) * 4$ byte 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 i limiti 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. Esempi di passaggio di array a 3 dimensioni sono trattati nel Capitolo 8

Esempio 7.5 *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.*

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

```

int * riga_max(int M[][10], int limite_1)
    // limite_1 = limite della prima dimensione.
    //10 = limite della 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
    {
        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 "rinascere" 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 `i`-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 5.4.

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 primo `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**=($1 \leq i \leq \text{limite_1}$, `punt_max` punta al primo elemento della riga di M che ha somma massima tra le righe `[0..i-1]` 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 semplice da capire. È ovviamente preferibile scegliere sempre questa soluzione.

Esercizio 7.6 Dimostrare la correttezza della versione della funzione dell'Esempio 7.5 in cui `somma_max` viene inizializzata a `INT_MIN`.

Esercizio 7.7 Per ciascun esercizio che segue, la funzione da produrre va corredata di pre-, postcondizione, 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[limite_1][5]`, ma questa volta vogliamo scrivere una funzione che determini se tra le colonne dalla 1 alla 4 di questa matrice, ce n'è almeno una che contiene tutti gli elementi che sono nella colonna 0 (possibilmente in un ordine diverso). Si osservi che qui stiamo usando la nozione di contenimento semplice della Definizione 6.1 e non l'*m*-contenimento e perciò, se, per esempio, la colonna 0 è composta solo da 1 (ripetuto `limite_1` volte), allora basta che una delle altre colonne contenga un 1 per rispondere sì alla domanda. Nel caso la risposta sia sì, la funzione deve restituire al chiamante l'indice (in `[1..4]`) di una qualsiasi delle colonne che contengono la colonna 0. Altrimenti, la funzione restituisce 0.

²Nel 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.

3. *Considerando ancora una matrice `char [] [5]`, ora vogliamo determinare se almeno una delle colonne dalla 1 alla 4 è una permutazione della colonna 0. Come nel caso precedente, in caso di risposta positiva, 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. La proprietà da testare qui è il mutuo contenimento delle 2 righe, nel senso della Definizione 6.1. In caso la risposta sia positiva, la funzione deve anche restituire al chiamante gli indici delle 2 righe in questione. Altrimenti deve restituire dei valori che avrete scelto in modo da mostrare che non è stata trovata alcuna soluzione.*

7.4 Variabili globali

Finora abbiamo visto solamente variabili dichiarate nel `main` oppure in una funzione. Esistono variabili dichiarate fuori da ogni funzione, cioè dichiarate nel file (o in uno dei file se ce n'è più d'uno, cf. Sezione 9.7) che contiene il programma al di fuori di ogni funzione. Questa zona esterna ad ogni funzione viene chiamata il **blocco globale** per indicare che si tratta del blocco più esterno possibile e nel quale sono annidati tutti gli altri blocchi. In accordo quindi con le regole di scope delle variabili, cf. Sezione 3.6, le variabili dichiarate nel blocco globale sono visibili in ogni funzione che segua la dichiarazione stessa. Per questo motivo queste variabili sono dette **globali**. Si noti che anche le funzioni sono globali allo stesso modo delle variabili globali. Una funzione è visibile nel blocco globale dal punto in cui compare il suo primo prototipo in poi. Le variabili globali vanno usate con prudenza e parsimonia per un motivo semplice: l'evoluzione del valore di una variabile globale è difficile da controllare in quanto una variabile globale può venire modificata da tutte le funzioni in cui è visibile. D'altra parte, 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 in precedenza, è buona norma usare poche variabili globali ed inoltre conviene dare loro nomi che indicano in modo chiaro che si tratta di globali come, per esempio, `global_nome` o simili.

Quando nel corpo di una funzione viene usato in un comando un nome, diciamo `q`, che non è il nome di alcuna variabile locale della funzione, allora il compilatore cerca nel blocco globale, prima della definizione della funzione, la dichiarazione della variabile globale `q` e, se una tale dichiarazione esiste, associa il nome usato nella funzione con questa variabile globale. Se invece una tale dichiarazione non esiste, allora il compilatore segnala un errore e interrompe la compilazione. Questa regola è chiamata dello **scoping statico** e merita attenzione per capirne tutte le

implicazioni. Il seguente esempio presenta un tipico caso in cui è indispensabile applicare appropriatamente la regola dello scoping statico.

Esempio 7.8

```
#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
}
```

Lo scoping statico assicura che la x usata nel corpo di f è la variabile globale dichiarata prima della definizione di f e non quella dichiarata nel `main` che invece è quella che viene stampata dal `main`.

*Per completezza osserviamo che esiste una regola diversa da quella dello scoping statico, detta di **scoping dinamico**, che nell'esempio appena esaminato, per l'invocazione della funzione f all'interno del `main`, associa la variabile x del corpo di f con quella dichiarata nel `main` in quanto è la dichiarazione più vicina all'invocazione di f . Quindi, adottando lo scoping dinamico, il programma dell'esempio stamperebbe 10 due volte. Si osservi che se il programma contenesse un'altra funzione g nel cui corpo ci fosse una dichiarazione `int x=20;` ed un'invocazione ad f , allora per questa invocazione la x di f verrebbe associata alla x di g . Al contrario, con lo scoping statico, anche per questa seconda invocazione di f la x sarebbe associata a quella globale ($x=1$). Insomma con lo scoping dinamico non c'è una sola associazione per la x di f , come per lo scoping statico, ma viene fatta un'associazione possibilmente diversa per ogni invocazione di f . È questo il motivo per cui le due regole sono chiamate di scoping **statico** e di scoping **dinamico**.*

Per evitare equivoci sottolineiamo che il C++ (come praticamente tutti i linguaggi di programmazione) adotta lo scoping statico. Lo scoping dinamico viene usato per gestire le eccezioni.

7.5 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 un L-valore e quindi anche un R-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. Per un tipo qualsiasi `T` diverso da `void`, se la funzione restituisce un R-valore di tipo `T`, basterà specificare `T` alla sinistra del nome della funzione. Se invece la funzione restituisce un L-valore allora si dovrà specificare `T&`, che chiaramente indica che si restituisce un riferimento. Se il tipo del risultato è `T`, 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`), mentre se il tipo del risultato è `T&` allora in `return e`; `e` deve avere per valore una variabile di tipo `T`. Va osservata la completa analogia tra quanto appena detto ed il passaggio dei parametri per valore e per riferimento studiati nella Sezione 7.2.

Esempio 7.9 *In questo esempio consideriamo il problema di trovare l'elemento di un array che contiene il massimo dell'array, ma anziché semplicemente ritornare il valore del massimo o la sua posizione nell'array, si vuole restituire un valore che permetta al chiamante di accedere direttamente all'elemento massimo. Ovviamente, il valore da restituire può essere o per valore l'L-valore dell'elemento massimo oppure un riferimento dell'elemento massimo. Esaminiamo entrambe le possibilità nel seguito. Si osservino le differenze rispetto al tipo del valore restituito. Per prima consideriamo la funzione che restituisce per valore l'L-valore dell'elemento massimo.*

```
int * massimo(int * A, int limite_1)
    // notare il tipo del risultato
{
    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;
}
```

Come esempio, il main usa il valore restituito da massimo per aggiunge 5 all'R-valore dell'elemento massimo dell'array. Vogliamo ora fare lo stesso usando il ritorno per riferimento.

```
#include<iostream>
int & massimo(int * A, int limite_1)
    // notare il tipo del risultato
{
    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;
}
```

Si osservi che in quest'ultimo esempio massimo restituisce un alias dell'elemento massimo dell'array e quindi l'invocazione della funzione può stare semplicemente alla sinistra del simbolo = dell'assegnazione (senza dereferenziazione). Questi esempi mostrano che, come nel caso del passaggio dei parametri, cf. Sezione 7.2, 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. Come per il passaggio dei parametri, anche nel caso della restituzione del risultato, l'uso dei riferimenti spesso consente di ottenere codice maggiormente 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 perché questa variabile sarà deallocata dopo la fine dell'esecuzione della funzione, vedi Sezione 7.1. 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 locale viene detto creare **un puntatore o un riferimento penzolante** che è la brutta traduzione dell'espressione inglese **dangling pointer/reference** che useremo nel seguito.

Come esempio di una funzione che crea un 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 perché 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 7.10 *Risolvere i seguenti esercizi specificando pre- e postcondizione per ciascuna funzione prodotta:*

1. *Scrivere una funzione che riceve un array 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 nell'ordine in cui l'array è allocato nella RAM, cf. Sezione 5.2) e calcola le seguenti 2 quantità. 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 la nuova funzione restituisca un riferimento al primo elemento della riga che contiene il massimo numero di valori uguali a c. Si osservi che il massimo numero di valori uguali a c potrebbe essere 0. In caso ci siano più righe che contengono il massimo numero di occorrenze di c, basta restituire il riferimento al primo elemento di una qualsiasi di queste righe.*

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 11.1.

7.6 Esercizi commentati sulle funzioni

Seguono alcuni esercizi risolti che propongono funzioni che utilizzano le diverse modalità viste in precedenza per il passaggio dei parametri e per la restituzione del risultato. Alcune delle funzioni proposte sono corrette ed altre sono (appositamente) errate. Invitiamo il lettore a cercare di determinare la correttezza o meno delle funzioni proposte prima di leggere la risposta.

Esercizio Risolto 7.11 *Si consideri il seguente programma:*

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

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

Risposta: Innanzitutto consideriamo 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 perché `x` ha tipo `int*` e quindi `*x=5`; assegna un intero alla variabile intera 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 ad una variabile intera. 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 è corretto e stampa 25.

Esercizio Risolto 7.12

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

Risposta: La funzione `f` riceve per valore un `int*` che diventa l'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 7.5 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 locale della funzione che viene quindi 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 questo programma in certe esecuzioni sembra corretto e in altre può esibire comportamenti inattesi. Ecco perché gli errori di dangling pointer sono difficili da trovare.

Esercizio Risolto 7.13

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

Risposta: 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 dangling, infatti viene restituito l'oggetto puntato da `x` che è alias di `z`. Quindi `f` restituisce la variabile `y` del `main`. In conclusione, il programma è corretto e stampa 25.

7.7 Esercizi proposti

Esercizio 7.14 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 7.15 Il seguente programma è corretto? Se sì, 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 7.16 Il seguente programma è corretto? Se sì 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 postcondizione ed i principali cicli iterativi devono avere il loro invariante.

Esercizio 7.17 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 7.18 Scrivere una funzione che riceve come parametri un array `int [] [5]` ed il numero di righe `limite1` e restituisce il riferimento all'elemento minimo dell'array. Modificare questa funzione in modo che restituisca il puntatore all'elemento minimo della matrice. A vostro avviso, conviene restituire questo puntatore per riferimento o per valore?

Esercizio 7.19 Scrivere un programma che legge da input standard i valori per inizializzare un array `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 7.20 Scrivere una funzione capace di ricevere un'array `int []`, il suo limite 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 7.21 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 7.22 Scrivere una funzione che riceve come parametri un array di `char` e il suo limite `limite1` 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 7.23 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 in posizione $k + n$ con $n > 0$ e la 'y' finale va cercata dalla posizione $k + n + 1$ in poi.

Esercizio 7.24 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.

Esercizio 7.25 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 complete (contenenti 20 elementi) e la terza riga con solo 3 elementi e quindi non completa. La funzione richiesta deve determinare se c'è una riga dell'array (considerando anche l'eventuale riga finale incompleta, 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 7.26 Questo esercizio serve a illustrare una tecnica che è spesso utile nel caso si debba scrivere una funzione che manipoli array a molte dimensioni. Abbiamo visto in Sezione 7.3 che il parametro formale capace di accogliere un array a più dimensioni, passato alla funzione, deve avere i limiti di 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` rispettino 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.

Capitolo 8

Esercizi sulle funzioni

In questo Capitolo svilupperemo funzioni che devono manipolare array a più dimensioni. A seconda dei casi passeremo l'array multidimensionale in modo esplicito, cioè specificando nel parametro formale il limite delle sue dimensioni dalla seconda in poi, oppure, seguendo l'idea dell'Esercizio 7.26, lo passeremo come fosse un array ad una sola dimensione, specificando i limiti delle varie dimensioni in appositi parametri formali. Ricordiamo che quest'ultima tecnica sfrutta il fatto che il C++ alloca gli array nella RAM in posizioni contigue, cf. Sezione 5.2. A causa di questa allocazione contigua un array a molte dimensioni con limiti $limite_1, \dots, limite_k$, risiede in memoria come un array ad una sola dimensione con $limite_1 * \dots * limite_k$ elementi e può quindi essere trattato come tale. Come sempre dimostreremo la correttezza parziale (e a volte anche totale) delle soluzioni proposte. I primi due problemi che affrontiamo sono i seguenti:

8.1 Problemi (1) e (2)

- (1) Sappiamo che un array a 3 dimensioni viene immagazzinato in memoria come una sequenza contigua di strati a 2 dimensioni che a loro volta sono immagazzinati come sequenze contigue di array a 1 dimensione che sono sequenze di elementi. Questa allocazione definisce quindi un ordine tra gli elementi di A. Nel seguito, quando facciamo riferimento ai primi n elementi di un array, intendiamo i primi n elementi rispetto a questo ordine. Si chiede di scrivere una funzione `int F(int A[][5][10], int limite1, int n_ele)` che soddisfi le seguenti condizioni:

- i) A ha `limite1` strati, e per $0 \leq n_ele \leq limite1 * 5 * 10$, si deve assumere che solo i primi `n_ele` elementi di A (rispetto all'ordine d'allocazione ricordato prima) siano definiti e che gli eventuali altri elementi siano indefiniti e che quindi non debbano venire considerati per le operazioni descritte nel seguito.

- ii) la funzione F deve restituire l'indice dello strato che abbia somma degli elementi massima in A . E' importante che nel calcolo della somma degli elementi di ciascuno strato siano considerati solamente gli elementi di A che sono definiti (ed esclusi quelli indefiniti).
 - iii) qualora ci fossero diversi strati con somma degli elementi uguale tra loro e maggiore di quella di tutti gli altri strati, F deve restituire l'indice di uno qualsiasi di questi strati a somma massima.
- (2) Usando una simbologia cara a molti, un array a 3 dimensioni può essere visto come una torta rettangolare composta da strati, ognuno dei quali è un array `int [5] [10]`. Continuando in questa simbologia della torta, possiamo anche definire le **fette** della torta come composte da righe degli strati: la prima fetta è composta dalle prime righe di tutti gli strati, la seconda fetta è composta dalle seconde righe di tutti gli strati e così via. Chiaramente gli elementi di una fetta non sono allocati in modo contiguo nella memoria come lo sono invece quelli degli strati. Si chiede di realizzare una funzione `int G(int* A, int limite1, int limite2, int limite3, int n_ele)` che soddisfi i seguenti punti:
- i) G riceve il puntatore al primo elemento dell'array A (che ha infatti tipo `int*` in questo caso), ma deve interpretare l'array come un array a 3 dimensioni `int [limite1] [limite2] [limite3]` che ha i primi `n_ele` elementi definiti (e gli eventuali altri indefiniti).
 - ii) G deve restituire l'indice della fetta la somma dei cui elementi sia massima in A . Quindi è lo stesso esercizio del punto (1), con la differenza che si devono considerare fette anziché strati. In particolare, come in (1), la somma degli elementi di una fetta deve considerare solo gli elementi definiti di A .
 - iii) Il caso in cui ci siano più fette a somma massima si risolve come spiegato in (1)(iii).

8.1.1 Soluzione del Problema (1)

In fondo si tratta di un problema di calcolo di un massimo e abbiamo già visto vari esempi di questo tipo. Si tratta di fare la somma dei diversi strati e conservare in ogni momento l'indice dello strato a somma massima incontrato fino a quel momento. C'è una (piccola) complicazione dovuta al fatto che l'array non è in generale completo, ma sono definiti solo i suoi primi `n_ele` elementi. A causa di questo fatto, in generale si dovrà calcolare la somma di alcuni strati completi, dopo di che ci si dovrà occupare di quello incompleto. Nel programma che segue, viene calcolato immediatamente il numero degli strati completi dell'array e verrà calcolato lo strato a somma massima tra questi e solo successivamente verrà considerato l'eventuale ultimo strato incompleto.

```

//PRE = (n_ele >= 0 e i primi n_ele elementi di A sono definiti)
int F(int (*A)[5][10], int limite1, int n_ele)
{
    int n_sp=n_ele/(5*10), n_eu=n_ele%(5*10);
    // n_sp=n. strati completi e n_eu = n. elementi ultimo strato

    int sommaM=INT_MIN, indice=0;

    for(int i=0; i<n_sp; i++)
    //R1=(se i=0 => sommaM=INT_MIN
    //se i>0 => A[indice] ha sommaM che è massima tra gli strati
    //A[0..i-1])&&(0<=i<=n_sp)
    {
        int S=somma(*(A+i));
        if(S > sommaM)
        {
            sommaM=S;
            indice=i;
        }
    }
    // POST1=(A[indice] ha sommaM che è massima tra
    //gli strati A[0..n_sp-1] completi)
    // consideriamo ora lo strato incompleto
    if(n_eu >0)
    {
        int S=somma_inc(*(A+n_sp), n_eu);
        if(S>sommaM)
            indice=n_sp;
    }
    //(A[indice] ha sommaM che è massima tra gli strati A[0..n_sp],
    //contando solo gli elementi definiti)
    return indice;
}
//POST=(A[indice] ha somma degli elementi definiti massima tra
//gli strati che contengono qualche elemento definito)

```

Per prima cosa nel precedente programma si deve osservare che il calcolo della somma degli strati pieni è affidato ad una funzione `somma` separata da `F` e similmente anche la somma dello strato finale incompleto è realizzata dalla funzione `somma_inc` (dove `inc` sta per incompleto). Conviene sempre organizzare il proprio programma in funzioni ognuna delle quali deve avere un compito ben chiaro. È comunque buona norma verificare che ogni funzione abbia un compito non troppo semplice in modo da evitare un'eccessiva frammentazione del programma.

La dimostrazione di correttezza di *F* segue la **ricetta di indicizzazione** presentata nel Capitolo 6. L'invariante **R1** del primo ciclo di *F* è ottenuto indicizzando la relativa postcondizione **POST1**. Quindi la prova della condizione d'uscita del ciclo è vera per costruzione.

È facile inoltre convincersi che (se *somma_inc* è corretta) allora viene considerato anche l'ultimo strato incompleto e quindi *F* sembra corretta. Usiamo *sembra* perché in realtà esiste un caso limite in cui la funzione potrebbe non essere corretta. È necessario considerare il caso in cui la somma di tutti gli strati (non vuoti) sia *INT_MIN*. In questo caso il valore di *sommaM* resta *INT_MIN* e quello di *indice* resta 0 ed è giusto, visto che lo strato 0 ha somma *INT_MIN* (come tutti gli altri strati). Quindi non è questo caso limite che ci mette in crisi, ma è invece il caso in cui *n_ele* sia 0. Infatti in questo caso la nostra *F* dà esattamente la stessa risposta del caso precedente. Insomma non riesce a distinguere i 2 casi. In realtà il caso in cui *n_ele* è uguale a 0 è un caso in cui il problema non ha senso e quindi potremmo escluderlo modificando la preconditione. Un'altra possibilità è di stabilire che in questo caso la funzione restituisca *indice* con valore -1, che non essendo un indice possibile per uno strato, identifica in modo univoco questo caso. Per ottenere questo comportamento basta introdurre all'inizio di *F* il test `if (n_ele==0) indice=-1;`. Ovviamente in questo caso la **POST** dovrebbe essere estesa (in modo ovvio) per considerare anche questo possibile output.

Esaminiamo adesso le due funzioni *somma* e *somma_inc*. Non presentano difficoltà particolari. Si osservi che *somma_inc* usa la stessa tecnica applicata in *F*: uno strato incompleto sarà costituito da alcune righe complete seguite da una incompleta (e le successive righe, se ci sono, sono completamente vuote e quindi vanno trascurate). Il numero di righe complete viene calcolato con la divisione intera tra il numero totale degli elementi presenti e la lunghezza di ogni riga, cioè 10, mentre il numero degli elementi sull'ultima riga è semplicemente il resto di questa stessa divisione. Si noti che il codice di *somma_inc* è corredato da invarianti e postcondizioni che soddisfano tutti la semplice **ricetta** del Capitolo 6. Per la più semplice funzione *somma* viene specificata solo la postcondizione.

