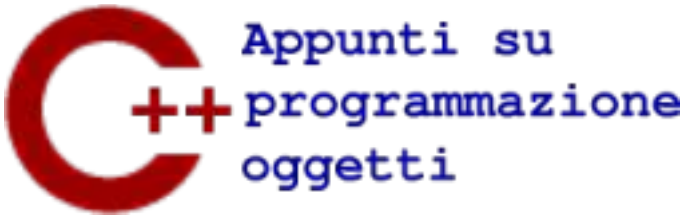


(2006.05)

Indice

1 Introduzione alla programmazione

1.1 Concetti fondamentali sugli algoritmi

Per molto tempo si pensò che il termine *algoritmo* derivasse da una storpiatura del termine *logaritmo*. L'opinione attualmente diffusa è invece che il termine derivi da *al-Khuwarizmi*, nome derivante a sua volta dal luogo di origine di un matematico arabo, autore di un libro di aritmetica e di uno di algebra: nel libro di aritmetica si parla della cosiddetta numerazione araba (quella attualmente usata) e si descrivono i procedimenti per l'esecuzione delle operazioni dell'aritmetica elementare. Questi procedimenti vennero in seguito chiamati algoritmi e il termine passò ad indicare genericamente qualunque *procedimento di calcolo*.

L'algoritmo esprime le *azioni* da svolgere su determinati *oggetti* al fine di produrre gli *effetti* attesi. Una azione che produce un determinato effetto è chiamata **istruzione** e gli oggetti su cui agiscono le istruzioni possono essere **costanti** (valori che restano sempre uguali nelle diverse esecuzioni dell'algoritmo) e **variabili** (contenitori di valori che vengono modificati ad ogni esecuzione dell'algoritmo). Si potrà dire brevemente che un algoritmo è una elaborazione di dati: i dati, cioè l'insieme delle informazioni che devono essere elaborate, sono manipolati, secondo le modalità descritte dalle istruzioni, per produrre altri dati. Ciò porta l'algoritmo ad essere una funzione di trasformazione dei dati di un insieme A (dati di input) in dati di un insieme B (dati di output).

In questi appunti, dato che ci si pone il fine di una introduzione alla programmazione, più che una definizione rigorosa di algoritmo se ne fornirà una definizione intuitiva. In questo senso si può definire l'algoritmo come *".. un insieme di istruzioni che definiscono una sequenza di operazioni mediante le quali si risolvono tutti i problemi di una determinata classe"*.

Per chiarire meglio il concetto di algoritmo è bene fare riferimento ad alcune proprietà che un insieme di istruzioni deve possedere affinché possa chiamarsi algoritmo:

- La **finitezza**. Il numero di istruzioni che fanno parte di un algoritmo è finito. Le operazioni definite in esso vengono eseguite un numero finito di volte.
- Il **determinismo**. Le istruzioni presenti in un algoritmo devono essere definite senza ambiguità. Un algoritmo eseguito più volte e da diversi esecutori, a parità di premesse, deve giungere a medesimi risultati. L'effetto prodotto dalle azioni descritte nell'algoritmo non deve dipendere dall'esecutore o dal tempo.
- La **realizzabilità pratica**. Tutte le azioni descritte devono essere eseguibili con i mezzi di cui si dispone.
- La **generalità**. Proprietà già messa in evidenza nella definizione che si è data: un algoritmo si occupa della risoluzione di famiglie di problemi.

1.2 Tipi di istruzioni

Per quanto osservato nell'ultima proprietà espressa, gli algoritmi operano principalmente su variabili che conterranno i dati sui quali si vuole svolgere una determinata elaborazione. I valori da elaborare devono essere *assegnati* alle variabili prima di effettuare l'elaborazione. Si pensi infatti ad una variabile come ad un contenitore. Le istruzioni operano sui valori contenuti: se questi non ci sono non ci si può attendere alcuna elaborazione. Ogni variabile è identificata da un nome che

permette di distinguerla dalle altre.

- L'**istruzione di assegnamento** fa in modo che un determinato valore sia conservato in una variabile. In questo modo si prepara la variabile per l'elaborazione o si conserva nella variabile un valore intermedio prodotto da una elaborazione precedente. Si può assegnare ad una variabile un valore costante come anche il valore risultante da una espressione aritmetica.
- L'**istruzione di input** fa in modo che l'algoritmo riceva dall'esterno un valore da assegnare ad una variabile. Nel caso di algoritmi eseguiti da un elaboratore, questi attende che da una unità di input (per esempio la tastiera) arrivi una sequenza di caratteri tipicamente terminanti con la pressione del tasto *Invio*. Il dato verrà assegnato alla variabile appositamente predisposta. Praticamente si tratta di una istruzione di assegnamento solo che, stavolta, il valore da assegnare proviene dall'esterno.
- L'**istruzione di output** fa in modo che l'algoritmo comunichi all'esterno i risultati della propria elaborazione. Nel caso di un elaboratore viene inviato su una unità di output (per esempio il video) il valore contenuto in una determinata variabile.

Vengono proposte, di seguito, due versioni di un algoritmo per il calcolo dell'area di un rettangolo (i nomi delle variabili sono scritti in maiuscolo):

Algoritmo A

```
Assegna a BASE il valore 3
Assegna a ALTEZZA il valore 7
Assegna a AREA valore BASE*ALTEZZA
Comunica AREA
```

Algoritmo B

```
Ricevi BASE
Ricevi ALTEZZA
Assegna a AREA valore BASE*ALTEZZA
Comunica AREA
```

Nell'algoritmo A si assegnano alle variabili `BASE` ed `ALTEZZA` dei valori costanti, l'algoritmo calcola l'area di un rettangolo particolare e se si vuole applicare l'algoritmo ad un diverso rettangolo è necessario modificare le due istruzioni di assegnamento a `BASE` e `ALTEZZA`: l'algoritmo ha perso la sua caratteristica di generalità.

Questo esempio non deve portare a concludere che non ha senso parlare di costanti in un algoritmo perché, per esempio, se si fosse trattato di un triangolo il calcolo dell'area avrebbe assunto l'aspetto: `Assegna a AREA valore BASE*ALTEZZA/2`. La costante `2` prescinde dai valori diversi che possono essere assegnati a `BASE` e `ALTEZZA`, essendo parte della formula del calcolo dell'area di un triangolo *qualsiasi*.

Nell'algoritmo B i valori da assegnare a `BASE` e `ALTEZZA` provengono da una unità di input. L'algoritmo ad ogni esecuzione si fermerà in attesa di tali valori e il calcolo verrà eseguito su tali valori: l'elaborazione è sempre la stessa (l'area di un rettangolo si calcola sempre allo stesso modo) ma i dati saranno di volta in volta diversi (i rettangoli hanno dimensioni diverse).

1.3 Le strutture di controllo

Gli algoritmi, a causa della loro generalità, lavorano utilizzando variabili. Non si conoscono, al momento della stesura dell'algoritmo stesso, i valori che possono assumere le variabili. Ciò se permette di scrivere algoritmi generali può comportare problemi per alcune istruzioni: si pensi al problema apparentemente banale del calcolo del quoziente di due numeri:

```
Ricevi DIVIDENDO
Ricevi DIVISORE
Assegna a QUOZIENTE valore DIVIDENDO/DIVISORE
```

```
Comunica QUOZIENTE
```

Il quoziente può essere calcolato se `DIVISORE` contiene un valore diverso da `0`, evento questo non noto al momento della stesura dell'algoritmo dipendendo, tale valore, dal numero proveniente da input. Inoltre è chiaro che, nella eventualità si presentasse il valore nullo, sarebbe priva di senso anche l'istruzione di output del valore di `QUOZIENTE` non essendoci nella variabile alcun valore.

È necessario introdurre, oltre alle istruzioni, degli strumenti che permettano di controllare l'esecuzione dell'algoritmo in conseguenza di eventi che si verificano in sede di esecuzione: le strutture di controllo. La **programmazione strutturata** (disciplina nata alla fine degli anni '60 per stabilire le regole per la scrittura di buoni algoritmi) impone l'uso di tre sole regole di composizione degli algoritmi:

- **la sequenza:** è l'unica struttura di composizione che si è utilizzata finora. In poche parole questa struttura permette di specificare l'ordine con cui le istruzioni si susseguono: ogni istruzione produce un risultato perché inserita in un contesto che è quello determinato dalle istruzioni che la precedono. Nell'esempio di prima il calcolo di `QUOZIENTE`, per poter contenere il valore atteso, deve essere eseguito dopo gli input.
- **la selezione:** questa struttura permette di scegliere tra due alternative la sequenza di esecuzione. È la struttura che ci permette, per esempio, di risolvere in modo completo il problema del calcolo del quoziente fra due numeri:

  ```
  Ricevi DIVIDENDO
  Ricevi DIVISORE
  Se DIVISORE <> 0
    Assegna a QUOZIENTE valore DIVIDENDO/DIVISORE
    Comunica QUOZIENTE
  Altrimenti
    Comunica 'Operazione senza senso'
  Fine-se
  ```

  La condizione espressa nella struttura `Se` permette di scegliere, in relazione al valore di verità o falsità, quale elaborazione svolgere. La sequenza contenuta nella parte `Altrimenti` potrebbe mancare se si volesse soltanto un risultato laddove possibile: in tale caso se la condizione `DIVISORE <> 0` risultasse non verificata, non si effettuerebbe alcuna elaborazione. L'esecuzione di questo algoritmo porterebbe l'esecutore, nel caso specifico il computer, a *decidere* cosa fare, quali istruzioni eseguire, in conseguenza dei dati che vengono introdotti. In sede di scrittura dell'algoritmo, non conoscendo i valori, si possono solo elencare le istruzioni da eseguire e fornire gli elementi (la condizione da verificare) per scegliere la corretta sequenza di elaborazione.
- **l'iterazione:** la struttura iterativa permette di ripetere più volte la stessa sequenza di istruzioni finché non si verifica una determinata condizione. Chiaramente non avrebbe alcun senso ripetere sempre le stesse istruzioni se non cambiassero i valori a cui si applicano le operazioni specificate nella sequenza. Le elaborazioni previste nella sequenza iterata ***devono*** potersi applicare a variabili che cambiano il loro valore: vuoi per una assegnazione diversa per ogni iterazione, vuoi per un input. A titolo di esempio, è riportato un algoritmo che *calcola e visualizza i quadrati di una serie di numeri positivi*. Si tratta, in altri termini, di effettuare la stessa elaborazione (calcolo e visualizzazione del quadrato di un numero) effettuata su numeri diversi (quelli che arriveranno dall'input):

```
Ricevi NUMERO
Mentre NUMERO > 0
  Assegna a QUADRATO valore NUMERO*NUMERO
  Comunica QUADRATO
  Ricevi NUMERO
Fine-mentre
```

Nel ***corpo*** della struttura iterativa (la parte compresa fra le parole `Mentre` e `Fine-mentre`) sono specificate le istruzioni per il calcolo del quadrato di un numero: l'iterazione permette di ripetere tale calcolo per tutti i numeri che verranno acquisiti tramite l'istruzione di input inserita nell'iterazione stessa che, non è superfluo sottolineare, fornisce un senso a tutta la struttura (i risultati, nonostante le stesse istruzioni, cambiano perché cambiano i valori). La condizione `NUMERO>0` viene chiamata ***condizione di controllo del ciclo*** e specifica quando deve terminare l'elaborazione (il valore introdotto da input è non positivo): si ricorda che l'algoritmo deve essere finito e non si può iterare all'infinito. Il primo input fuori ciclo ha lo scopo di permettere l'impostazione della condizione di controllo sul ciclo stesso e stabilire, quindi, quando terminare le iterazioni.

In generale si può dire che la struttura di una elaborazione ciclica, controllata dal verificarsi di una condizione, assume il seguente aspetto:

```
Considera primo elemento
Mentre elementi non finiti
  Elabora elemento
  Considera prossimo elemento
Fine mentre
```

Le strutture di controllo devono essere pensate come schemi di composizione: una sequenza può contenere una iterazione che, a sua volta, contiene una selezione che a sua volta può contenere dell'altro e così via. Rappresentano in pratica i mattoncini elementari di una scatola di montaggio le cui diverse combinazioni permettono la costruzione di architetture di varia complessità.

1.4 Accumulatori e contatori

L'elaborazione ciclica è spesso utilizzata per l'aggiornamento di totalizzatori o contatori. Per chiarire meglio il concetto di totalizzatore, si pensi alle azioni eseguite dal cassiere di un supermercato quando si presenta un cliente con il proprio carrello pieno di merce. Il cassiere effettua una elaborazione ciclica sulla merce acquistata: ogni oggetto viene esaminato per acquisirne il prezzo. Lo scopo della elaborazione è quello di cumulare i prezzi dei prodotti acquistati per stabilire il totale che il cliente dovrà corrispondere.

Dal punto di vista informatico si tratta di utilizzare una variabile (nell'esempio potrebbe essere rappresentata dal totalizzatore di cassa) che viene aggiornata per ogni prezzo acquisito: ogni nuovo prezzo acquisito non deve sostituire il precedente ma aggiungersi ai prezzi già acquisiti precedentemente. Tale variabile:

1. dovrà essere azzerata quando si passa ad un nuovo cliente (ogni cliente dovrà corrispondere solamente il prezzo dei prodotti che lui acquista)
2. si aggiornerà per ogni prodotto esaminato (ogni nuovo prezzo acquisito verrà cumulato ai precedenti)
3. finito l'esame dei prodotti acquistati la variabile conterrà il valore totale da corrispondere.

La variabile di cui si parla nell'esempio è quella che, nel linguaggio della programmazione, viene definita un **totalizzatore** o **accumulatore**: cioè una variabile nella quale ogni nuovo valore non sostituisce ma si aggiunge a quelli già presenti in precedenza. Se la variabile si aggiorna sempre di una quantità costante (per esempio viene sempre aggiunta l'unità) viene chiamata **contatore**.

In generale si può dire che l'uso di un totalizzatore prevede i seguenti passi:

```
Inizializzazione totalizzatore
Inizio ciclo aggiornamento totalizzatore
  ...
  Aggiornamento totalizzatore
Fine ciclo
Uso del totalizzatore
```

L'inizializzazione serve sia a dare senso all'istruzione di aggiornamento (cosa significherebbe la frase: *aggiorna il valore esistente con il nuovo valore* se non ci fosse un valore esistente?), sia a fare in modo che l'accumulatore stesso contenga un valore coerente con l'elaborazione da svolgere. Nell'esempio di prima il nuovo cliente non può pagare prodotti acquistati dal cliente precedente: il totalizzatore deve essere azzerato, prima di cominciare l'elaborazione, affinché contenga un valore che rispecchi esattamente tutto ciò che è stato acquistato dal cliente esaminato.

L'aggiornamento viene effettuato all'interno di un ciclo. Se infatti si riflette sulla definizione stessa di totalizzatore, è facile prendere atto che avrebbe poco significato fuori da un ciclo: come si può cumulare valori se non si hanno una serie di valori?

Quando i valori da esaminare terminano, il totalizzatore conterrà il valore cercato. Nell'esempio di prima tutto ciò si tradurrebbe: finito l'esame dei prodotti acquistati, si potrà presentare al cliente il totale da corrispondere.

A titolo di esempio di utilizzo di un accumulatore, viene presentata la risoluzione del seguente problema: *data una sequenza di numeri positivi, se ne vuole calcolare la somma*.

```
Inizializza SOMMA con valore 0
Ricevi NUMERO
Mentre NUMERO > 0
  Aggiorna SOMMA sommando NUMERO
  Ricevi NUMERO
Fine-mentre
Comunica SOMMA
```

Il totalizzatore `SOMMA` è inizializzato, prima del ciclo, al valore nullo poiché deve rispecchiare la somma dei numeri introdotti da input e, quindi, non essendo ancora stata effettuata alcuna elaborazione su alcun numero, tale situazione viene espressa assegnando il valore neutro della somma.

L'output di `SOMMA` alla fine del ciclo indica il fatto che si può utilizzare (in questo caso per la conoscenza del valore contenuto) il totalizzatore quando il suo contenuto è coerente con il motivo della sua esistenza: `SOMMA` deve accumulare tutti i valori e ciò avverrà quando tutti i numeri da considerare saranno stati elaborati, cioè in uscita dal ciclo.

1.5 Cicli a contatore

Una applicazione diffusa dei contatori è quella di controllo delle elaborazioni iterative. Ci sono delle elaborazioni cicliche in cui è noto a-priori il numero degli elementi da elaborare e, in questi

casi, un contatore, che si aggiorna ad ogni elaborazione effettuata (si pensi ad esempio ad un contachilometri di una automobile che si aggiorna in automatico ad ogni chilometro percorso), conteggia gli elementi che vengono trattati mano a mano. Appena il contatore raggiunge la quantità prestabilita, l'elaborazione ha termine.

In questi casi lo schema generale dell'elaborazione ciclica può assumere questo aspetto:

```
Ricevi Quantità elementi da elaborare
Per contatore da 1 a quantità elementi da elaborare
  Ricevi elemento
  elabora elemento
Fine-Per
```

Si tratta di un caso particolare dell'elaborazione ciclica. Rispetto all'elaborazione ciclica trattata precedentemente (il ciclo `Mentre` controllato dall'avere, una certa variabile, assunto un valore particolare) si possono notare alcune differenze:

Ciclo Mentre	*Ciclo Per*

`Considera primo elemento`
`Mentre elementi non finiti`
`  Elabora elemento`
`  Considera prossimo elemento`
`Fine mentre`
	`Ricevi Quantità elementi da elaborare`
	`Per contatore da 1 a quantità elementi`
	`  Ricevi elemento`
	`  elabora elemento`
	`Fine-Per`

Nel ciclo `Mentre` si acquisisce il primo elemento prima dell'inizio del ciclo e, quindi, quando si entra nel ciclo la prima cosa da fare è effettuare l'elaborazione dell'elemento che è stato ricevuto; l'input successivo prepara il prossimo elemento. Viene acquisito un elemento in più che funge da *tappo*: cioè serve solo ad avvisare che l'elaborazione è finita. Se ci sono, per esempio, 8 elementi da elaborare devono essere forniti 9 elementi (gli 8 da elaborare e l'*elemento tappo*). Il conteggio degli elementi elaborati, se necessario, può essere effettuato utilizzando un contatore.

Nel ciclo `Per` si acquisisce per prima cosa il numero rappresentante la quantità delle iterazioni. Subito dopo si può procedere con l'elaborazione ciclica: un contatore automatico si occupa di verificare se il valore contenuto nel contatore abbia raggiunto la quantità prefissata e, in questo caso, di bloccare l'iterazione. Non è necessario alcun input suppletivo.

2 Fondamenti di C++: costrutti di base

2.1 Cenni sui linguaggi di programmazione

I linguaggi di programmazione permettono di scrivere algoritmi eseguibili da un sistema di elaborazione. Un algoritmo scritto in un linguaggio di programmazione viene chiamato **programma** e il processo di scrittura del programma, a partire dall'algoritmo, viene chiamato **codifica**. I linguaggi di programmazione sono costituiti da un insieme di *parole chiavi* (le parole che hanno un senso in quel linguaggio), un insieme di *simboli speciali* (caratteri con particolari significati come separatori, simboli di fine istruzione, ecc) e da un *insieme di regole* (la sintassi del linguaggio) che devono essere rispettate per scrivere programmi sintatticamente corretti.