```
int somma(int (*S)[10])
{
    int somma=0;
    for(int i=0; i<5; i++)
        for(int j=0; j<10; j++)
            somma+=S[i][j];
    return somma;
} //(somma=somma di S[5][10])

int somma_inc(int (*S)[10], int ne)
{
    int nrp=ne/10, somma=0, eur=ne%10;
```

```

for(int i=0; i<nrp; i++)
//R=(somma=somma delle righe S[0..i-1])&& (0<=i<=nrp)
{
    for(int j=0; j<10; j++)
        somma+=S[i][j];
}
// (somma=somma delle righe S[0..nrp-1]) complete
for(int i=0; i<eur; i++)
    somma+=S[nrp][i]
// (somma=somma degli elementi definiti di righe S[0..nrp])
return somma;
}

```

È interessante notare che, visto il modo in cui il C++ alloca ogni strato dell'array (come sequenza di righe contigue ognuna delle quali è una sequenza contigua di elementi), in fondo entrambe le funzioni fanno la stessa cosa: dato un indirizzo della memoria, sommano un certo numero di interi che sono allocati a partire da quell'indirizzo. Quindi, potremmo fare entrambe le operazioni con una stessa funzione come segue:

```

int T(int* A,int quanti)//Osservare il tipo di A !!
{
    int somma=0;
    for(int i=0;i<quanti;i++)
        somma=somma+A[i];
    return somma;
}

```

Ovviamente le attuali invocazioni in F dovrebbero venire modificate. La prima deve diventare $S=T(**(A+i), 50)$. La trasformazione della seconda invocazione è lasciata come esercizio.

C'è un'ultima considerazione da fare su questa soluzione. Molti programmatori vedendola potrebbero pensare (e generalmente lo fanno) che sia inutile distinguere il trattamento degli strati pieni da quello dell'ultimo strato incompleto e, di conseguenza, cercherebbero di inglobare le due cose in un unico ciclo. Questo generalmente significa che questo ciclo deve, ad ogni iterazione, testare se si è arrivati all'ultimo strato in modo da trattarlo appropriatamente. Una tale soluzione oltre ad essere poco elegante, introdurrebbe test inutili, e probabilmente sarebbe meno chiara di quella proposta e quindi, nella migliore delle ipotesi, dimostrare la sua correttezza sarebbe più difficile, e nella peggiore, sarebbe semplicemente sbagliata. Insomma, in coerenza col sottotitolo del libro *semplice è bello*, l'approccio che consigliamo sempre è quello più semplice.

8.1.2 Soluzione del Problema (2)

Specifichiamo innanzitutto la pre- e la postcondizione della funzione G richiesta.

PRE=(A punta al primo elemento di $\text{limite1}*\text{limite2}*\text{limite3}$ elementi, dei quali solo i primi n_ele sono definiti, con $n_ele > 0$).

POST=(G restituisce l'indice di una fetta che ha somma (degli elementi definiti) di valore maggiore o uguale rispetto alla somma (dei valori definiti) delle altre fette con almeno un valore definito).

Vediamo subito alcuni punti rilevanti sulla costruzione di G :

- (a) Per prima cosa si noti che nella **PRE**, per semplicità, si assume che $n_ele > 0$, cioè che nell'array ci sia almeno un elemento definito. Se volessimo invece gestire anche il caso $n_ele = 0$, potremmo farlo allo stesso modo dell'Esercizio precedente.
- (b) È importante notare che sarebbe un errore cercare di calcolare la somma di fette che non contengono alcun elemento definito. Infatti, se lo facessimo, rischieremmo che la somma di tali fette risulti uguale a 0 e, nel caso che le fette valide abbiano somma negativa, rischieremmo di considerare massima una fetta che in realtà non dovrebbe neppure venire considerata. Per risolvere il problema, prima di iniziare a considerare una nuova fetta possiamo semplicemente controllare se la fetta contiene almeno un valore definito. Per farlo basta controllare che la posizione del primo elemento della fetta sia minore di n_ele . Questo controllo viene realizzato dalla condizione, $f*\text{limite3} < n_ele$ nel test di permanenza del ciclo che scandisce le fette.
- (c) La funzione dovrà essere costituita da 3 cicli annidati, dove il primo scandisce le fette (cioè le righe di ogni strato), il secondo scandisce gli strati ed il terzo scandisce le colonne delle righe su ogni strato. Gli invarianti di questi cicli sono facili e seguono la **ricetta di indicizzazione** del Capitolo 6. Per esempio l'invariante del ciclo più esterno che scandisce le fette può essere ricavato facilmente dalla **POST**. Se f è l'indice di questo **for**, allora **R1**=(**conosco l'indice della fetta a somma massima tra le fette da 0 a f-1**)&& (**$0 \leq f \leq \text{limite2}$**). Gli invarianti dei cicli più interni sono altrettanto semplici.
- (d) Per ogni fetta dobbiamo considerare solo gli elementi definiti. Il modo di farlo è semplice. Sappiamo che solo i primi n_ele elementi di A sono definiti, cioè $A[0] \dots A[n_ele-1]$. Quindi, se stiamo considerando un elemento della fetta f , strato s e colonna c , ci basta calcolare la sua posizione tra gli elementi di A , cioè $\text{pos} = s * \text{limite2} * \text{limite3} + f * \text{limite3} + c$ e poi chiederci se $\text{pos} < n_ele$ oppure no. Solo se $\text{pos} < n_ele$ l'elemento che stiamo considerando è definito e va quindi sommato alla somma della fetta che stiamo calcolando.

La funzione G che risponde ai punti appena elencati, è la seguente.

```

int G(int* A, int limite1, int limite2, int limite3, int n_ele)
{
    int sommaM=INT_MIN, indice=0;

    for(int f=0; f<limite2 && f*limite3 < n_ele; f++)//f=fette

//R1=(la fetta indice ha somma (degli elementi definiti)
//pari a sommaM che è massima tra le fette da 0 a f-1
//ognuna con almeno un elemento definito)&&(0<=f<=limite2)
    {
        int somma=0;
//la fetta f contiene almeno un elemento definito
        for(int s=0; s<limite1; s++) // s = strato

//R2=(somma è la somma (degli elementi definiti) delle prime
//s-1 righe della fetta f-sima) //&&(0<=s<=limite1)
        {
            int sommar=0;
            for(int c=0; c<limite3 ; c++) // c = colonna

//(sommar è la somma degli elementi definiti di A[s][f][0..c-1] )
            if(s*(limite2*limite3)+f*limite3+c < n_ele)
// condizione da soddisfare per essere definito
                sommar+=A[s*(limite2*limite3)+f*limite3+c];
//uscita da ciclo c, vale (sommar=somma degli elementi
//definiti della riga f dello strato s)
                somma+= sommar;
//somma=somma degli elementi definiti della fetta f fino
//allo strato s) con s++ vale ancora R2
        }
//uscita dal ciclo s, vale (somma = somma degli elementi definiti del
//fetta r e ce n'è almeno //uno definito, quindi è una somma valida)
        if(somma > sommaM)
        {
            sommaM=somma;
            indice=f;
        }
    }
//la fetta indice ha somma (degli elementi definiti)
//pari a sommaM che è massima
//tra le fette da 0 a f ognuna con almeno un elemento)&&
//(0<=f<limite2), con f++ vale nuovamente R1

```

```

    }
//POST
    return indice;
}

```

Chiariamo subito che questa funzione fa delle cose inutili, perché, una volta che inizia a considerare una fetta (cioè quando la fetta contiene almeno un elemento definito), allora considera sempre tutti gli elementi della fetta, mentre potrebbe fermare il calcolo della somma della fetta non appena incontra un elemento indefinito, in quanto è facile constatare che anche tutti i successivi elementi della fetta saranno indefiniti. Comunque consideriamo per il momento questo programma un po' naïf e vediamo se è corretto. Il fatto che i 3 cicli scandiscano le fette, con almeno un elemento definito, e calcolino per ciascuna la somma degli elementi definiti è facile: essi scandiscono tutti gli elementi delle fette (con almeno un valore definito) e per ogni elemento viene controllato se è definito o no e solo nel primo caso il suo valore viene sommato. Inoltre il fatto che solo fette con almeno un elemento definito vengono considerate, implica che per ciascuna di esse la somma calcolata è valida, cioè non è lo 0 assegnato inizialmente a somma. Dobbiamo ora convincerci che la condizione $f * \text{limite3} < n_ele$ garantisca, se soddisfatta, che la fetta f ha almeno un elemento definito. Basta osservare che $f * \text{limite3}$ è la posizione del primo elemento della fetta f e se esso è definito sappiamo che la fetta f contiene almeno lui, mentre se non è definito, allora nessun altro elemento della fetta sarà definito, visto che essi hanno posizione successiva a $f * \text{limite3}$ nell'allocazione di A . Consideriamo per finire la condizione d'uscita dal ciclo f : **R1** $\&\& !(f < \text{limite2} \ \&\& f * \text{limite3} < n_ele) = \text{R1} \ \&\& (f == \text{limite2} \ || \ f * \text{limite3} \geq n_ele) \Rightarrow$ **POST**. Dobbiamo considerare separatamente i diversi casi di uscita dal ciclo:

- (i) Se $f == \text{limite2}$, la **R1** ci dice che *(la fetta indice ha somma (degli elementi definiti) pari a sommaM che è massima tra le fette da 0 a limite2-1 (cioè tutte), ognuna con almeno un elemento definito)*. Da questo segue la **POST**.
- (ii) Se invece usciamo dal ciclo perché $f * \text{limite3} \geq n_ele$, allora **R1** ci dice che *(la fetta indice ha somma (degli elementi definiti) pari a sommaM che è massima tra le fette da 0 a f-1 ognuna con almeno un elemento definito)* e la condizione $f * \text{limite3} \geq n_ele$, ci dice che la prossima fetta f non ha elementi definiti e quindi non va considerata e, visto che anche le successive fette saranno nella stessa condizione, *indice* è il valore giusto. Quindi vale la **POST**.

Quindi il nostro programma naïf è (parzialmente) corretto. Vediamo di migliorarlo evitando di fargli fare operazioni inutili. L'osservazione da fare è semplice. Visto che percorriamo ogni fetta (per calcolarne la somma) scorrendo gli strati in modo crescente, allora quando in questo calcolo, raggiungiamo un elemento indefinito, anche i successivi elementi di quella fetta saranno indefiniti. Inutile quindi

continuare il calcolo. Per implementare questa idea, basta introdurre un booleano che ci permetta di uscire dai cicli con indici *c* e *s* non appena incontriamo un elemento indefinito. Nel seguito mostriamo questi 2 cicli modificati:

```
int somma=0; bool fine_fetta=false;

for(int s=0; s<limite1 && !fine_fetta; s++) // s = strato
{
    int sommar=0;
    for(int c=0; c<limite3 && !fine_fetta; c++) // c = colonna

        if(s*(limite2*limite3)+f*limite3+c < n_ele)
            sommar+=A[s*(limite2*limite3)+f*limite3+c];
        else
            fine_fetta=true;

    somma+= sommar;
}
```

Visto che le nuove condizioni di uscita sui 2 cicli non fanno altro che interrompere calcoli che non davano alcun contributo ai valori di *sommar* e quindi di *somma* che erano calcolati, il nuovo programma calcola esattamente la stessa cosa del precedente. Quindi è anch'esso (parzialmente) corretto.

Il lettore potrebbe chiedersi perché la funzione *G* non usa un'altra funzione per calcolare la somma di ciascuna fetta (come faceva la funzione *F* dell'Esercizio precedente per gli strati), ma procede a fare tutti i calcoli localmente con tre cicli annidati. Il motivo di questa scelta è che gli elementi delle fette non sono allocati in posizioni contigue nella memoria (come sono invece gli elementi degli strati) e quindi per accedere agli elementi di una fetta *c*'è bisogno di tutti e tre gli indici *f*, *s* e *c*. Questo fatto implica che l'eventuale funzione che calcolasse la somma di una fetta dovrebbe ricevere sempre l'intero array *A* assieme all'indice *f* del ciclo che scorre le fette (che resterebbe in *G*). Quindi una tale funzione sarebbe comunque legata alla funzione chiamante *G*, a causa dell'indice *f* e del fatto che necessita dell'intero array (e non solo di un suo strato o di una riga). Queste considerazioni ci hanno fatto optare per non definire una funzione ausiliaria. Naturalmente sarebbe accettabile anche fare la scelta opposta.

8.2 Problemi (3) e (4)

In un problema di pattern matching abbiamo un testo *T*, per esempio `char T[1000]`, ed un pattern *P*, per esempio, `char P[dim_P]`, ed esaminiamo *T* da sinistra a destra cercando una porzione di *dim_P* elementi contigui di *T* che siano identici agli elementi di *P*. Più precisamente, cerchiamo un *k* tale che `T[k..k+dim_P-1] = P[0..dim_P-1]`. Quando questo si verifica diremo di avere trovato un **match** di

P in T e diremo che questo match comincia nella posizione k di T. Due match di P in T, che iniziano nelle posizioni k_1 e k_2 di T, sono **sovrapposti**, se gli intervalli $[k_1, k_1 + \dim_P - 1]$ e $[k_2, k_2 + \dim_P - 1]$ hanno qualche indice in comune e altrimenti sono detti **non sovrapposti**. Nel seguito proponiamo due versioni diverse di pattern matching:

- (3) Si chiede di scrivere una funzione `char* F(char (*T)[10][15], int limite1, char P[], int dim_P)` che soddisfi i seguenti punti:
 - (i) `limite1` è il limite della prima dimensione di T, cioè T è un array `char T[limite1][10][15]` ed è completamente definito;
 - (ii) Per ogni strato `T[i]` e per ogni sua riga `T[i][j]`, F deve calcolare il numero N_j di match di P (anche sovrapposti) sulla riga. Poi deve calcolare la somma $N_0 + N_1 + \dots + N_9 = M_i$, dove M_i è il **valore dello strato** `T[i]`. E' importante osservare che un match su una riga deve essere completamente contenuto sulla riga. Non può continuare alla riga successiva;
 - (iii) F deve restituire il puntatore al primo elemento dello strato di T che ha valore massimo (in caso ci siano più strati a valore massimo, basta restituire il puntatore ad uno qualsiasi di questi strati);
- (4) Si tratta di scrivere una funzione `char* G(char (*T)[10][15], int limite1, char P[], int dim_P)` per i cui parametri vale quanto detto in (3) e che soddisfi i seguenti punti:
 - (i) Per ogni strato `T[i]`, G deve calcolare il numero di match di P non sovrapposti che esistono su `T[i]`. Si osservi che, a differenza dell'esercizio (3), qui un match può iniziare su una riga `T[i][j]` e continuare sulla riga successiva `T[i][j+1]`, a condizione che $j+1 < 10$, cioè che la riga $j+1$ sia ancora una riga dello strato `T[i]`; il numero totale di questi match è il **valore dello strato** `T[i]`;
 - (ii) G deve restituire il puntatore al primo elemento dello strato di valore massimo (uno qualsiasi se ce n'è più di uno con valore massimo)

8.2.1 Soluzione del Problema (3)

Conviene sviluppare oltre a F alcune funzioni ausiliarie che permettono di mantenere F più chiara. Abbiamo la funzione `match` che conta il numero di match di P su una riga di T e che, per farlo, usa la funzione `MT` la quale, dato un array T, stabilisce se c'è un match completo di P che inizia in `T[0]`. Le funzioni seguono. Esse sono cordate da pre- e postcondizioni ed invarianti. Gli invarianti sono ricavati dalle relative postcondizioni applicando la **ricetta di indicizzazione** del Capitolo 6. Quindi la prova di correttezza vale (praticamente) per costruzione. C'è solo un

dettaglio sul quale conviene soffermarsi. Il test di permanenza del ciclo della funzione `match` è più complicato di quanto si potrebbe in un primo momento pensare. Il test è $i < \text{lim} - \text{dim_P} + 1$, perché solo i che soddisfano questa condizione sono dei possibili punti di partenza di `match`. Infatti per i maggiori di $\text{lim} - \text{dim_P}$, la porzione di array che resta per tentare il match è più corta del pattern P e quindi il match certamente fallirebbe. Ovviamente, questo ragionamento servirà quando dimostreremo che all'uscita del ciclo vale la postcondizione. Vediamo. Dobbiamo dimostrare che $R \wedge i = \text{lim} - \text{dim_P} + 1 \Rightarrow \text{POST}$. Per dimostrare che vale la **POST** dovremo dimostrare che gli indici da $\text{lim} - \text{dim_P}$ a $\text{lim} - 1$ che non vengono considerati dal ciclo, non possono dare ulteriori match (che non sarebbero contati in `nmatch`) e questo segue dall'osservazione fatta in precedenza.

```
bool MT(char *, char *, int);
```

```
int match(char* T, int lim, char *P, int dim_P)
{
    int nmatch=0;
    // N.B. basta considerare inizi fino a lim-dim_P
    for(int i=0; i< lim-dim_P+1; i++)
    //R=(nmatch=n. match di P con inizio in T[0..i-1])
        if(MT(T+i, P, dim_P))
            nmatch++;
    return nmatch;
} //POST=(nmatch=n. match di P in T[0..lim-1])
```

```
bool MT(char *T, char*P, int dim_P)
{
    bool ok = true;
    for(int i=0; i<dim_P && ok; i++) //R=(ok <=> T[0..i-1]==P[0..i-1])
        if(T[i]!=P[i])
            ok=false;
    return ok;
} //(ok <=> T[0..dim_P-1]=P[0..dim_P-1])
```

```
char * F(char (*T)[10][15], int limite1, char*P, int dim_P)
{
    int nmax=0;
    char* Q=**T;
    for(int i=0; i<limite1; i++) // i = strati
    // nmax=n. match di P massimo sugli strati 0..i-1,
    //Q punta al primo elemento di quello strato
    {
        int n=0;
        for(int j=0; j<10; j++) // j = righe
```

```

        n+=match(T[i][j],15, P, dim_P);
// n=n. match di P nello strato i

        if (n>nmax)
        {
            nmax=n;
            Q=T[i][0];
        }

    }
    return Q;
//POST=(Q punta al primo elemento di uno strato con
//massimo n. di match)
}

```

8.2.2 Soluzione del Problema (4)

Visto che in questo esercizio il match può andare accapo tra le righe di uno stesso strato, conviene considerare uno strato come un unico array ad una dimensione di 150 elementi. Se considerassimo ancora matching potenzialmente sovrapposti, potremmo usare ancora la funzione `match` vista prima. Per soddisfare la richiesta di considerare match non sovrapposti, cambiamo `match` nella funzione `matchG` che considera i vari punti di partenza su `T`, ma anziché ripartire dalla posizione successiva indipendentemente da come vada l'invocazione di `MT`, qualora `MT` abbia successo, sposta il prossimo punto di partenza di `dim_P` posizioni. Che questo *salt* serva ad evitare di trovare match sovrapposti, mi pare intuibile. Vediamo come funziona la dimostrazione di correttezza di `matchG`. La postcondizione è simile a quella di `match` con la sola modifica di contare match non sovrapposti. Il fatto che all'uscita del ciclo, `R` e la condizione `i=lim-dim_P+1` implichino la postcondizione, si dimostra esattamente come per la funzione `match`, vedi Soluzione (3). Può destare qualche perplessità il fatto che `R` sia effettivamente invariante a causa dell'incremento *saltellante*. Supponiamo che `R && (i<lim-dim_P+1)`. Nel corpo del ciclo si tenta un match a partire da `T[i]`. Se il match fallisce, `inc = 1` e quindi `R` ridiventa vera prima di riprovare di nuovo il test di permanenza. Se invece il match ha successo allora `nmatch++`; `incr = dim_P`; quindi, contabilizziamo il nuovo match trovato, ma a causa dell'incremento, l'asserzione iniziale diventa (`nmatch=n. match non sovrapposti di P con inizio in [0..i-dim_P]`). Per rendere di nuovo vero `R`, dobbiamo dimostrare che tutti gli inizi da `i-dim_P+1` a `i-1` non sono inizi di match non sovrapposti ai precedenti. Questo fatto è ovvio visto che tutte queste posizioni di `T` fanno parte del match appena trovato. Quindi, anche in questo caso `R` risulta vero quando si torna a riprovare il test di permanenza nel ciclo.

```
int matchG(char* T, int lim, char *P, int dim_P)
```

```

{
    int nmatch=0, incr=1;

    for(int i=0; i< lim-dim_P+1;i=i+incr)

//R=(nmatch=n. match non sovrapposti di P con inizio in T[0..i-1])

        if (MT(T+i,P,dim_P))
        {
            nmatch++;
            incr=dim_P;
        }
        else
            incr=1;
    return nmatch;
} //(nmatch=n. match non sovrapposti di P in T[0..lim-1])

char * G(char (*T) [10] [15], int limite1, char*P, int dim_P)
{
    int nmax=0;
    char* Q=**T;
    for(int i=0; i<limite1; i++)

// nmax=n. match non sovrapposti di P massimo sugli strati 0..i-1,
//Q punta al primo elemento di quello strato
    {
        int n=0;
        n+=matchG(T[i] [0],150, P, dim_P);
        // n=n. match di P nello strato i (con match anche su più righe
        if (n>nmax)
        {
            nmax=n;
            Q=T[i] [0];
        }
    }
    return Q;
} //POST=Q punta al primo elemento di uno strato con massimo n. di mat

```


8.3 Problema (5)

Si tratta di realizzare una funzione `void F(int *A, int & top, int y)` che soddisfi la seguente pre- e postcondizione:

PRE $= (A[0..top-1]$ **è ordinato in modo non decrescente e** $top \geq 0$ **e** y **definito)** $\&\& (A[0..top-1] = [p_1, \dots, p_k, u_1, \dots, u_q, g_1, \dots, g_w],$ **con** $k+q+w = top, k, q, w \geq 0, \forall j \in [1, k], p_j < y, \forall j \in [1, q], u_j = y, \forall j \in [1, w], g_j > y)$

POST $= (A[0..top-1]$ **è uguale a** $[p_1, \dots, p_k, g_1, \dots, g_w]).$

Si osservi che la postcondizione implica che alla fine della funzione, $top = k + w$ e che l'array $A[0..top-1]$ è ordinato e non contiene y . In programmazione è molto importante dividere un problema complesso in sottoproblemi più semplici. In questo modo il programma diventi una collezione di funzioni con un compito molto chiaro (specificato da pre- e postcondizioni) e la cui correttezza sia ragionevolmente facile da dimostrare. Per illustrare questa maniera di procedere organizziamo F in modo che usi 2 altre funzioni ausiliarie il cui compito è come segue:

- una funzione che cerca y in $A[0..top-1]$ e restituisce se l'ha trovato, dove l'ha trovato e quante volte y appare in $A[0..top-1]$;
- una funzione che elimina da A tutte le copie di y ;

8.3.1 Soluzione del Problema (5)

Nel seguito viene descritto un programma che risolve il Problema (5). Le funzioni dei punti (a) e (b) del Problema (5) sono *cerca* e *shift*, rispettivamente. Ad ogni ciclo è associato un invariante e le pre- e postcondizioni delle funzioni sono specificate. Visto che *cerca* ha 2 compiti, essa è organizzata in 2 cicli separati. Sarebbe possibile svolgere entrambi i compiti con un solo ciclo, ma diverrebbe più complicato e quindi anche la sua correttezza diventerebbe difficile da dimostrare. Questa maniera di procedere è in accordo con la divisione dei problemi in sottoproblemi più semplici.