Il linguaggio macchina costituito da zero ed uno è l'unico che può controllare direttamente le unità fisiche dell'elaboratore in quanto è l'unico comprensibile dall'elaboratore stesso. È però estremamente complicato scrivere programmi in tale linguaggio, naturale per la macchina, ma completamente *innaturale* per l'uomo. Per poter permettere un dialogo più semplice con la macchina sono nati i linguaggi di programmazione.

Il più *vecchio* linguaggio di programmazione è il linguaggio assembly. Il linguaggio assembly è una rappresentazione simbolica del linguaggio macchina. La scrittura di programmi è enormemente semplificata: il linguaggio assembly utilizza simboli facili da ricordare e non incomprensibili sequenze binarie. Per essere eseguito dall'elaboratore un programma in linguaggio assembly deve essere tradotto in linguaggio macchina; tale lavoro è a carico di un programma detto *assemblatore.* Questi due tipi di linguaggi, detti anche linguaggi di *basso livello* sono propri di ogni macchina.

I linguaggi di *alto livello* sono più vicini al linguaggio naturale, sono orientati ai problemi piuttosto che all'architettura della macchina, rendono cioè la scrittura di un programma molto vicina a quella che si produrrebbe se l'esecutore fosse un umano, piuttosto che una macchina con esigenze e, per esempio, modi di gestire le parti di un elaboratore, che dipendono da come sono costruite e non dalle funzioni svolte. Non fanno riferimento ai registri fisicamente presenti sulla macchina ma a variabili. Per essere eseguiti devono essere tradotti in linguaggio macchina, e tale traduzione viene svolta da un programma detto **compilatore**.

I linguaggi di alto livello sono in larga misura indipendenti dalla macchina, possono essere eseguiti su qualsiasi elaboratore a patto che esista il corrispondente compilatore che ne permetta la traduzione.

I linguaggi di alto livello si caratterizzano per essere orientati a specifiche aree applicative. Questi linguaggi vengono anche detti della terza generazione.

Per ultimi in ordine di tempo sono arrivati i linguaggi della quarta generazione, ancora più spiccatamente rivolti a specifiche aree applicative e utilizzabili in modo intuivo dall'utente non esperto. Il più famoso di questi è SQL (Structured Query Language), che opera su basi dati relazionali. I linguaggi di IV generazione sono detti *non procedurali* poiché l'utente specifica la funzione che vuole svolgere senza entrare nel dettaglio di come verrà effettivamente svolta.

2.2 Il linguaggio C e il C++

Nel 1972, presso i Bell Laboratories, Dennis Ritchie progettava e realizzava la prima versione del linguaggio C. Ritchie aveva ripreso e sviluppato molti dei principi e dei costrutti sintattici del

linguaggio BCPL, sviluppato da Martin Richards, e del linguaggio B, sviluppato da Ken Thompson, l'autore del sistema operativo Unix. Successivamente gli stessi Ritchie e Thompson riscrissero in C il codice di Unix.

Il C si distingueva dai suoi predecessori per il fatto di implementare una vasta gamma di tipi di dati (carattere, interi, numeri in virgola mobile, strutture) non originariamente previsti dagli altri due linguaggi. Da allora ad oggi il C ha subito trasformazioni: la sua sintassi è stata affinata, soprattutto in conseguenza della estensione *object-oriented (C++)*. Il C++, come messo in evidenza dallo stesso nome, rappresenta una evoluzione del linguaggio C: il suo progettista (Bjarne Stroustrup) quando si pose il problema di trovare uno strumento che implementasse le classi e la programmazione ad oggetti, invece di costruire un nuovo linguaggio di programmazione, pensò bene di estendere un linguaggio già esistente, il C appunto, aggiungendo nuove funzionalità. In questo modo, contenendo il C++ il linguaggio C come sottoinsieme, si poteva riutilizzare tutto il patrimonio di conoscenze acquisito dai programmatori in C (linguaggio estremamente diffuso in ambito di ricerca) e si poteva fare in modo che tali programmatori avessero la possibilità di acquisire le nuove tecniche di programmazione senza essere costretti ad imparare un nuovo linguaggio e quindi senza essere costretti a disperdere il patrimonio di conoscenze già in loro possesso. Così le estensioni ad oggetti hanno fornito ulteriore linfa vitale al linguaggio C.

Questi appunti fanno riferimento all'ultima definitiva standardizzazione del linguaggio: il cosiddetto ANSI/ISO C++ del Dicembre 1997.

2.3 Struttura di un programma

In questa parte si esamineranno le strutture minime di C++ che permettono di codificare gli algoritmi, prodotti utilizzando le strutture di controllo, introducendo poche caratteristiche specifiche del linguaggio. I frammenti di programmi riportati utilizzeranno soltanto un tipo di dati. In questo contesto le differenze con la codifica in altri linguaggi di programmazione, sono minime. Dal capitolo successivo si cominceranno ad esaminare le caratteristiche peculiari del C++.

Il linguaggio C++, utilizzando un termine proprio, è *case-sensitive*: fa cioè distinzione fra maiuscole e minuscole. Le parole chiavi devono essere scritte utilizzando lettere minuscole. I nomi che sceglie il programmatore (per esempio i nomi di variabili) possono essere scritti utilizzando qualsiasi combinazione di caratteri minuscoli o maiuscoli. È necessario, però tenere in considerazione che, qualora si utilizzi un sistema misto nei nomi a scelta, essendo il linguaggio case-sensitive, c'è molto rischio di commettere errori. Anche se sono possibili altre soluzioni, è convenzione usare, anche nei nomi a scelta di chi scrive il programma, sempre lettere minuscole.

Qualsiasi programma in C++ segue lo schema:

```
#include <iostream>
using namespace std;

main(){
  // dichiarazioni delle variabili utilizzate
  ...
  // istruzioni del programma
  ...
}
```

La prima riga del listato serve per includere, nel programma, le funzionalità per l'utilizzo degli ***stream*** (flussi) di input e output. Il linguaggio C++ fornisce delle librerie già pronte contenenti

funzionalità riguardanti elaborazioni varie. Il meccanismo che sta alla base delle librerie, e il significato esatto della riga, saranno chiariti successivamente. Per il momento basta sapere che per poter utilizzare le funzionalità, disponibili in una determinata libreria, è necessario aggiungere una riga del genere con il nome della libreria stessa. Nel caso specifico, se non si aggiungesse tale riga, il programma non potrebbe comunicare con l'esterno per esempio con una tastiera e un video.

Anche il significato esatto della riga successiva (`using ...`) verrà chiarito in seguito. Per il momento basta dire che è necessaria quando si utilizzano librerie standard (`std`) del C++. La riga è conclusa con un `;` che è il carattere terminatore di una istruzione qualsiasi.

Il programma vero e proprio, per il momento composto da una sola parte, è racchiuso nel blocco, delimitato dalla coppia di parentesi `{}`, e preceduto da `main()`.

Il programma, nello schema proposto, è suddiviso, per motivi di leggibilità, in due parti distinte: dichiarazione delle variabili da utilizzare e istruzioni. Nel C++ una variabile può essere dichiarata in qualsiasi punto purché prima del suo utilizzo. Quest'ultima è una possibilità comoda quando si sviluppano programmi complessi, si ha necessità di utilizzare una variabile in un punto preciso e si vuole evitare di andare indietro nel listato per dichiarare la variabile e avanti nel punto interessato per utilizzarla, tuttavia penalizza la leggibilità e comprensibilità del programma.

In questi appunti le variabili saranno sempre dichiarate, tutte, nella parte iniziale.

Le righe precedute da `//` sono commenti: vengono tralasciate dal compilatore in sede di traduzione, ma sono utili al programmatore per chiarire il senso delle righe che seguono. Questo è un aspetto importante perché nella codifica si utilizzano linguaggi simbolici e le istruzioni non chiariscono il senso delle operazioni che si stanno facendo, a maggior ragione quando passa del tempo fra la stesura del programma e la sua lettura o quando il programma viene esaminato da persona diversa rispetto a quella che lo ha sviluppato. È necessario quindi aggiungere righe di commento in modo da isolare e chiarire le singole parti del programma.

I commenti possono, secondo l'uso del linguaggio C, iniziare dopo `/*` ed essere conclusi da `*/`. In questo caso il commento può espandersi in più righe: è il terminatore `*/` che dice al compilatore dove finisce il commento.

In questi appunti `/*` e `*/` sono utilizzati per segnare le righe dei listati che saranno commentate nel testo.

2.4 Lo stream di output

Quando si parla di input o di output senza meglio precisare, si fa riferimento alle *periferiche di default*: nel caso specifico tastiera e video. Se si avrà necessità di utilizzare periferiche diverse si dovrà specificare esplicitamente.

Il sistema operativo fornisce una interfaccia ad alto livello verso l'hardware: le periferiche sono mappate in memoria, è utilizzata cioè, in pratica, una parte della memoria centrale (il *buffer*) come deposito temporaneo dei dati da e verso le periferiche. In questo modo, per esempio, le operazioni di output possono essere effettuate sempre allo stesso modo a prescindere dalla periferica: sarà il sistema che si occuperà della gestione della specificità dell'hardware. Il sistema di I/O fornisce il concetto astratto di flusso o canale (lo *stream*). Con tale termine si intende un dispositivo logico indipendente dalla periferica fisica: chi scrive il programma si dovrà occupare dei dati che transitano per il canale prescindendo dalle specifiche del dispositivo fisico che sta usando (il video,

un lettore di dischi magnetici, una stampante). Lo stream di output, nel caso del video, è usato in maniera sequenziale: si possono accodare tutti gli output nel canale e il sistema provvede a stamparli uno di seguito all'altro.

```
// Programma per mostrare diversi modi di
// stampare su video

#include <iostream>
using namespace std;

main(){
  cout << "Prima riga ";                                          /*1*/
  cout << "seguito della riga" << endl;                           /*2*/
  cout << "Nuova riga molto lunga"
    " la riga continua ancora su video"
    "\n ora siamo passati ad una nuova riga"
       << endl;                                                   /*3*/
}
```

Se si compila, e si lancia l'esecuzione del programma, si ottiene:

```
Prima riga seguito della riga
Nuova riga molto lunga la riga continua ancora su video
 ora siamo passati ad una nuova riga
```

Nella `1` si incanala (l'operatore <<) nello stream `cout` la stringa `Prima riga`. Tutto ciò che è racchiuso fra " e " (il carattere doppio apice), come si nota nell'esecuzione, verrà visualizzato su video così come è. Il `;` chiude l'istruzione.

Nella `2`, nonostante si tratti di una nuova istruzione, la stringa, come si può notare nell'esecuzione, è visualizzata di seguito alla prima. Lo stream è utilizzato in modo sequenziale. Per andare a riga nuova si è accodato (operatore <<) il modificatore `endl` (*end-line*) che, appunto, termina la linea e fa passare alla prossima riga nel video.

L'istruzione `3` è suddivisa in più righe di listato. L'istruzione termina, al solito, con il `;` e consente di distribuire il testo da visualizzare su più righe fisiche. In ognuna si è scritta una stringa racchiusa dai soliti caratteri: `la riga continua...` verrà visualizzata dopo `Nuova riga...`. Per passare ad una nuova riga su video, si è utilizzato in questo caso il carattere di controllo `\n` dentro la stringa da visualizzare. Lo spazio successivo al carattere di controllo non è necessario ma è stato introdotto solo per evidenziarlo. In definitiva per poter passare alla linea successiva si può accodare allo stream di output `endl` o inserire nella stringa da stampare il carattere di controllo `\n`.

All'interno della stringa da stampare si possono inserire anche altri codici di controllo, tutti preceduti dal carattere di *escape* (\), di cui qui si fornisce un elenco di quelli che possono essere più utili:

- `\n` porta il cursore all'inizio della riga successiva
- `\t` porta il cursore al prossimo fermo di tabulazione (ogni fermo di tabulazione è fissato ad 8 caratteri)
- `\'` stampa un apice
- `\"` stampa le virgolette

2.5 Variabili ed assegnamenti

Si propone adesso un programma per il calcolo dell'area di un rettangolo i cui lati hanno valori interi. In questo caso sono necessarie due variabili per gli input, la base e l'altezza, una di output per contenere l'area.

```
//  Calcolo area rettangolo

#include <iostream>
using namespace std;

main(){
  // dichiarazioni delle variabili

  int base;
  int altezza;
  int area;

  // elaborazione richiesta

  base = 3;
  altezza = 7;
  area = base*altezza;
  cout << area;
}
```

Subito dopo `main()` sono presente le dichiarazioni delle variabili intere necessarie:

```
int base; int altezza; int area;
```

La parola chiave `int` specifica che l'identificatore che lo segue si riferisce ad una variabile di tipo intero; dunque `base`, `altezza` e `area` sono variabili di questo tipo.

Anche le dichiarazioni così come le altre istruzioni devono terminare con un punto e virgola. Nel nostro esempio alla dichiarazione del tipo della variabile corrisponde anche la sua definizione che fa sì che le venga riservato uno spazio in memoria centrale.

Il nome di una variabile la identifica, il suo tipo ne definisce la dimensione e l'insieme delle operazioni che vi si possono effettuare. La dimensione può variare rispetto all'implementazione del compilatore. Nella macchina su cui sono state effettuate le prove dei programmi riportati in questi appunti, ogni variabile di tipo `int` occupa, in memoria, 32 bit e può contenere un numero compreso fra -2147483648 e +2147483647. Si mostrerà in seguito un modo per controllare l'occupazione, per esempio di un `int`, in memoria e quindi i limiti di rappresentatività (il più piccolo e il più grande numero rappresentabile).

Tra le operazioni permesse fra variabili di tipo `int` vi sono: la somma (`+`), la sottrazione (`-`), il prodotto (`*`) e la divisione (`/`). È opportuno osservare che la divisione fra due `int` produce sempre un `int`: il valore del risultato è troncato alla parte intera.

Effettuata la dichiarazione, la variabile può essere utilizzata. L'istruzione

```
base = 3;
```

assegna alla variabile `base` il valore `3`; cioè inserisce nello spazio di memoria riservato a tale variabile il valore indicato. Effetto analogo avrà `altezza=7`. L'assegnamento è dunque realizzato mediante l'operatore `=`. L'assegnamento può essere effettuato contestualmente alla dichiarazione:

```
...
int base = 3;
int altezza = 7;
...
```

Nel linguaggio C++ è possibile assegnare lo **stesso valore** a più variabili contemporaneamente. Per esempio se le dimensioni riguardavano un quadrato, si sarebbe potuto scrivere:

```
base = altezza = 5;
```

In questo caso prima verrebbe assegnato il valore `5` alla variabile `altezza` e quindi, il risultato dell'assegnazione (cioè `5`), viene assegnato alla variabile `base`.

L'istruzione:

```
area = base * altezza;
```

assegna alla variabile `area` il prodotto dei valori di `base` e `altezza` .

L'ultima istruzione

```
cout << area;
```

visualizza `21`, il valore della variabile `area`. In questo caso `area`, non essendo racchiusa fra doppi apici, non è interpretata come stringa ma come variabile di cui visualizzare il contenuto.

Le dichiarazioni delle variabili dello stesso tipo possono essere scritte in sequenza separate da una virgola:

```
int base,altezza,area;
```

Dopo la dichiarazione di tipo sono specificati gli identificatori di variabile, che possono essere in numero qualsiasi, separati da virgola e chiusi da un punto e virgola. In generale quindi la dichiarazione di variabili ha la seguente forma:

```
tipo lista_di identificatori;
```

Esistono inoltre delle regole da rispettare nella costruzione degli identificatori: devono iniziare con una lettera o con un carattere di sottolineatura _ e possono contenere lettere, cifre e _. Per quanto riguarda la lunghezza occorre tenere presente che soltanto i primi trentadue caratteri sono significativi, anche se nelle versioni del C meno recenti questo limite scende a otto caratteri. Sarebbe comunque opportuno non iniziare il nome della variabile con il carattere di sottolineatura ed è bene tenere presente che le lettere accentate, permesse dalla lingua italiana, non sono considerate lettere ma segni grafici e le lettere maiuscole sono considerate diverse dalle rispettive minuscole.

Oltre a rispettare le regole precedentemente enunciate, un identificatore **non può essere una parola chiave del linguaggio**, **né può essere uguale ad un nome di funzione libreria o scritta dal programmatore**.

Allo scopo di rendere più chiaro l'effetto ottenuto dal programma dell'esempio precedente, si possono visualizzare i valori delle variabili `base` e `altezza`. È opportuno, per motivi di chiarezza, far precedere la visualizzazione dei valori da una descrizione.

```
cout << "Base: "     << base;
cout << " Altezza: " << altezza;
```

```
cout << " Area: "    << area;
```

Quello che si ottiene in esecuzione è

```
Base: 3 Altezza: 7 Area: 21
```

Per ottenere una distanza maggiore tra il valore di base e altezza e le seguenti descrizioni si possono aggiungere ulteriori spazi

```
cout << "Base: "      << base;
cout << "\tAltezza: " << altezza;
cout << "\tArea: "    << area;
```

così da avere

```
Base: 3    Altezza: 7    Area: 21
```

oppure formattare l'output in maniera che sia visualizzato in più righe:

```
// Calcolo area rettangolo

#include <iostream>
using namespace std;

main(){
  // dichiarazioni delle variabili

  int base,altezza,area;

  // elaborazione richiesta

  base = 3;
  altezza = 7;
  area = base*altezza;

  cout << “Base: ” << base << “ Altezza: ”<< altezza << endl;
  cout << “Area: ” << area;
}
```

L'esecuzione del programma sarà:

```
Base: 3 Altezza: 7
Area: 21
```

Mentre `int` è una parola chiave del C++ e fa parte integrante del linguaggio, `base`, `altezza` e `area` sono identificatori di variabili scelti a discrezione di chi scrive il programma. Gli stessi risultati si sarebbero ottenuti utilizzando, al loro posto, nomi generici quali `x`, `y` e `z` solo che il programma sarebbe risultato meno comprensibile.

La forma grafica data al programma è del tutto opzionale; una volta rispettata la sequenzialità e la sintassi, la scrittura del codice è libera. In particolare più istruzioni possono essere scritte sulla stessa linea. È indubbio però che, in questo ultimo caso, il programma risulterà notevolmente meno leggibile del precedente.

Lo stile grafico facilita enormemente il riconoscimento dei vari pezzi di programma, delle operazioni in essi effettuati e consente una diminuzione di tempo nelle modifiche, negli ampliamenti e nella correzione degli errori. In generale è inoltre bene dare alle variabili dei nomi

significativi, in modo che, quando si debba intervenire a distanza di tempo sullo stesso programma, si possa facilmente ricostruire l'uso che si è fatto di una certa variabile.