```
//PRE= la preconditione di F descritta prima
bool cerca(int*A, int top, int y, int & start, int & quanti)
{
    bool trovato=false;
    for(int i=0; i< top && A[i]<=y && !trovato; i++)
/*R=(per ogni j in [0,i-2], A[j]<y)&&
(trovato => start=i-1 && A[i-1]=y)&&
(!trovato => A[0..i-1]<y) && (0<=i<=top)*/

    if (A[i]==y)
    {
        trovato=true;
```

```

        start=i;
    }
/*POST=(trovato => start=k) &&
( !trovato=> per ogni j in [0,top-1], A[j]!=y)*/

    if(trovato)
    {
        quanti=1;
        for(int i=start+1; i<top && A[i]==y;i++)
/*R1=(per ogni j in [start,i-1], A[j]=y)&&(quanti = i-start)*/
            quanti++;
    }
/*POST1=(trovato => quanti=q)*/

}
return trovato;
}
/*POST=(trovato => (start = k && quanti=q && (0<=k<top && q>0))
&& (!trovato => per ogni j in [0..top-1] A[j]!=y) */

/*PRE=(A[0..top-1] definito && 0<=start<=start+quanti-1<top)
&& ( vA e vtop sono il valore iniziale di A e top) */
void shift(int *A, int& top, int start, int quanti)
{
    for(int i=start; i<top-quanti; i++)
/*R=(A[0..start-1]=vA[0..start-1] &&
A[start..i-1]=vA[start+quanti..i+quanti-1]) &&
(A[i..top-1]=vA[i..top-1])&&(start<=i<=top-quanti) */
        A[i]=A[i+quanti];

/*(A[0..start-1]=vA[0..start-1] &&
A[start..vtop-quanti-1]=vA[start+quanti..vtop-1]) */
    top-=quanti;
}
/*POST=(A[0..start-1]=vA[0..start-1] &&
A[start..vtop-quanti-1]=vA[start+quanti..vtop-1]
&& top=vtop-quanti) */

void F(int*A, int &top, int y)
{
    int start, quanti;
    if(cerca(A,top, y, start,quanti))

```

```

    shift(A, top, start, quanti);
}

```

Visto che la preconditione di *F* stabilisce completamente com'è fatto $A[0..top-1]$, possiamo essere molto semplici e precisi nella postcondizione di *cerca*. È facile vedere che all'uscita del primo ciclo di *cerca*, se *trovato* è vero, allora $start=k$. Infatti *R* in questo caso implica che $A[start]=y$ e che $A[0..start-1]<y$ e quindi $start=k$. Il secondo ciclo, inizia da $start+1$ e termina non appena finisce l'array o quando si trova $A[i]>y$. Visto che l'array è ordinato, questo significa che $i=k+q$ e visto che *R2* stabilisce che $i-start=quanti$, ne segue che $k+q-k=quanti$ e quindi $q=quanti$, come volevamo. A questo punto basta mettere assieme le dimostrazioni dei due cicli per dimostrare che se *trovato* è falso, il secondo ciclo non viene neppure eseguito e si arriva alla fine con $!trovato \ \&\& \ \forall j \in [0, top-1], A[j] \neq y$. Se invece *trovato* è vero, alla fine di *cerca* abbiamo che $start=k$ e $quanti=q$.

Anche in questo caso, coerenti con l'approccio *semplice è bello*, nella funzione *cerca* abbiamo calcolato *start* con un ciclo e *quanti* con un secondo ciclo, perché in questo modo è più facile dimostrare la correttezza di ciascuno dei 2 e quindi dell'intera funzione. Era ovviamente possibile calcolare entrambe le quantità *start* e *quanti* con un solo ciclo. L'invariante di questo ciclo unico sarebbe l'unione degli invarianti dei nostri due cicli e quindi sarebbe più complesso. Perciò abbiamo preferito la soluzione con due cicli.

Consideriamo ora la funzione *shift*. È importante osservare che la sua pre- e postcondizione sono le più generali possibile. Esse asseriscono che, dati *start* e *quanti* che individuano un intervallo di indici contiguo contenuto in $[0..top-1]$, la funzione elimina i valori che si trovano in quelle posizioni, spostando i valori che seguono quelli eliminati di *quanti* posizioni verso sinistra. Questa operazione viene fatta indipendentemente dai valori contenuti nella porzione di array che viene eliminata. Una volta dimostrato che *shift* è corretta rispetto a questa pre- e a questa postcondizione, il fatto che in *F*, invocando *shift*, si eliminano tutti e soli gli *y* producendo un array ancora ordinato in modo non decrescente, segue dalla postcondizione di *cerca* (che garantisce che *start* e *quanti* individuano tutti gli *y* in *A*), assieme al fatto che $A[0..top-1]$ è inizialmente ordinato in modo non decrescente.

Per quanto riguarda la dimostrazione di correttezza della *shift*, è bene notare che l'invariante *R* del suo ciclo non è ottenuto completamente applicando la **ricetta di indicizzazione** alla postcondizione della funzione. La differenza è la condizione $(A[i..top-1]=vA[i..top-1])$ che serve a provare l'invarianza di *R* in quanto garantisce che ogni $A[i]$ riceve $vA[i+quanti]$. Basta riflettere sul significato della condizione mancante per capire il perché di questa situazione. All'uscita del ciclo $(A[i..top-1]=vA[i..top-1])$ diventa $(A[top-quanti..top-1]=vA[top-quanti..top-1])$, cioè asserisce che i *quanti* elementi finali di *vA* si trovano ancora in *A* nelle stesse posizioni.

Visto che questi ultimi elementi del vecchio `va` non sono più elementi del nuovo `A`, che infatti si è accorciato di `quanti` elementi, è ovvio che questa condizione non sarebbe di alcun interesse nella postcondizione di `shift`.

Capitolo 9

Estensioni del C++

In questo Capitolo presenteremo diversi nuovi e importanti concetti e costrutti del linguaggio C++.

9.1 Costanti

In Sezione 2.2 abbiamo visto che nei programmi è possibile dichiarare delle costanti, come per esempio in `const double pigreco=3.14;`. Un punto importante è che le costanti vanno sempre inizializzate al momento della loro dichiarazione. Ora verranno considerati aspetti avanzati dell'uso delle costanti, come i puntatori costanti, o i puntatori a costanti o ancora l'utilità di parametri formali costanti nelle funzioni. Un esempio della dichiarazione di un puntatore ad una costante `double` è il seguente: `const double pigreco=3.14, *punt_pigreco=&pigreco;`. È importante capire bene cosa viene dichiarato da questa istruzione: `pigreco` è costante, mentre `punt_pigreco` punta ad una costante, ma non è costante, cioè il suo R-valore può cambiare. Va osservato che `const double pigreco=3.14; double *punt_pigreco=&pigreco;` darebbe un errore in compilazione. La dichiarazione infatti specifica che `punt_pigreco` punta ad una variabile (quindi non costante) di tipo `double` e quindi potremmo usare questo puntatore per modificare la variabile cui punta. Questo sarebbe un errore, visto che esso punta ad una costante. Quindi la dichiarazione è semplicemente rifiutata dal compilatore. Al contrario, le seguenti dichiarazioni, `int y=3; const int* p=&y;` sono accettate dal compilatore. Questa apparente incoerenza diventa assolutamente ragionevole se si considera che la specifica `const` protegge l'oggetto a cui è applicata nel senso che, con questa specifica, l'oggetto non può essere modificato. Quindi un `const` non può venire cancellato (come avviene nelle dichiarazioni `const double pigreco=3.14; double *punt_pigreco=&pigreco;`), mentre è sempre possibile aggiungere un `const` (come avviene in `int y=3; const int* p=&y;`). È importante capire che questa dichiarazione concede di modificare l'R-valore di `y` e

anche quello di `p`, ma non consente di 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 oppure degli array passati 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 per il corrispondente parametro formale il seguente tipo: `const char * A` oppure `const char A[]`. In questo caso, se la funzione contenesse istruzioni che modificassero qualche elemento di `A`, come per esempio `*(A+i)=exp`; oppure `A[i]=exp`; il compilatore darebbe un errore e il programma non compilerebbe. 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 R-valore `V` di tipo `T` come corrispondente parametro attuale (anziché un L-valore di tipo `T`). 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`. La situazione appena descritta è preferibile a quello che succederebbe se usassimo la maniera "giusta" di passare un valore `V`, cioè il passaggio per valore, in quanto, col passaggio per valore, andrebbero costruite due copie di `V` e questo potrebbe essere oneroso qualora `V` occupasse molto spazio nella memoria. Il risparmio di memoria è rilevante nel caso in cui sia necessario passare alle funzioni degli oggetti (nel senso di istanze di classi) di grandi dimensioni e infatti questa tecnica è molto comune nei programmi C++ che usano gli oggetti.

9.2 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 si 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 dai confini di questo testo.

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 lunedì. Evidentemente questi dubbi non ci sarebbero se potessimo usare direttamente le costanti `lunedì`, `martedì`, `mercoledì`, `giovedì`, `venerdì`, `sabato`, `domenica`. Questo è possibile attraverso la seguente dichiarazione di tipo: `enum Giorno {lunedì, martedì, mercoledì, giovedì, venerdì, sabato, domenica};`. Questa dichiarazione introduce un nuovo tipo `Giorno` dando esplicitamente i valori del tipo. Questi valori sono `lunedì`, `martedì`, `mercoledì`, `giovedì`, `venerdì`, `sabato` e `domenica` che sono quindi costanti (anche dette **token**) utilizzabili nel programma. Dopo questa dichiarazione potremo definire variabili di tipo `Giorno` che potranno assumere come R-valore uno dei token elencati per il tipo `Giorno`. Per esempio, la seguente può essere una dichiarazione con inizializzazione corretta: `Giorno g=lunedì;`. 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: `{lunedì=10, martedì=1, mercoledì=2, giovedì=9, venerdì=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 << lunedì <<endl;` produce "solo" la stampa dell'intero corrispondente al token `lunedì`. 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 perché 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 definiti dall'utente. 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 quindi gli operatori aritmetici non sono applicabili per ottenere valori di tipo enumerazione. Insomma

i tipi enumerazione hanno dei limiti d'uso importanti e possiamo dire che essi fondamentalmente servono a rendere i programmi più leggibili. Cosa che è comunque da non sottovalutare.

I tipi **struttura** permettono di definire collezioni di valori di tipi possibilmente diversi. Per esempio `struct S{int pippo; char pluto; double dafne;};` introduce un nuovo tipo struttura di nome `S` la cui dichiarazione specifica che ogni valore di tipo `S` consiste di tre componenti che potranno venire accedute attraverso i loro nomi che sono `pippo`, `pluto` e `dafne` e i cui tipi sono `int`, `char` e `double`, rispettivamente. Le tre componenti vengono dette **campi** della struttura `S`. Una volta introdotta la dichiarazione del tipo `S`, esso diventa un tipo con tutti i diritti dei tipi predefiniti. Per esempio, la seguente dichiarazione `S x;` introduce una variabile `x` di tipo `S` i cui R-valori consisteranno delle 3 componenti (campi) specificati nella dichiarazione di `S`, e che vengono indicati rispettivamente con `x.pippo`, `x.pluto` e `x.dafne`. L'inizializzazione dei campi di `x` può essere fatta o esplicitamente con assegnazioni come le seguenti, `x.pippo=1`; `x.pluto='a'`; `x.dafne=1.35`; oppure specificando nella dichiarazione del tipo `S` anche una funzione chiamata **costruttore del tipo** `S` che avrà la seguente forma, `S(int w1=0, char w2=0, double w3=0.0){pippo = w1; pluto = w2; dafne = w3;};`. Si osservi che questa funzione ha lo stesso nome di `S` appunto perché 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`. La definizione del costruttore di un tipo struttura va inserita nella dichiarazione del tipo stesso. Quindi nel caso della struttura `S`, la dichiarazione dovrebbe essere: `struct S{int pippo; char pluto; double dafne; S(int w1=0, char w2=0, double w3=0.0){pippo = w1; pluto = w2; dafne = w3;}};`. 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 presenza dei valori di default nel costruttore, permette di invocarlo con 0, 1, 2 e 3 parametri, cioè le seguenti dichiarazioni sarebbero tutte corrette: `S a`, `b(1)`, `c(2, 's')`, `d(3, 'w', 3.4)`; . Inserire valori di default impedisce spiacevoli interferenze con il costruttore di default che il C++ fornisce automaticamente per ogni tipo definito dall'utente. I tipi struttura verranno usati nei Capitoli 11 e 12 per costruire liste concatenate e alberi binari.

La maniera più semplice per fare l'output di una variabile di tipo struttura è di stampare ogni campo separatamente. Naturalmente una struttura può contenere un campo `C` di un (altro) tipo struttura 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 struttura che consiste nello sfruttare il sovraccaricamento per introdurre una definizione della funzione di output `<<` per il tipo struttura che ci interessa. Una tale definizione per un tipo `struct papero{...}`; ha la seguente forma:

```
ostream & operator<<(const ostream & SS, papero & P)
```

```
{
  SS << ....//stampa su SS i vari campi di P a piacimento
  return SS;
}
```

Il frammento papero `P(...); .... cout<< P; stamperebbe i campi di P` come specificato nella ridefinizione di `operator<<`.

La visibilità delle dichiarazioni di tipo segue le stesse regole di quella delle variabili normali. Quindi il tipo dichiarato in un blocco è visibile in quel blocco (dal punto della dichiarazione in poi) e non fuori da esso. Spesso si desidera che un tipo ad hoc sia visibile da tutte le funzioni del programma. Se il programma risiede su un unico file, per avere visibilità globale di un nuovo tipo, si deve mettere la sua dichiarazione nel blocco globale del file, cf. Sezione 7.4. Il caso di programmi che risiedono su diversi file è discusso in Sezione 9.7.

9.3 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. Ogni blocco ha un solo punto di entrata ed un solo punto di uscita ed a questi punti si associano pre- e postcondizione del 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. Nella Sezione 7.1 abbiamo spiegato che l'istruzione di `return` è realizzata ricordando l'indirizzo a cui ritornare (quando la funzione conclude la sua esecuzione) nel record di attivazione che è inserito nella pila dei dati automatici al momento dell'invocazione della funzione. La realizzazione del `throw` avviene di nuovo con l'ausilio della pila dei dati automatici nella quale vengono inseriti puntatori ai `catch` man mano che l'esecuzione entra nei relativi blocchi `try`. L'esecuzione di una `throw` consiste nell'eliminare, partendo dalla cima della pila, tutti i record d'attivazione fino a quando non si incontra un puntatore ad un `catch` che abbia un parametro dello stesso tipo dell'eccezione sollevata dalla `throw`. L'esecuzione continuerà con le istruzioni del corpo del `catch` individuato in questo modo e con quello che resta della pila dei dati.

Ora introduciamo 2 nuove istruzioni di salto che rispettano la struttura a blocchi. Premettiamo che queste istruzioni vanno comunque usate il meno possibile e solo a condizione che il codice ne guadagni 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 del 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 il `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 vie che si chiama `switch`. Esso prende la seguente forma.

```
enum colore{bianco, giallo, rosso, blu, verde} X=giallo;
// la precedente dichiarazione dichiara il tipo enum colore
// e la variabile X di tipo colore e R-valore = giallo
..... // comandi in cui X potrebbe cambiare di valore
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 keyword `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 due 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` perché 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;
}
```

9.4 Conversioni ed operatori di cast

In Sezione 2.3 si è visto che nei programmi è normalissimo scrivere espressioni che mescolano valori di tipi diversi e si è mostrato che la valutazione di una tale espressione comporta l'uso di conversioni automatiche per fare in modo che ciascuna operazione sia eseguita con operandi di tipo uguale. Le conversioni automatiche sono fatte in modo da non perdere informazioni il che in pratica significa che i tipi la cui rappresentazione usa meno byte sono convertiti in tipi che occupano più byte. Queste conversioni vengono chiamate **promozioni** e 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 `enum` a `int`;

Per la verità, queste promozioni non sono tutte sicure. Convertire un `int` in un `float` potrebbe comportare una perdita di informazioni. Infatti se cercassimo di rappresentare in floating point con 23 bit per la mantissa un valore intero la cui rappresentazione binaria avesse dei bit 1 a distanza di più di 24 bit, la rappresentazione dell'intero sarebbe inesatta. In pratica se consideriamo l'intero $2^{24}+1$ e lo assegniamo ad un `float`, e poi riconvertiamo il valore da `float` a `int`, 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 valore di tipo `boolean`, `short int` ed `enum`.

Da quanto detto sulle conversioni automatiche operate dal C++ è facile dedurre che in questo linguaggio ogni espressione che contiene valori e variabili 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}` $\rightarrow$ `{short int}` $\rightarrow$ `{int}` $\rightarrow$ `{float}` $\rightarrow$ `{double}`

Anche le assegnazioni possono richiedere conversioni, ma in questo caso non c'è, in generale, la libertà di scegliere le conversioni che non producono perdita

di informazione. In un'assegnazione C++, $T_x \ x = \text{exp}$; il valore V prodotto dalla valutazione dell'espressione exp avrà un certo tipo T che potrà in generale essere diverso dal tipo T_x di x . In questo caso il valore V dovrà essere convertito dal suo tipo T in un valore *equivalente* di tipo T_x . Ovviamente qui non c'è libertà di scelta, ma una sola conversione **deve** essere fatta; da T a T_x . Se questa conversione ricade tra le promozioni, elencate prima, essa viene inserita nel codice oggetto prodotto dal compilatore, senza che il programmatore sia neppure avvertito della cosa. Se invece la conversione non è una promozione e quindi può comportare perdita di informazione, il compilatore, come prima, produce codice che esegue la conversione, ma in questo caso emette anche un *warning* (ammonimento) per avvertire del fatto che il programma effettua una conversione non sicura. Ci sono anche conversioni che non vengono accettate dal compilatore che, in questi casi, segnala un errore di tipo, terminando immediatamente il processo di compilazione. Un esempio di una conversione che è rifiutata dal compilatore è (forse inaspettatamente) quella tra `int` e i tipi enumerazione (mentre la conversione opposta è una promozione). Altre conversioni che producono errori in compilazione sono quelle tra 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` $\rightarrow$ `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. Quindi nel caso di un codice ASCII positivo, la conversione aggiunge 3 byte tutti a 0, mentre nel caso di codice ASCII negativo, la conversione rappresenta il valore negativo in complemento a 2 con 4 byte. Una conversione non sicura è, per esempio, quella inversa `int` $\rightarrow$ `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 questa 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 serve ad eseguire solo un particolare tipo di conversione. È importante capire qual'è il vantaggio di avere 4 cast anziché 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 tra gli anni 60 ed gli 80. 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 promozioni e più spesso per chiedere conversioni che vanno nella direzione opposta alle promozioni. 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 per 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: stampa 100 101 101. Sembrerebbe che ci siano 2 variabili `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 comparire anche altre costanti introdotte prima). Quindi nell'esempio precedente `cout << x << ...` 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 (e naturalmente resta lo stesso anche l'oggetto puntato). Va tenuto a mente che lo standard C++ sconsiglia di usare il `const_cast` per modificare valori dichiarati costanti in precedenza, come dell'esempio precedente. Queste azioni possono infatti causare comportamenti diversi a seconda del compilatore usato, in quanto lo standard non prevede sempre un comportamento da seguire.

- `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, come per il `const_cast` e al contrario dello `static_cast`, questa conversione non produce 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 che valeva prima del cast. Quindi, per esempio, dopo aver eseguito, `char *x; double *y;... x=reinterpret_cast<char*>(y);` `x` e `y` avranno lo stesso valore, ma, usando `x`, possiamo accedere il primo degli 8 byte che contengono l'R-valore di `y` interpretando quello che questo byte contiene come un codice ASCII esteso. Per inciso, in questo modo è possibile "leggere" byte a byte i valori di ogni tipo, anche di tipo struttura. Va da sé che questi cast sono estremamente pericolosi e, se fatti con poca attenzione, possono causare errori difficili 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. Il tipo a run-time del valore di una variabile è detto **tipo dinamico**, mentre il tipo dichiarato per quella variabile nel programma sorgente è chiamato **tipo statico**. Il `dynamic_cast` permette di conoscere il tipo dinamico delle variabili ed a scegliere le azioni da fare in base a questo tipo. Il suo uso verrà spiegato nel dettaglio in successivi corsi 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 7.3, 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 7.3, 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` $\rightarrow$ `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 9.8 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 un'espressione il cui valore è un L-valore del tipo specificato nel corrispondente parametro formale. L'unica eccezione è quella già spiegata nella Sezione 9.1 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`.

9.5 Gestione delle eccezioni

In Sezione 4.2.1 abbiamo visto che durante l'esecuzione di un programma possono verificarsi delle situazioni inattese, chiamate **eccezioni** e che il C++ offre delle apposite istruzioni per gestirle in modo elegante. Nella presente Sezione l'argomento viene approfondito.

Il programmatore degli anni 70 trattava le eccezioni inserendo output di appropriati messaggi d'errore 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 e del `return`. Si tratterebbe di prevedere che certe funzioni possano ritornare in modo "normale", ma anche in modo "anormale" compiendo cose diverse nei 2 casi. Per evitare questi problemi, i linguaggi di programmazione moderni, tra cui il C++, offrono dei comandi che permettono di gestire le eccezioni in modo semplice ed elegante. In pratica questi comandi permettono di definire dei punti (a piacimento) del programma in cui, in caso un'eccezione intervenga, l'esecuzione *salta* in un altro punto del programma in cui l'eccezione è gestita. Dopo la gestione di un'eccezione l'esecuzione continua normalmente con le istruzioni che seguono quelle di gestione dell'eccezione. Come già brevemente descritto in Sezione 4.2.1, un'eccezione viene *lanciata* con l'esecuzione di un'operazione (`throw`) che *salta* al blocco `catch` (*afferra*) più vicino alla `throw`. Il `catch` più vicino viene trovato grazie alla pila dei dati automatici con la procedura spiegata nella Sezione 9.3. Dopo l'esecuzione di un blocco `catch`, l'esecuzione continua con le istruzioni che lo seguono. Se il blocco `catch` conclude il corpo di una funzione, allora, dopo l'esecuzione del `catch`, la funzione terminerà.

I comandi per la gestione delle eccezioni sono descritti nel seguito in maggiore dettaglio rispetto a quanto già visto nella Sezione 4.2.1. Si tratta di tre comandi:

1. **try**: serve a individuare un blocco di una funzione che contiene istruzioni che possono causare il lancio di eccezioni. Alla fine del blocco `try` ci sono istruzioni `catch`, descritte dopo, che servono a gestire le eventuali eccezioni lanciate all'interno del blocco `try`.
2. **throw(T)**: serve a 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)` *afferra* 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 afferrata da uno dei due `catch` presenti. Se questo non fosse vero il programma sarebbe sbagliato, in quanto ogni eccezione lanciata deve venire afferrata da qualche `catch`, come vedremo meglio successivamente. In questo programma la funzione `g` lancia un'eccezione col 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` (che è il valore di `true`).

Restano vari aspetti da chiarire. Che succede se viene lanciata un'eccezione di un tipo `T` che nessun `catch` afferra? Per prima cosa entrano in gioco le conversioni automatiche del C++ (vedi Sezione 9.4). Se neppure con le conversioni si trova un `match` allora semplicemente l'esecuzione termina con un appropriato messaggio d'errore.

Quando un `throw` viene eseguito, ci possono essere diversi `catch` il cui parametro ha il tipo giusto per raccogliere l'eccezione. Abbiamo già detto che viene scelto il `catch` più vicino al `throw`. Per illustrare questo fatto, modifichiamo la funzione `f` dell'esempio precedente 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 lanciata (cioè `g`) (rispetto al `catch` del `main`) e quindi verrà scelta al posto di quella. Una seconda cosa importante illustrata da questo esempio è che dopo l'esecuzione del corpo del `catch` l'esecuzione continua normalmente, cioè viene eseguito 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).

9.6 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 **string**, **vector**, e **map**. Le ultime due classi sono definite nella **Standard Template Library (STL)**. La STL contiene molte altre classi. Informazioni più dettagliate si possono 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 di 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 efficiente alla 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 9.2. 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 alla C, vedi Sezione 2.1. Si ricordi che in questo caso il carattere `'\0'` non compare in *X*. Per appendere una porzione di stringa (stringa alla C) ad una stringa *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 11.1. 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 su `map`.

9.7 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 diversi gruppi di sviluppatori che ovviamente produrranno più file. Pensare di riunire tutti i pezzi su un unico file alla fine del lavoro è assurdo in quanto ogni gruppo di sviluppo ha l'esigenza di compilare la propria parte di programma, indipendentemente dal resto, in modo da riconoscere al più presto, grazie al compilatore, gli errori contenuti in questa parte. Quindi, in generale, un programma C++ è suddiviso in più file (che chiameremo **unità di compilazione**) ed il compilatore consente di compilare una singola unità, indipendentemente dalle altre unità, producendo un corrispondente file compilato. Il comando per ottenere la compilazione di una singola unità di nome `U.1.cpp` è `g++ -c U.1.cpp` che, se ha successo, produce il file compilato in `U.1.o`. Un modulo del compilatore, chiamato **linker**, successivamente collega i diversi file compilati in un unico programma oggetto eseguibile. Se abbiamo n unità compilate `U.1.o`, ..., `U.n.o`, esse vengono collegate con il comando `g++ U.1.o ... U.n.o`. Il programma oggetto risultante viene messo nel file `a.exe` (`a.out`). Quando il programma viene diviso in varie unità, possono diventare necessari molti comandi per compilare le unità e per collegarle 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 con un semplice esempio nella Sezione 9.9 che chiude questo Capitolo.

Torniamo al nostro programma composto da varie unità. Ogni unità di compilazione 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. Visto che in C++ l'uso di ogni oggetto (funzione/tipo/variabile globale) deve essere preceduto da un'opportuna dichiarazione, dobbiamo capire quali dichiarazioni siano necessarie in un'unità per *importare* oggetti definiti in altre unità. Nel seguito descriveremo le dichiarazioni che servono per *importare* funzioni, variabili globali e dichiarazioni di tipo:

- (a) Per quanto riguarda le funzioni la questione è semplice. Se l'unità U vuole usare una funzione la cui definizione risiede 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 3.6) impedisce dichiarazioni multiple di variabili con lo stesso nome in uno stesso blocco. Questa regola vale anche per le variabili globali e per applicarla correttamente,

quando il programma risiede su più unità di compilazione, è necessario essere consapevoli che le diverse unità di compilazione condividono lo stesso blocco globale. Insomma i blocchi globali delle diverse unità di compilazione formano un unico blocco globale (nel quale l'ordine dei diversi pezzi non è importante). Quindi, se in 2 unità di compilazione diverse apparissero 2 dichiarazioni di variabili globali con nome uguale, la compilazione separata di ciascuna unità non darebbe problemi, ma il linker darebbe un errore in quanto, collegando le 2 unità, si accorgerebbe che esse violano la regola della dichiarazione singola. 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, `extern int Globale`; . La keyword `extern` 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 `extern`? 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, nella pila dei dati, lo spazio RAM per contenere l'R-valore di questa variabile. Invece, quando il compilatore incontra la dichiarazione `extern int Globale`;¹, esso la usa come semplice informazione sul tipo di `Globale` in modo da poter eseguire il normale controllo dei tipi. Se nelle diverse unità che costituiscono un programma, ci fossero delle dichiarazioni di `Globale` con `extern` e nessuna dichiarazione *normale* di `Globale`, allora il linker fallirebbe con un opportuno messaggio d'errore. Per quanto detto finora, una dichiarazione `extern int Globale=2`; in una unità di compilazione `U` non ha senso. Infatti, 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'`extern` 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, purché tutte le dichiarazioni

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

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 globali.

Da quanto visto finora, le diverse unità che compongono un programma devono spesso includere dichiarazioni di oggetti definiti su altre unità. Le dichiarazioni devono essere corrette rispetto alle definizioni. Incongruità tra dichiarazioni dello stesso oggetto sulle diverse unità risulterebbero 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 *esportate* da ciascuna unità. A questo fine si usano dei file particolari, detti **header** che sono contrassegnati dall'estensione `.h`. La tecnica è semplice. Supponiamo di avere n unità di compilazione $U_1, \dots, U_n$ ciascuna sviluppata da un diverso gruppo di specialisti. A monte dello sviluppo separato delle diverse unità, si stabilisce con precisione quali oggetti ciascuna unità *esporta* verso le altre. Questo insieme di oggetti viene detto l'**interfaccia** dell'unità di compilazione. L'interfaccia dell'unità U_i viene scritta nel file header $U_i.h$. Quindi ogni unità U_i è costituita da 2 file: il file $U_i.cpp$ che contiene le definizioni delle funzioni ed il file $U_i.h$ che contiene l'interfaccia di U_i . Il file $U_i.cpp$ contiene l'istruzione `#include "U_i.h"` che ovviamente include fisicamente il contenuto di $U_i.h$ in $U_i.cpp$. Per fare in modo che U_i possa *importare* gli oggetti *esportati* da U_j basterà inserire l'istruzione `#include "U_j.h"` in $U_i.h$ ². Visto che i file header possono contenere comandi di inclusione di altri file header e visto che includere più di una volta un file header in una unità di programmazione causa un errore di compilazione perché il compilatore troverebbe dichiarazioni (apparentemente) multiple degli oggetti dell'header ripetuto, è buona norma inserire in ogni file header alcune istruzioni per il pre-processor del compilatore (quello che esegue le `#include`). Supponiamo che si tratti del file `pippo.h`, allora le seguenti istruzioni vanno inserite all'inizio ed alla fine di `pippo.h`:

```
#ifndef PIPPO_H
#define PIPPO_H

--dichiarazioni di pippo.h--

#endif
```

Dovrebbe essere facile capire che il pre-processor tiene memoria dei nomi come `PIPPO_H` che sono definiti quando un header è incluso per la prima volta e usa questa memoria per evitare di includere più volte lo stesso header.

²Si osservi che il file $U_j.h$ che viene incluso in $U_i.h$ differisce da quello che viene incluso in $U_j.cpp$ per quanto riguarda le dichiarazioni delle variabili globali che nell'unità U_j vanno introdotte con dichiarazioni *normali*, mentre nelle altre unità vanno introdotte come *extern*

Ovviamente le interfaccia (realizzate con i file header) costituiscono un accordo tra i diversi gruppi di sviluppo che permette a ciascun gruppo di sviluppare la propria unità, contando sul fatto che le altre unità siano sviluppate in modo coerente rispetto alle interfaccia stabilite.

Organizzare lo sviluppo di un programma in unità separate che possono venire realizzate in parallelo è una politica spesso dettata da questioni di produttività. D'altra parte, un programma di grandi dimensioni presenta generalmente una struttura in parti che realizzano funzionalità diverse. Questa struttura viene detta *logica*. È certamente raccomandabile che la strutturazione in unità di compilazione sia coerente con quella logica, ma non è sempre detto che questo si verifichi. Comunque vadano le cose, è importante che il linguaggio di programmazione utilizzato sia sufficientemente espressivo da esprimere in modo vantaggioso la strutturazione logica dei programmi che si sviluppano. Il C++ fornisce lo strumento dei namespace per ottenere questo risultato. 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.....}
```

La nozione di namespace è un concetto del C++ che non è presente nel C. Per fare in modo che i moderni compilatori C++ accettino e compilino senza storie i programmi C, cioè senza dichiarazioni di namespace (come tutti i programmi che abbiamo scritto in questo testo), un tale programma viene visto dal compilatore C++ come racchiuso in un namespace implicito detto **globale**. In

sostanza, 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 che è stato discusso prima in questa Sezione. Un namespace è un dominio di oggetti (variabili, tipi e funzioni) la cui visibilità è regolata come descritto nelle Sezioni 3.6 e 7.4.

Namespace diversi possono vedersi l'un l'altro solamente usando delle apposite istruzioni C++. 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, tra gli altri, gli `stream` di i/o e 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.

9.8 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);`. Successivamente, quando il compilatore troverà l'invocazione `f(5);` sceglierà la prima funzione mentre quando troverà `f(3.5);` sceglierà la seconda. Ovviamente il compilatore basa la scelta sul tipo del parametro attuale delle invocazioni. Questo fenomeno avviene talmente spesso nel C++ che si rischia di prenderlo per scontato. Per esempio, ci pare naturale di 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 quando ci sono differenze tra questi tipi, è necessario fare ricorso alle conversioni automatiche di tipo e quindi il compilatore può facilmente trovarsi nella situazione in cui, applicando conversioni diverse, una certa invocazione potrebbe venire soddisfatta da 2 o

anche più funzioni sovraccaricate. Nel seguito, queste funzioni verranno chiamate informalmente *le funzioni che possono andare bene* per l'invocazione considerata.

Per risolvere casi come questo, il C++ usa un algoritmo che si chiama algoritmo di **risoluzione dell'overloading**. L'idea di fondo dell'algoritmo è di ordinare le diverse conversioni possibili in C++ attribuendo a ciascuna un valore intero che cresce in funzione del grado di *pericolosità* della conversione. Coerentemente con quanto illustrato nelle Sezioni 2.3 e 9.4, una conversione è pericolosa se rischia di produrre perdita parziale o totale dell'informazione contenuta nel valore originale. In sostanza, le conversioni vengono catalogate per *pericolosità* e, per una certa invocazione, tra tutte le funzioni che potrebbero andare bene per essa, viene scelta dal compilatore quella che causa le conversioni strettamente meno pericolose. Nel seguito viene considerata una coppia costituita da un parametro attuale (quindi usato in un'invocazione) ed il corrispondente parametro formale (di una funzione che potrebbe andare bene per quell'invocazione). I diversi gradi di pericolosità della conversione che questa coppia può richiedere sono i seguenti:

1. tra parametro attuale e formale c'è perfetta uguaglianza oppure è necessaria una conversione triviale come tra $T\&$ e T , tra $\text{int}[]$ e int^* , e tra T e $\text{const } T$;
2. per convertire il tipo del parametro attuale in quello del parametro formale è sufficiente una promozione. Si ricordi che le promozioni sono: $\{\text{char}, \text{bool}, \text{enum}\} \rightarrow \{\text{short int}\} \rightarrow \{\text{int}\} \rightarrow \{\text{float}\} \rightarrow \{\text{double}\}$, cf. Sezione 9.4;
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, $\text{int} \rightarrow \text{char}$ e $\text{double} \rightarrow \text{int}$;
4. per convertire il tipo del parametro attuale in quello del parametro formale è necessaria una conversione definita dall'utente³.

Supponiamo di considerare l'invocazione $f(p_1, \dots, p_n)$ e di avere k definizioni di funzioni che possono andare bene per quest'invocazione. L'algoritmo di risoluzione dell'overloading calcola k n -tuple $\langle P_1^1, \dots, P_n^1 \rangle, \dots, \langle P_1^k, \dots, P_n^k \rangle$ in cui per la i -esima funzione che potrebbe andare bene, $\langle P_1^i, \dots, P_n^i \rangle$ è la sequenza degli indici di pericolosità delle conversioni richieste per passare dai tipi di $p_1, \dots, p_n$ ai tipi dei rispettivi parametri formali della i -esima funzione. Tra queste n -tuple definiamo la seguente relazione di ordine parziale⁴ $\leq$: $\langle P_1^i, \dots, P_n^i \rangle \leq \langle P_1^j, \dots, P_n^j \rangle$ se $\forall q \in [1..n], P_q^i \leq P_q^j$. Varrà il minore stretto, $\langle P_1^i, \dots, P_n^i \rangle < \langle P_1^j, \dots, P_n^j \rangle$, quando vale il $\leq$ ma non l'uguaglianza, da cui segue che $\exists m \in$

³Questa possibilità riguarda soprattutto le classi del C++ e qui è menzionata solo per completezza

⁴Una relazione è di ordine parziale quando soddisfa la proprietà riflessiva, antisimmetrica e transitiva.

$[1..n]$, $P_m^i < P_m^j$. Se, tra le k funzioni che potrebbero andare, ce n'è una la cui lista di indici di pericolosità è $<$ (minore stretto) della lista di indici di pericolosità di tutte le altre funzioni che potrebbero andare, allora l'algoritmo di risoluzione dell'overloading sceglie questa funzione. Se invece una tale funzione non esiste, allora l'algoritmo non riesce a decidere quale funzione si debba usare per l'invocazione considerata e, di conseguenza, il compilatore emette un messaggio d'errore ed interrompe la compilazione.

Vediamo ora 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 int); // (4)
void print(char); // (5)

void f(char c, int i, short int s, float f)
{
    print(c); // invoca (5) senza conversioni,
              //livello di pericolosità (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), essa viene preferita alla (5). Non ci fosse (1), l'algoritmo di risoluzione dell'overloading fallirebbe a causa del fatto che le funzioni (3) e (4) richiedono entrambe una conversione di livello (2). In casi come questo il compilatore dà un errore di **ambiguità** del programma: non c'è un'unica funzione che sia la migliore tra tutte quelle disponibili. Solo eliminando (1), (3) e (4), per `print(49)`; verrebbe scelta (5) che non avrebbe più rivali.

Vediamo un altro esempio per mostrare conversioni di livello (4). A questo fine, introduciamo un tipo struttura con un costruttore (cf. Sezione 9.2) e con una novità: operatori di conversione che specificano come un valore del tipo struttura può all'occorrenza essere convertito in un valore di qualche altro tipo. L'esempio completo segue:

```
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,
// cioè 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 delle 2 disponibili, 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 (classi) che ereditano campi e funzioni da altre strutture (classi). Le prime vengono chiamate **sottoclassi** e le seconde **classi base**. Generalmente, una relazione di ereditarietà tra una sottoclasse B ed una classe base A definisce una conversione (di livello (4)) tra oggetti di tipo B in oggetti di tipo A.

Nell'illustrazione dell'algoritmo di risoluzione dell'overloading fatta finora non abbiamo considerato la presenza di parametri con valori di default. In realtà questi parametri non entrano in gioco perché l'algoritmo di risoluzione dell'overloading considera solo i parametri attuali presenti nell'invocazione considerata. Quindi, se tra le funzioni che potrebbero andare, ce n'è qualcuna con dei parametri formali

con valori di default per cui l'invocazione non specifica dei corrispondenti parametri attuali, questi parametri in più non sono considerati dall'algoritmo di risoluzione dell'overloading. Chiariamo con un esempio.

```
void f(int x, int y=10)  // funzione (1)
{
    cout <<"con default"<<endl;
}
void f(double 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)) per il 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 parametro⁵. Invece la funzione (2) richiede una conversione di livello (2) per il (solo) argomento. Viene scelta quindi la funzione (1). In sostanza vengono considerate solo le conversioni dei parametri attuali presenti nell'invocazione e quindi gli eventuali valori di default non entrano mai in gioco.

C'è un'ultima considerazione da fare riguardo al tipo ritornato dalle funzioni. In quanto detto finora il tipo ritornato dalle funzioni sovraccaricate non è mai stato considerato. Da questo segue ovviamente che esso 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 maggiore complicazione e possibili incoerenze. Illustriamo questa affermazione con il seguente esempio, di nuovo tratto dal libro di B.Stroustrup citato prima. Con la risoluzione dell'overloading adottata dal C++ ed illustrata 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)
```

⁵Si osservi che in ogni caso i valori di default richiedono sempre conversioni a pericolosità minima.


```
double d=sqrt(da); // invoca (2)
fl=sqrt(fl); // invoca (1)
d=sqrt(fl); // invoca (1)
}
```

Se facessimo dipendere la scelta della funzione invocata dal contesto in cui si trova l'invocazione, troveremmo difficile decidere cosa fare per l'invocazione, `float fl=sqrt(da);` perché il contesto richiederebbe di chiamare (1) mentre il parametro richiederebbe (2). Quindi il compilatore dovrebbe emettere un messaggio d'errore per un'invocazione che nel comando seguente verrebbe invece accettata senza problemi. Se invece, per l'invocazione in `float fl=sqrt(da);` usassimo la funzione (1), questo sarebbe comunque strano visto che per l'identica invocazione del comando seguente dovremmo usare la (2).

9.9 Makefile

Nel seguito, illustriamo il comando Makefile con un semplice esempio. Immaginiamo di avere un programma che consiste di 3 unità `P1.cpp`, `P2.cpp`, e `P3.cpp` e tale che alcune di esse includono i file header, `D1.h` e `Global.h`. Il Makefile specifica tutte le operazioni che si devono fare per arrivare all'eseguibile complessivo. Quindi, specifica la compilazione separata delle 3 unità ed il loro linkaggio. Per ogni operazione specifica da quali file essa dipende. In questo modo, automaticamente, l'operazione viene eseguita solo quando uno dei files da cui dipende è cambiato rispetto all'ultima volta in cui quell'operazione è stata eseguita. Per eseguire questo controllo basta confrontare la data di ultima modifica del file prodotto dall'operazione con la data di ultima modifica dei file da cui l'operazione dipende. Se la prima data precede una delle seconde date, allora l'operazione va rifatta, altrimenti no. Il Makefile è organizzato in compiti (task). Ogni compito ha un'etichetta (nell'esempio seguente i compiti si chiamano "completo", "P1" eccetera) seguito da ':' e dalla lista dei file e/o dei compiti da cui l'operazione nella linea seguente dipende. La linea seguente inizia con un `tab` (per distinguerla dalle dipendenze che potrebbero estendersi per più righe di video) e riporta semplicemente il comando da eseguire per assolvere quel compito. Il Makefile seguente dovrebbe essere facilmente comprensibile:

```
completo: P1 P2 P3
        g++ -o completo P1 P2 P3
P1: P1.cpp D1.h D2.h
        g++ -c -o P1 P1.cpp
P2: P2.cpp D2.h
        g++ -c -o P2 P2.cpp
P3: P3.cpp D2.h
        g++ -c -o P3 P3.cpp
```

Ovviamente i nomi di questo esempio sono arbitrari. Se un tale *Makefile* è nella directory corrente, allora il comando da shell “make” esegue il primo compito (nel nostro esempio “completo”). Visto che esso dipende dai compiti precedenti, questi devono essere aggiornati prima di eseguire “completo”. Le dipendenze tra i diversi compiti possono essere complicate a piacimento. Se vogliamo invece eseguire un compito diverso dal primo, dobbiamo specificare la sua etichetta. Per esempio “make P3” per eseguire il compito P3.

In conclusione, tengo a chiarire che il comando Unix *Makefile* offre enormemente più opzioni di quanto mostrato qui, ma il nostro scopo era solo quello di permettere agli studenti di usarlo per i loro programmi.

Capitolo 10

Ricorsione

La ricorsione è una tecnica di programmazione che permette in molti casi di realizzare soluzioni eleganti, concise e corrette. Nella sua forma più semplice, una funzione ricorsiva invoca se stessa. L'idea è che le invocazioni risolvono sottoproblemi del problema iniziale e che, dato che ad ogni passo di ricorsione il problema da risolvere diventa più semplice, prima o poi verrà raggiunto un problema così semplice da non necessitare ulteriori invocazioni ricorsive. A quel punto l'ultima invocazione eseguirà un return e così via fino a ritornare alla prima invocazione della funzione ricorsiva. Da quanto detto si osserva immediatamente che dimostrare la correttezza di una funzione ricorsiva pone un problema nuovo. Il corpo di una funzione ricorsiva contiene invocazioni della funzione stessa, quindi per procedere nella dimostrazione dobbiamo già sapere che la funzione è corretta ! Insomma pare di essere caduti in un circolo vizioso. Per uscire da questa impasse useremo il **principio di induzione** che ci permetterà di realizzare dimostrazioni convincenti di funzioni ricorsive. In questo Capitolo considereremo solo esempi che utilizzano dati automatici, cioè gestiti automaticamente nella pila dei dati, come spiegato in Sezione 3.6. Nei Capitoli seguenti studieremo invece la ricorsione su dati gestiti dinamicamente, cioè dati la cui creazione e distruzione viene gestita in modo esplicito dai programmi attraverso opportune operazioni C++.

10.1 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**. Si dice invece che un certo insieme di funzioni sono **mutuamente ricorsive** se una delle funzioni ne invoca un'altra e così via finché 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 pila dei dati automatici, dello spazio necessario a contenere gli R-valori delle variabili locali e dei parametri formali della funzione e anche l'indirizzo di ritorno, cf. Sezione 7.1.

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 gli indirizzi di ritorno di queste invocazioni saranno tutti indirizzi di qualche istruzione della stessa funzione ricorsiva invocata ripetutamente. 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) *ripartiranno* dall'istruzione successiva all'invocazione ricorsiva che le ha messe in attesa, quando la funzione invocata eseguirà il `return`, cf. Sezione 7.1. Avvertiamo il lettore che, nonostante che l'intuizione delle copie multiple delle funzioni ricorsive non corrisponda a quello che succede realmente, nel seguito qualche volta ricorreremo ad essa per rendere più semplici le spiegazioni.

Ogni funzione ricorsiva deve possedere almeno un **caso base**, cioè una condizione che, qualora soddisfatta, causi 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 il che è ovviamente un errore.

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. Una dimostrazione formale di questa affermazione esce ovviamente dallo scopo di questo testo. Ci limiteremo a mostrare degli esempi per sostenere a livello intuitivo la validità di questa affermazione. Nella Sezione 11.3 mostreremo come una funzione ricorsiva può venire simulata da una funzione iterativa. Qui di seguito consideriamo l'altra direzione, che è più semplice, cioè mostreremo come simulare un ciclo iterativo con una funzione ricorsiva. Si consideri il seguente semplice 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 $1 + 10 + 9 + 8 + \dots + 1 = 56$. Vediamo come fare lo stesso calcolo con una funzione ricorsiva. In realtà esistono (almeno) due modi ricorsivi per eseguire lo stesso calcolo del precedente `while`. Iniziamo da quello che simula il `while` più da vicino.

```
int simula_while_1(int x, int y, int z)
{
    if (x<=y)
```

```

    return z; // caso base
else
    return simula_while_1(x-1,y,z+x);
}

// invocazione iniziale
int x=10, y=0, z=1;
cout<<simula_while_1(x,y,z)<<endl;

```

L'esecuzione prodotta dall'invocazione iniziale causa 11 invocazioni ricorsive di `simula_while_1` con parametro formale `x` che assume valori 10, 9, ..., 0. Per le prime 10 esecuzioni, il caso base ($x \leq y$) non si applica e quindi il parametro formale `z` assume valori 1+10, 1+10+9, ..., 1+10+9+...+1. L'ultima invocazione con $x=0=y$, non fa altro che restituire $z=1+10+9+...+1$. Questo valore è semplicemente restituito dalla undicesima invocazione alla decima che lo restituisce alla nona e così via fino alla prima invocazione. Quindi alla fine 1+10+9+...+1 viene stampato su `cout`. La funzione `simula_while_1` quindi calcola e stampa lo stesso valore del programma iterativo precedente. I calcoli che nel ciclo `while` sono eseguiti nel corpo del ciclo, in `simula_while_1` vengono eseguiti dalle espressioni che costituiscono i parametri attuali di ciascuna invocazione ricorsiva: `x-1` simula l'istruzione `x--`; mentre `z+x` simula `z=z+x`; . A questo proposito va osservato che ogni invocazione di `simula_while_1` ha proprie variabili locali `x`, `y` e `z` (con `y` sempre uguale a 0), mentre nel ciclo `while` c'è solo una `x`, una `y` ed una `z`. Si osservi che il test del caso base di `simula_while_1` è la negazione del test di permanenza nel ciclo `while`. Questo fatto non dovrebbe sorprendere visto che il test del caso base diventa vero quando la ricorsione finisce, mentre l'iterazione finisce quando il test di permanenza diventa falso.

E' interessante osservare anche una seconda funzione ricorsiva che simula il precedente ciclo `while` e che chiamiamo `simula_while_2`

```

int simula_while_2(int x, int y, int z)
{
    if(x<=y) return z; // caso base
    else
        return x+simula_while_2(x-1,y,z);
}

// invocazione iniziale
int x=10, y=0, z=1;
cout<<simula_while_2(x,y,z)<<endl;

```

Vale la pena di confrontare questa soluzione con la precedente. Il caso base è lo stesso e la `x` diminuisce di 1 ad ogni invocazione ricorsiva, quindi anche per questa nuova funzione verranno eseguite 11 invocazioni ricorsive con parametro `x`

uguale a $10, 9, \dots, 0$, ma in questa funzione la z resta a 1 attraverso tutte le invocazioni e fornisce il valore iniziale della somma che viene calcolata dalle istruzioni `return x+simula_while_2(x-1, y, z);`. Quindi `simula_while_2`, prima esegue l'invocazione ricorsiva `simula_while_2(x-1, y, z)`; e poi somma x al valore che l'invocazione restituisce e restituisce, a sua volta, il risultato. Quindi `simula_while_2`, in un certo senso, fa il contrario di `simula_while_1` che fa i suoi calcoli prima di eseguire l'invocazione ricorsiva. Nel seguito chiameremo queste 2 diverse maniere di fare i calcoli da parte delle funzioni ricorsive, **fare i calcoli al ritorno** (`simula_while_2`) e **fare i conti all'andata** (`simula_while_1`) della ricorsione. In generale le funzioni ricorsive eseguono calcoli sia all'andata che al ritorno della ricorsione. Le funzioni ricorsive che fanno tutti i loro calcoli solo all'andata, come `simula_while_1`, vengono dette **ricorsive terminali** (in inglese **tail recursive**). Le funzioni ricorsive terminali ammettono sempre una semplice traduzione in un equivalente programma iterativo (insomma un ciclo `while`). Questo fatto verrà illustrato nella Sezione 11.3.

10.2 Esempi di Ricorsione

In questa Sezione discuteremo alcuni esempi di funzioni ricorsive. Per dimostrare la correttezza di queste funzioni useremo delle dimostrazioni speciali dette **induttive** perchè si basano sul **principio di induzione**. Inizieremo col descrivere una dimostrazione induttiva per un semplice esempio. Successivamente descriveremo lo schema generale di una prova induttiva mostrando che lo schema è sensato. Infine studieremo alcuni esempi ai quali applicheremo lo schema generale.

Esercizio Risolto 10.1 *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(); // (1)

    cout << x ;
}
// (2)
```

Nel seguito useremo il simbolo `·` per indicare la concatenazione tra caratteri e stringhe. Con **C(cin)** indichiamo il contenuto di `cin` e lo stesso per **C(cout)**. La pre- e la postcondizione di questa funzione sono:

PRE = (cout è vuoto e C(cin) contiene una stringa della forma $w \cdot ' ; ' ; '$ in cui w potrebbe essere la sequenza vuota di caratteri)¹;

POST = (C(cout) è $' ; ' \cdot w^R$)²;

A parole, **POST** asserisce che tutti i caratteri che all'inizio sono in cin, fino al $' ; ' ; '$ compreso, verranno scritti su cout in ordine inverso rispetto a quello che avevano in cin.

La dimostrazione che **PRE** <reverse()> **POST** è vera, considera innanzitutto il caso base e poi gli altri casi.

Caso base: supponiamo che valga **PRE** e che C(cin) sia semplicemente $' ; ' ; '$ (quindi w è vuota), allora reverse semplicemente scrive su cout il carattere $' ; ' ; '$ e questo soddisfa **POST**.

Passo induttivo: consideriamo il caso in cui w non sia vuota. Quindi, assumiamo che C(cin) sia $w \cdot ' ; ' ; '$, con $w = x \cdot r$, con r possibilmente vuota. In questo caso reverse esegue un'invocazione ricorsiva con stato del calcolo in cui su C(cin) è la stringa $r \cdot ' ; ' ; '$. Quindi il problema che questa nuova invocazione di reverse deve affrontare è più piccolo di quello originale, e perciò possiamo assumere che questa invocazione sia corretta e quindi, visto che C(cin) è $r \cdot ' ; ' ; '$ che soddisfa **PRE**, possiamo assumere che al suo ritorno valga **POST** e cioè che C(cout) sia $' ; ' ; ' \cdot r^R$. Visto che dopo il ritorno dall'invocazione ricorsiva, la funzione reverse stampa x , alla fine del corpo della funzione, C(cout) sarà $' ; ' ; ' \cdot r^R \cdot x = ' ; ' ; ' \cdot w^R$, cioè esattamente quanto è richiesto da **POST**.

Quella appena descritta è una dimostrazione induttiva. Il punto essenziale di questa dimostrazione è l'ipotesi che l'invocazione ricorsiva sia corretta rispetto a **PRE** e **POST**. È ragionevole oppure stiamo mordendoci la coda? Potrebbe sembrare che prendiamo per vera la correttezza che vogliamo dimostrare! Ribadiamo i 2 punti che garantiscono che la nostra dimostrazione è matematicamente corretta:

- l'invocazione ricorsiva è fatta con contenuto di cin più corto di quello che avevamo all'inizio della funzione: $r \cdot ' ; ' ; '$ invece di $x \cdot r \cdot ' ; ' ; '$;
- l'invocazione ricorsiva viene effettuata in uno stato del calcolo che soddisfa la **PRE**.

Il primo punto garantisce che assumere la correttezza dell'invocazione ricorsiva rispetto a **PRE** e **POST** non è un ragionamento circolare. Il secondo punto ci permette di usare l'ipotesi di correttezza dell'invocazione ricorsiva, cioè di usare il fatto che la **POST** sia vera al ritorno dell'invocazione stessa. Ma attenzione, come la **PRE** va adattata allo stato del calcolo al momento dell'invocazione ricorsiva per dimostrare che la **PRE** è verificata in quel momento, così anche la **POST** che dobbiamo assumere al ritorno dall'invocazione ricorsiva, va adattata allo stato del

¹Si osservi che questa asserzione equivale a richiedere che cin contenga il $' ; ' ; '$. Dare un nome, w , alla sequenza di caratteri che precede $' ; ' ; '$, serve ad esprimere con precisione la postcondizione. Si noti inoltre che dopo il $' ; ' ; '$ potrebbero esserci altri caratteri che è inutile esplicitare in quanto non entrano nella postcondizione.

² w^R indica la sequenza ottenuta invertendo l'ordine dei caratteri in w

calcolo. Quindi dovremo assumere che, dopo il ritorno dell'invocazione ricorsiva, $C(cout)$ è $' ; ' \cdot r^R$. Nel seguito chiameremo **POST_ric** le postcondizioni come quella appena vista, cioè le postcondizioni da assumere vere dopo un'invocazione ricorsiva. Questo serve a distinguerle dalla **POST** che deve valere alla fine della funzione. Nel nostro esempio **POST_ric** vale (se l'invocazione soddisfa **PRE**) alla riga (1) di *reverse*, mentre **POST** deve valere alla riga (2).

Maggiori spiegazioni sulle dimostrazioni induttive sono date nella Sezione 10.2.1, ma prima diamo alcuni esercizi non risolti

Esercizio 10.2 Si chiede di realizzare una funzione ricorsiva `int conta()` che legge da `cin` una sequenza di caratteri fino a trovare `' ; '` (come nell'Esercizio 10.1) e conta quanti caratteri diversi da `' ; '` sono letti. Scrivere la pre e la postcondizione della funzione e dimostrare la sua correttezza seguendo lo schema induttivo usato nell'Esercizio Risolto 10.1.

Esercizio 10.3 Ecco due variazioni della funzione `conta` dell'esercizio precedente. Per entrambe le funzioni richieste scrivere pre- e postcondizione e cercare di abbozzare la dimostrazione di correttezza seguendo lo schema induttivo dell'Esercizio Risolto 10.1.

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

10.2.1 Le prove induttive

Precisiamo ora lo schema generale delle prove induttive che verranno usate estensivamente nel resto del testo. Successivamente discuteremo della validità di questo tipo di dimostrazione.

Definizione 10.1 Per dimostrare la correttezza di una funzione ricorsiva F rispetto a pre- e postcondizioni, **PRE** e **POST**, si segue il seguente schema induttivo:

- **Caso base:** Assumendo che valga **PRE**, si deve dimostrare che nel caso in cui si applicano i casi base di F , allora, dopo l'esecuzione delle istruzioni che questi casi prevedono, vale **POST**.
- **Passo induttivo:**
 - **Ipotesi induttiva:** si assume che le invocazioni ricorsive che compaiono nel corpo di F sono corrette rispetto a **PRE** e **POST**;
 - **Dimostrazione:**

1. **PRE** e il corpo di F devono essere tali da garantire che, le invocazioni ricorsive sono sempre eseguite in stati del calcolo che soddisfano **PRE**;
2. l'ipotesi induttiva, garantisce che al ritorno delle invocazioni ricorsive vale la **POST_ric**;
3. la **POST_ric** e la parte di F che segue le invocazioni stesse devono essere tali da poter dimostrare che F termina sempre in uno stato del calcolo che soddisfa **POST**.

Fondamenti delle prove induttive: Nel seguito vogliamo fugare i dubbi che possono nascere dal fatto che lo schema di prova induttivo sembra seguire una strada circolare in cui si assume per vero quello che si sta dimostrando. Il punto fondamentale da capire è che non si assume per vero *esattamente* quello che si sta dimostrando, ma si assume la correttezza della funzione ricorsiva che si applica ad un problema più piccolo di quello originale. Facendo riferimento all'Esercizio Risolto 10.1, le invocazioni ricorsive avvengono quando `cin` contiene un carattere in meno della sua situazione iniziale. Cerchiamo di essere più precisi. Supponiamo di voler dimostrare la correttezza, rispetto a **PRE** e **POST**, di una funzione ricorsiva $F(\text{int } n)$ il cui parametro formale assume valori naturali, cioè $n \geq 0$ e supponiamo che il caso base di F sia $n=0$ e che, nel corpo di F , le invocazioni ricorsive siano $F(n-1)$. Applichiamo lo schema di dimostrazione della Definizione 10.1:

1. Il **Caso base** richiede di dimostrare che se vale **PRE** e $n=0$, allora $F(0)$ è tale da soddisfare **POST** alla fine dell'esecuzione. Supponiamo che la prova riesca.
2. La correttezza di $F(1)$ può usare il fatto che l'invocazione ricorsiva di questa funzione è $F(0)$ che, per il punto precedente, è corretta rispetto a **PRE** e **POST**. Quindi quello che si deve dimostrare per avere la correttezza di $F(1)$ è esattamente il **Passo induttivo** della Definizione 10.1.
3. Nello stesso modo si dimostra la correttezza di $F(2)$ che contiene l'invocazione ricorsiva $F(1)$ che è corretta per il punto precedente.
4. Nello stesso modo si dimostra la correttezza (rispetto a **PRE** e **POST**) di $F(3)$, $F(4)$, ..., $F(n)$, per un qualsiasi n .
5. È importante osservare che le dimostrazioni per la correttezza di $F(1)$, $F(2)$, ..., $F(n)$ sono tutte uguali al **Passo induttivo** della Definizione 10.1. Quindi basta fare una volta sola il **Passo induttivo** per averle dimostrate tutte!

Chiaramente l'argomento appena descritto assume una funzione ricorsiva $F(n)$ particolarmente semplice (caso base $n=0$ e n che diminuisce di 1 ad ogni ricorsione). Le cose non sono sempre così semplici, ma sono spesso riconducibili a questo

caso semplice. Si riprenda l'Esercizio Risolto 10.1. La ricorsione avviene su quanti caratteri sono presenti su `cin` prima di leggere la sentinella `';` e questo numero è un naturale. Quindi siamo proprio nel caso di $F(n)$ appena considerato. In tutti i problemi ricorsivi considerati in questo testo sarà possibile ricondursi al caso della ricorsione sui numeri naturali, ma, in generale, ci sono funzioni ricorsive in cui la ricorsione ha uno schema più complicato. Per questi casi è necessario dimostrare che l'insieme delle possibili invocazioni ricorsive forma un insieme parzialmente ordinato tale che le invocazioni ricorsive risultano sempre più piccole (rispetto a quest'ordine) dell'invocazione originale e tale che ogni catena discendente di invocazioni termina sempre in un caso base della funzione.

Esercizio Risolto 10.4 *Si vuole realizzare una funzione ricorsiva `int ins(char *A, int limite)` che deve leggere da `cin` una sequenza di caratteri per inserirli (nello stesso ordine) nell'array `A`, con `limite` posizioni, fino a che l'array non sia pieno oppure fino a che non legga la sentinella `';`. La pre- e postcondizione cui deve obbedire la funzione cercata sono le seguenti:*

- **PRE** = (`cin` contiene $a_1 \dots a_k$; $k \geq 0$, `A` ha `limite` ≥ 0 posizioni);
- **POST** = (se $k \geq \text{limite}$ allora $A[0..limite-1] = a_1 \dots a_{limite}$ e viene restituito `limite`, altrimenti $A[0..k] = a_1 \dots a_k$; e viene restituito $k+1$).

```
int ins(char *A, int limite)
{
    if(!limite) // finito il posto in A, 0 letture
        return 0;

    char x;
    cin >> x;
    if (x==';') //c'e' posto in A, ma leggo ';'
        //quindi la ricorsione
        //finisce e ho fatto l'ultima lettura
    {
        A[0]=x;
        return 1;
    }
    else // inserisco il carattere e ricorro con il resto di A
        // che ha 1 posizione in meno di prima
        //le letture fatte sono 1+ quelle che saranno
        //eseguite dalla ricorsione
    {
        A[0]=x;
        return 1+ins(A+1,limite-1);
    }
}
```

In questa funzione ricorsiva ci sono due casi base: il primo si verifica quando la condizione `!limite` (cioè `limite==0`) 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**? $k \geq \text{limite}=0$ e quindi `A[0..0-1]` 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` come richiesto.
- Nel secondo caso base, $k=0$ mentre `limite > 0` e quindi deve essere verificata la seconda parte di **POST** ed infatti così è, visto che `A[0..0]=A[0]=';`' e che viene restituito 1 che è il numero di caratteri letti, cioè la sentinella `';`'.

Resta da considerare il **Passo induttivo**, cioè il caso in cui `limite>0` e il carattere letto non è la sentinella e quindi viene effettuata l'invocazione ricorsiva. In questo caso il carattere letto in `x` (diverso da `';`') viene inserito all'inizio dell'array con `A[0]=x`. Dopo di che viene eseguita l'invocazione ricorsiva `ins(A+1, limite-1)`. Per questa invocazione vale **PRE**, dato che `limite > 0` e quindi `limite - 1 ≥ 0` e `cin` contiene ancora il carattere `';`', visto che lo conteneva prima dell'ultima lettura che non ha letto `';`'. Quindi l'**Ipotesi induttiva** garantisce che al ritorno dell'invocazione ricorsiva `ins(A+1, limite-1)` vale **POST ric**, cioè a partire da `A+1` viene inserito in `A` quello che c'è su `cin` e viene restituito il numero dei caratteri letti (ed inseriti). Perciò `A[0]=x; return 1+ins(A+1, limite-1)` inserisce correttamente `x` al primo posto di `A` e restituisce il numero totale di letture effettuate. Quindi **POST** vale alla fine del corpo della funzione `ins`.

Esercizio Risolto 10.5 Cerchiamo ora una soluzione ricorsiva del Problema (5) già affrontato nel Capitolo 8. In realtà realizziamo una funzione iterativa `F` che si serve di alcune funzioni ausiliarie ricorsive per raggiungere lo scopo dell'esercizio. Anche questa nuova versione di `F` deve soddisfare la pre- e post-condizione specificata per la soluzione del Problema (5) del Capitolo 8. Comunque, per mostrare che per un dato problema possono esistere molte coppie di pre- e postcondizioni che servono allo scopo di dimostrare la correttezza di una soluzione, specifichiamo qui una coppia di pre- e postcondizioni più semplici di quelle del Capitolo 8.

PRE = ($0 \leq \text{top}$, `A[0..top-1]` è ordinato in modo non decrescente, `y` è definito) && (siano $[a_1, \dots, a_n]$ gli elementi di `A[0..top-1]` diversi da `y` nell'ordine che hanno in `A[0..top-1]`);

POST = (`top=n` && `A[0..n-1] = [a1, ..., an]`)

La funzione ausiliaria iterativa cerca del Capitolo 8 contiene 2 cicli separati. In corrispondenza di ciascun ciclo realizzeremo ora una funzione ricorsiva. La prima funzione ricorsiva si chiama `cercal` e il suo compito è di calcolare il più piccolo indice `start` tale che `A[start]=y`. Qualora un tale `start` non esista, `cercal` deve restituire il valore booleano `false`. La funzione `cercal` ha come

parametri l'array *A*, il suo numero di posizioni *top*, il valore *y* da cercare, il parametro *start*, passato per riferimento, in cui *cercal* restituisce il risultato di una ricerca coronata da successo, ed anche *pos* che indica fin dove l'array *A* è stato percorso nella ricerca di *y*. Quindi inizialmente *pos* avrà valore 0 e crescerà durante la ricorsione. La funzione *cercal* con la sua coppia di pre- e postcondizioni segue:

```
/*PRE1=(0<=pos<=top, A[pos..top-1] è definito
ed ordinato in modo non decrescente) */
bool cercal(int*A, int top, int pos, int y, int & start)
{
    if (pos==top) return false;

    if (A[pos]==y)
        {start=pos; return true;}

    if (A[pos]<y)
        return cercal(A,top,pos+1,y,start);

    return false;
}
/*POST1=(restituisce true=> pos<=start<=top-1 && A[start]=y &&
A[pos..start-1]<y) && (restituisce false => per ogni j
in [pos..top-1] A[j]!=y)*/
```

Caso base: ci sono 3 casi da considerare,

- *pos=top*: l'array è vuoto e perciò non può contenere *y*. Quindi la risposta *false* soddisfa **POST1**.
- *pos<top && A[pos]=y*: la risposta in questo caso è *true* e *start=pos*. Questo soddisfa **POST1** perché *pos* è il primo elemento di *A[pos..top-1]* e quindi, in questo caso, *A[pos..start-1]* è vuoto e ovviamente tutti i suoi elementi (cioè nessuno) sono minori di *y*.
- *pos<top && A[pos]>y*: in questo caso la funzione restituisce *false* e questo soddisfa **POST1**, visto che, essendo *A[pos..top-1]* ordinato, da *A[pos]>y* segue che *A[pos..top-1] > y* e quindi *A[pos..top-1]!=y* come richiesto da **POST1**.

Passo Induttivo: supponiamo valga **PRE1** e che l'esecuzione della funzione porti ad eseguire l'invocazione ricorsiva *cercal(A, top, pos+1, y, start)*. *A[pos+1..top-1]* è ordinato, visto che *A[pos..top-1]* lo è e da *pos < top*, segue che *pos+1<=top*. Quindi l'invocazione ricorsiva è eseguita in uno stato del calcolo che soddisfa **PRE1**. Per l'ipotesi induttiva, al ritorno dall'invocazione ricorsiva, vale la **POST1.ric** e immediatamente dopo questo ritorno, la funzione *cercal* esegue l'istruzione di *return*:

```
return cercal(A, top, pos+1, start);
```

Questo soddisfa POST1? La risposta è positiva e per dimostrarlo si ragiona come segue. Consideriamo separatamente il caso in cui l'invocazione ricorsiva restituisca true e quello in cui restituisca false:

- **l'invocazione restituisce true:** dal fatto che **POST1_ric** vale al ritorno dell'invocazione ricorsiva `cercal(A, top, pos+1, y, start)`, segue che $pos+1 \leq start < top$, che $A[start] = y$ e anche che $A[pos+1..start-1] \neq y$. Da questo segue immediatamente che $pos \leq start < top$ e, dato che $A[pos] < y$, segue anche che $A[pos..start-1] \neq y$. Insomma vale **POST1**.
- **l'invocazione restituisce false:** allora $A[pos+1..top-1] \neq y$ e, visto che $A[pos] < y$, ne segue che $A[pos..top-1] \neq y$, come richiesto da **POST1**.

*Quindi abbiamo dimostrato che la funzione `cercal` è corretta rispetto a **PRE1** e **POST1**. Realizziamo ora la funzione che conta quanti y sono presenti in A . La chiamiamo `cerca2`. Si osservi la condizione $A[0..t-1] \geq y$ di **PRE2** che indica che la funzione `cerca2` scorre A partendo dalla posizione `start` o superiore, infatti solo gli elementi in $A[start..top-1]$ soddisfano la condizione $\geq y$.*

```
/*PRE2=(t>=0, A[0..t-1] è ordinato in modo non decrescente,
  A[0..t-1]>=y, y è definito) */
int cerca2(int *A, int t, int y)
{
    if(t)
        if(A[0]==y)
            return 1+cerca2(A+1,t-1,y);
    // si arriva qui con A vuoto o A[0]>y
    return 0;
}

// POST2=(restituisce il numero di y in A[0..t-1])
```

*Si osservi che in `cerca2` viene usata una tecnica per scorrere l'array A diversa da quella usata in `cercal`. Infatti, in `cerca2` non si vuole calcolare un particolare indice nell'array $A[0..top-1]$ (come `start` in `cercal`), ma si vuole solo percorrere l'array e contare quanti elementi uguali a y ci sono. Visto che la **PRE2** ci assicura che il primo elemento di A è maggiore o uguale a y , i casi base si hanno quando l'array finisce, oppure quando si incontra un elemento maggiore di y (si ricordi che A è ordinato!). Visto che in questi casi la funzione restituisce 0, la **POST2** è verificata.*

*Consideriamo ora il **Passo ricorsivo**. È facile constatare che l'invocazione ricorsiva `cerca2(A+1, t-1, y)` è sempre eseguita in uno stato del calcolo che*

soddisfa **PRE2**: visto che $t > 0$, $t-1 \geq 0$, inoltre, dal fatto che $A[0..t-1]$ è ordinato segue che $A[1..t-1]$ lo è, e infine, da $A[0..t-1] \geq y$ segue che $A[1..t-1] \geq y$. Quindi vale **PRE2** per l'invocazione ricorsiva e perciò, per l'**Ipotesi induttiva**, vale **POST2_ric** al suo ritorno che, precisamente, garantisce che il valore restituito dall'invocazione ricorsiva `cerca2(A+1, t-1, y)` è il numero di y presenti in $A[1..t-1]$ e visto che $A[0]=y$, basta aggiungere 1 al valore restituito da `cerca2(A+1, t-1, y)` per ottenere il numero di y in $A[0..t-1]$.