2.6 Lo stream di input

Il programma scritto in precedenza non calcola l'area di un qualsiasi rettangolo ma soltanto di quello che ha per base `3` e per altezza `7`, supponiamo centimetri.

Per rendere il programma più generale, si deve permettere a chi lo sta utilizzando di immettere i valori della base e dell'altezza; in questo modo l'algoritmo calcolerà l'area di un qualsiasi rettangolo.

```
cin >> base;
```

L'esecuzione di questa istruzione fa sì che il sistema attenda l'immissione di un dato da parte dell'utente e che lo vada a conservare nella variabile specificata.

In questa istruzione viene utilizzato il canale di input `cin` e l'*operatore di estrazione* `>>`. Si potrebbe interpretare l'istruzione come: estrai dal canale di input un dato e conservalo nella variabile specificata. Come già specificato in precedenza la definizione del canale di input si trova, come quella del canale di output, nella libreria `iostream`.

Durante l'esecuzione di un programma può essere richiesta all'utente l'immissione di più informazioni, è opportuno, allora, visualizzare delle frasi esplicative facendo precedere le istruzioni di input da opportuni messaggi chiarificatori.

```
cout << "Valore base: ";
cin >> base;
```

Quello che apparirà all'utente in fase di esecuzione del programma sarà

```
Valore base: _
```

In questo istante si attende che nel canale di input sia disponibile un valore da estrarre. Se l'utente digita `15` seguito da *Invio*

```
Valore base: 15
```

questo dato verrà assegnato alla variabile `base`.

Il programma del calcolo dell'area di un rettangolo, modificato in modo da consentire l'immissione di dati e con aggiunti commenti esplicativi, potrebbe assumere la forma:

```
// Calcolo area rettangolo

#include <iostream>
using namespace std;

main(){
  // dichiarazioni variabili

  int base,altezza,area;

  // descrizione programma e input
  cout << "Calcolo AREA RETTANGOLO \n \n";
  cout << "Valore base: ";
  cin >> base;
  cout << “Valore altezza: ”;
```

```
  cin >> altezza;

  // elaborazione e output

  area = base*altezza;

  cout << "\nBase: " << base << " Altezza: " << altezza << endl;
  cout << "Area: " << area << endl;
}
```

L'esecuzione del programma, nell'ipotesi che l'utente inserisca i valori `10` e `13`, sarà.

```
Calcolo AREA RETTANGOLO

Valore base: 10
Valore altezza: 13

Base: 10 Altezza: 13
Area: 130
```

Con una sola istruzione di input è possibile acquisire più di un valore, per cui i due input dell'esempio precedente, avrebbero potuto essere sostituiti da:

```
cout << "Introdurre Base e Altezza separati da uno spazio" << endl;
cin >> base >> altezza;
```

In questo caso, quando il programma viene eseguito, alla richiesta di input si risponderà con due numeri (che saranno assegnati rispettivamente a `base` e ad `altezza`), separati da uno spazio.

2.7 Costrutto if e dichiarazioni di costanti

Il costrutto `if` permette di codificare una struttura di selezione. La sintassi dell'istruzione `if` è:

```
if(espressione)
istruzione
```

dove la valutazione di `espressione` controlla l'esecuzione di `istruzione`: se `espressione` è vera viene eseguita `istruzione`.

A titolo di esempio viene proposto un programma che richiede un numero all'utente e, se tale numero è minore di `100`, visualizza un messaggio.

```
#include <iostream>
using namespace std;

const int limite=100;                                          /*1*/

main(){
  int i;

  cout << "Introdurre un valore intero ";
  cin >> i;
  if (i<limite)                                                /*2*/
    cout << "numero introdotto minore di " << limite << endl;  /*3*/
}
```

Il programma proposto utilizza la costante `100` che può essere utilizzata direttamente nelle istruzioni che la coinvolgono, ma è più conveniente assegnare ad essa un nome simbolico da utilizzare al

posto del valore. L'istruzione contenuta nella `1` dichiara una costante di tipo `int` con nome `limite` e valore `100`. Nelle istruzioni presenti nel programma, quando viene coinvolta la costante, viene utilizzato il nome, così come si può notare nelle istruzioni contenute nelle righe `2` e `3`.

La differenza fra la dichiarazione di variabile e la dichiarazione di una costante, dal punto di vista della sintassi del linguaggio, prevede la presenza, nel secondo caso, della parola chiave `const` e dell'assegnazione di un valore. Dal punto di vista dell'esecuzione una variabile, dichiarata come costante, non può essere modificata: se nel programma è presente qualche istruzione che comporta la modifica, per esempio nel caso proposto, di `limite`, il compilatore genera un messaggio di errore.

Il motivo dell'uso delle dichiarazioni di costanti risiede nel fatto che, in questo modo, all'interno del programma, compare solo il nome simbolico. Il valore compare una sola volta all'inizio del programma stesso e, quindi, se c'è necessità di modificare tale valore, per esempio, valutando se l'input dell'utente non superi `150`, qualsiasi sia la lunghezza del programma e quanti che siano gli utilizzi della costante, basta modificare l'unica linea della dichiarazione e ricompilare il programma affinché questo funzioni con le nuove impostazioni.

L' espressione `i<limite` presente in `2` è la *condizione logica* che controlla l'istruzione di stampa e pertanto la sua valutazione potrà restituire soltanto uno dei due valori booleani **vero** o **falso** che in C++ corrispondono rispettivamente ai valori interi **uno** e **zero**. È appunto per tale ragione che un assegnamento del tipo `a=i<limite`, è del tutto lecito. Viene infatti valutata l'espressione logica `i<limite`, che restituisce `1` (vero) se `i` è minore di `100` e `0` (falso) se `i` è maggiore uguale a `100`: il risultato è dunque un numero intero che viene assegnato alla variabile `a`.

Chiedersi se il valore di `a` è diverso da zero è lo stesso che chiedersi, per quanto osservato prima, se il valore di `a` è vero, il che, in C++, corrisponde al controllo eseguito per *default* (effettuato in mancanza di differenti indicazioni), per cui avremmo anche potuto scrivere

```
...
cin >> i;
a = i<limite;
if (a)
  cout << "numero introdotto minore di " << limite << endl;
```

La sintassi completa dell'istruzione `if` è la seguente:

```
if(espressione)
  istruzione1
[else
  istruzione2]
```

dove la valutazione di `espressione` controlla l'esecuzione di `istruzione1` e `istruzione2`: se `espressione` è vera viene eseguita `istruzione1`, se è falsa viene eseguita `istruzione2`.

Nell'esempio riportato precedentemente è stato omesso il ramo `else`: il fatto è del tutto legittimo in quanto esso è opzionale. Le parentesi quadre presenti nella forma sintattica completa hanno questo significato.

La tabella seguente mostra gli operatori utilizzabili nell'istruzione `if` :

Operatore	***Esempio***	***Risultato***
`!`	`!a`	(NOT logico) `1` se `a` è `0`, altrimenti `0`

Operatore	*Esempio*	*Risultato*
<	a < b	1 se a<b, altrimenti 0
<=	a <= b	1 se a<=b, altrimenti 0
>	a > b	1 se a>b, altrimenti 0
>=	a >= b	1 se a>=b, altrimenti 0
==	a == b	1 se a è uguale a b, altrimenti 0
!=	a != b	1 se a non è uguale a b, altrimenti 0
&&	a && b	(AND logico) 1 se a e b sono veri, altrimenti 0
||	a || b	(OR logico) 1 se a è vero, (b non è valutato), 1 se b è vero, altrimenti 0

È opportuno notare che, nel linguaggio C++, il confronto dell'eguaglianza fra i valori di due variabili viene effettuato utilizzando il doppio segno ==.

Esempio A

```
if (a==b)
  cout << "Sono uguali";
```

Esempio B

```
If (a=b)
  cout << "Valore non zero";
```

Nell'esempio **A** si **confronta** il contenuto della variabile a ed il contenuto della variabile b: se sono uguali viene stampata la frase specificata.

Nell'esempio **B** si **assegna** ad a il valore attualmente contenuto in b e si verifica se è diverso da zero: in tal caso viene stampata la frase specificata.

Una ulteriore osservazione va fatta a proposito degli operatori logici && (AND logico) e || (OR logico) che vengono usati per mettere assieme più condizioni. Es.

```
if (a>5 && a<10)
  cout << "a compreso fra 5 e 10";
```

```
if (a<2 || a>10)
  cout << "a può essere <2 oppure >10";
```

2.8 Istruzioni composte

L'istruzione composta, detta anche *blocco*, è costituita da un insieme di istruzioni inserite tra parentesi graffe che il compilatore tratta come se fosse un'istruzione unica.

Un'istruzione composta può essere scritta nel programma **dovunque** possa comparire un'istruzione semplice.

Esempio A

```
if (a>100)
  cout << "Prima frase \n";
cout << "Seconda frase \n";
```

Esempio B

```
if (a>100) {
  cout << "Prima frase \n";
  cout << "Seconda frase \n";
};
```

Nell'esempio **A** verrà visualizzata "Prima frase" solo se a è maggiore di 100, "Seconda frase" verrà visualizzato in ogni caso: *la sua visualizzazione prescinde infatti dalla condizione*.

Nell'esempio **B**, qualora a non risulti maggiore di 100, non sarà visualizzata alcuna frase: le due cout infatti sono raggruppate in un *blocco la cui esecuzione è vincolata dal verificarsi della condizione*.

Un blocco può comprendere anche una sola istruzione. Ciò può essere utile per aumentare la chiarezza dei programmi. L'istruzione compresa in una if può essere opportuno racchiuderla in un blocco anche se è una sola: in tal modo risulterà più evidente la dipendenza dell'esecuzione della

istruzione dalla condizione.

2.9 l'operatore ?

L'operatore `?` ha la seguente sintassi:

```
espr1 ? espr2 : espr3
```

Se `espr1` è **vera** restituisce `espr2` **altrimenti** restituisce `espr3`.

Si può utilizzare tale operatore per assegnare, condizionatamente, un valore ad una variabile. In questo modo può rendere un frammento di programma meno dispersivo:

Esempio A

```
if (a>100)
  sconto=10;
else
  sconto=5;
```

Esempio B

```
sconto=(a>100 ? 10 : 5);
```

In tutte e due gli esempi proposti viene assegnato alla variabile `sconto` un valore in dipendenza della condizione specificata, solo che, nell'esempio B, è più chiaramente visibile che si tratta di una assegnazione. Cosa non immediatamente percepibile, se non dopo aver letto le istruzioni, nel costrutto dell'esempio A.

2.10 Autoincremento ed operatori doppi

Il linguaggio C++ dispone di un operatore speciale per incrementare una variabile di una unità. Scrivere:

```
n++;
```

equivale a scrivere:

```
n = n+1;
```

Cioè ad incrementare di una unità il valore della variabile `n`. L'operatore `++` è l'operatore di **autoincremento**. L'operatore reciproco `--` (due simboli meno) *decrementa* di una unità il valore di una variabile:

```
m--; // riduce il valore della variabile di 1
```

L'operatore `--` è l'operatore di **autodecremento**. Gli operatori di autoincremento e autodecremento sono utilizzati nei contatori.

Anche per gli accumulatori sono previsti nel linguaggio C++ degli operatori particolari.

```
x += 37;          k1 += k2;          a += (b/2);
```

l'utilizzo del doppio operatore += rende le espressioni equivalenti rispettivamente a:

```
x = x+37;         k1 = k1+k2;        a = a+(b/2);
```

Le due espressioni sono equivalenti, dal punto di vista del risultato finale, solo che l'utilizzo del doppio operatore aumenta la leggibilità dell'assegnazione: diventa molto più chiaro che si tratta di un aggiornamento e non dell'immissione di un nuovo valore, come ci si potrebbe attendere dall'utilizzo del simbolo di assegnazione =.

Nel doppio operatore si possono usare *tutti gli operatori aritmetici*.

Nel seguito di questi appunti si converrà di utilizzare:

- gli operatori di autoincremento e di autodecremento tutte le volte che una variabile dovrà essere aggiornata con l'unità
- il doppio operatore (es. `+=`, `-=` ecc…) tutte le volte che si parlerà di aggiornamento generico di una variabile (per es. negli accumulatori)
- l'operatore di assegnamento generico (cioè `=`) in tutti gli altri casi.

2.11 Pre e post-incremento

Per incrementare di `1` la variabile `z` si può scrivere in due modi:

```
z++;                                    ++z;
```

cioè mettere l'operatore `++` prima o dopo del nome della variabile.

In questo caso, le due forme sono equivalenti. La differenza importa solo quando si scrive una *espressione* che contiene `z++` o `++z`.

Scrivendo `z++`, il valore di `z` viene *prima usato poi incrementato*:

```
int x,z;   // due variabili intere
z = 4;     // z vale 4
x = z++;   // anche x vale 4 ma z vale 5
```

Scrivendo `++z`, il valore di `z` viene *prima incrementato e poi usato*:

```
int x,z;  // due variabili intere
z = 4;    // z vale 4
x = ++z;  // ora x vale 5 come z
```

In definitiva basta tenere presente che l'ordine delle operazioni, nell'espressione dopo il simbolo di assegnazione, avviene da sinistra verso destra.

2.12 Cicli e costrutto while

Le strutture cicliche assumono nella scrittura dei programmi un ruolo fondamentale, non fosse altro per il fatto che, utilizzando tali strutture, si può istruire l'elaboratore affinché esegua azioni ripetitive su insiemi di dati diversi: il che è, tutto sommato, il ruolo fondamentale dei sistemi di elaborazione.

È in ragione delle suddette considerazioni che i linguaggi di programmazione mettono a disposizione del programmatore vari tipi di cicli in modo da adattarsi più facilmente alle varie esigenze di scrittura dei programmi. Il costrutto più generale è il ciclo `while` (ciclo iterativo con *controllo in testa*):

```
while(esp)
  istruzione
```

Viene verificato che `esp` sia vera, nel qual caso viene eseguita `istruzione`. Il ciclo si ripete mentre `esp` risulta essere vera.

Naturalmente, per quanto osservato prima, `istruzione` può essere un blocco e, anche in questo caso, può essere utile racchiudere l'istruzione in un blocco anche se è una sola.

Come esempio delle istruzioni trattate fino a questo punto, viene proposto un programma che, data una sequenza di numeri interi positivi, fornisce la quantità di numeri pari della sequenza e la loro somma. Un qualsiasi numero negativo, o il valore nullo, ferma l'elaborazione.

```
#include <iostream>
using namespace std;

main(){
  // dichiarazioni variabili

  int vn,conta,somma;

  // inizializzazione accumulatori

  cout << "Conteggio e somma dei numeri pari\n\n";
  conta = somma = 0;                                          /*1*/

  // esame dei numeri

  cout << "Inserire numero positivo ";
  cin >> vn;                                                  /*2*/
  while (vn>0){

    // verifica se, numero inserito, pari

    if(!(vn%2)){                                              /*3*/
      conta++;                                                /*4*/
      somma += vn;                                            /*4*/
    }

    // prossimo numero da elaborare

    cout << "Inserire numero positivo ";
    cin >> vn;
  }

  // risultati elaborazione

  cout << "Nella sequenza c\'erano "
       << conta << " numeri pari" << endl;
  cout << "La loro somma e\' " << somma << endl;
}
```

Nella `1` si inizializzano al valore `0` il contatore e l'accumulatore dei pari.

Nella `2`, in accordo a quanto scritto in (1.3), si acquisisce il primo valore numerico da elaborare.

Il controllo della `3` permette di stabilire se il numero introdotto è pari. Viene usato l'operatore modulo `%` che fornisce il resto della divisione intera fra `vn` e `2` e, subito dopo, viene controllato se tale resto è nullo in modo che, nelle `4`, si possano aggiornare il contatore e l'accumulatore.

2.13 Cicli e costrutto for

L'istruzione `for` viene utilizzata tradizionalmente per codificare cicli a contatore: istruzioni cicliche

cioè che devono essere ripetute un numero definito di volte. Il formato del costrutto `for` è il seguente:

```
for(esp1; esp2; esp3)
  istruzione
```

Il ciclo inizia con l'esecuzione di `esp1` (*inizializzazione del ciclo*) la quale non verrà più eseguita. Quindi viene esaminata `esp2` (*condizione di controllo del ciclo*). Se `esp2` risulta vera, viene eseguita `istruzione`, altrimenti il ciclo non viene percorso neppure una volta. Conclusa l'esecuzione di `istruzione` viene eseguita `esp3` (*aggiornamento*) e di nuovo valutata `esp2` che se risulta essere vera dà luogo ad una nuova esecuzione di `istruzione`. Il processo si ripete finché `esp2` risulta essere falsa.

Conoscendo la quantità di valori numerici da elaborare, il programma precedente potrebbe essere codificato:

```
#include <iostream>
using namespace std;

main(){
  // dichiarazioni variabili

  int vn,conta,somma;
  int qn,i;                                                    /*1*/

  // inizializzazione accumulatori

  cout << "Conteggio e somma dei numeri pari\n\n";
  cout << "Quanti numeri devono essere elaborati? ";
  cin >> qn;                                                   /*2*/

  conta = somma = 0;

  // esame dei numeri

  for(i=1;i<=qn;i++){                                          /*3*/

    cout << "Inserire numero positivo ";
    cin >> vn;

    // verifica se, numero inserito, pari

    if(!(vn%2)){
      conta++;
      somma += vn;
    }
  }

  // risultati elaborazione

  cout << "Nella sequenza c\'erano "
       << conta << " numeri pari" << endl;
  cout << "La loro somma e\' " << somma << endl;
}
```

Vengono aggiunte nella `1` due nuove variabili per la quantità dei numeri da elaborare e il contatore del ciclo.

Viene effettuato nella 2 l'input della quantità dei numeri da elaborare che, con il ciclo che inizia da 3, vengono elaborati secondo uno schema messo in evidenza in (1.5).

Il programma per prima cosa assegna il valore 1 alla variabile i (la prima espressione del for), si controlla se il valore di i è non superiore a qn (la seconda espressione) e poiché l'espressione risulta vera verranno eseguite le istruzioni inserite nel ciclo. Terminate le istruzioni che compongono il ciclo si esegue l'aggiornamento di i così come risulta dalla terza espressione contenuta nel for, si ripete il controllo contenuto nella seconda espressione e si continua come prima finché il valore di i non rende falsa la condizione.

Questo modo di agire del ciclo for è quello comune a tutti i cicli di questo tipo messi a disposizione dai compilatori di diversi linguaggi di programmazione. Il linguaggio C++, considerando le tre parti del costrutto come espressioni generiche, espande abbondantemente le potenzialità del for generalizzandolo in maniera tale da comprendere, per esempio, come caso particolare il ciclo while. La prima versione del programma, nella parte del ciclo di elaborazione, potrebbe essere codificata:

```
...
for(conta=somma=0;vn>0;){

    // verifica se, numero inserito, pari

    if(!(vn%2)){
      conta++;
      somma += vn;
    }

    // prossimo numero da elaborare

    cout << "Inserire numero positivo ";
    cin >> vn;
  }
...
```

Il ciclo esegue l'azzeramento di conta e somma (che verrà eseguito una sola volta) e subito dopo il controllo se il valore di vn è positivo e, in questo caso, verranno eseguite le istruzioni del ciclo. Terminate le istruzioni, poiché manca la terza espressione del for, viene ripetuto il controllo su vn.