Per concludere l'esercizio, dobbiamo fornire una funzione `shift` ricorsiva ed una funzione (non ricorsiva) `F` che invoca le 3 funzioni ricorsive per ottenere il risultato richiesto dalla **POST** finale.

```
//PRE3=(A[0..newtop+quant-1] definito, quant>0, newtop>=0)
void shift(int *A, int newtop, int quant)
{
    if (newtop>0)
    {

        A[0]=A[quant];

        shift(A+1, newtop-1, quant);
    }
}

/*POST3=(A[0..newtop-1] = A[quant..newtop+quant-1]) */
```

POST3 asserisce che la parte di $A[quant..newtop+quant-1]$ viene spostata in $A[0..newtop-1]$ eliminando i valori originalmente contenuti nella porzione $A[0..quant-1]$. Insomma viene fatto uno `shift` verso sinistra di `quant` posizioni. Che `shift` faccia proprio questo lo si dimostra con una prova induttiva. Il **Caso base** è semplice. Quando `newtop=0` allora entrambi gli array in **POST3** sono vuoti. Quindi essi soddisfano banalmente l'uguaglianza. Per il **Passo ricorsivo**, `newtop > 0` e quindi certamente si deve spostare $A[quant]$ nella posizione 0 di A . In questo modo resteranno `newtop-1` elementi da spostare e naturalmente da $A[quant+1..newtop+quant-1]$ alle posizioni $[1..newtop-1]$ di A . Questa è **POST_ric** dell'invocazione ricorsiva `shift(A+1, newtop-1, quant)`. Per poterla usare, basterà dimostrare che questa invocazione avviene sempre in uno stato del calcolo che soddisfa **PRE3**. Questo è facile:

- $A[1..(newtop-1)+quant-1]$ è definito in quanto **PRE3** all'inizio della funzione implica che $A[0..newtop+quant-1]$ lo è e questa contiene la porzione di array precedente;
- visto che il caso base non si applica `newtop > 0` e quindi `newtop-1 > 0`, come richiede la **PRE3**;

- infine quanti non viene cambiato dalla funzione quindi resta maggiore di 0.

Quindi anche shift è corretta rispetto a **PRE3** e **POST3**. Resta ora solamente da scrivere la funzione F che coordina le tre funzioni realizzate finora. F non è ricorsiva.

```
void F(int*A, int & top, int y)
{
    int start=0, quanti;
    if(cerca1(A,top,0,y,start))
    {
        quanti=cerca2(A+start, top-start, y);
        shift(A+start, top-quanti-start, quanti);
        top=top-quanti;
    }
}
```

Per dimostrare la correttezza di F dobbiamo usare quella appena dimostrata delle 3 funzioni ausiliarie. Ci sono vari punti da considerare:

- **PRE** implica **PRE1** quindi possiamo usare **POST1**;
- quindi se cerca1 ritorna false significa che non ci sono y e quindi non fare nulla soddisfa **POST**;
- se invece cerca1 ritorna true allora A[start]=y ed è il minimo indice con questa proprietà;
- quindi vale **PRE2** e quindi possiamo assumere che al ritorno da cerca2, valga **POST2** e quindi che start e quanti individuino tutte le occorrenze di y in A;
- resta il ragionamento più complicato: shift viene invocato con A=A+start e newtop=top-quanti-start, cioè shift agisce solo sulla parte di A in cui ci sono le occorrenze di y, lasciando stare i primi start elementi che non devono subire cambiamenti. In A[start..top-1] ci sono naturalmente top-start elementi. Da questi vogliamo eliminare le quanti occorrenze di y e quindi ne rimarranno solo newtop=top-quanti-start.
- visti i valori dei parametri attuali dell'invocazione di shift, **PRE3** prescrive che (A+start)[0.. top-quanti-start+quanti-1] sia definito, e che quanti>0 e che newtop>=0:
 - per la prima parte basta semplificare e osservare che se abbiamo un array di top e elementi e partiamo dalla posizione start, allora da quella posizione restano top-start elementi (che saranno in [start..top-start-1]),

- che $\text{newtop} = \text{top} - \text{quanti} - \text{start}$ sia maggiore o uguale a 0 segue dal fatto che start e quanti indicano la dimensione di parti distinte dell'array A che ha top elementi;
- che quanti sia maggiore di 0 segue da **POST1** e **POST2**.
- quindi vale **POST3** da cui sappiamo che $(A + \text{start})[\text{top} - \text{start} - 1]$ viene modificato eliminando i suoi primi quanti elementi e lasciando quindi $\text{top} - \text{start} - \text{quanti}$ elementi. Aggiungendo la prima parte di start elementi (non toccata da shift), restano in totale $\text{top} - \text{quanti}$ elementi e questa quantità diventerà quindi il nuovo valore di top .

Questo dimostra la correttezza di F rispetto a **PRE** e **POST**.

L'Esercizio Risolto 10.5 è molto istruttivo. I seguenti punti vanno notati:

1. Un punto delicato di F è l'invocazione di shift . In particolare non è semplice convincersi che i parametri attuali dell'invocazione siano corretti. Un modo per farlo è quello di attribuire dei valori alle diverse variabili e poi controllare che i conti tornano. Per esempio, potremmo supporre che $\text{top} = 10$, $\text{start} = 3$ e $\text{quanti} = 3$. Allora l'invocazione diventerebbe $\text{shift}(A + 3, 4, 3)$. Da **POST1** segue che $A + 3$ è l'inizio della fascia di A che contiene le 3 occorrenze di y . $(A + 3)[0..10 - 3 - 1]$ contiene 7 elementi. Di questi 3 vanno tolti e quindi ne resteranno 4 e infatti il secondo parametro attuale dell'invocazione è 4. Il che significa che ci sono 4 elementi che vanno spostati in questa porzione di array. Essi sono $(A + 3)[3..6]$ e vanno spostati in $(A + 3)[0..3]$. Si osservi che la parte $(A + 3)[3..6]$ consiste dei 4 ultimi elementi di A , mentre $(A + 3)[0..3]$ consiste dei 4 elementi di A che iniziano da start , cioè da dove iniziano le occorrenze di y . Lo spostamento dei 4 elementi finali di A viene fatto ricorsivamente e inizia con $(A + 3)[\text{quanti} = 3]$ che si sposta in posizione $(A + 3)[0]$. L'invocazione ricorsiva si occupa di spostare i rimanenti 3 elementi. Dato che l'invocazione ricorsiva viene fatta con i parametri attuali, $3A + 3 + 1, 4 - 1, 3$ essa dovrà nuovamente spostare $(A + 3 + 1)[3]$ in $(A + 3 + 1)[0]$ e così via per le successive invocazioni ricorsive che terminano dopo 4 invocazioni, cioè quando il parametro newtop diventa 0.
2. Consideriamo di nuovo la pre- e postcondizione di shift . **PRE3** non specifica che gli elementi da eliminare contengono y . Infatti la funzione fa il suo lavoro qualsiasi sia il contenuto degli elementi che vengono sovrascritti. Lo stesso vale per **POST3**. Essa specifica solo che quella determinata porzione di A viene spostata all'inizio sovrascrivendo quello che c'era prima. Questo esempio ci insegna a non inserire nelle nostre asserzioni dei dettagli irrilevanti rispetto al lavoro fatto dalla funzione stessa.
3. Consideriamo ora **PRE2**, essa richiede che t , cioè il numero di elementi dell'array A , sia maggiore o uguale a 0. In realtà nella prima invocazione di

cerca2, quella fatta da *F*, **POST1** ci assicura che $t \geq 0$. Dobbiamo aggiungere anche il caso uguale a 0 perché **PRE2** non deve essere vera solo per la prima invocazione, ma anche per tutte le altre possibili e quindi anche per quella con $t=0$ che termina la sequenza di invocazioni ricorsive. Questa è una caratteristica delle pre- e postcondizioni ricorsive. La preconditione deve valere per tutte le possibili invocazioni mentre la postcondizione deve occuparsi solo del calcolo fatto dalla funzione e non da quanto potrebbe essere stato fatto dalle precedenti invocazioni ricorsive. Come esempio di questa caratteristica, **POST3** si occupa di spostare i suoi *newtop* elementi e non si preoccupa degli altri elementi eventualmente spostati prima. Chiaramente la **POST3** della prima invocazione dice che tutto quello che doveva essere spostato è stato correttamente spostato, ma questo è fatto grazie alle invocazioni ricorsive ognuna delle quali fa solo il suo pezzetto di lavoro.

Proponiamo ora un esercizio interessante. Vale la pena di provare a risolverlo cercando di scrivere con cura la pre- e postcondizione della funzione.

Esercizio 10.6 *Si tratta di un esercizio simile all'Esercizio 10.5, ma ora l'array $A[0 \dots \text{top}-1]$ non è ordinato. Vogliamo eliminare da A tutte le occorrenze di un valore y dato e vogliamo restituire gli elementi di A rimasti, compattati verso sinistra. Si chiede di scrivere una pre- ed una postcondizione che descrivano in modo preciso cosa deve effettivamente fare la funzione, poi di sviluppare il codice di una funzione ricorsiva che esegua questo compito ed infine, di dimostrare induttivamente che essa è corretta rispetto alla pre- e postcondizione data.*

Capitolo 11

Gestione dinamica dei dati e liste

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

11.1 Gestione dinamica dei dati

Le istruzioni del C++ per la gestione dinamica dei dati sono le seguenti. L'operazione di allocazione della memoria è la funzione `new` 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 la pila in cui vengono gestite le variabili automatiche. La funzione `new` ha il seguente prototipo: `T* new (T) ;`. L'invocazione di `new`, se ha successo, riserva sullo **heap** una sequenza di byte della dimensione giusta a contenere un valore di tipo `T` e restituisce l'indirizzo del primo byte di questa sequenza. Se invece l'allocazione non ha successo, il valore restituito è 0. In programmi in 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;};
main()
{
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' $(i+1)$ -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 logico quindi 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` ad 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 puntato da `p`. Le parentesi `[]` servono ad indicare che si vuole deallocare un array e questo vale per ogni array, indipendentemente 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 (non array), allora si invoca `delete` semplicemente. Vediamo un esempio:

```
int * p=new int(10); //alloca un intero con valore 10 puntato da p
// ....usa *p....
delete p; // dealloca *p
```

Il puntatore `p` punterà a 4 byte sullo heap che contengono il valore 10 e questo spazio verrà deallocato dalla `delete`. A quel punto `p` è un dangling pointer e quindi deve venire usato con le cautele del caso.

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
```

11.2 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 11.1, 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 il tipo di questo campo è arbitrario (potrebbe essere anche di tipo struttura). 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 alla costante `NULL=0` predefinita nel C++) 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 semplicemente da una variabile `nodo*` che abbia valore 0 e che quindi non punti a nessun nodo. Nel seguito chiameremo **corretta** una lista che sia o vuota oppure costituita da un nodo il cui campo `next` punti ad una lista corretta. Insomma una lista corretta è vuota (puntatore `nodo*` uguale a 0) oppure una lista concatenata il cui ultimo nodo abbia campo `next` a 0. Nel seguito le liste dovranno sempre essere corrette. Nel seguito consideriamo alcune operazioni sulle liste concatenate e mostriamo vari modi di realizzarle attraverso funzioni ricorsive e iterative. Negli esercizi che seguono assu-

miamo il tipo `struct nodo{int info; nodo* next;};` con costruttore `nodo::nodo(int a=0, nodo* b=0){info=a; next=b;}`.

Esercizio Risolto 11.1 *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 la lista 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_{ric}** quando essa termina (stiamo usando la notazione introdotta nella Definizione 10.1) e cioè che il valore che essa restituisce sia 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 11.2 *Per tutti i successivi esercizi si chiede di definire pre- e postcondizione di ogni funzione e di delineare una prova induttiva della sua correttezza.*

1. Modificare la funzione dell'Esercizio 11.1 in modo che i k nodi che vengono costruiti, anziché avere campi `info` uguali a 0, contengano valori $1, \dots, k$ con 1 nel primo nodo e k nell'ultimo. Si possono aggiungere parametri alla funzione nell'Esercizio 11.1. La nuova funzione deve essere ricorsiva.

2. Scrivere una funzione ricorsiva che riceve una lista concatenata e restituisce il numero dei nodi della lista stessa. Questa quantità si chiama la **lunghezza** della lista.
3. Scrivere una funzione ricorsiva che, data una lista concatenata ed un carattere x , restituisce il numero di nodi della lista il cui campo `info` sia uguale a x .
4. Scrivere una funzione che, data una lista concatenata ed un intero k , dealloca i primi k nodi della lista se essa ha k o più nodi, mentre, se ha meno di k nodi, allora dealloca tutti i nodi che ci sono. La funzione deve restituire la lista che resta dopo le deallocazioni. Per esempio, se la lista ha 3 nodi e $k=2$, allora la funzione deve deallocare i primi 2 nodi e restituire la lista che contiene il terzo nodo. Se $k=0$ la funzione deve restituire la lista originale senza fare nulla. Infine se $k=4$, la funzione deve deallocare tutti e 3 i nodi presenti e deve restituire la lista vuota (cioè 0 o NULL).

Esercizio Risolto 11.3 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* L, int x)
{
    return new nodo(x,L);
}
```

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* &L, int x)
{
    L=new nodo(x,L);
}
```

Consideriamo ora di voler aggiungere un nodo con campo `info` uguale a x in fondo alla lista L . Dobbiamo quindi scorrere la lista fino ad arrivare in fondo, cioè all'ultimo nodo della lista, e dobbiamo cambiare il suo campo `next` in modo che punti al nuovo nodo (con campo `info` uguale a x). Riconoscere l'ultimo nodo di una lista è facile: è quello con campo `next=0`. Naturalmente, dobbiamo capire che, se la nostra strategia è quella di cercare l'ultimo nodo della lista, stiamo assumendo che la lista abbia sempre almeno un nodo! Quindi la pre- e postcondizione di una funzione che segue questa strategia sono:

PRE $= (L \text{ è una lista non vuota, sia } v_L \text{ il suo valore iniziale})$

POST=($L=vL+nodo(x, 0)$), in cui con + si indica l'aggiunta di un nuovo nodo con campi x e 0 in fondo a vL .

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

Si osservi che soddisfare **PRE** è essenziale, infatti qualora la funzione venisse invocata con $L=0$, il test $L->next==0$ causerebbe un errore di . Insomma 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 (L==0) L=new(x, 0); else ins_fine(L, x);`, ma sarebbe più semplice non doversene preoccupare. È possibile scrivere una funzione ricorsiva che inserisca un nodo in fondo ad una lista qualsiasi (vuota o no)? Ovviamente la risposta è positiva e nel seguito vedremo 2 modi diversi di farlo. Pre- e postcondizione per le funzioni che seguono questa diversa strategia sono le seguenti:

PRE=(L è una lista (eventualmente vuota) e sia vL il suo valore iniziale)

POST=($L=vL+nodo(x, 0)$), in cui con + si indica l'aggiunta di un nuovo nodo con campi x e 0 in fondo a vL . Visto che ora vL potrebbe essere 0 , $0+nodo(x, 0)$ è semplicemente l'indirizzo di `nodo(x, 0)`.

La prima funzione che proponiamo per risolvere il problema usa il passaggio della lista L per valore.

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

Che questa funzione sia corretta rispetto a **PRE** e **POST** viene dimostrato in modo induttivo come segue. Il **Caso base** è che L sia vuota ($!L$). In questo caso, la funzione restituisce il puntatore al nuovo nodo `nodo(x, 0)` il che, ovviamente, garantisce **POST**.

Nel **Caso ricorsivo**, **PRE** vale prima dell'invocazione ricorsiva perché $L->next$ è una lista corretta, visto che $L \neq 0$. Da questo segue che vale **POST.ric** dopo l'invocazione. Basta ora capire che cosa faccia l'istruzione,

```
L->next=ins_fine_val(L->next, x);
```

Per capirlo introduciamo la seguente notazione. Sia m il nodo puntato da vL , quindi $vL = m + L'$ in cui $L' = vL \rightarrow next = m \rightarrow next$. Allora l'istruzione

```
L → next = ins_fine_val (L → next, x);
```

concatena il nodo m con la lista restituita dall'invocazione ricorsiva

```
ins_fine_val (L → next, x)
```

che, è $L' + \text{nodo}(x, 0)$ (è **POST-ric**), quindi, l'istruzione complessiva produce la lista $m + L' + \text{nodo}(x, 0)$, che soddisfa **POST** e che viene restituita dal successivo `return L;`.

Quindi la funzione è corretta rispetto a **PRE** e **POST** e gestisce anche il caso di lista vuota. Però `ins_fine_val` ha un difetto: esegue delle operazioni inutili. Si riconsideri la notazione introdotta per la prova del precedente **Caso ricorsivo**. Qualora L' non fosse vuota, il primo nodo di $L' + \text{nodo}(x, 0)$ è anche il primo nodo di L' e quindi l'assegnazione,

```
L → next = ins_ins_val (L → next, x)
```

asigna a $L \rightarrow next$ lo stesso R -valore che esso già possiede! L'unico campo `next` che viene effettivamente modificato è quello dell'ultimo nodo della lista originale che, dopo l'esecuzione della funzione, da 0 passa a puntare al nuovo nodo. Il problema è che la funzione non può distinguere l'ultimo nodo da quelli intermedi, altrimenti ritorneremmo alla funzione `ins_fine` che accetta solo liste non vuote. Quindi è necessario eseguire `L → next = ins_fine_val (L → next, x);` per ogni nodo della lista originale, anche se è quasi sempre inutile.

Riassumendo, il problema è che, dovendo accettare anche la lista vuota, il caso base deve essere appunto $L = 0$ (anziché $L \rightarrow next = 0$). Solo quando questa condizione è verificata, si deve creare il nuovo nodo e si deve inserire il puntatore a questo nodo nel campo `next` dell'ultimo nodo della lista originale. Osserviamo che questo campo `next` è esattamente il parametro attuale dell'invocazione finale della funzione `ins_ins_val`. Purtroppo il passaggio di questo parametro avviene per valore, e quindi viene passato solo l' R -valore 0 di questo campo `next`, ma se passassimo il parametro per riferimento, allora l'invocazione finale avrebbe l' L -valore del campo `next` dell'ultimo nodo e potrebbe quindi assegnargli immediatamente il puntatore al nuovo nodo. La funzione che usa quest'idea è la seguente:

```
void ins_fine_rif(nodo*& L, int x)
//passaggio di L per riferimento
{
    if(!L)
        L=new nodo(x,0);
    else
        ins_fine_rif(L->next,x);
}
```

La dimostrazione di correttezza di questa funzione rispetto a **PRE** e **POST** segue. Il **Caso base** è uguale a quello visto per `ins_fine_val`. Il **Passo induttivo** usa il fatto che il parametro $L \rightarrow next$ è passato per riferimento e quindi

***L'Ipotesi induttiva** garantisce, non solo che l'invocazione ricorsiva produca la lista $L' + \text{nodo}(x, 0)$, ma che inserisca il puntatore a questa lista nel campo `next` del nodo m puntato da L . Quindi, al ritorno dall'invocazione ricorsiva, senza fare nulla, L punta a $m + L' + \text{nodo}(x, 0)$ e quindi soddisfa **POST**.*

Va chiarito che ad ogni invocazione della funzione `ins_fine_rif`, il parametro L , grazie al passaggio per riferimento, è un alias del campo `next` del nodo precedente, se c'è un nodo precedente, mentre se non c'è un nodo precedente allora sarà un alias della variabile del chiamante passata come parametro attuale. In generale si tratta della lista all'intera lista a cui si vuole aggiungere il nodo finale. Insomma, anche se solo per l'ultima invocazione, in $L = \text{new nodo}(x, 0)$;, sfruttiamo questo aliasing di L , esso è presente per tutte le invocazioni.

Ora consideriamo lo stesso esercizio appena visto, ma lo vogliamo risolvere in modo iterativo.

Esercizio Risolto 11.4 *Si tratta di appendere un nuovo nodo con campo `info=x` in fondo ad una lista L in modo iterativo. Offriamo 2 soluzioni.*

```
//PRE=(L è lista corretta)
nodo* ins_fine_iter(nodo* L, int x)
{
    if(!L) return new nodo(x);
    nodo*X=L;
    int n=0;
    while(L->next) //R=(L = suffisso di X con |X|-n>0 nodi)
        {L=L->next; n++;}
    L->next=new nodo(x);
    return X;
}
//POST=(restituisce L+nodo(x,0))
```

La notazione $|X|$ sta per il numero di nodi della lista X . Sia chiaro che n è nel programma solo per scrivere **R** in modo da esprimere che il ciclo avanza L fino a puntare all'ultimo nodo. Dopo di che è chiaro che l'istruzione $L \rightarrow \text{next} = \text{new nodo}(x)$; appende il nuovo nodo all'ultimo nodo già presente. Quindi ovviamente l'unica cosa di cui convincersi è che dal ciclo si esce con L che punta all'ultimo nodo. Se non ci curassimo della terminazione, basterebbe osservare il test di permanenza del ciclo: esso è falso solo se $!L \rightarrow \text{next}$, cioè se L è l'ultimo nodo. Con n è facile garantire anche la terminazione perché n cresce ad ogni giro e quindi $|L|$ diminuisce ad ogni giro e quindi prima o poi consisterà del solo ultimo nodo di X .

Questa soluzione è inefficiente perché ogni volta si tratta di percorrere tutta la lista per arrivare al suo ultimo nodo. Si potrebbe cercare di rappresentare una lista con un doppio puntatore, uno al suo inizio ed uno alla fine. Se la lista è vuota entrambi i puntatori sono a 0. Vediamo come potrebbe funzionare.