Le inizializzazioni di conta e somma avrebbero potuto essere svolte fuori dal for: in tal caso sarebbe mancata anche la prima espressione.

2.14 Cicli e costrutto do-while

L'uso della istruzione while prevede il test sulla condizione all'inizio del ciclo stesso. Ciò vuol dire che se, per esempio, la condizione dovesse risultare falsa, le istruzioni facenti parte del ciclo verrebbero saltate e non verrebbero eseguite nemmeno una volta.

Quando l'istruzione compresa nel ciclo deve essere comunque eseguita almeno una volta, è più comodo utilizzare il costrutto:

```
do
  istruzione
while(espr);
```

In questo caso viene eseguita istruzione e successivamente controllato se espr risulta vera, nel

qual caso il ciclo viene ripetuto.

Come sempre l'iterazione può comprendere una istruzione composta.

È bene precisare che in un blocco `for`, `while` o `do...while`, così come nel blocco `if`, può essere presente un numero qualsiasi di istruzioni di ogni tipo ivi compresi altri blocchi `for`, `while` o `do...while`. I cicli possono cioè essere *annidati*.

3 Vettori, stringhe, costrutti avanzati

3.1 Tipi di dati e modificatori di tipo

Le elaborazioni di un computer coinvolgono dati che possono essere di vari tipi. La stessa elaborazione può fornire risultati diversi a seconda del tipo di dato: se si considerano , per esempio, i dati 2 e 12 e si vuole stabilire un ordine, la risposta sarà diversa a seconda se si intendono i due dati come numeri o come caratteri. Nel primo caso, facendo riferimento al valore, la risposta sarà 2, 12. Se, invece, l'ordinamento va inteso in senso alfabetico, la risposta sarà 12, 2 (allo stesso modo di AB, B). È necessario stabilire in che senso vanno intesi i valori.

In generale, in rapporto al tipo di elaborazione, in Informatica si fa distinzione fra tipo carattere o alfanumerico e tipi numerici. Se sui dati si vogliono effettuare operazioni aritmetiche, questi devono essere definiti come tipo numerico, in caso contrario vanno definiti come tipo carattere.

In realtà il discorso è più complesso: se per il tipo carattere c'è ben poco da dire, il tipo numerico pone problemi non indifferenti per la conservazione nella memoria di un computer. Basta dire che i numeri sono infiniti e che lo può essere pure la parte decimale di un numero, che la memoria di un computer è limitata, che la codifica interna è in binario e quindi rappresentazioni molto estese per numeri anche piccoli come valore, per affermare che possono esserci più alternative in ragione dello spazio occupato in memoria e della precisione con cui si vuole conservare il numero.

Si aggiunga, a quanto sopra, che nelle applicazioni reali i dati, quasi mai, sono in formato semplice ma si presentano spesso in strutture più complesse.

I linguaggi di programmazione mettono a disposizione un insieme di *tipi elementari* e dei metodi per costruire strutture complesse: i *tipi utente* che verranno trattati successivamente.

C++ mette a disposizione 6 tipi elementari identificati dalle parole chiavi: `char`, `int`, `float`, `double`, `bool`, `void`. A parte il tipo `void` che verrà affrontato più avanti:

- le variabili di tipo `char` possono conservare, ciascuna, un carattere. Il carattere può essere conservato nella variabile utilizzando, al solito modo, lo stream di input `cin`, o, in una istruzione di assegnazione, racchiudendolo fra singoli apici

```
...
char alfa, beta;
cin >> alfa;  // riceve da tastiera un carattere e lo mette nella variabile
beta = 'Z';   // assegna direttamente il carattere
...
```

- le variabili dei tipi `int`, `float`, `double` possono conservare numeri. Questi tipi permettono anche l'uso di ***modificatori*** che variano l'occupazione di memoria della variabile e, quindi, le caratteristiche del numero conservato. I modificatori ammessi sono `short`, `long`, `unsigned`.

```
#include <iostream>
#include <limits>
using namespace std;

main(){

  cout << "bit intero " << numeric_limits<int>::digits << endl;
  cout << "cifre intero " << numeric_limits<int>::digits10 << endl;
  cout << "minimo intero " << numeric_limits<int>::min() << endl;
```

```
    cout << "massimo intero " << numeric_limits<int>::max() << endl;

    cout << "\nbit short " << numeric_limits<short>::digits << endl;
    cout << "cifre short " << numeric_limits<short>::digits10 << endl;
    cout << "minimo short " << numeric_limits<short>::min() << endl;
    cout << "massimo short " << numeric_limits<short>::max() << endl;

    cout << "\nbit float " << numeric_limits<float>::digits << endl;
    cout << "cifre float " << numeric_limits<float>::digits10 << endl;
    cout << "minimo float " << numeric_limits<float>::min() << endl;
    cout << "massimo float " << numeric_limits<float>::max() << endl;

    cout << "\nbit double " << numeric_limits<double>::digits << endl;
    cout << "cifre double " << numeric_limits<double>::digits10 << endl;
    cout << "minimo double " << numeric_limits<double>::min() << endl;
    cout << "massimo double " << numeric_limits<double>::max() << endl;

    cout << "\nbit long double " << numeric_limits<long double>::digits << endl;
    cout << "cifre long double " << numeric_limits<long double>::digits10 << endl;
    cout << "minimo long double " << numeric_limits<long double>::min() << endl;
    cout << "massimo long double " << numeric_limits<long double>::max() << endl;

}
```

il programma proposto serve per far visualizzare alcune caratteristiche importanti delle variabili dichiarate in quel modo. In particolare viene chiesta la visualizzazione, per ogni tipo, di quantità bit occupati (`digits`), quantità cifre di precisione (`digits10`), valore minimo (`min()`) e valore massimo (`max()`) conservabili. Non è importante spiegare il senso delle singole righe, è importante solo sapere cosa produce il programma perché è fondamentale, nella scrittura di un programma, conoscere il grado di affidabilità di un valore numerico conservato e come dichiarare la variabile in modo da consentirne la corretta conservazione.

Nella macchina utilizzata per le prove dei programmi riportati in questi appunti, e con il compilatore utilizzato, si ottengono questi risultati:

```
bit intero 31
cifre intero 9
minimo intero -2147483648
massimo intero 2147483647

bit short 15
cifre short 4
minimo short -32768
massimo short 32767

bit float 24
cifre float 6
minimo float 1.17549e-38
massimo float 3.40282e+38

bit double 53
cifre double 15
minimo double 2.22507e-308
massimo double 1.79769e+308

bit long double 64
cifre long double 18
```

```
minimo long double 3.3621e-4932
massimo long double 1.18973e+4932
```

Ogni variabile di tipo `int` occupa 31 bit in memoria, può contenere solo numeri interi e i valori ammissibili variano tra i margini riportati (***margine di rappresentatività***). Se si applica il modificatore `short`, l'occupazione di memoria cambia e di conseguenza anche il margine di rappresentatività.

```
...
int a;
short b;   // non e' necessario specificare int
...
a = 15;
b = 20;
...
```

Se si vuole conservare un numero con parte intera e parte decimale è necessario dichiarare la variabile di tipo virgola mobile (*floating point*) `float` o `double`. In questi casi il problema, da considerare, non è il margine di rappresentatività, ambedue consentono la conservazione di numeri con valori estremamente elevati (si vedano il valore minimo e massimo scritti in notazione esponenziale), ma la ***precisione***. Il tipo `float` utilizza 24 bit e garantisce sei cifre di precisione (*singola precisione*). In pratica il numero può essere molto grande come valore (10^{38}) ma non può contenere più di 6 cifre diverse da 0: va bene 12340000000000000 ma non 123456789. Il secondo valore non è conservato in modo corretto.

Il tipo `double`, al costo di maggiore occupazione di memoria, garantisce un numero maggiore di cifre precise (*doppia precisione*). Se si ha necessità di una precisione maggiore si può dichiarare la variabile di tipo `long double` e si arriva a 18 (*quadrupla precisione*).

```
...
float c;
double d;
long double e;
...
c = 123.23;
d = 456970.345;
e = 789.0       // in ogni caso deve essere inserita la parte decimale
...
```

Il modificatore `unsigned` potrebbe, per esempio, esserci necessità di utilizzarlo per aumentare il margine di rappresentatività di un `int`: invece di utilizzare 31 bit per il numero e 1 bit per il segno, si utilizzerebbero tutti e 32 bit per conservare il valore numerico. La variabile potrebbe contenere numeri, fino al doppio del valore precedente, ma solo positivi.

- le variabili di tipo `bool` sono utilizzate quando serve conservare indicatori del verificarsi/non verificarsi di eventi e permettono la conservazione dei valori simbolici `true` e `false`.

```
...
bool test;
test = false;
if (i<100)
  test = true;
...
```

3.2 Il costrutto cast

A volte può interessare effettuare dei cambiamenti al volo per conservare i risultati di determinate operazioni in variabili di tipo diverso principalmente quando si va da un tipo *meno capiente* ad un tipo *più capiente*. In tali casi il linguaggio C mette a disposizione del programmatore un costrutto chiamato ***cast***.

```
main(){
  int uno, due;
  float tre;

  uno = 1;
  due = 2;
  tre = uno/due;
  cout << tre;
}
```

Questo programma nonostante le aspettative (dettate dal fatto che la variabile `tre` è dichiarata `float`) produrrà un risultato nullo. Infatti la divisione viene effettuata su due valori di tipo `int`, il risultato viene conservato temporaneamente in una variabile di tipo `int` e, solo alla fine, conservato in una variabile `float`. È evidente, a questo punto, che la variabile `tre` conterrà solo la parte intera (cioè `0`).

Affinché la divisione produca il risultato atteso, è necessario avvisare C++ di convertire il risultato intermedio prima della conservazione definitiva nella variabile di destinazione.

Tutto ciò è possibile utilizzando il costrutto ***cast*** che ha la seguente sintassi:

```
(nome-di-tipo) espressione
```

Utilizzando questo costrutto, il programma, si scriverebbe così:

```
main(){
  int uno, due;
  float tre;

  uno = 1;
  due = 2;
  tre = (float) uno/due;
  cout << tre;
}
```

In questo caso il programma fornisce il risultato atteso. Infatti il quoziente viene calcolato come `float` e quindi, dopo, assegnato alla variabile `tre`.

In definitiva il costrutto *cast* forza una espressione a essere di un tipo specifico (nell'esempio una divisione intera viene forzata a fornire un risultato di tipo virgola mobile).

Viene adesso proposta una interessante applicazione del casting per la trasformazione di un carattere minuscolo nella sua rappresentazione maiuscola. Il programma sfrutta, per ottenere il suo risultato, il fatto che una variabile di tipo `char`, in definitiva, conserva un codice numerico; qui viene inoltre usato il casting per mostrare tale codice.

```
#include <iostream>
using namespace std;

main(){
```

```
  char min,mai;
  const int scarto=32;                                          /*1*/

  cout << "Conversione minuscolo-maiuscolo" << endl;
  cout << "Introduci un carattere minuscolo ";
  cin  >> min;                                                  /*2*/

  if (min>=97 && min<=122) {                                    /*3*/
    mai = min-scarto;                                           /*4*/
    cout << "\n Rappresentazione maiuscola " << mai << endl;    /*5*/
    cout << "Codice ASCII " << (int)mai << endl;                /*6*/
  }else
    cout << "\n Carattere non convertibile" << endl;
}
```

Nella riga con etichetta `1` è definita la costante `scarto`. Il valore `32` dipende dal fatto che, nel codice ASCII, tale è la distanza fra le maiuscole e le minuscole (es. '`A`' ha codice `65`, '`a`' ha codice `97`).

Nella riga con etichetta `2` si effettua l'input del carattere da elaborare.

Nella riga con etichetta `3` si controlla, utilizzando la sua rappresentazione numerica, se il carattere immesso rientra nei limiti della codifica ASCII delle lettere minuscole. Le lettere minuscole, in ASCII, hanno codice compreso fra `97` (la lettera minuscola `a`) e `122` (la lettera minuscola `z`). Il confronto poteva anche essere fatto sulla rappresentazione alfanumerica:

```
if (min>='a' && min<='z')
```

Nella riga con etichetta `4` si effettua in pratica la trasformazione in maiuscolo. Al codice numerico associato al carattere viene sottratto il valore `32`. In questo caso è utilizzato il codice numerico del carattere. Si noti che in questo contesto ha senso assegnare ad un `char` il risultato di una sottrazione (operazione numerica).

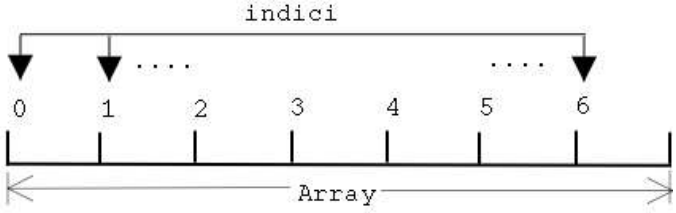
Nella riga con etichetta `5` si effettua l'output di `mai` inteso come carattere (così è infatti definita la variabile), laddove nella riga `6` si effettua l'output del suo codice numerico (è usato un casting sulla variabile).

3.3 Array, vettori e nuovi tipi

L'**array** è la prima struttura di dati trattata in Informatica. È quella più comunemente usata ed è quella che sta a fondamento di quasi tutte le altre.

Un array è un insieme di variabili dello stesso tipo cui è possibile accedere tramite un nome comune e referenziare uno specifico elemento tramite un indice.

Si può pensare all'array come ad un insieme di cassetti numerati: per poter accedere al contenuto di un cassetto deve essere specificato il numero ad esso associato (l'**indice**) che indica la posizione relativa dell'elemento, rispetto al punto iniziale della struttura.



Nelle variabili semplici per accedere al valore contenuto in esse è necessario specificare il nome ed, inoltre, una variabile che dovrà conservare un valore diverso avrà un nome diverso. Nell'array esiste un nome che però stavolta identifica l'array come struttura (il nome farà riferimento all'insieme dei cassetti dell'esempio precedente: questo concetto sarà chiarito meglio più avanti). I singoli elementi dell'array verranno referenziati specificando la loro posizione relativa all'interno della struttura (l'indice): l'elemento a cui ci si riferisce dipende dal valore assunto dall'indice.

Gli elementi dell'array vengono allocati in posizioni di memoria adiacenti. In Informatica una allocazione in memoria centrale per byte consecutivi, viene chiamata **vettore**.

Il diverso tipo di dichiarazione nelle variabili di tipo elementare, come si è già fatto notare, comporta un diverso modo di svolgere le operazioni fra i dati in esse contenute. Nel caso di un vettore, essendo questa una struttura complessa costituita da un gruppo di variabili, si possono distinguere due livelli:

- la struttura che ha proprie caratteristiche: una certa dimensione, un certo ordine per cui se si inserisce o si elimina un elemento, in ogni caso, deve essere garantita l'allocazione per byte contigui.
- Le singole variabili che , in rapporto al tipo, consentono alcune operazioni ed altre no.

C++, nel caso di strutture (insiemi di dati) complesse, mette a disposizione un meccanismo (*classi*) che consente, una volta dichiarato un oggetto di quel tipo, all'oggetto stesso, di possedere dei comportamenti tipici di tutti gli oggetti della stessa famiglia.

La definizione di vettore è contenuta nella libreria `vector` che quindi va inclusa nel programma tutte le volte che si ha necessità di definire una variabile di tipo vettore. La libreria fa parte di quelle che vengono definite STL (Standard Template Library).

Per l'esposizione delle principali caratteristiche e delle applicazioni concrete di questa libreria, viene proposto un programma che, acquisito un array, per esempio di numeri interi, ordinato e un numero intero, inserisca il numero intero, all'interno del vettore, nel posto che gli compete. Il programma permette anche, alla fine, l'eliminazione di un elemento dalla struttura.

```
#include <iostream>
#include <vector>                                                    /*1*/
using namespace std;

main(){
  vector<int> numeri;                                                /*2*/
  int i,temp;
  int cosa,pos;

  cout << "Inserisce un numero in un vettore ordinato ed" << endl;
  cout << "Elimina un elemento, conoscendone la posizione \n\n";

  // Riempimento di un vettore contenete 10 elementi

  cout << "Inserimento elementi ordinati" << endl;
  for(i=0; i<10; i++){                                               /*3*/
    cout << "elemento " << i << " -->";
    cin  >> temp;                                                    /*4*/

    numeri.push_back(temp);                                          /*5*/
  };
```

```
  cout << "Elemento da inserire ";
  cin >> cosa;                                                    /*6*/

  // Ricerca posizione

  pos=-1;                                                         /*7*/
  for(i=0; pos<0 && i<numeri.size(); i++){                        /*8*/
    if(cosa<numeri[i])                                            /*9*/
      pos = i;
  }

  // Inserimento nella posizione trovata

  if(pos>=0)
    numeri.insert(numeri.begin()+pos,cosa);                       /*10*/
  else
    numeri.push_back(cosa);                                       /*11*/

  // Vettore con nuovo elemento inserito

  cout << "\nNuovo vettore " << endl;
  for(i=0; i<numeri.size(); i++)                                  /*12*/
    cout << numeri[i] << "\t";
  cout << endl;

  // Eliminazione di un elemento dal vettore

  cout << "Da quale posizione togliere? ";
  cin >> pos;
  numeri.erase(numeri.begin()+pos);                               /*13*/

  // Nuova visualizzazione

  cout << "\nNuovo vettore " << endl;
  for(i=0; i<numeri.size(); i++)
    cout << numeri[i] << "\t";
  cout << endl;
}
```

Nella `1` si dichiara l'inclusione della libreria per l'uso dei vettori.

Nella `2` si dichiara una variabile (`numeri`) di tipo vettore. Ogni elemento presente nel vettore sarà di tipo `int`. Il tipo di elemento è specificato fra parentesi angolari (`<` e `>`) subito dopo `vector`. Naturalmente se c'era necessità di un vettore di `double` con nome `fatturati`, la dichiarazione sarebbe stata del tipo:

```
vector<double> fatturati;
```

`numeri`, per quanto osservato in precedenza, è il nome di tutta la struttura.

Dalla `3` comincia un ciclo per l'input, e la successiva conservazione nel vettore, di 10 variabili di tipo `int`. Prima (`4`) viene effettuato l'input in una variabile temporanea e, subito dopo (`5`), viene ordinato al vettore di inserire il valore ricevuto, all'interno della struttura. Tutto ciò viene fatto passando il valore `temp` (fra parentesi) alla *funzione membro* `push_back`, applicata (l'operatore `.`) al vettore `numeri`. La `push_back` si occupa di creare, nel primo posto disponibile in coda, un elemento nel vettore e di conservarci il valore ricevuto. Questa è una capacità che hanno tutti gli

oggetti che sono definiti come `vector` e, quindi anche `numeri`. È una *competenza* che hanno tutti gli oggetti di tipo `vector`.

Nella `6` viene acquisito il numero da inserire nel vettore. Il primo problema da risolvere, a questo punto, è quello di trovare la posizione corretta in cui inserire il numero e, di ciò, si occupa il ciclo successivo. In dettaglio il ciclo si occuperà di trovare la posizione del primo numero, presente nel vettore, maggiore, in valore, del valore contenuto in `cosa`. Quella è la posizione che deve occupare il numero acquisito come input. Trovata la posizione si tratterà semplicemente di inserire l'elemento.

Nella `7` la posizione viene inizializzata a `-1` che, quindi, sarà l'indicatore di posizione non trovata. Il controllo presente in `8` permette di uscire fuori dal ciclo nel caso in cui si trova un elemento del vettore maggiore del numero dato (controllo `pos<0`) o nel caso in cui termina il vettore (`i<numeri.size()`). Può infatti accadere che nessun elemento del vettore sia maggiore del dato di input e, in questo caso specifico, il numero dovrà essere inserito come ultimo elemento. Si permane dentro il ciclo se sono verificate contemporaneamente (operatore `&&`) le condizioni `pos` ancora con valore negativo e non si è ancora concluso l'esame degli elementi del vettore.

La funzione membro `size()` applicata al vettore `numeri` fornisce la quantità degli elementi presenti nel vettore. Nel caso specifico si conosce il numero degli elementi e, infatti, si sarebbe potuto specificare 10, ma, in generale, si può ricorrere a detta funzione per conoscere la quantità degli elementi presenti in un vettore.

La `9` coinvolge un confronto con un elemento del vettore (quello di posizione generica `i`) e l'accesso ad esso avviene specificandone l'indice.

In `10`, se si è trovato un elemento del vettore maggiore del numero da inserire (informazione fornita dalla posizione non negativa), si inserisce il numero fornendolo, assieme alla posizione di inserimento, alla funzione `insert` applicata al vettore. La posizione è ottenuta sommando il valore relativo che si è trovato (`pos`) a `numeri.begin()`, che indica la posizione iniziale di memorizzazione della struttura. Se non si è trovata una posizione, il numero è maggiore di tutti gli elementi del vettore e si invia al vettore senza specificare una posizione (`11`) in modo da inserirlo in coda a quelli esistenti.

Il ciclo che comincia da `12` si occupa di effettuare l'output del vettore in modo da controllare l'avvenuto inserimento.

Per ordinare ad un vettore di eliminare un elemento, si richiama la funzione `erase` del vettore, specificando, come in `13`, la posizione dell'elemento da eliminare. Anche qui la posizione come in `10` è espressa come somma della posizione relativa (*scostamento*) rispetto all'inizio del vettore.

Il ciclo per l'inserimento dei numeri della sequenza ordinata, per come è stato implementato, non fa alcun controllo sul fatto che la sequenza sia effettivamente crescente. Può essere interessante modificare il ciclo in modo che faccia in modo che ogni input abbia un valore maggiore del precedente, prima di inviarli al vettore per la conservazione, allo scopo di avere una sequenza crescente.

```
...
for(i=0; i<10; i++){
  cout << "elemento " << i << " -->";
  cin  >> temp;

  // verifica ordinamento
```

```
  if(!i || temp>numeri[i-1])                                    /*1*/
    numeri.push_back(temp);                                     /*2*/
  else{
    cout << "La sequenza deve essere ordinata" << endl
         << "Ripetere l\'inserimento" << endl;
    i--;                                                        /*3*/
  };
};
...
```

È il controllo in `1` che, sostanzialmente, permette di stabilire se la sequenza è crescente. Se si tratta del primo elemento (`!i` cioè se `i` è `0`) o l'input è maggiore dell'elemento inserito precedentemente (quello con indice `i-1`), si inserisce in coda (`2`). L'ultima condizione non avrebbe senso per il primo elemento, ma si ricorda, come fatto rilevare in precedenza, che per l'operatore OR (`||`), basta che la prima condizione sia verificata per non procedere oltre.

Se il valore inserito non è maggiore del precedente, basta decrementare l'indice del ciclo (`3`). Questa operazione è necessaria poiché, in ogni caso, per effetto del costrutto `for`, l'indice verrebbe incrementato prima di passare al controllo del ciclo.

3.4 Elaborazione di stringhe: primo esempio

Le variabili di tipo `char` consentono di conservare un singolo carattere. Se si voglio conservare sequenze di caratteri come, per esempio, una parola o una intera frase, si può utilizzare la libreria `string` che contiene la definizione della classe `string` che consente la dichiarazione di oggetti di quel tipo. Anche in questo caso si tratta di una struttura simile a quella trattata nei vettori, con propri comportamenti come ricerca al suo interno, estrazione di sottostringa, cancellazione di caratteri al suo interno, possibilità di trattarla come un insieme unico e possibilità di accesso ai singoli caratteri che ne fanno parte. I singoli caratteri sono accessibili utilizzando l'indice. Dal punto di vista dei caratteri che la compongono, la stringa, si comporta come vettore di caratteri.

Il seguente programma acquisisce una stringa e un carattere e restituisce le ricorrenze del carattere all'interno della stringa:

```
#include <iostream>
#include <string>                                               /*1*/
using namespace std;

main(){
  string dove;                                                  /*2*/
  char cosa;
  int i,volte,pos;
  bool continua;

  // input stringa in cui cercare e carattere da cercare

  cout << "Cerca un carattere in una stringa\n"
    "e ne mostra le ricorrenze";
  cout << "\n\nStringa di ricerca\n\n";
  cin >> dove;                                                  /*3*/
  cout << "\nCosa cercare? ";
  cin >> cosa;

  // inizializzazione contatore ricorrenze,
```

```
  // posizione carattere trovato e controllo ciclo elaborazione

  volte=0;
  pos=-1;
  continua = true;                                                    /*4*/

  // continua elaborazione finche' trovata una ricorrenza del carattere

  while(continua){
    pos = dove.find(cosa,pos+1);                                      /*5*/

    if(pos==-1)                                                       /*6*/
      continua = false;                                               /*7*/
    else
      volte++;
  }

  cout << "\n\nil carattere " << cosa << " si presenta "
       << volte << " volte" << endl;
}
```

Per poter dichiarare oggetti di tipo `string`, come in `2`, è necessario includere la relativa libreria (`1`).

L'input di una stringa (`3`) viene effettuato come per le variabili di tipo elementare. Questo però va bene se nella stringa fornita in input non si inserisce il carattere *Spazio*. In questo caso infatti, come notato in precedenza, le due parti della stringa ai lati dello *Spazio* verrebbero percepiti come due input diversi. Anche l'assegnazione ad una stringa viene effettuata per mezzo del solito operatore `=`, è solo necessario racchiudere la stringa fra doppi apici (`dove = “stringa di prova”;`). Anche nelle operazioni di confronto fra stringhe si utilizzano i soliti operatori (`<`, `<=`, `>`, `>=`, `==`, `!=`).

La variabile booleana `continua`, inizializzata in `4`, controlla il ciclo successivo.

Il messaggio `find`, inviato in `5` alla stringa `dove`, comanda di cercare il primo parametro, immesso fra parentesi, a partire dalla posizione specificata come secondo parametro. Il risultato di questa ricerca viene fornito come valore intero che è conservato in `pos`. La posizione iniziale di ricerca è `0` (il primo carattere della stringa: `pos` è inizializzato a `-1` e, come parametro, viene fornito `pos+1`). Le ricerche successive cominceranno dalla posizione successiva a quella trovata in precedenza.

Se non si trovano ulteriori ricorrenze, la funzione membro `find` associata a `dove`, fornisce il valore `-1` (`6`) e l'elaborazione può terminare: la `7` fa in modo di rendere falsa la condizione di controllo del ciclo.

3.5 Stringhe: elaborazioni comuni

Il primo programma proposto acquisita da input una stringa contenete cognome e nome separati da uno spazio, estrae il cognome, il nome e genera una nuova stringa con i dati in ordine inverso (prima il nome e poi il cognome).

```
#include <iostream>
#include <string>
using namespace std;

main(){
  string nomecogn, nome, cognome, nmcg;
  int pos;
```

```
  cout << "Da una stringa con cognome e nome separati da spazio" << endl
       << "estrae cognome nome e inverte ordine\n\n";

  cout << "Cognome e nome separati da uno spazio ";
  getline(cin,nomecogn);                                          /*1*/

  // ricerca spazio ed estrazione

  pos = nomecogn.find(' ');                                       /*2*/
  if(pos>=0){                                                     /*3*/
    cognome = nomecogn.substr(0,pos);                             /*4*/
    nome = nomecogn.substr(pos+1);                                /*5*/
  }

  nmcg = nome+" --- "+cognome;                                    /*6*/
  cout << "Cognome: " << cognome << endl;
  cout << "Nome: " << nome << endl;
  cout << "Inverso: " << nmcg << endl;
}
```

Per acquisire la stringa, nella `1`, è usata la funzione `getline` che permette di scrivere una stringa contenente spazi e che viene considerata completata con *Invio*. Alla funzione vengono passati il canale da dove ricevere la linea (`cin`) e la stringa dove conservare la linea (`nomecogn`). La funzione estrae dal *buffer di tastiera* (la zona di memoria dove vengono inseriti i caratteri digitati) tutti i caratteri fino a *Invio*, li mette nella variabile specificata e scarta il carattere *Invio*. È questo comportamento che se, prima dell'input della stringa, fosse stato effettuato un input di una variabile di altro tipo, produce come effetto il salto dell'istruzione. Nel buffer è contenuto *Invio* e una `getline` preleva quello che trova fino all'*Invio* stesso, cioè niente. Quando, in una istruzione precedente l'uso di `getline`, c'è un input diverso (una variabile di uno dei tipi elementari) è opportuno inserire subito dopo l'input, e prima dell'acquisizione della stringa, la riga `cin.ignore();` che ha l'effetto di ignorare (scartare) il prossimo carattere presente nel buffer.

Nella `2` viene inviato, alla stringa `nomecogn`, il messaggio `find` passando, stavolta, solo il parametro di cosa cercare. In `pos`, al solito, ci sarà la posizione dello spazio all'interno della stringa.

Se esiste nella stringa uno spazio (controllo della `3`), la stringa viene divisa in due parti. Nella `4` viene messa in `cognome` la sottostringa che si forma estraendo da `nomecogn` una quantità `pos` di caratteri dalla posizione `0`. Quelli dalla posizione `pos+1` alla fine vanno, invece, in `nome` (`5`).

la `6` costruisce la nuova stringa `nmcg` mettendo assieme (operatore `+`) stringhe già esistenti e nuove (quella centrale).

Il prossimo programma proposto, acquisita una stringa e una parola, sostituisce la seconda parola della stringa con la parola acquisita da input.

```
#include <iostream>
#include <string>
using namespace std;

main(){
  string dove,metti;
  int inizio,fine,quanti;

  cout << "Data una stringa e una parola in input" << endl
       << "la inserisce al posto della seconda parola della stringa";
  cout << "\n\nInserire stringa da elaborare ";
```

```
  getline(cin,dove);
  cout << "Parola da inserire ";
  getline(cin,metti);

  // ricerca spazi che includono la seconda parola

  inizio = dove.find(' ');                                    /*1*/
  fine = dove.find(' ',inizio+1);                             /*2*/

  // eliminazione parola, inserimento nuova

  quanti = (fine-inizio)-1;                                   /*3*/
  dove = dove.erase(inizio+1,quanti);                         /*4*/
  dove = dove.insert(inizio+1,metti);                         /*5*/

  cout << "Nuova stringa" << endl << dove << endl;
}
```

Nella `1` si cerca la posizione del primo spazio e nella `2` quella del secondo in modo da isolare la parola da eliminare.

Nella `3` si calcola la quantità dei caratteri da eliminare. Si ricorda, per il conteggio, che le posizioni vengono contate a partire da 0.

Il messaggio `erase` lanciato a `dove`, nella `4`, ha l'effetto di eliminare dalla stringa una certa quantità di caratteri. È necessario specificare, come parametri, la posizione a partire da dove cancellare (quella successiva al primo spazio: `inizio+1`) e la quantità di caratteri da cancellare. Il risultato dell'operazione viene depositato nuovamente in `dove`.

Eliminati i caratteri, si può inserire nella stringa la nuova parola. A ciò provvede la funzione membro `insert` della `5`, applicata a `dove`. Come parametri si specificano la posizione a partire da dove inserire e la stringa da inserire. Anche in questo caso il risultato è una nuova stringa, ma il programma richiede che le variazioni siano fatte nella stringa di partenza e, quindi, come nell'istruzione precedente, il risultato viene assegnato a `dove`.

Il programma funziona anche quando la stringa di partenza è composta da una sola parola. In questo caso viene sostituita quella parola.

3.6 La scelta multipla: costrutto switch-case

Può essere necessario, nel corso di un programma, variare l'elaborazione in seguito a più condizioni. Potrebbero essere, per esempio, i diversi valori che può assumere una variabile. Esiste nel linguaggio C++ un costrutto per codificare la scelta multipla. Sintatticamente la struttura si presenta in questo modo:

```
switch(espressione)
  case valore:
    istruzione
  case valore:
    istruzione
    ...
  default:
    istruzione
```

Nelle varie istruzioni `case` si elencano i valori che può assumere `espressione` e che interessano

per l'elaborazione. Valutata l'espressione specificata, se il valore non coincide con alcuno di quelli specificati, viene eseguita l'istruzione compresa nella clausola `default`. Occorre tenere presente che, la differenza sostanziale rispetto ad una struttura `if`, consiste nel fatto che, nella `if`, le varie strade sono alternative. I vari `case` esistenti agiscono invece da etichette: se espressione assume un valore specificato in un `case`, vengono eseguite le istruzioni **a partire da quel punto in poi**. Il valore specificato in `case`, in definitiva, assume funzione di *punto di ingresso* nella struttura.

Se si vuole delimitare l'istruzione da eseguire a quella specificata nella `case` che verifica il valore cercato, occorre inserire l'istruzione `break` che fa proseguire l'elaborazione dall'istruzione successiva alla chiusura della struttura.

Per chiarire meglio il funzionamento della struttura viene presentato un programma che effettua il conteggio delle parentesi di una espressione algebrica.

```
// Conta i vari tipi di parentesi contenute in una espressione
// algebrica (non fa distinzione fra parentesi aperte e chiuse)

#include <iostream>
#include <string>
using namespace std;

main(){
  string espress;
  int pargraf,parquad,partond;
  int i;

   // acquisizione espressione come stringa

  cout << "\nEspressione algebrica ";
  getline(cin,espress);

 // ricerca parentesi nella espressione

  pargraf=parquad=partond=0;
  for(i=0;i<espress.length();i++) {                                  /*1*/

    switch(espress[i]) {                                             /*2*/
      case '{':; case '}':                                           /*3*/
        pargraf++;
        break;                                                       /*4*/
      case '[':; case ']':                                           /*3*/
        parquad++;
        break;                                                       /*4*/
      case '(':; case ')':                                           /*3*/
        partond++;
        break;                                                       /*4*/
    }
  }

// presentazione dei risultati

  cout << "\nGraffe = " << pargraf << endl;
  cout << "Quadre = " << parquad << endl;
  cout << "Tonde  = " << partond << endl;
}
```

Il ciclo che comincia da `1` si occupa di effettuare una *scansione lineare* della stringa: verranno

esaminati tutti i caratteri che la compongono. Se ne può conoscere la lunghezza applicando `length()` alla stringa da elaborare, quindi il ciclo deve contare (la variabile `i`) da `0` (la posizione del primo carattere della stringa) fino al numero che indica la quantità di caratteri che la compongono, estremo escluso, perché il conteggio parte dal valore `0`.

Nella riga con etichetta `2` viene specificata l'espressione da valutare: `espress[i]` cioè il carattere estratto dall'espressione algebrica.

Nelle righe con etichetta `3` si esaminano i casi parentesi aperta o parentesi chiusa. I singoli valori sono seguiti dalla istruzione nulla (il solo carattere `;`) e, poiché l'elaborazione continua da quel punto in poi, sia che si tratti di parentesi aperta che di parentesi chiusa si arriva all'aggiornamento del rispettivo contatore.

Nelle righe con etichetta `4` si blocca l'esecuzione del programma altrimenti, per esempio, una parentesi graffa oltre che come graffa verrebbe conteggiata anche come quadra e tonda, una parentesi quadra verrebbe conteggiata anche come tonda. Si noti che, anche se sono presenti due istruzioni, non vengono utilizzate parentesi per delimitare il blocco: il funzionamento della `switch-case` prevede infatti la continuazione dell'elaborazione con l'istruzione successiva. L'ultima istruzione `break` è inserita solo per coerenza con gli altri casi. Inoltre se in seguito si dovesse aggiungere una istruzione `default`, il programma continuerebbe a dare risultati coerenti senza necessità di interventi se non nella parte da inserire.

3.7 Vettori di stringhe

Il primo programma proposto si occupa di verificare se alcune parole acquisite da input, sono contenute in un vocabolario. Il vocabolario è un vettore di stringhe che contiene tutte le parole che ne fanno parte e che sono acquisite da input.

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

main(){
  vector<string> vocab;                                          /*1*/
  string parcerc,stinp;
  int i,pos;
  bool trovata,continua;

  // Acquisisce parole da inserire nel vocabolario

  cout << “Vocabolario” << endl;
  continua=true;
  for(i=0;continua;i++) {                                        /*2*/
    cout << "\nParola " << i << " " ;
    getline(cin,stinp);
    if(stinp!="")                                                /*3*/
      vocab.push_back(stinp);
    else
      continua=false;                                            /*4*/
  }

  // Acquisisce la parola da cercare
```

```
  cout << "\n\nParola da cercare (Invio per finire) ";
  getline(cin,parcerc);

  while (parcerc!="") {                                          /*5*/

    // Cerca la parola

    trovata=false;
    for (i=0;i<vocab.size();i++) {                               /*6*/
      if (parcerc==vocab[i]) {                                   /*7*/
        trovata=true;
        pos=i;
        break;                                                   /*8*/
      }
    }

    if (trovata)
      cout << "\nParola trovata, posizione " << pos;
    else
      cout << "\nParola non trovata";

    // Prossima parola da cercare

    cout << "\n\nParola da cercare (Invio per finire) ";
    getline(cin, parcerc);
  }
}
```

Nella `1` si dichiara `vocab` come vettore di stringhe.

Il ciclo di acquisizione delle parole contenute nel vocabolario, è controllato dal valore della variabile booleana `continua` (`2`). Se l'input è vuoto (`3`), il valore viene posto a `false` (`4`) e il ciclo termina.

Anche il ciclo per l'input delle parole da cercare (`5`) si comporta allo stesso modo: se la stringa è vuota, l'elaborazione termina. L'unica differenza con il ciclo precedente è che, qui, non si visualizza un conteggio delle parole.

Nel ciclo `6` viene effettuata una scansione del vocabolario alla ricerca, se esiste, di una corrispondenza con la stringa cercata (`7`). Se la parola è presente nel vocabolario la `8` si occupa di forzare l'uscita dal ciclo. È inutile continuare la scansione delle parole del vocabolario.

Il secondo programma proposto estrae da un testo tutte le parole che lo compongono:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

main(){
  string testo,estratta;
  vector<string> token;
  int inizio,fine,i,lparola;
  bool continua;

  cout << "Estrae tutte le parole contenute in un testo" << endl;
  cout << "\nTesto : ";
  getline(cin,testo);
```

```
    // Estrazione parole (parola racchiusa fra spazi)
    // fra due parole c'e' un solo spazio

    continua = true;
    inizio = -1;                                                    /*1*/
    while(continua){

      fine = testo.find(' ',inizio+1);                              /*2*/

      // ultima parola

      if(fine==-1){                                                 /*3*/
        continua = false;
        fine = testo.length();                                      /*4*/
      }

      // Estrazione parola

      lparola = (fine-inizio)-1;                                    /*5*/
      estratta = testo.substr(inizio+1,lparola);                    /*6*/
      token.push_back(estratta);                                    /*7*/

      inizio = fine;                                                /*8*/
    }

    // Elenco parole

    cout << "\nParole che compongono il testo" << endl;

    for(i=0;i<token.size();i++)
      cout << token[i] << endl;
  }
```

Si comincia l'elaborazione inizializzando la variabile `inizio` (`1`) che sarà la posizione, dello spazio che precede la parola, da cui iniziare la ricerca dello spazio che delimita la parola da estrarre (`2`). La parola è delimitata dagli spazi che si trovano nella posizione `inizio` e nella posizione `fine` (sempre `2`). La coppia di valori, contenuti nelle suddette variabili, viene utilizzata per calcolare la quantità di caratteri che compongono la parola.