```

/* PRE=(o L=F=0 oppure L punta all'inizio di una lista corretta
e F punta all'ultimo nodo di questa lista)*/
void ins_fine_iter2(nodo* &L, nodo* &F, int x)
{
    if (!L)
        {L=F=new nodo(x, 0); }
    else
        {F->next=new nodo(x); F=F->next; }
}
//POST=(L=L+nodo(x,0) e F punta al nuovo nodo)

```

Ovviamente in questo modo si evita il ciclo. Visto che la funzione restituisce 2 risultati, L ed F sono passati per riferimento.

Esercizio 11.5 Considerare il problema di appendere 2 liste tra loro. Quindi far puntare il campo next dell'ultimo nodo della prima lista al primo della seconda. Risolvere l'esercizio sia ricorsivamente che iterativamente. Estendere entrambe le soluzioni iterative viste nell'Esercizio Risolto 11.2.

Esercizio 11.6 Nei seguenti esercizi sono sulle liste concatenate e chiedono soluzioni ricorsive. Per ogni esercizio si richiede di specificare pre- e postcondizione di ogni funzione e di delineare la relativa dimostrazione di correttezza.

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 la lista P+Q. Nessun nuovo nodo deve essere creato e nessuno deve essere distrutto. Semplicemente le due liste date vanno concatenate 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 verranno 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 $k \geq 0$, 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.
6. Risolvere lo stesso esercizio del punto precedente, ma invertendo la restituzione delle due liste risultato: col `return` si vuole restituire la lista dei primi k nodi di P e con il parametro P quella che resta.

Esercizio 11.7 I seguenti esercizi considerano varie operazioni in cui un nodo viene eliminato da una lista. Per tutti gli esercizi si può assumere che inizialmente la lista Q non sia vuota. Per ogni esercizio si richiede di specificare pre- e postcondizione di ogni funzione e di delineare la relativa dimostrazione di correttezza.

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 `char pop(nodo*& Q)` che elimini il primo nodo di Q , restituisca la nuova lista attraverso il parametro passato per riferimento, e restituisca 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 `char del_end_rif(nodo* & Q)` che elimini l'ultimo nodo della lista e restituisca la lista rimanente attraverso il parametro Q , appositamente passato per riferimento, e il campo informativo del nodo eliminato col `return`.

Esercizio 11.8 Consideriamo ora operazioni su liste ordinate, cioè tali che il campo informativo dei suoi nodi non decresca percorrendo la lista dall'inizio alla fine. Per ogni esercizio si richiede di specificare pre- e postcondizione di ogni funzione e di delineare la relativa dimostrazione di correttezza.

1. Scrivere una funzione ricorsiva `nodo* ins_ord(nodo* Q, char 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, char x)` simile alla precedente, ma restituisca la nuova lista attraverso il parametro passato per riferimento.

3. Scrivere una funzione `nodo * del_ord(nodo* Q, char x)` che elimini dalla lista ordinata `Q` tutti i nodi (ce ne 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, char 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`.

11.3 Ricorsione ed iterazione

Un punto importante della ricorsione è che ogni funzione ricorsiva può essere simulata da una iterativa (e viceversa). In generale la funzione iterativa deve definire e usare una struttura dati che simula la pila delle chiamate ricorsive ed è quindi parecchio più complicata della funzione ricorsiva che essa simula. Si osservi infatti che chi scrive programmi ricorsivi non deve preoccuparsi di realizzare la pila delle invocazioni ricorsive in quanto questo è compito dell'ambiente di esecuzione dei programmi! In realtà, come anticipato in Sezione 10.1, ci sono 2 tipi di ricorsione: uno semplice da simulare iterativamente ed uno più difficile da simulare con l'iterazione. Il caso semplice è la ricorsione terminale in cui i calcoli vengono eseguiti solo all'andata e quindi nessun calcolo segue una invocazione ricorsiva. In questa Sezione considereremo una semplice funzione ricorsiva terminale e mostreremo come trovare un'equivalente funzione iterativa. La funzione in questione esegue il seguente calcolo: essa stampa i campi `info` dei nodi di una lista concatenata dal primo all'ultimo elemento.

```
void stampa(nodo *L)
{
    if (L)
    {
        cout<<L->info<<' ';
        stampa(L->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 un ciclo `while` nel modo seguente:

```

while (L)
{
    cout<<L->info<<' ';
    L=L->next;
}

```

Questo ciclo percorre la lista dall'inizio alla fine stampando i campi `info` dei nodi, esattamente come la funzione ricorsiva `stampa`. La sola differenza tra i due modi di produrre la stampa della lista è che la funzione `stampa`, una volta raggiunta la fine della lista, risale all'inizio della lista a causa dei `return` (impliciti). D'altra parte, essendo la ricorsione terminale, durante questo *ritorno indietro* non viene calcolato nulla e quindi evitarlo, come fa il ciclo `while`, 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 di `L->info` da fare.

```

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

```

Nonostante diventi più difficile, è possibile trasformare anche funzioni non *tail recursive* in cicli iterativi. 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*, ma serve invece nel caso di funzioni non *tail recursive*). 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.

```

int nodi=conta(L); // conta i nodi di L
int *A=new int [nodi];
int pos=0;
while (L)
{A[pos]=L->info; pos++; L=L->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* delle invocazioni ricorsive.

Capitolo 12

Alberi Binari

Questo Capitolo è dedicato ad un'altra struttura dati ricorsiva: gli alberi binari. Il Capitolo è organizzato in una prima sezione che contiene le definizioni di base, seguita da una sezione con molti esercizi (in maggioranza risolti, ma non solo). Segue una sezione sugli alberi binari di ricerca. Il Capitolo termina con un esercizio impegnativo sugli alberi. Tutti i programmi sono accompagnati da una prova di correttezza.

12.1 Nozioni di base sugli alberi binari

La definizione di albero binario è ricorsiva.

Definizione 12.1 *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 discendenti di n compreso n stesso. Un **cammino** in un albero è una sequenza di archi che collegano un nodo a qualche suo discendente (o viceversa). 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 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.

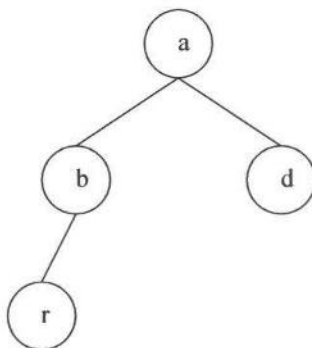


Figura 12.1: L'albero binario con 4 nodi costruito da BUILD_TREE

In C++ un nodo di un albero binario viene modellato da una struttura `nodo` 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; right=c;}`. 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 l'albero binario con 4 nodi che è raffigurato nella Figura 12.1.

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

Nella Figura 12.1 ogni nodo è rappresentato con un cerchio 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 12.1. Inoltre, `terzo=primo->right` e `quarto=primo->left->left`.

È molto utile avere anche una rappresentazione lineare degli alberi binari, cioè rappresentare un albero con una stringa di simboli. È utile per esempio per l'input/output di alberi. L'albero di Fig. 12.1 ha la seguente rappresentazione lineare:

$a(b(r(-, -), -), d(-, -))$, in cui $-$ indica un sotto-albero vuoto. La relazione padre-figlio_destro-figlio_sinistro è rappresentata con le parentesi e la virgola. Per esempio il fatto che a sia padre di b e d è rappresentato da $a(b(\dots), d(\dots))$ e $b(r(-, -), -)$ indica che b ha il solo figlio sinistro, mentre questo figlio sinistro r non ha alcun figlio, cioè è una foglia.

Nel seguito considereremo solo alberi binari **corretti**, dove è corretto l'albero vuoto (puntatore di tipo `nodo*` uguale a 0) ed è corretto un albero costituito da un nodo i cui campi `left` e `right` puntano ad alberi corretti. Visto che un albero è sempre acceduto tramite un puntatore al suo nodo radice, anziché scrivere frasi come, *l'albero la cui radice è puntata dal puntatore p* , scriveremo più semplicemente, *l'albero p* e scriveremo anche che p è la radice dell'albero.

Visto che consideriamo solo alberi binari, i cammini possono venire rappresentati come sequenze di 0/1, dove 0 rappresenta la scelta di andare al figlio sinistro e 1 quella di andare al figlio destro. Per esempio, nell'albero di Fig. 12.1, il cammino dalla radice alla foglia r è 0 0, mentre il cammino dalla radice al nodo d è 1. Nel seguito faremo sempre terminare un cammino con la sentinella -1, quindi il primo cammino considerato verrà rappresentato da 0 0 -1 e l'altro da 1 -1. Il cammino -1 è il cammino vuoto, cioè il cammino che parte da un qualsiasi nodo e resta in quel nodo. Si osservi che per parlare di cammini, anche vuoti, è necessario che l'albero abbia almeno 1 nodo.

Percorrere un albero (binario) significa visitare ogni suo nodo. Fondamentalmente esistono 2 maniere di percorrere un albero: **in larghezza** e **in profondità** (breadth-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 per ridiscendere lungo il cammino successivo 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 dato un albero non vuoto, prima si visita la radice, poi il sotto-albero sinistro e per ultimo quello destro. Nell'ordine infisso invece si visita prima il sotto-albero sinistro, poi la radice e per ultimo il suo sotto-albero destro. Infine l'ordine postfisso prescrive che si visiti prima il sotto-albero sinistro, poi quello destro e per ultimo la radice.

Il seguente programma esegue il percorso in profondità secondo l'ordine infisso e stampa i campi `info` di tutti i nodi secondo questo ordine.

```

\\PRE=(n è un albero corretto)
void infix_stampa(nodo * n)
{
    if(n)
    {
        infix_stampa(n->left);
    }
}

```

```

        cout<<n->info<<' ';
        infix_stampa(n->right);
    }
}
/*POST=(stampa i campi info di tutti i nodi
di n in ordine infisso) */

```

Esercizio 12.1 *Scrivere funzioni ricorsive che stampano tutti i nodi di n in ordine pre e postfisso.*

12.2 Esercizi sugli alberi binari

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

Esercizio Risolto 12.2 *Vediamo ora una funzione ricorsiva che dealloca tutti i nodi di un albero binario. La ricorsione rende il compito particolarmente semplice. La cosa principale da capire è che, prima di deallocare la radice di un albero, si deve 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 ad un nodo deallocato e che quindi punta ad un'area di memoria (che risiede sullo heap) che non è più a disposizione del programma. Ovviamente lo stesso varrebbe per x->right.*

```

//PRE=(x e' albero corretto)
void dealloca_albero(nodo * x)
{
    if(x) //l'albero non e' vuoto
    {
        dealloca_albero(x->left);
        dealloca_albero(x->right);
        delete x;
    }
}
//POST=(tutti i nodi dell'albero x sono deallocati)

```

È facile mostrare la correttezza del programma dealloca_albero:

- Il caso base è che l'albero sia vuoto e quindi non c'è nulla da deallocare e la **POST** è banalmente verificata.
- nel caso ricorsivo, x non è vuoto, e quindi x->left/right è corretto, per cui vale la **PRE** prima delle invocazioni ricorsive e quindi vale la **POST_ric**

al ritorno dalle invocazioni. Per cui l'unico nodo rimasto dell'albero x è la sua radice e l'istruzione `delete x`; elimina anche questo nodo.

Esercizio Risolto 12.3 *Dato Un albero qualsiasi vogliamo ricorsivamente stampare la sua rappresentazione lineare.*

```
/*PRE=( r è un albero corretto) */
void stampa_lin(nodo *r)
{
    if(r)
    {
        cout<<r->info<<' (';
        stampa_lin(r->left);
        cout<<',';
        stampa_lin(r->right);
        cout<<')';
    }
    else
        cout<< '_';
}
/*POST=(produce su cout la rappresentazione lineare
di r ) */
```

Il caso base soddisfa la postcondizione. Le invocazioni ricorsive sono fatte con sotto-alberi corretti, quindi producono la stampa desiderata. La funzione poi stampa `r->info`, le parentesi e la virgola nell'ordine richiesto. Il lettore dovrebbe cercare di classificare questa funzione usando i diversi percorsi in profondità introdotti nella Sezione 12.1.

Esercizio Risolto 12.4 *Vogliamo calcolare il numero di nodi di un albero n .*

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

Dimostrare la correttezza di `conta_nodi` è molto facile. PRE =(n è un albero binario corretto). POST =(la funzione restituisce il numero dei nodi nell'albero n).

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 $n \rightarrow \text{left}$ e $n \rightarrow \text{right}$ sono alberi corretti (eventualmente vuoti). Quindi per entrambe le invocazioni ricorsive, vale la post-condizione **POST_ric** e cioè esse restituiscono il numero di nodi del sotto-albero sinistro e destro del nodo n e quindi, sommando 1 (per contare la radice) alla somma dei valori restituiti dalle 2 invocazioni, si ottiene il numero di nodi di tutto l'albero n . Questo dimostra che il return restituisce il valore richiesto e quindi la funzione è corretta rispetto a PRE e POST.*

Esercizio Risolto 12.5 *Vogliamo ora realizzare l'operazione di ricerca di un nodo di un albero che contenga un campo informativo di un valore particolare, dato. Potrebbero esserci più nodi nell'albero che contengono il valore cercato, 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 alla 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 diventa 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, ma per prima cosa specifichiamo la pre- e postcondizione della funzione da fare.*

PRE = (n è un albero corretto).

POST = (se la funzione restituisce un valore diverso da 0, allora il valore punta ad un nodo N dell'albero n , tale che $N \rightarrow \text{info} = x$, altrimenti nessun nodo esiste nell'albero n che soddisfa questa condizione).

```
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);
}

```

Come sempre usiamo l'induzione per dimostrare la correttezza di ricerca rispetto a **PRE** e **POST**. Il **Caso base** è ovvio: l'albero vuoto non ha nodi e quindi certamente non contiene un nodo con campo `info=x`. Nel **Passo induttivo**, sappiamo che `n` punta ad un nodo e quindi eseguire il test `n->info==x` non dà errore. Se il test è `true`, allora `n` soddisfa la condizione cercata e quindi restituire `n` soddisfa **POST**. Se invece la condizione è falsa, dovremo ispezionare i sotto-alberi di `n`. Visto che `n` è un albero corretto e non vuoto, `n->left/right` sono corretti e quindi vale **PRE** per `ricerca(n->left, x)`; . Quindi vale **POST.ric** dopo `nodo* z=ricerca(n->left, x)`; da cui segue che se `z` non è 0, il nodo puntato da `z` è quello che cerchiamo e quindi restituirlo soddisfa **POST**. Se invece `z=0`, allora **POST.ric** implica che l'albero `n->left` non ha nodi che soddisfano la condizione, per cui è necessario cercare un tale nodo nel sotto-albero destro. Cosa che viene eseguita dalla seconda invocazione ricorsiva. Come prima, la **PRE** vale per questa seconda invocazione ricorsiva e quindi al ritorno dall'invocazione va da **POST.ric** da cui segue che il valore restituito dall'invocazione è corretto per l'albero `n->right`. Visto che, dai precedenti tentativi, sappiamo che solo in `n->right` ci può ancora essere un nodo che soddisfa la condizione, restituire col `return` la stessa risposta dell'invocazione ricorsiva su `n->right` soddisfa **POST**.

Esercizio Risolto 12.6 Consideriamo ora un'operazione nuova sugli alberi binari. Quella di aggiungere un nodo ad un albero. Va subito chiarito che il nuovo nodo verrà sempre inserito sulla frontiera dell'albero, cioè esso sarà sempre inserito come foglia dell'albero. L'albero iniziale potrebbe 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. Come al solito iniziamo scrivendo la pre- e postcondizione della funzione che vogliamo realizzare. **PRE** = (`n` è un albero corretto, sia `vn=n`), chiamiamo `vn` il valore iniziale dell'albero `n`, in modo da poterlo paragonare nella **POST** con quello ottenuto dopo l'esecuzione della funzione. `vn` sta per vecchio `n`.

La postcondizione è più difficile da specificare perché la funzione deve eseguire una sequenza di scelte non semplice da descrivere. Introduciamo una notazione per aiutarci nel compito. Consideriamo un qualsiasi albero binario `T`. La funzione che vogliamo realizzare deve aggiungere un nuovo nodo a `T` e diremo che questo nodo è **messo bene** in `T` se soddisfa i seguenti punti:

- (a) se T è l'albero vuoto, allora il nuovo nodo diventa la radice;
- (b) se T non è vuoto, allora il nuovo nodo viene aggiunto come una foglia f , tale che, 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 (senza contare f) è minore di quello del sotto-albero sinistro di m e, altrimenti, f è nel sotto-albero sinistro di m .

POST = (l'albero n è vn a cui è stato aggiunto un nuovo nodo con $info=x$ e questo nodo è messo bene in vn).

```
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);
    }
}
```

È importante osservare che il primo parametro formale di `inserisci` è passato per riferimento.

È facile vedere che nel **caso base**, cioè quando vale `!n`, vale **POST**: siamo nel caso (b) della definizione di messo bene e n è l'albero costituito dal solo nuovo nodo. Inoltre, va osservato che, visto che n è passato per riferimento, il chiamante riceve il puntatore al nuovo albero.

Nel **passo induttivo** possiamo contare sulla correttezza della funzione `conta_nodi`, che abbiamo dimostrato nell'Esercizio 12.4. Dalla **PRE** di `inserisci` e dal fatto che n non sia 0, segue che `n->left` e `n->right` puntano ad alberi corretti 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 sul 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 verificata e quindi, per l'**Ipotesi induttiva**, anche la sua **POST_ric** è verificata. Quindi l'invocazione restituisce l'albero di partenza (`n->left` o `n->right`) col nuovo nodo messo bene e, quanto asserito su come la funzione decide quale sotto-albero scegliere per fare l'invocazione ricorsiva, segue che anche nell'albero n il nuovo nodo è messo bene e quindi vale la **POST**.

È molto importante osservare la similarità tra questa funzione e la funzione `ins.fine_rif` dell'Esercizio 11.3. In entrambe le funzioni è fondamentale

il passaggio del sotto-albero corrente per riferimento (parametro n). Anche in inserisci, nel caso base in cui il parametro n è uguale a 0, il parametro è un alias del campo left o right del nodo padre e quindi l'assegnazione n=new nodo(x, 0, 0) attacca il nuovo nodo al padre. Si osservi che, sempre grazie al passaggio per riferimento, la funzione è corretta anche nel caso venga invocata inizialmente su un albero vuoto.

Esercizio 12.7 *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 Risolto 11.3.*

Esercizio Risolto 12.8 *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 tutti essere l'altezza di un albero non vuoto. 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 ricorsivo 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 ricorsivo.

```
//PRE=(n è albero corretto)
int altezza(nodo * n)
{
    if(!n) // albero vuoto
        return -1;
    else
```

```

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

```

Dimostrare la correttezza di questa funzione è facile. PRE =(n è un albero binario corretto) e POST =(la funzione restituisce l'altezza dell'albero 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;
- **Passo induttivo:** n non è 0 e da questo segue che le 2 invocazioni ricorsive sono eseguite con **PRE** vera e quindi, per l'**Ipotesi induttiva**, 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 12.9 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 12.8?
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`. Per questo esercizio il campo `info` deve essere intero.

Esercizio Risolto 12.10 In questo esercizio assumiamo di considerare alberi con nodi il cui campo `info` abbia tipo intero. Vogliamo realizzare 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 12.8, 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

$n \rightarrow \text{left} \rightarrow \text{info}$ e $n \rightarrow \text{right} \rightarrow \text{info}$ 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:

```
//PRE=(n è albero corretto)
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;
    }
}
/*POST=(ogni nodo di n ha in info la distanza
        minima da una foglia) */
```

L'ultimo comando condizionale potrebbe venire evitato inserendo degli *else*, ma il programma presentato ci sembra più semplice e quindi preferibile. La dimostrazione della correttezza di questa funzione è semplice: i casi base sono ovviamente corretti rispetto a **POST**. Le invocazioni ricorsive garantisco che i rispettivi sotto-alberi soddisfino **POST_ric** e quindi visto che il caso in cui n ha 2 figli è distinto da quelli in cui ne ha uno solo, è chiaro che alla fine anche $n \rightarrow \text{info}$ riceve il valore giusto.

Nonostante sia corretta, il codice di questa funzione appare di una lunghezza 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 12.8: 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 12.8) 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. La PRE è la solita. La POST invece è:

POST =(se `n` è albero vuoto, allora restituisce `INT_MAX`, altrimenti, alla fine dell'esecuzione ogni nodo dell'albero `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 non vuoto di `n`. Quindi almeno uno tra `h_l` e `h_r` è un valore valido e perciò scegliere il minore ed inserirlo in `n->info` dopo aver sommato 1 è il calcolo corretto che soddisfa POST. Si osservi che il ragionamento appena fatto non funzionerebbe se `n` fosse una foglia. Ecco perchè è necessario considerare a parte questo caso.

Esercizio 12.11 *Per i seguenti esercizi scrivere pre- e postcondizione, realizzare i programmi richiesti e infine fare la loro dimostrazione di correttezza.*

1. Usare le idee discusse nell'Esercizio 12.10 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 12.10), 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'Esercizio 12.10 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 del sotto-albero radicato in n 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 un albero binario e inserisca nel campo `info` di ogni nodo n il numero dei nodi con al più 1 figlio che si trovano nel sottoalbero radicato in n .

Esercizio Risolto 12.12 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->info > M->info`. Dopo di che invoca semplicemente se stessa sui figli di n . Le invocazioni con `n=0` non modificano `M`. L'invocazione iniziale è: `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`)¹; e la **postcondizione** è: **POST** = (`M` punta al nodo massimo tra quelli che precedono `n`, `n` stesso ed 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 ora il **Passo induttivo**, cioè che `n` sia un albero non vuoto. Consideriamo prima il caso particolare che `M=0`. Questo implica che `n` è la radice dell'albero intero e quindi è il primo nodo considerato. 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_ric**, da cui segue che vale la **PRE** per la seconda invocazione ricorsiva e quindi anche per essa vale la **POST_ric**, da cui segue che vale **POST** per l'intera funzione che quindi restituisce in `M` il nodo massimo dell'intero albero. 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 tutti 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).

Nel seguito consideriamo una seconda funzione per calcolare il nodo massimo, che chiamiamo `max2`. Essa è parecchio più complicata della precedente, ma vale la pena di studiarla comunque. `max2` 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 sotto-alberi 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 guadagniamo in concisione (pagando, forse, con una minore semplicità).

¹Se non ci sono nodi che precedono `n`, allora `M=0`


```

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 postcondizione di questa funzione e di dimostrare per induzione la sua correttezza.

Esercizio 12.13 Risolvere i seguenti esercizi con funzioni ricorsive di cui è richiesta la scrittura della pre- e postcondizione e la dimostrazione di correttezza.

1. In un albero in cui i nodi hanno valori tutti distinti nei campi `info`, diciamo che un nodo `q` è secondo quando il suo campo `info` è secondo nell'ordine crescente tra i valori dei campi `info` dei nodi. Si scriva una funzione che restituisca il puntatore al secondo nodo di un albero dato. Particolare cura andrà spesa per il trattamento dei casi limite.
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 Risolto 12.14 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 negli elementi dell'array i campi `info` di tutti i nodi dell'albero secondo l'ordine prefisso. Ovviamente i nodi dell'albero hanno campo `info` di tipo intero. 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 campo `info` del nodo corrente nella prossima posizione libera dell'array. Il punto essenziale della soluzione è l'utilizzo del parametro `x` di tipo `int*&` che ad

ogni istante punta alla prossima posizione libera dell'array A. Quando un nuovo info viene inserito in *x, allora x viene incrementato di 1, in modo che punti alla prossima posizione libera. Si osservi che la prima invocazione ricorsiva di `fai_ric` (quella che si occupa di $n \rightarrow \text{left}$) modifica il valore di x facendolo avanzare di tante posizioni quanti sono i nodi di quel sotto-albero e, grazie al passaggio di x per riferimento, il nuovo valore di x è passato alla seconda invocazione ricorsiva (quella che si occupa di $n \rightarrow \text{right}$).

```
int conta(nodo *n)
{
    if(n)
        return 1+conta(n->left)+conta(n->right);
    else
        return 0;
}
/* PRE=(n è un albero corretto con k>=0 nodi e x un array
   con almeno k elementi, sia vx=x)*/
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);
    }
}
/* POST=(in vx[0..k-1] contiene i campi info dei nodi
   di n considerati in ordine prefisso e x=vx+k)

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
}
```

Il lettore è invitato a pensare alla dimostrazione della correttezza di `fai_ric`.