Se si è arrivati alla fine del testo da elaborare (controllo della `3`), oltre che rendere falsa la condizione di controllo del ciclo al fine di chiudere l'elaborazione, viene assegnata a `fine` la lunghezza del testo (`4`). È come se si posizionasse fine nello *spazio dopo l'ultimo carattere* del testo.

La lunghezza della parola, e quindi la quantità di caratteri da estrarre, è calcolata in `5`. L'estrazione dei caratteri è effettuata dal metodo `substr` richiamato per la stringa `testo`, passandogli la posizione da cui estrarre (quella dopo la posizione dello spazio che precede la parola) e la quantità di caratteri da estrarre (`6`). La parola viene successivamente (`7`) inserita in un vettore.

L'assegnazione della `8` ha lo scopo di passare avanti nella ricerca della prossima parola, assegnando come punto di partenza, il punto di arrivo precedente.

4 Il paradigma procedurale

4.1 Costruzione di un programma: lo sviluppo top-down

Un paradigma è "*un insieme di idee scientifiche collettivamente accettato per dare un senso al mondo dei fenomeni" (T.S.Khun)*. Applicare un paradigma significa applicare una tecnica per scrivere buoni programmi. In questo e nei seguenti paragrafi verrà esposto il paradigma procedurale che può essere così sintetizzato: "*si definiscano le procedure desiderate; si utilizzino gli algoritmi migliori*".

Accade spesso, specie nei problemi complessi, che una stessa sequenza di istruzioni compaia nella stessa forma in più parti dello stesso programma o che sia utilizzata in più programmi. Gli algoritmi riguardano elaborazioni astratte di dati che possono essere adattate a problemi di natura apparentemente diversi (dal punto di vista informatico due problemi sono diversi se *necessitano di elaborazioni diverse* e non se *trattano di cose diverse*). Per fare un esempio riguardante altre discipline basta pensare, per esempio, alla Geometria: il calcolo dell'area di una superficie varia in relazione alla forma geometrica diversa e non alla natura dell'oggetto. L'area di una banconota o di una lastra di marmo si calcolerà sempre allo stesso modo trattandosi in ambedue i casi di rettangoli. L'elaborazione riguardante il calcolo dell'area di un rettangolo ricorrerà nei problemi di calcolo di blocchi di marmo così come nei problemi di calcolo di fogli su cui stampare banconote.

Per risparmiare un inutile lavoro di riscrittura di parti di codice già esistenti, i linguaggi di programmazione prevedono l'uso dei ***sottoprogrammi***. Sostanzialmente un sottoprogramma è una parte del programma che svolge una funzione elementare.

L'uso di sottoprogrammi non è solo limitato al risparmio di lavoro della riscrittura di parti di codice, ma è anche uno strumento che permette di affrontare problemi complessi riconducendoli a un insieme di problemi di difficoltà via via inferiore. Tutto ciò consente al programmatore un controllo maggiore sul programma stesso *nascondendo* nella fase di risoluzione del singolo sottoprogramma, le altri parti in modo tale da *isolare* i singoli aspetti del problema da risolvere.

Si tratta del procedimento di stesura *per raffinamenti successivi* (o ***top-down***). Quando la complessità del problema da risolvere cresce, diventa difficoltoso tenere conto *contemporaneamente* di tutti gli aspetti coinvolti, fin nei minimi particolari, e prendere *contemporaneamente* tutte le decisioni realizzative: in tal caso sarà necessario procedere per approssimazioni successive, cioè decomporre il problema iniziale in sottoproblemi più semplici. In tal modo si affronterà la risoluzione del problema iniziale considerando in una prima approssimazione risolti, da altri programmi di livello gerarchico inferiore, gli aspetti di massima del problema stesso. Si affronterà quindi ciascuno dei sottoproblemi in modo analogo.

In definitiva si comincia specificando la sequenza delle fasi di lavoro necessarie anche se, in questa prima fase, possono mancare i dettagli realizzativi: si presuppone infatti che tali dettagli esistano già. Se poi si passa all'esame di una singola fase di lavoro, questa potrà ancora prevedere azioni complesse ma riguarderà, per come è stata derivata, una *parte* del problema iniziale. Iterando il procedimento, mano a mano, si prenderanno in esame programmi che riguardano parti sempre più limitate del problema iniziale. In tal modo la risoluzione di un problema complesso è stata ricondotta alla risoluzione di più problemi semplici (tanti quante sono le funzioni previste dalla elaborazione originaria).

Le tecniche di sviluppo per raffinamenti successivi suggeriscono cioè di scrivere subito il

programma completo, come se il linguaggio di programmazione a disposizione fosse di livello molto elevato ed orientato proprio al problema in esame.

Tale programma conterrà, oltre alle solite strutture di controllo, anche operazioni complesse che dovranno poi essere ulteriormente specificate. Queste operazioni verranno poi descritte in termini di operazioni ancora più semplici, e così via fino ad arrivare alle operazioni elementari fornite dal linguaggio di programmazione utilizzato.

4.2 Un esempio di sviluppo top-down

In una località geografica sono state rilevate ogni 2 ore le temperature di una giornata. Si vuole conoscere la temperatura media, l'escursione termica e lo scostamento medio dalla media.

Si tratta di scrivere un programma che richiede alcune elaborazioni statistiche su una serie di valori. Si ricorda che la media aritmetica di una serie *n* di valori è data dal rapporto fra la somma dei valori della serie e il numero *n* stesso. L'escursione termica è in pratica il campo di variazione cioè la differenza fra il valore massimo e il valore minimo della serie di valori. Lo scostamento medio è la media dei valori assoluti degli scostamenti dalla media aritmetica, dove lo scostamento è la differenza fra il valore considerato della serie e la media aritmetica.

Innanzi tutto si può osservare che, qualunque sia il problema da risolvere, il processo di sviluppo di un programma avanza attraversando tre stadi: *input dati da elaborare*, *elaborazione*, *output dei risultati ottenuti*. Nell'esempio proposto possono essere tradotti in:

```
Inizio
  Acquisizione temperature rilevate
  Elaborazioni sulle temperature
  Comunicazione risultati
Fine
```

Se, poi, si tiene conto delle elaborazioni da compiere e si specificano meglio le richieste, si avrà:

```
Elaborazioni sulle temperature
Inizio
  Calcolo media e ricerca massimo e minimo
  Calcolo escursione termica
  Calcolo scostamento medio
Fine
```

In questa prima approssimazione si sono evidenziati i risultati intermedi da conseguire affinché il problema possa essere risolto. Non si parla di istruzioni eseguibili ma di *stati di avanzamento* del processo di elaborazione: per il momento non c'è niente di preciso ma il processo di risoluzione del problema, è stato ricondotto a fasi elementari ognuna delle quali si occupa di una determinata elaborazione. Viene evidenziata la sequenza delle operazioni da effettuare: l'escursione termica si può, per esempio, calcolare solo dopo la ricerca del massimo e del minimo.

Si noti che ad ogni fase di lavoro è assegnato un compito specifico ed è quindi più facile la ricerca di un eventuale sottoprogramma errato: se lo scostamento medio è errato e la media risulta corretta è chiaro che, con molta probabilità, l'errore è localizzato nel sottoprogramma che si occupa di tale calcolo.

Il primo sottoprogramma si può già tradurre in istruzioni eseguibili. È opportuno tenere presente che a questo livello il problema da risolvere riguarda solamente l'acquisizione delle temperature rilevate. Il resto del programma, a questo livello, non esiste.

```
Acquisizione temperature
Inizio
  Per indice da 0 a 11
    Ricevi e conserva temperatura rilevata
  Fine-per
Fine
```

Passando al dettaglio della prima elaborazione richiesta, la si può pensare composta da una fase di inizializzazione dell'accumulatore della somma dei termini della serie e delle variabili che conterranno il massimo ed il minimo della serie stessa. La seconda fase è il calcolo vero e proprio.

```
Calcolo media e ricerca massimo e minimo
Inizio
  Inizializza Somma Valori
  Considera primo elemento serie come Massimo e Minimo
  Per indice da 1 a 11
    Aggiorna Somma con elemento considerato
    calcola minimo fra minimo ed elemento
    calcola massimo fra massimo ed elemento
  Fine-per
Fine
```

La seconda fase dell'elaborazione da svolgere è immediata:

```
Calcolo escursione termica
Inizio
  Escursione = Massimo – Minimo
Fine
```

Il dettaglio del calcolo successivo potrebbe essere:

```
Calcolo scostamento medio
Inizio
  Azzera Somma scostamenti
  Per indice da 0 a 11
    Aggiorna Somma scostamenti con scostamento in valore assoluto
  Fine-per
  scostamento medio=Somma scostamenti/12
Fine
```

La stesura del sottoprogramma, che si occupa dell'output dei risultati ottenuti, è immediata:

```
Comunicazione risultati
Inizio
  Comunica Media
  Comunica Escursione termica
  Comunica Scostamento medio
Fine
```

Il programma a questo punto è interamente svolto. Per ogni sottoprogramma ci si è occupati di un solo aspetto dell'elaborazione: ciò rende la stesura del programma, e la sua manutenzione, più semplici. Ogni sottoprogramma diventa più agevole da controllare rispetto al programma nel suo complesso.

Il *processo di scomposizione successiva* non è fissato in maniera univoca: dipende fortemente dalla *soggettività* del programmatore. Non ci sono regole sulla *quantità di pezzi* in cui scomporre il programma. Ci sono delle indicazioni di massima che suggeriscono di limitare il singolo segmento

in maniera sufficiente a che il codice non superi di molto la pagina in modo da coprire, con lo sguardo, l'intero o quasi programma e limitare il singolo segmento a poche azioni di modo che sia più semplice isolare eventuali errori. Inoltre il sottoprogramma deve essere quanto più possibile isolato dal contesto in cui opera, cioè il sottoprogramma deve ***avere al suo interno tutto ciò di cui ha bisogno*** e non fare riferimento a dati particolari presenti nel programma principale. Ciò porta ad alcuni indubbi vantaggi:

- Il sottoprogramma è facilmente esportabile. Se dalla scomposizione di altri programmi si vede che si ha necessità di utilizzare elaborazioni uguali si può riutilizzare il sottoprogramma. È evidente che affinché ciò sia possibile è necessario che il sottoprogramma non faccia riferimento a contesti che in questo caso potrebbero essere diversi. Se, inoltre, il sottoprogramma effettua una sola operazione si potrà avere più opportunità di inserirlo in nuove elaborazioni.
- La manutenzione del programma è semplificata dal fatto che, facendo il sottoprogramma una sola elaborazione e, avendo al suo interno tutto ciò che serve, se c'è un errore nella elaborazione questo è completamente isolato nel sottoprogramma stesso e, quindi, più facilmente rintracciabile.
- Qualora si avesse necessità di modificare una parte del programma, ciò può avvenire facilmente: basta sostituire solamente il sottoprogramma che esegue l'elaborazione da modificare. Il resto del programma non viene interessato dalla modifica effettuata.

L'utilizzo di sottoprogrammi già pronti per la costruzione di un nuovo programma porta ad una metodologia di sviluppo dei programmi che viene comunemente chiamata ***bottom-up*** poiché rappresenta un modo di procedere opposto a quello descritto fino ad ora. Si parte da sottoprogrammi già esistenti che vengono assemblati assieme a nuovi per costruire la nuova elaborazione. In definitiva *"… si può affermare che, nella costruzione di un nuovo algoritmo, è dominante il processo top-down, mentre nell'adattamento (a scopi diversi) di un programma già scritto, assume una maggiore importanza il metodo bottom-up."* (N.Wirth).

4.3 Comunicazioni fra sottoprogrammi

Seguendo il procedimento per scomposizioni successive si arriva alla fine ad un programma principale e ad una serie di sottoprogrammi. Il programma principale *chiama* in un certo ordine i sottoprogrammi; ogni sottoprogramma oltre che *chiamato* può anche essere il *chiamante* di un ulteriore sottoprogramma, come nell'esempio proposto delle temperature, il sottoprogramma delle elaborazioni. Terminato il sottoprogramma l'esecuzione riprende, nel chiamante, dall'istruzione successiva alla chiamata.

Si può dire che tutti i sottoprogrammi fanno parte di un insieme organico: ognuno contribuisce, per la parte di propria competenza, ad una elaborazione finale che è quella fissata dal programma principale. L'elaborazione finale richiesta è frutto della cooperazione delle singole parti; ogni sottoprogramma (unità del sistema) riceve i propri input (intesi come somma delle informazioni necessarie all'espletamento delle proprie funzioni) dai sottoprogrammi precedenti, e fornisce i propri output (intesi come somma delle informazioni prodotte al suo interno) ai sottoprogrammi successivi. Per questi ultimi, le informazioni suddette saranno gli input della propria parte di elaborazione.

Quanto detto porterebbe alla conclusione che tutti i sottoprogrammi, facendo parte di un insieme organico, lavorano sulle stesse variabili. D'altra parte se, per esempio, il reparto carrozzeria e il

reparto verniciatura operano nella stessa fabbrica di automobili, è ovvio che il reparto verniciatura si occuperà delle stesse carrozzerie prodotte dall'altro reparto. Così in effetti è stato per esempio per i linguaggi di programmazione non strutturati: i sottoprogrammi condividevano le stesse variabili, il sottoprogramma eseguiva una parte limitata di elaborazione ma era fortemente legato al contesto generale. La portabilità di un sottoprogramma, in queste condizioni, è estremamente problematica richiedendo pesanti modifiche al codice stesso (si pensi al fatto che il programma destinazione non usa le stesse variabili del programma che ospitava originariamente il sottoprogramma o, peggio ancora, usa variabili con nomi uguali ma con significati diversi).

Per garantire quanto più possibile la portabilità e l'indipendenza dei sottoprogrammi, i linguaggi strutturati adottano un approccio diverso distinguendo le variabili in base alla ***visibilità*** (in inglese *scope*). In relazione alla visibilità le variabili si dividono in due famiglie principali:

- Variabili **globali** visibili cioè da tutti i sottoprogrammi. Tutti i sottoprogrammi possono utilizzarle e modificarle. Sono praticamente patrimonio comune.
- Variabili **locali** visibili solo dal sottoprogramma che li dichiara. Gli altri sottoprogrammi, anche se chiamati, non hanno accesso a tali variabili. La variabile locale è definita nel sottoprogramma ed è qui utilizzabile. Se viene chiamato un sottoprogramma le variabili del chiamante sono mascherate (non accessibili) e riprenderanno ad essere visibili quando il chiamato terminerà e si tornerà al chiamante. L'ambiente del chiamante (l'insieme delle variabili con i rispettivi valori) a questo punto verrà ripristinato esattamente come era prima della chiamata.

Per quanto ribadito più volte sarebbe necessario utilizzare quanto meno possibile (al limite eliminare) le variabili globali per ridurre al minimo la dipendenza dal contesto da parte del sottoprogramma.

Riguardando però l'elaborazione dati comuni, è necessario che il programma chiamante sia in condizioni di poter comunicare con il chiamato. Devono cioè esistere delle *convenzioni di chiamata* cioè delle convenzioni che permettono al chiamante di comunicare dei *parametri* che rappresenteranno gli input sui quali opererà il chiamato. D'altra parte il chiamato avrà necessità di tornare al chiamante dei parametri che conterranno i risultati della propria elaborazione e che potranno essere gestiti successivamente. Queste convenzioni sono generalmente conosciute come ***passaggio di parametri***.

Il passaggio di parametri può avvenire secondo due modalità:

- Si dice che un parametro è **passato per valore** (dal chiamante al chiamato) se il chiamante comunica al chiamato il valore che è contenuto, in quel momento, in una sua variabile. Il chiamato predisporrà una propria variabile locale nella quale verrà ricopiato tale valore. Il chiamato può operare su tale valore, può anche modificarlo ma tali modifiche riguarderanno solo la copia locale su cui sta lavorando. Terminato il sottoprogramma la variabile locale scompare assieme al valore che contiene e viene ripristinata la variabile del chiamante con il valore che essa conteneva prima della chiamata al sottoprogramma.
- Si dice che un parametro è **passato per riferimento o per indirizzo** se il chiamante comunica al chiamato *l'indirizzo di memoria* di una determinata variabile. Il chiamato può utilizzare, per la variabile, un nome diverso ma le locazioni di memoria a cui ci si riferisce sono sempre le stesse. Viene semplicemente stabilito un riferimento diverso alle stesse posizioni di memoria: ogni modifica effettuata si ripercuoterà sulla variabile originaria anche se il nuovo nome cessa di

esistere alla conclusione del sottoprogramma.

Per sintetizzare praticamente su cosa passare per valore e cosa per riferimento, si può affermare che gli input di un sottoprogramma sono passati per valore mentre gli output sono passati per riferimento. Gli input di un sottoprogramma sono utili allo stesso per compiere le proprie elaborazioni mentre gli output sono i prodotti della propria elaborazione che devono essere resi disponibili ai sottoprogrammi successivi.

4.4 Visibilità e namespace

L'applicazione del paradigma funzionale porta a costruire delle librerie contenenti funzioni che si occupano delle elaborazioni riguardanti famiglie di problemi. Le librerie espandono le potenzialità del linguaggio: agli strumenti resi disponibili dal linguaggio standard, si possono aggiungere tutte le funzionalità che servono per agevolare la risoluzione di un determinato problema. Basta includere, nel programma la libreria che si intende utilizzare.