Esercizio 12.15 1. Si modifichi l'Esercizio risolto 12.14 in modo che l'array abbia elementi di tipo `nodo*` anziché `int` e che, alla fine del calcolo, ogni elemento dell'array punti ad un diverso nodo dell'albero.

2. Si chiede di scrivere un programma che faccia lo stesso calcolo del programma dell'Esercizio 12.14 seguendo lo stesso schema con le due funzioni `faie` e `fairic`, ma, nella funzione `fairic`, il parametro `x` deve venire passato per valore anziché per riferimento. Per compensare la perdita del passaggio per riferimento, sarà necessario modificare il prototipo di `fairic` in qualche modo.

12.3 Alberi binari di ricerca

Gli alberi binari sono una struttura dati utile per gestire insiemi di dati. In generale un dato consiste di una **chiave** che identifica univocamente il dato stesso e un insieme di informazioni, spesso chiamato un record o una struttura in C++. Queste informazioni dipendono dal campo d'applicazione considerato. Tutto quello che segue è completamente indipendente dalle specifiche informazioni che compongono ciascun dato. Ci interessa solo la chiave che deve soddisfare il requisito di assumere valori da un insieme su cui è definito un ordine totale, cioè tale che per ogni coppia di valori distinti, uno è necessariamente maggiore dell'altro. In quanto segue un dato verrà implementato da un nodo di un albero binario. Quindi d'ora in avanti i nodi degli alberi verranno realizzati dalla seguente struttura:

```
struct nodo{int key,info; nodo* left,*right;
nodo(int a=0, int b=0, nodo* c=0, nodo* d=0)
{key=a; info=b; left=c; right=d;}};
```

Ovviamente la chiave di un nodo sarà contenuta nel suo campo `key`, mentre `info` conterrà le informazioni che, come già spiegato, sono irrilevanti per i nostri esercizi. Per semplicità il campo `key` (e anche l'`info`) è dichiarato intero. Ovviamente gli interi sono totalmente ordinati.

Gestire dei dati come quelli descritti significa riuscire ad eseguire delle operazioni su questi dati in modo efficiente rispetto al numero dei dati gestiti. Le operazioni assolutamente essenziali sono la ricerca di un dato data una chiave, la cancellazione di un dato data la chiave e l'inserimento di un nuovo dato nell'insieme dei dati. Gli alberi binari, *opportunamente organizzati*, permettono di eseguire in modo piuttosto efficiente queste 3 operazioni (e anche altre). Per prima cosa vediamo come *organizzare opportunamente* gli alberi binari.

Definizione 12.2 *Un albero binario si dice di ricerca se ogni suo nodo n ha campo `key` maggiore dei campi `key` dei nodi del suo sotto-albero sinistro e minore dei campi `key` dei nodi del sotto-albero destro.*

Si osservi che dalla precedente Definizione segue che un albero binario di ricerca ha nodi con chiavi tutte distinte tra loro. Un albero binario di ricerca ha delle proprietà sorprendenti. Per prima cosa osserviamo che la stampa infissa delle chiavi contenute nei nodi di un albero binario di ricerca, produce la stampa ordinata delle chiavi dalla più piccola alla più grande. La funzione di stampa è `infix_stampa` descritta in Sezione 12.2 con la sola modifica di stampare il campo `key` anziché

info. Che questa funzione stampi i dati in ordine lo si può dimostrare con l'induzione. Il **Caso base** è, come al solito, l'albero vuoto. Non c'è niente da stampare, quindi la stampa è ordinata banalmente. Per il **Passo induttivo** in cui cioè n sia diverso da 0, osserviamo che in primo luogo vengono stampate le chiavi del sottoalbero sinistro di n e questa stampa è in ordine per l'**Ipotesi induttiva**. Poi viene stampato $n \rightarrow \text{key}$ che è maggiore delle chiavi stampate prima, visto che l'albero è binario di ricerca. Quindi fino alla stampa di $n \rightarrow \text{key}$ i valori stampati sono in ordine e continuano ad esserlo anche dopo la seconda invocazione ricorsiva perchè le chiavi del sottoalbero destro sono maggiori di $n \rightarrow \text{key}$ e sono tra loro ordinati per l'**Ipotesi induttiva**.

Non dovrebbe sorprendere che cercare un nodo con una data chiave in un albero binario di ricerca sia molto più efficiente (in generale) che cercarlo in un albero binario qualsiasi (non di ricerca). La nuova funzione di ricerca segue.

```
/*PRE=(R punta ad un albero binario
di ricerca, possibilmente vuoto) */
nodo* ricerca_abr(nodo*R, int y)
{
    if(R)
    {
        if(R->key==y)
            return R;
        if(R->key >y)
            return ricerca_abr(R->left,y);
        else
            return ricerca_abr(R->right,y);
    }
    return 0;
}
/*POST=(se restituisce n!=0 => n punta ad un nodo con chiave
uguale a y)&&(se restituisce 0 => non esiste alcun nodo con
chiave uguale a y) */
```

È utile confrontare questa nuova funzione con quella vista nell'Esercizio 12.5. Nella `ricerca_abr` se R non è il nodo cercato, allora confrontando $R \rightarrow \text{key} > y$ è possibile decidere se continuare la ricerca nel sotto-albero sinistro oppure in quello destro. Quindi ad ogni nodo R considerato, si elimina metà dell'albero rimasto da esaminare (naturalmente se l'albero è bilanciato). È facile capire che questa ricerca può al massimo percorrere tutti i nodi di un cammino quindi il lavoro da fare è legato all'altezza dell'albero. Facciamo qualche conto. Se l'albero binario è bilanciato ed ha altezza n allora avrà $2^{n+1} - 1$ nodi. Quindi `ricerca_abr` visita $n + 1$ nodi per eseguire la ricerca tra $2^{n+1} - 1$ nodi e questo è vero sia che la ricerca abbia successo sia che non riesca. La funzione di ricerca descritta nell'Esercizio 12.5 può invece dover visitare tutti i $2^{n+1} - 1$ nodi nel caso peggiore

(di insuccesso). Una bella differenza! Bisogna capire bene che questo argomento è vero solo quando l'albero è bilanciato.

L'inserimento di un nuovo nodo in un albero binario di ricerca non può venire fatto come nell'Esercizio 12.6, infatti la sua posizione è determinata dalla chiave y che il nuovo nodo deve possedere, rispetto ai valori già presenti. In effetti il nuovo nodo va inserito esattamente dove lo cercheremmo con `ricerca_abr` e la similarità tra la funzione di ricerca e quella di inserimento che segue va assolutamente osservata.

```
/*PRE=(R è un albero binario di ricerca corretto) */
void inserimento_abr(nodo*&R, int y)
{
    if(!R)
        R=new nodo(y); // info=left=right=0
    else
        if(R->key==y)
            throw(...)// vietato inserire chiave già presente
        if(R->key > y)
            inserimento_abr(R->left,y);
        else
            inserimento_abr(R->right,y);
}
/*POST=(R è albero binario di ricerca corretto,
ottenuto da quello iniziale
aggiungendo una foglia con chiave uguale a y)*/
```

La precondizione di `inserimento_abr` è che inizialmente l'albero con radice R è albero binario di ricerca. La postcondizione è che il nuovo albero, ottenuto dopo l'inserimento, sia ancora binario di ricerca. Si osservi che la funzione può sollevare un'eccezione nel caso scopra che si cerca di inserire una chiave già presente. Ovviamente la prova di correttezza non esamina questo caso, ma solo quello di terminazione normale della funzione. Il **Caso base** è ovvio: se l'albero è vuoto all'inizio, esso avrà un solo nodo alla fine ed un albero con un solo nodo è binario di ricerca. Per il **Passo induttivo**, basta osservare che la funzione sceglie di inserire il nuovo nodo nel sotto-albero che è determinato dal rapporto tra y e $R->key$ che mantenga la proprietà di essere un albero binario di ricerca. L'**Ipotesi induttiva** garantisce che l'inserimento in questo sotto-albero produce un albero binario di ricerca con il nuovo nodo con chiave y e, da quanto detto prima sulla scelta del sotto-albero in cui inserire il nuovo nodo, anche il nuovo albero con radice R è ancora binario di ricerca.

Il nodo con chiave minima e quello con chiave massima sono facili da trovare in un albero binario di ricerca. La funzione che trova il nodo minimo è la seguente.

```
//PRE=(R è un albero binario di ricerca corretto)
nodo* min(nodo* R)
```

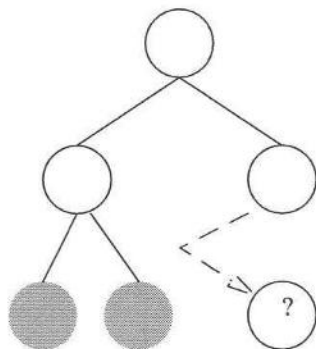


Figura 12.2: Due nodi si possono aggiungere, ma uno no

```

{
  if (R)
    if (!R->left)
      return R;
    else
      return min(R->left);
  return 0;
}
/* POST=(se restituisce n !=0 => n punta al nodo con chiave
  minima) && (se restituisce 0 => l'albero è vuoto) */

```

Per capire che `min` è corretta, basta osservare che il nodo che questa funzione individua è sempre quello la cui chiave è la prima che viene stampata da `stampa_abr` (cioè la prima nell'ordine infisso) e quindi sempre la più piccola.

12.4 Un esercizio da esame

Il problema che vogliamo affrontare è quello di aggiungere nodi ad un albero corretto qualsiasi (anche vuoto). Lo faremo sia in modo iterativo che ricorsivo. I nodi da aggiungere sono rappresentati da una sequenza di cammini composti da 0/1 e per ogni cammino si tratta di percorrere il cammino nell'albero per aggiungere un nuovo nodo alla fine del cammino percorso. Ovviamente, per un dato albero, ci sono cammini che non permettono di aggiungere un nodo all'albero, per esempio, può succedere che il cammino sia *troppo corto* e quindi il posto in cui aggiungere il nodo già occupato da un nodo dell'albero, oppure può succedere che il cammino sia *troppo lungo* e che il nodo da aggiungere non abbia un nodo padre nell'albero a cui venire appeso. Vediamo un esempio.

Esempio 12.16 Si consideri l'albero composto dai 3 nodi bianchi della Fig. 12.2. Si chiede di aggiungere a quest'albero 4 nodi, uno per ciascuno dei seguenti cammini: 0 0 -1 0 1 -1 1 0 1 -1 -1. Ogni cammino termina con -1. Si

osservi che il quarto cammino consiste solo di -1 e quindi si tratta del cammino vuoto. Il primo cammino causa l'aggiunta del nodo scuro più a sinistra dell'albero di Fig. 12.2, il secondo cammino causa l'aggiunta del secondo nodo scuro. Il terzo cammino non funziona perché richiede l'esistenza del figlio sinistro del figlio destro della radice che non c'è nell'albero di partenza. Questo fatto è indicato in Fig. 12.2 con la freccia tratteggiata. Quindi nessun nodo verrà aggiunto in corrispondenza di questo cammino. Anche il cammino vuoto -1 non porta alcun nodo nuovo perché l'albero possiede già la radice. Quindi il terzo cammino è troppo lungo mentre il quarto è troppo corto. build_ric

Cominciamo dalla soluzione ricorsiva. La prima funzione è `build_ric` che riceve comè parametri un array `C` che contiene la sequenza dei cammini da usare per aggiungere nodi all'albero `r`. La funzione deve restituire col `return` l'albero prodotto dall'aggiunta dei nodi corrispondenti a tutti i cammini in `C`. Questa sequenza di cammini è sempre conclusa dalla sentinella `-2`. Quindi la sequenza dell'esempio precedente in `C` sarebbe: `0 0 -1 0 1 -1 1 0 1 -1 -1 -2`. Inoltre, ogni volta che si deve aggiungere un nuovo nodo, il suo campo `info` verrà letto da `cin`.

```
/* PRE=(C contiene una sequenza di cammini,
   ciascuno di essi termina con -1 e la
   sequenza termina con -2, r è un albero corretto, sia r=vr)
nodo* build_ric(int*C, nodo*r, ifstream & INP, ofstream & OUT)
{
    bool fine=false;
    for(int i=0; !fine; i++)
    {
        int j=inizio(C,i);

        if(j==-2)
            fine=true;
        else
        {
            add_ric(C+j, r);
        }
    }
    return r;
}
/* POST=(r è vr a cui sono stati aggiunti nodi corrispondenti
   ai cammini in C esaminati da sinistra a destra. Se un cammino
   è troppo corto o troppo lungo, lo si è scritto su cout) */
```

`build_ric` usa due funzioni ausiliarie: `inizio` e `add_ric`. La prima è iterativa, mentre la seconda è ricorsiva. Ecco le loro pre- e postcondizioni assieme

al loro prototipo.

PRE_inizio=(C contiene n cammini, ogni cammino termina con -1, e dopo l'ultimo cammino (che termina, come tutti i cammini, con -1) c'è la sentinella -2, $n \geq 0$, $i \geq 0$)

int inizio(int* C, int i)

POST_inizio=(se $i \in [0..n-1]$ allora inizio_cam restituisce l'indice di C in cui inizia il cammino i, altrimenti, inizio_cam restituisce -2)

PRE_add=(C[0...] contiene una sequenza w di 0 e 1 (possibilmente vuota) seguita da -1, r è un albero corretto, sia $r=vr$)

void add_ric(int* C, nodo*& r)

POST_add=(se w è troppo corto o troppo lungo rispetto a vr, allora v resta uguale a vr e su cout verrà scritto *cammino troppo corto* oppure *cammino troppo lungo*, altrimenti r è un albero corretto che si ottiene da vr aggiungendogli un nuovo nodo tale che il cammino dalla radice di r al nuovo nodo sia w e tale che il suo campo info sia un valore letto da cin)

La funzione inizio presenta maggiori difficoltà di quanto ci si potrebbe aspettare. Come al solito esse si nascondono nei casi limite: nessun cammino, cammino vuoto, e i maggiore del numero dei cammini presenti. C'è anche un'altra difficoltà. Se vogliamo trovare l'inizio del cammino di indice i, dobbiamo scorrere i primi $i-1$ cammini e a quel punto non sappiamo se il prossimo cammino c'è oppure se in C c'è -2. Dobbiamo essere pronti a entrambi i casi. Questo viene fatto nel ciclo di inizio. Una soluzione corretta rispetto alle pre- e post-condizioni date, è la seguente. Essa usa una funzione ausiliaria scorri che scorre C fino al prossimo cammino, se c'è.

```
/* PRE=(C[x..] contiene un cammino terminato da -1)*/
int scorri(int* C, int x)
{
    int i;
    for( i=0; C[x+i]!=-1; i++);
    return x+i+1;
}
/*POST=(restituisce la posizione in C
immediatamente dopo il cammino che inizia in C[x]) */
int inizio(int* C, int i)
{
    int x=0;
    for(int count=0; (count < i) && (C[x]!=-2); count++)
        x=scorri(C,x);

    if(C[x]!=-2)
        return x;
    else
```

```

    return -2;
}

```

La correttezza di `scorri` è facile, vista la sua **PRE**: `C[x..]` contiene sicuramente un cammino quindi prima o poi si trova il `-1`. In particolare non ci può essere un `-2` prima del `-1`.

L'invariante del ciclo di `inizio` è come segue:

R = (x è tale che $0 \leq x < C[0]$ oppure $C[x] = -2$) è l'inizio del cammino di indice `count` e `count ≤ i`)

Visto che la negazione della condizione di permanenza è: `count=1 || C[x]=-2`, chiaramente in entrambi i casi il test finale fa la cosa giusta per garantire la **POST** di `inizio`.

Veniamo ora alla `add_ric`.

```

/*PRE=(C[0..] contiene un cammino
ed r è un albero corretto, sia r=vr) */
void add_ric(int*C, nodo*&r)
{
    if (C[0]==-1)
        if (!r)
        {
            r=new nodo();
            cout<<endl<<"inserire un intero ";
            cin>>r->info;
        }
    else
        cout<<"cammino troppo corto"<<endl;
    else
    {
        if (r)
            if (C[0]==0)
                add_ric(C+1, r->left);
            else
                add_ric(C+1, r->right);
        else
            cout<<endl<<"cammino troppo lungo";
    }
}

/*POST=(se il cammino in C è troppo lungo/corto per r
stampa un messaggio d'errore, altrimenti
r è vr con un nodo aggiunto che corrisponde al cammino,
il suo campo info è letto da cin) */

```

I casi base sono evidentemente corretti: se finisce l'albero prima del cammino oppure finisce il cammino prima dell'albero, siamo in presenza di un cammino troppo

lungo o troppo corto, rispettivamente. Se cammino e albero finiscono contemporaneamente, si crea un nuovo albero che consiste del solo nuovo nodo e lo si assegna a *r* che, grazie al passaggio per riferimento, aggancia il nuovo nodo al padre e all'albero iniziale considerato. Nei casi ricorsivi si arriva quando il cammino non è finito su *C* e l'albero non è vuoto, quindi la **PRE** vale per le invocazioni ricorsive e la **POST_ric** ci garantisce che esse faranno la cosa giusta nel sotto-albero di cui si occupano. Nel caso in cui il nodo viene aggiunto, è ovvio che esso venga aggiunto lungo il cammino richiesto, visto che la ricorsione percorre l'albero seguendo i valori del cammino. Resta da osservare che se le invocazioni ricorsive arrivano al caso di cammino troppo lungo/corto, significa che anche il cammino originale era troppo lungo/corto per l'albero *r*.

Vogliamo ora eseguire la stessa operazione di `add_ric`, ma iterativamente. Lo facciamo con la seguente funzione `add_iter`. Pre- e postcondizione sono le stesse della funzione `add_ric`.

```
void add_iter(int* C, nodo*& r)
{
    nodo*avanti=r, *indietro=0;
    bool lungo=false;
    int i=0;
    while(C[i]!=-1 && !lungo)
    {
        if(avanti)
        {
            if(C[i]==0)
            {
                indietro=avanti;
                avanti=avanti->left;
            }
            else
            {
                indietro=avanti;
                avanti=avanti->right;
            }
        }
        else
        {cout<<endl<<"cammino troppo lungo "; lungo=true;}
        i++;
    }
    //fine ciclo while
    if(!lungo)
    {//qui è sempre vero che C[i]=-1

        if(avanti)
```

```

        {cout<<"cammino troppo corto ";}
    else
    {
        int x;
        cout<<"Inserire un intero ";
        cin>>x;
        if(!indietro)
            {r=new nodo(x,0,0);}
        else
        {
            if(C[i-1]==0)
                indietro->left=new nodo(x,0,0);
            else
                indietro->right=new nodo(x,0,0);
        }
    }
}
}

```

`add_iter` prima cerca di percorrere il cammino in `C` e contemporaneamente lo stesso cammino sull'albero `r`. Lo fa usando 2 variabili `avanti` e `indietro`, in cui la prima indica dove si è arrivati nell'albero seguendo il cammino, mentre `indietro` punta al padre di `avanti`. Questo servirà nel caso in cui il cammino funziona e il nodo va aggiunto. Infatti in questo caso troviamo `-1` in `C`, `avanti` è `0` e il nuovo nodo andrà appeso a `indietro`, e per sapere se deve essere appeso a sinistra o a destra, dovremo guardare il valore dell'elemento del cammino che precede il `-1`. In questo ciclo potremo quindi accorgerci se l'albero finisce troppo presto rispetto al cammino e quindi segnalarlo con l'apposito output. Se usciamo dal ciclo con `lungo=false`, l'albero non è troppo corto rispetto al cammino, ma potrebbe essere il contrario. Ovviamente per saperlo basta testare `avanti`. Se non è `0`, stampiamo che il cammino è troppo corto. Se `avanti` è `0` dobbiamo creare il nuovo nodo e appenderlo all'albero, ma c'è ancora da considerare il caso speciale in cui l'albero iniziale fosse vuoto (e quindi anche il cammino fosse vuoto). Questo caso lo riconosciamo perché `indietro` è ancora a `0`, il che indica appunto che il ciclo non è stato mai eseguito. In quel caso `r` diventa il nuovo nodo. Se invece `indietro` non è `0`, allora appendiamo il nuovo nodo a `indietro`. Capire se a sinistra o a destra è l'ultima piccola difficoltà dell'esercizio. L'invariante del ciclo di `add_iter` è il seguente:

$$R = \begin{cases} \text{se } i=0 \Rightarrow (\text{indietro}=0, \text{avanti}=r) \\ \text{se } i>0 \ \&\& \ !\text{lungo} \Rightarrow (\text{avanti} \text{ è nodo di } r \text{ che corrisponde} \\ \text{al percorso di } C[0..i-1] \text{ partendo da } r \text{ e } \text{indietro} \text{ è il padre di } \text{avanti}) \\ \text{se } \text{lungo} \Rightarrow (C[0..i-1] \text{ è troppo lungo per } r) \end{cases}$$

Per avere una soluzione iterativa completa, basta modificare `build_ric` in modo che invochi `add_iter`.

Programmazione consapevole

Scrivere un programma non è una cosa semplice. Per farlo bene serve innanzitutto capire il problema che si vuole risolvere con il programma. Sembra un'affermazione ovvia, ma non è per nulla ovvia e ancora meno ovvio è il fatto che nella descrizione precisa di un problema ci sia già molto della sua soluzione e del programma che la realizza. Cardine della trattazione è l'utilità di sviluppare programmi la cui correttezza sia dimostrabile in modo semplice e convincente.

Il testo fornisce una grande collezione di esercizi risolti attraverso un ragionamento che conduce alla scelta di una soluzione semplice. Infatti convincersi della correttezza dei programmi è tanto più semplice quanto più semplici sono i programmi stessi.

Lo scopo principale del testo è quello di sviluppare negli studenti la capacità di ragionare sui problemi in modo da riuscire a trovare delle soluzioni chiare e semplici. Questa capacità di ragionare deve venire sostenuta da un'approfondita conoscenza del linguaggio di programmazione usato. Il testo contiene una descrizione del linguaggio C++ (senza le classi) che, oltre a spiegare i comandi del linguaggio, illustra cosa succede quando i programmi vengono eseguiti. Queste nozioni sono necessarie per una programmazione veramente consapevole.

Gilberto Filè è ordinario di Informatica presso la Facoltà di Scienze MM.FF.NN. dell'Università di Padova dal 1987.

La sua carriera sia scientifica che didattica è incentrata sullo studio dei linguaggi di programmazione e della programmazione.

In particolare, dal 2000 il Prof. Filè ha la responsabilità del corso di Programmazione che fa parte del primo anno della Laurea triennale in Informatica. Questo testo è quindi il frutto di una grande esperienza di insegnamento della programmazione.