A parte una visione più dettagliata di quanto detto, che verrà fornita nei prossimi paragrafi, è un procedimento che si è adottato fin dalla scrittura del primo programma, quando si è dichiarata la volontà di utilizzare le funzioni, per esempio, della `iostream`.

L'utilizzo di più librerie può portare ad ambiguità se, per esempio, esistono nomi uguali in più librerie. C++ rende disponibili i namespace, che delimitano la visibilità dei nomi in essi dichiarati, per risolvere problemi di questo genere.

```
#include <iostream>
using namespace std;                                          /*1*/

namespace prova1{                                             /*2*/
  int a=5;
};
namespace prova2{                                             /*3*/
  int a=10;
  int b=8;
};

main(){
  cout << prova1::a << endl;                                  /*4*/
  cout << prova2::a << endl;                                  /*5*/
  cout << prova2::b << endl;                                  /*6*/
}
```

La dichiarazione di `1` rende accessibili tutti i nomi del namespace `std`. Le librerie disponibili nel C++ (*iostream*, *string*, *vector*, ecc...) hanno i nomi (*cin*, *cout*, ecc...) inclusi in questo namespace.

Nella `2` viene dichiarato `prova1` che contiene al suo interno una variabile `a` che assume un valore diverso da quello assunto da una variabile, con lo stesso nome, dichiarata nel namespace `prova2` (`3`).

Nella `4` si richiede la stampa della variabile `a` definita nel namespace `prova1`. L'operatore `::`, operatore di visibilità, consente di specificare in *quale namespace* deve essere cercata la variabile *a*. Senza questo operatore, il compilatore genererebbe un errore: non è infatti definita alcuna variabile `a`.

Il valore stampato in conseguenza della esecuzione della `5` è diverso dal precedente, poiché qui ci si

riferisce alla variabile a definita nel namespace `prova2`.

Si poteva aggiungere una riga del tipo:

```
using namespace prova1;
```

e, in questo caso, non era necessario utilizzare l'operatore di visibilità nella `4`.

```
#include <iostream>
using namespace std;

namespace prova1{
  int a=5;
};
namespace prova2{
  int a=10;
  int b=8;
};

using namespace prova1;                                    /*1*/
using namespace prova2;                                    /*1*/

main(){
  cout << prova1::a << endl;                               /*2*/
  cout << prova2::a << endl;                               /*2*/
  cout << b << endl;                                       /*3*/
}
```

Si può fare in modo, come nelle `1`, di rendere accessibili i nomi dei due namespace, e, quindi, avere la possibilità di accedere a quanto contenuto senza necessità di specificare l'operatore di visibilità `::`, ma, in questo caso, tale possibilità può essere sfruttata solo nel caso della `3`. Negli altri due casi di uso (`2`) è necessario specificare il namespace perché il riferimento è ambiguo. Tale sarebbe anche la natura dell'errore evidenziato dal compilatore, se non si usasse l'operatore di visibilità: la variabile è definita in tutte e due i namespace e il compilatore non può decidere a quale riferirsi.

4.5 Tipi di sottoprogrammi

Nei paragrafi precedenti si è parlato di sottoprogrammi in modo generico perché si volevano evidenziare le proprietà comuni. In genere si fa distinzione fra due tipi di sottoprogrammi:

- Le **funzioni**. Sono sottoprogrammi che *restituiscono* al programma chiamante un valore. La chiamata ad una funzione produce, al ritorno, un valore che potrà essere assegnato ad una variabile. Le funzioni vengono utilizzate principalmente a destra del segno di assegnamento essendo, sostanzialmente, sostituite da un valore.
- Le **procedure**. Sono sottoprogrammi che *non restituiscono* alcun valore; si occupano di una fase della elaborazione

È opportuno osservare che quanto espresso prima non esaurisce le comunicazioni fra sottoprogrammi. Da quanto detto infatti potrebbe sembrare che tutte le comunicazioni fra chiamante e chiamato si esauriscano, nella migliore delle ipotesi (funzioni), in un unico valore. In realtà la comunicazione si gioca principalmente sul passaggio di parametri, quindi una procedura può modificare più variabili: basta che riceva per riferimento tali variabili.

Nel linguaggio C++ ogni sottoprogramma ha un nome e i sottoprogrammi vengono chiamati

specificandone il nome e l'ambito di visibilità.

4.6 Le funzioni in C++. Istruzione return

Nel linguaggio C++ la maggior parte di quello che si usa è costituito da funzioni. Per poter simulare le procedure che non ritornano alcun valore è stato introdotto il tipo `void`. Il tipo `void` o *tipo indefinito* è utilizzato dal C++ tutte le volte che il valore di ritorno di una funzione non deve essere preso in considerazione. In pratica nel linguaggio C++ le procedure sono funzioni che restituiscono un `void`.

La costruzione e l'uso di una funzione, può essere schematizzata in tre fasi:

- La **definizione** della funzione cioè l'elenco delle operazioni svolte dalla funzione stessa. La definizione comincia specificando il tipo di valore ritornato, subito dopo viene specificato il nome scelto a piacere dal programmatore e seguente le regole della scelta del nome delle variabili, segue poi l'elenco dei parametri e infine le dichiarazioni locali e le istruzioni così come dallo schema seguente:

  ```
  tipo-ritornato nome-funzione(dichiarazione parametri)
  {
    dichiarazioni ed istruzioni
  }
  ```

  Le definizioni di funzioni possono essere scritte in qualsiasi punto del programma: verranno mandate in esecuzione in seguito alla chiamata e quindi non ha alcuna importanza il posto fisico dove sono allocate. Sono però comuni delle convenzioni secondo le quali le definizioni delle funzioni è bene codificarle dopo il `main`. D'altra parte il `main` stesso non è altro che una funzione particolare che viene eseguita per prima. Il programma quando viene eseguito effettua una chiamata alla funzione `main`.

  Fra le istruzioni contenute nella definizione della funzione particolare importanza assume l'istruzione `return` utilizzata per ritornare al chiamante un valore. La sintassi dell'istruzione prevede di specificare dopo la parola chiave `return` un valore costante o una variabile compatibile con il tipo-ritornato dalla funzione. Es.

  ```
  return 5; // Ritorna al chiamante il valore 5
  return a; // Ritorna al chiamante il valore contenuto nella variabile a
  ```

  Non è importante che le definizioni di tutte le funzioni usate in un programma seguano l'ordine con cui sono chiamate sebbene, per motivi di chiarezza e leggibilità, è opportuno che sia così e che si segua l'ordine specificato prima. In ogni caso la funzione con il nome `main` è eseguita per prima, in qualunque posto sia messa, e le funzioni sono eseguite nell'ordine in cui sono chiamate.

- Il **prototipo** della funzione. Si tratta in sintesi di ripetere all'inizio del programma, prima della definizione della funzione `main`, le dichiarazioni delle funzioni definite dopo. Si riscrive in pratica quanto specificato nella prima riga della definizione della funzione.

  I prototipi sono stati introdotti per permettere al compilatore di effettuare un maggiore controllo sui tipi di parametri: conoscendoli in anticipo, infatti, all'atto della chiamata è possibile stabilire se i parametri passati sono congruenti con quelli attesi. Per questo motivo nel prototipo non è necessario specificare il nome dei parametri: sono indispensabili solo la quantità e il tipo.

Nella costruzione di programmi complessi capita di utilizzare molte funzioni. In questo caso le funzioni sono raggruppate in **librerie** e i rispettivi prototipi sono raggruppati nei files di intestazione (**header** files). Si è avuto modo di utilizzare librerie di funzioni fin dall'inizio. Per esempio sia `cin` che `cout` sono funzioni (in realtà si tratta di oggetti, ma ciò sarà chiarito in seguito) contenute in una libreria di sistema che è *inclusa*, all'atto della compilazione, nel nostro programma. Tali funzioni sono definite nel namespace `std` dello header `iostream` che viene incluso all'inizio del programma.

- La **chiamata** della funzione. Consiste semplicemente nello specificare, laddove occorre utilizzare l'elaborazione fornita dalla funzione, il nome della funzione stessa, eventualmente preceduto dall'operatore di visibilità e dal namespace in cui è definito, e l'elenco dei parametri passati. Il programma chiamante può utilizzare il valore restituito dalla funzione e in tal caso il nome della funzione figurerà, per esempio, in una espressione. Il chiamante può anche trascurare il valore restituito anche se non è di tipo `void`. Basta utilizzare la funzione senza assegnare il suo valore restituito.

4.7 Funzioni e parametri in C++: un esempio pratico

Nel linguaggio C++ tutti i parametri sono passati alle funzioni **per valore**, e ciò allo scopo di isolare quanto più possibile la funzione e renderla riutilizzabile. Per passare alla funzione un parametro per riferimento è necessario utilizzare il simbolo `&` attaccato alla fine della dichiarazione di tipo della variabile o all'inizio del nome della variabile stessa.

La funzione, nella sua definizione, preparerà variabili locali per ricevere i parametri passati dal chiamante. Nel caso di passaggio per indirizzo, ogni variazione effettuata dalla funzione sul contenuto della variabile locale si ripercuoterà sulla variabile corrispondente del chiamante.

I parametri passati dal chiamante sono comunemente chiamati argomenti o **parametri reali**, mentre quelli presenti nella definizione della funzione sono chiamati **parametri formali**.

A questo punto viene proposta la codifica del programma sulle elaborazioni statistiche di temperature.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
using namespace std;

namespace temperature{                                              /*1*/
  const int numero=12;                                              /*2*/

  void rileva(vector<float>& t);                                    /*3*/
  void elab(vector<float> t,float& tm,float& et,float &sm);         /*3*/
  void statistiche(float tm,float et,float sm);                     /*3*/

  void calcMedMnMx(vector<float> t2,float& m,float& mn,float& mx); /*4*/
  float calcEscursione(float mn,float mx){return mx-mn;};           /*4*/
  float calcScostMedio(vector<float> t2,float m);                   /*4*/
}

main(){
  vector<float> temp;                                               /*5*/
  float tempMedia,escursione,scostamento;                           /*5*/
```

```
  cout << "Data una serie di rilevazioni di temperature" << endl
       << "calcola temperatura media, escursione termica, "
       << "scostamento medio" << endl;

  temperature::rileva(temp);                                          /*6*/
  temperature::elab(temp,tempMedia,escursione,scostamento);           /*6*/
  temperature::statistiche(tempMedia,escursione,scostamento);         /*6*/
}

// Rilevazione temperature

void temperature::rileva(vector<float>& t){                           /*7*/
  float tempinp;
  int i;

  cout << "\nRilevazione di " << temperature::numero << " temperature" << endl;
  for(i=0;i<temperature::numero;i++){
    cout << "Introdurre rilevazione " << i+1 << " ";
    cin >> tempinp;
    t.push_back(tempinp);
  };
}

// Elaborazioni sulle temperature rilevate

void temperature::elab(vector<float> t,float& tm,float& et,float &sm){
  float tMin,tMax;

  temperature::calcMedMnMx(t,tm,tMin,tMax);
  et = temperature::calcEscursione(tMin,tMax);                        /*8*/
  sm = temperature::calcScostMedio(t,tm);                             /*8*/
}

// Output statistiche richieste

void temperature::statistiche(float tm,float et,float sm){
  cout << "\nTemperatura media :" << tm << endl;
  cout << "Escursione termica :"  << et << endl;
  cout << "Scostamento medio :"   << sm << endl;
}

// Calcolo temperature media, minima, massima

void temperature::calcMedMnMx(vector<float> t2,float& m,float& mn,float& mx){
  float sommatemp=0.0;
  int i;

  min=max=t2[0];
  sommatemp += t2[0];
  for(i=1;i<temperature::numero;i++){
    sommatemp += t2[i];
    mn = min(mn,t2[i]);                                               /*9*/
    mx = max(mx,t2[i]);                                               /*9*/
  }
  m = sommatemp/temperature::numero;
}

// Calcolo scostamento medio
```

```
float temperature::calcScostMedio(vector<float> t2,float m){
  float sommaScost=0.0,scost;
  int i;

  for(i=0;i<temperature::numero;i++)
    sommaScost += fabs(t2[i]-m);                                    /*10*/

  return sommaScost/temperature::numero;
}
```

Nella `1` viene definito uno spazio per i nomi che conterrà tutte le dichiarazioni che verranno utilizzate nel programma. È questa una buona abitudine che permette di fare riferimento, nel corso del programma, a quanto contenuto in `temperature`, senza preoccuparsi se in altre librerie ci sono funzioni che hanno lo stesso nome.

Nella `2` viene definita una costante per la quantità di rilevazioni di temperature effettuate, in modo da utilizzarla tutte le volte che sarà necessario riferirsi a tale dato. Se si vuole modificare la quantità, basta modificare il valore assegnato e ricompilare il programma.

Nelle `3` sono riportati i prototipi delle funzioni principali in cui è stato suddiviso il programma. La `rileva` si occupa di effettuare l'input del vettore delle temperature. Il vettore, essendo per la funzione un output, è passato per indirizzo. La funzione `elab` partendo dal vettore delle temperature, che per essa è un input (parametro passato per valore), fornisce temperatura media, escursione termica e scostamento medio che, dal suo punto di vista, sono output e, di conseguenza, sono parametri passati per riferimento. La funzione `statistiche` si occupa della visualizzazione dei risultati. I tre dati calcolati sono passati per valore: la funzione deve solo conoscerli, non effettua su di essi alcuna elaborazione.

Le `4` riportano i prototipi delle funzioni che si occupano delle tre elaborazioni richieste. La prima funzione dall'input del vettore fornisce in output temperatura media, minima e massima. La seconda dalla temperatura minima e massima calcola l'escursione termica. La terza dall'input del vettore e dalla temperatura media calcola lo scostamento medio.

La funzione `calcEscursione` che si occupa semplicemente del calcolo del risultato di una differenza, è implementata direttamente nel prototipo. Sono quelle che vengono chiamate ***funzioni inline***. Questa è una tecnica che, normalmente, si usa quando si tratta, come in questo caso, di funzioni che comprendono poche righe di codice. La differenza con la definizione delle altre funzioni consiste nel modo in cui vengono trattate dal compilatore:

- le funzioni inline sono trattate come macroistruzioni. Le istruzioni sono sostituite al nome della funzione: se ci sono, per esempio, due chiamate alla funzione, per due volte sarà sostituito il codice corrispondente.
- nel caso generale, invece, la chiamata alla funzione viene trattata come una chiamata ad un sottoprogramma e viene effettuato un salto al codice della funzione che è tradotto una sola volta: se ci sono, per esempio, due chiamate alla funzione, ci saranno due salti alla stesso posto che contiene il codice della funzione (il codice è presente solo una volta).

In definitiva le funzioni inline sono eseguite in modo più rapido ma occupano più spazio essendo ogni volta, per ogni chiamata, presente il codice della funzione. Questa è sostanzialmente la ragione perché è una tecnica che viene utilizzata per piccole funzioni.

Nelle `5` sono dichiarate le variabili che necessitano al programma che, per come è stato scomposto, si riducono alle variabili di input e di output del programma stesso. Qualsiasi altra variabile sarà definita nella funzione che ne avrà bisogno.

Nelle `6` vengono richiamate in sequenza le tre funzioni principali in cui è stato suddiviso il programma. È usato l'operatore di visibilità per dichiarare lo spazio dei nomi in cui sono dichiarate le funzioni richiamate; se non fosse stato specificato, il compilatore avrebbe generato errore. Nella chiamata alle funzioni sono passati i parametri reali: le variabili per come sono conosciute dalla `main`.

La definizione della funzione in `7`, che richiede le temperature rilevate e le conserva in un vettore, è formalmente uguale ad altri frammenti di codice già esaminati in precedenza. Si fa solo notare, in questa sede, l'uso di variabili locali `tempinp` e `i` utilizzate dentro la funzione e il vettore `t` che, pur avendo visibilità locale, per il fatto di essere dichiarato come parametro, fra l'altro passato per indirizzo, permette alla funzione di operare sul vettore passato dal chiamante, all'atto della chiamata stessa.

Le chiamate presenti nelle `8` riguardano funzioni che ritornano un valore diverso da `void`, nel caso specifico di tipo `float`. È necessario *assegnare* ad una variabile di tipo `float` la chiamata alla funzione: infatti l'esecuzione della chiamata provoca, nei due casi, il calcolo di un valore di tipo `float` così come dichiarato nelle `4`.

Nelle `9` sono usate due funzioni contenute nella libreria `algorithm` inclusa nel programma. Si tratta di due funzioni che restituiscono, rispettivamente, il valore minimo e il valore massimo dei due parametri. Nel programma vengono utilizzate per controllare se le variabili, che conservano tali valori, necessitano di aggiornamento in relazione al confronto con l'elemento del vettore.

Nella `10` è utilizzata una funzione, contenuta nella libreria `cmath`, che calcola il valore assoluto del parametro. La libreria `cmath` contiene altre funzioni (tutte restituiscono un valore in virgola mobile) che possono essere utili nelle applicazioni matematiche:

- `sqrt` per il calcolo della radice quadrata. Alla funzione va passato il valore di cui calcolare la radice
- `pow` per l'elevamento a potenza. Alla funzione vanno passati, come parametri, la base e l'esponente

La libreria contiene anche funzioni per calcoli trigonometrici, logaritmici.

5 Puntatori, strutture, tabelle e classi

5.1 Puntatori ed operatori new e delete

Si è avuto modo di conoscere l'operatore & (*indirizzo di*) utilizzato nel passaggio di un parametro, per riferimento, ad una funzione. L'operatore * è il secondo operatore utilizzato per la manipolazione di indirizzi di memoria: è utilizzato per specificare un **puntatore**.

Un puntatore è una variabile che contiene un indirizzo di memoria. Un puntatore è quindi una variabile che non contiene dati ma l'indirizzo dei dati. Una variabile per essere utilizzata come puntatore è necessario che sia dichiarata come tale. La codifica seguente è relativa ad un programma che esegue la somma di due numeri interi contenuti in variabili allocate dinamicamente e accessibili tramite puntatori:

```
#include <iostream>
using namespace std;

main(){
  int *a,*b,*c;                                          /*1*/

  a = new (int);                                         /*2*/
  b = new (int);                                         /*2*/
  c = new (int);                                         /*2*/

  cout << "\nPrimo numero ";
  cin  >> *a;                                            /*3*/
  cout << "\nSecondo numero ";
  cin  >> *b;                                            /*3*/
  *c = *a + *b;                                          /*3*/
  cout << "\nSomma = " << *c;

  delete a;                                              /*4*/
  delete b;                                              /*4*/
  delete c;                                              /*4*/
}
```

Nella `1` vengono dichiarati tre *puntatori ad interi*. Con tale terminologia si vuole intendere che tali variabili (`a`, `b` e `c`) conterranno ognuna il primo indirizzo di una quantità di locazioni di memoria tali da poter contenere un dato del tipo specificato.

Nelle `2` si alloca in memoria lo spazio dove conservare i valori dei tipi specificati. La variabile `a`, per esempio, non può essere utilizzata per conservare dei valori essendo stata dichiarata come puntatore: in pratica l'operatore `new` alloca in memoria uno spazio raggiungibile attraverso la variabile `a`. Fra le parentesi che seguono l'operatore, e che possono essere omesse, si specifica quanto spazio di memoria deve essere allocato (tanto quanto ne basta per conservare un dato di tipo `int`).

Nelle `3` viene utilizzato lo spazio allocato per conservare i dati provenienti da input o calcolati. In questo caso, attraverso l'operatore `*`, si accede alla zona di memoria dove conservare i dati.

Nelle `4` si libera lo spazio di memoria allocato nelle `2`. Tale operazione si rende necessaria per recuperare lo spazio di memoria dedicato alla conservazione di dati che ora non servono più: in questo esempio tale operazione non è strettamente necessaria, visto che subito dopo il programma termina, ma in generale è bene liberare lo spazio con l'uso esplicito dell'operatore `delete`, in modo

che il compilatore possa riallocare, qualora serva, lo spazio.

L'uso dei puntatori e dell'allocazione dinamica della memoria è dettato da una doppia esigenza:

1. *Gestire in maniera ottimale la memoria*: lo spazio si può allocare giusto per il tempo necessario. Quando si dichiara una variabile, lo spazio allocato dipende dalla dimensione prevista per quella variabile, se invece si dichiara un puntatore, lo spazio allocato è quello necessario per conservare un indirizzo di memoria che è inferiore, specie se la variabile dichiarata non è di un tipo elementare. Se poi si libera la memoria, lo spazio è occupato giusto per il tempo minimo indispensabile. Nell'esempio, essendo lo spazio dedicato ad un `int` esiguo, non è giustificato l'uso di puntatori, ma nei prossimi paragrafi si esamineranno casi in cui la differenza può essere rilevante.
2. *Velocizzare l'accesso ai dati*: nel passaggio ad una funzione di un parametro una cosa è passare un puntatore, altra una variabile soprattutto se quest'ultima contiene dati estesi.

5.2 Le strutture

Una struttura è un insieme di variabili di uno o più tipi, raggruppate da un nome in comune. Anche i vettori sono collezioni di variabili come le strutture, solo che un vettore può contenere solo variabili dello stesso tipo, mentre le variabili contenute in una struttura non devono essere necessariamente dello stesso tipo.

Le strutture del linguaggio C++ coincidono con quelli che in Informatica sono comunemente definiti **record**. Il raggruppamento sotto un nome comune permette di rappresentare, tramite le strutture, *entità* logiche in cui le variabili comprese nella struttura rappresentano gli *attributi* di tali entità.

Per esempio con una struttura si può rappresentare l'entità `dipendente` i cui attributi potrebbero essere: `reparto`, `cognome`, `nome`, `stipendio`. In tale caso la definizione potrebbe essere:

```
struct dipendente{
  string reparto;
  string cognome;
  string nome;
  float stipendio;
};
```

La sintassi del linguaggio prevede, dopo la parola chiave `struct`, un nome che identificherà la struttura (il *tag* della struttura). Racchiuse tra le parentesi sono dichiarate le variabili che fanno parte della struttura (i *membri* della struttura). È bene chiarire che in questo modo si definisce la struttura logica `dipendente`, che descrive l'aspetto della struttura, e non un posto fisico dove conservare i dati. In pratica si può considerare come se si fosse definito, per esempio, com'è composto il tipo `int`: ciò è necessario per dichiarare variabili di tipo `int`.

Per mostrare l'uso elementare di una struttura, viene proposto un semplice programma che riceve da input i dati di un dipendente e mostra i dati ricevuti:

```
#include <iostream>
#include <string>
using namespace std;

namespace azienda{                                          /*1*/
  struct dipendente{
```

```
    string reparto;
    string cognome;
    string nome;
    float stipendio;
  };
}

main(){
  azienda::dipendente dip1;                                    /*2*/

  cout << "Esempio di uso di una struttura in C++" << endl;
  cout << “Inserire reparto in cui lavora il dipendente “;
  getline(cin,dip1.reparto);                                   /*3*/
  cout << "Inserire cognome dipendente ";
  getline(cin,dip1.cognome);                                   /*3*/
  cout << "Inserire nome ";
  getline(cin,dip1.nome);                                      /*3*/
  cout << "Inserire stipendio ";
  cin >> dip1.stipendio;                                       /*3*/

  cout << "Dati dipendente" << endl;
  cout << dip1.reparto << endl;                                /*3*/
  cout << dip1.cognome << endl;                                /*3*/
  cout << dip1.nome << endl;                                   /*3*/
  cout << dip1.stipendio << endl;                              /*3*/
}
```

La struttura viene definita, nella `1`, nello spazio di nomi `azienda`.

Nella `2` viene dichiarata una variabile del tipo `dipendente` definito nello spazio `azienda`. La variabile dichiarata avrà i membri definiti nella struttura: ci sarà, per esempio, un `cognome` per il dipendente `dip1`.

L'accesso ai membri della struttura, come evidenziato nelle `3`, avviene utilizzando l'operatore di appartenenza (il punto). In questo modo si può distinguere se, per esempio, il cognome si riferisce al dipendente `dip1` o al dipendente `dip2`, se fosse stata dichiarata un'altra variabile di tipo `dipendente` con quel nome.

5.3 Puntatori a strutture

Una struttura può occupare parecchio spazio in memoria centrale, in relazione alla quantità di membri contenuti nella struttura stessa e al tipo di membri. Ricorre spesso la necessità di usare puntatori a strutture: per esempio per passare ad una funzione una o più strutture. Utilizzando riferimenti invece di passare le strutture per valore si può ottenere una elaborazione più veloce: si evita, infatti, di ricopiare nelle variabili locali della funzione le strutture passate.

Tali necessità sono così frequenti che il linguaggio C++ mette a disposizione l’operatore `->` (il trattino seguito dal simbolo di maggiore) per accedere ai membri della struttura.

Utilizzando l'allocazione dinamica della memoria, il programma di visualizzazione dei dati di un dipendente potrebbe diventare:

```
#include <iostream>
#include <string>
using namespace std;
```

```
namespace azienda{
  struct dipendente{
    string reparto;
    string cognome;
    string nome;
    float stipendio;
  };
}

main(){
  azienda::dipendente *dip1;                                    /*1*/
  dip1 = new(azienda::dipendente);                              /*2*/

  cout << "Esempio di uso di una struttura in C++" << endl;
  cout << “Inserire reparto in cui lavora il dipendente “;
  getline(cin,dip1->reparto);                                   /*3*/
  cout << "Inserire cognome dipendente ";
  getline(cin,dip1->cognome);                                   /*3*/
  cout << "Inserire nome ";
  getline(cin,dip1->nome);                                      /*3*/
  cout << "Inserire stipendio ";
  cin >> dip1->stipendio;                                       /*3*/

  cout << "Dati dipendente" << endl;
  cout << dip1->reparto << endl;                                /*3*/
  cout << dip1->cognome << endl;                                /*3*/
  cout << dip1->nome << endl;                                   /*3*/
  cout << dip1->stipendio << endl;                              /*3*/
}
```

Nella `1` viene definito `dip1` come puntatore ad una variabile di tipo `dipendente`.

Nella `2` si richiede uno spazio di memoria tale da poter conservare una struttura del tipo specificato.

L'accesso ai membri, come evidenziato nelle `3`, avviene utilizzando l'operatore `->` invece dell'operatore punto.

5.4 Tabelle: vettori di strutture

Una tabella è costituita da una successione di elementi detti ***record*** costituiti da due o più ***campi***, il primo è chiamato ***chiave*** e distingue un record da un altro. Anche un vettore di tipi elementari, per esempio di `int`, può definirsi tabella, ma nella realtà delle applicazioni informatiche è comune trovare vettori contenenti strutture.

Sulle tabelle sono comuni due tipi di algoritmi: la ricerca (data una tabella, cercare la chiave associata ad un determinato record), la selezione (data una tabella, estrarne un altra che contiene i record della prima tabella che soddisfano a determinati requisiti).

Il primo programma proposto, data una tabella contenete i dipendenti di una azienda, cerca se una persona, di cui vengono forniti cognome e nome, è un dipendente dell'azienda e, in questo caso, visualizza i suoi dati:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
```

```
namespace azienda{                                                    /*1*/
  const int numerodip=10;

  struct dipendente{
    string reparto;
    string cognome;
    string nome;
    float stipendio;
  };

  void insertDip(vector<dipendente>& d);
  void insertCogNom(string& c,string& n);
  int cerca(vector<dipendente> d,string c,string n);
  void stampa(string r,float s);
}

main(){
  vector<azienda::dipendente> libropaga;                              /*2*/
  string cogncerca,nomcerca,repcerca;
  float stipcerca;
  int pos;

  // Inserimento dati dei dipendenti,
  // cognome e nome dipendente da cercare

  azienda::insertDip(libropaga);                                      /*3*/
  azienda::insertCogNom(cogncerca,nomcerca);                          /*3*/

  // Ricerca dipendente

  pos = azienda::cerca(libropaga,cogncerca,nomcerca);                 /*4*/

  // Visualizza dati dipendente

  if(pos>=0)                                                          /*5*/
    azienda::stampa(libropaga[pos].reparto,libropaga[pos].stipendio);
  else
    cout << "Non risulta come dipendente" << endl;
}

// Inserimento dipendenti

void azienda::insertDip(vector<dipendente>& d,string& c,string& n){
  azienda::dipendente tempdip;
  int i;

  cout << "Inserimento dati dei dipendenti" << endl;
  for(i=0;i<azienda::numerodip;i++){
    cout << "\nDipendente " << i << endl;
    cout << "Reparto ";
    getline(cin,tempdip.reparto);                                     /*6*/
    cout << "Cognome ";
    getline(cin,tempdip.cognome);                                     /*6*/
    cout << "Nome ";
    getline(cin,tempdip.nome);                                        /*6*/
    cout << "Stipendio ";
    cin >> tempdip.stipendio;                                         /*6*/
    cin.ignore();                                                     /*7*/
```

```
    d.push_back(tempdip);                                          /*8*/
  }
}

// Cognome e nome dipendente da cercare

void azienda::insertCogNom(string& c,string& n){
  cout << "Dipendente da cercare" << endl
       << "Cognome ";
  getline(cin,c);
  cout << "Nome ";
  getline(cin,n);
}

// Ricerca

int azienda::cerca(vector<dipendente> d,string c,string n){
  int i,p;

  p=-1;                                                            /*9*/
  for(i=0;i<azienda::numerodip;i++){
    if(c==d[i].cognome && n==d[i].nome){                           /*10*/
      p=i;                                                         /*11*/
      break;                                                       /*12*/
    }
  }
  return p;                                                        /*13*/
}

// Dati dipendente trovato

void azienda::stampa(string r,float s){
  cout << "\nReparto di appartenenza " << r << endl;
  cout << "Stipendio " << s << endl;
}
```

Come in altri esempi, in `1` è definito uno spazio di nomi in cui dichiarare la quantità di dipendenti, la struttura con i dati di interesse del dipendente, i prototipi delle funzioni.

Nella `2` viene dichiarato un vettore di tipo `dipendente`, tipo a sua volta definito in `azienda`.

Nelle `3` si chiamano le funzioni per l'input della tabella dei dipendenti e dei dati della persona da cercare.

Nella `4` viene chiamata la funzione di ricerca che restituisce il numero della riga della tabella che contiene le informazioni cercate. Se non esiste una riga della tabella che soddisfa i requisiti richiesti, la funzione, ritorna il valore `-1`. Il valore ritornato può essere testato (`5`) per il tipo di stampa da effettuare.

Nella funzione di inserimento dei dati dei dipendenti, prima vengono conservati gli input in una struttura temporanea (`6`) e, quindi, l'elemento viene inserito nel vettore (`8`). La `7` svuota il buffer di tastiera del carattere *Invio*, lasciato dall'input numerico precedente, al fine di consentire il prossimo `getline`.

La funzione di ricerca inizializza al valore `-1` la posizione (`9`), se, poi, il cognome e nome passati come parametri hanno gli stessi valori dei rispettivi della riga della tabella considerata (`10`), la posizione viene conservata (`11`) e si forza una uscita anticipata dal ciclo (`12`). La posizione trovata,

il valore assegnato in `9` o quello assegnato in `11`, viene restituito al chiamante (`13`).

Il prossimo programma proposto effettua una selezione. Acquisita la tabella dei dipendenti e un reparto, viene stampato l'elenco dei dipendenti che lavorano nel reparto.

Il programma è, in buona parte, uguale al precedente, cambiano solo poche funzioni che saranno riportate nel listato seguente. Le parti di codice uguali, per motivi di chiarezza, non sono riportate. La numerazione delle righe commentate è riportata in modo da evidenziare le differenze con il programma precedente:

```
...
namespace azienda{
  ...
  string insertRep();
  void selezione(vector<dipendente> d1,string r,vector<dipendente>& d2);
  void stampa(vector<dipendente> d2);
}
...
main(){
  vector<azienda::dipendente> libropaga,dipRep;                    /*2*/
  string repcerca;
  ...
  repcerca = azienda::insertRep();                                 /*3*/
  ...
  azienda::selezione(libropaga,repcerca,dipRep);                   /*4*/
  ...
  if(dipRep.size())                                                /*5*/
    azienda::stampa(dipRep);
  else
    cout << "Non risultano dipendenti nel reparto" << endl;
}
...
// Inserimento reparto da selezionare

string azienda::insertRep(){
  string r;
  cout << "Reparto da selezionare" << endl;
  getline(cin,r);
  return r;
}
// Seleziona in base al reparto

void azienda::selezione(vector<azienda::dipendente> d1,string r,
                        vector<azienda::dipendente>& d2){
  int i;

  for(i=0;i<azienda::numerodip;i++){
    if(r==d1[i].reparto)                                           /*11*/
      d2.push_back(d1[i]);
  }
}

// Stampa dipendenti del reparto cercato

void azienda::stampa(vector<azienda::dipendente> d2){
  int i;

  cout << "\nDipendenti che lavorano nel reparto" << endl;
```

```
  for(i=0;i<d2.size();i++)
    cout << d2[i].cognome   << " "
         << d2[i].nome      << " "
         << d2[i].stipendio << endl;
}
```

Nello spazio di nomi `azienda` la funzione `insertCogNom` è sostituita dalla `insertRep`, `cerca` è sostituita da `selezione` e la funzione `stampa` è modificata in modo da stampare non più singole informazioni ma un intero vettore. Così come le dichiarazioni di variabili della `2` si adattano alla nuova elaborazione: un nuovo vettore `dipRep` e la stringa `repcerca`.

La chiamata alla funzione di inserimento del reparto, della `3`, ritorna un dato di tipo `string`.

La `4`, rispetto al programma precedente, è modificata in modo da chiamare la nuova funzione. Il controllo in `5` non si deve più occupare dell'esistenza di una posizione, ma di un vettore.

La funzione di selezione, dopo aver verificato nella `11` che il reparto è quello interessato, invia l'elemento al nuovo vettore.

5.5 Estensione delle strutture: le classi

I membri di una struttura possono anche essere funzioni oltre che dati. Per una biblioteca che effettua operazioni di prestito ai soci, un libro, per esempio, non è solo un insieme di attributi ma anche una *cosa* che può essere prestata:

```
namespace biblioteca{
  struct libro {
    string titolo;
    string autore;
    string editore;
    float prezzo;
    bool presente;

    bool prestato(){                                          /*1*/
      bool prestitoOk=false;
      if (presente){
        presente   = false;                                   /*2*/
        prestitoOk = true;
      };
      return prestitoOk;
    };
  };
}
```

Nella `1` viene definita la funzione per effettuare il prestito di un libro: tale funzione setta a `false` il valore del membro `presente` e ritorna un valore logico sul risultato dell'operazione effettuata (se il libro è già stato prestato l'operazione non può avere luogo). La funzione è inserita nella struttura per intendere che è una proprietà distintiva del libro, così come il titolo o l'autore. L'accesso al membro `presente`, nella `2`, non necessita dell'operatore `::` perché si trova nella stesso namespace così come dell'operatore punto, poiché avviene dentro la struttura stessa.

```
...
biblioteca::libro lib1;                                       /*1*/
...
cout << "Inserire titolo :";
getline(cin,lib1.titolo);                                     /*2*/
```

```
cout << "Titolo :" << lib1.titolo << endl;
if(!lib1.prestato())                                          /*3*/
  cout << "Libro gia\' prestato" << endl;
else
  cout << "Prestito effettuato" << endl;
...
```

Il frammento di programma riportato, dopo aver dichiarato nella `1` una variabile di tipo `libro`, chiede il titolo di un libro e ne effettua il prestito. Allo stesso modo come nella `2` si ha accesso alla variabile membro `titolo` di `lib1`, nella `3` si ha accesso alla funzione membro `prestato` riferita sempre a `lib1`. Detta funzione modifica il valore `presente` di `lib1`.

Le classi sono uno degli elementi fondamentali della programmazione orientata agli oggetti (OOP). *Una classe è un tipo di dato definito dall'utente che ha un proprio insieme di dati e di funzioni* (Abstract Data Type).

```
// esempio non funzionante !!

class libro {
 string titolo;
 string autore;
 string editore;
 float prezzo;
 bool presente;

 bool prestato(){
   bool prestitoOk=false;
   if (presente){
     presente   = false;
     prestitoOk = true;
   };
   return prestitoOk;
 };
};
```

In questo modo viene definita la classe libro. Un tipo con attributi e comportamenti (le funzioni definite nella classe).

La definizione di `libro`, tranne che per la sostituzione della parola chiave `struct` con la parola chiave `class`, sembra identica a quella adottata precedentemente, solo che ora, come d'altra parte messo in evidenza dalla riga di commento, c'è una differenza sostanziale: il compilatore se si tenta di accedere ad un attributo dell'oggetto, fornisce un errore che evidenzia la non visibilità dello stesso. Ciò è dovuto alle regole di visibilità dei vari elementi: se, infatti, non si specifica altrimenti, la visibilità è limitata solo all'interno della classe, per esempio la funzione `prestato` può accedere a `presente`.

In generale in una classe possono essere specificati tre livelli di visibilità:

```
class libro {
 public:
   ...
 protected:
   ...
 private:
   ...
};
```

Nella sezione `public` si specificano i membri accessibili agli altri membri della classe, alle istanze della classe e alle classi discendenti (quelle che si definiscono a partire da `libro` e che ne ***ereditano*** le proprietà).

Nella sezione `protected` si specificano i membri accessibili agli elementi della classe, alle classi discendenti ma non alle istanze della classe. È la definizione assunta per default ed è quindi questo il motivo perché nell'esempio proposto non erano visibili i vari membri. Nelle strutture invece la definizione di default è `public` ed è questo che ha garantito l'accesso ai membri negli esempi riportati in precedenza.

Nella sezione `private` si specificano i membri che devono essere accessibili solamente agli altri membri della stessa classe.

Le sezioni possono essere disposte in modo qualsiasi, figurare più volte nella definizione della classe e non è obbligatorio inserirle tutte.

La possibilità di definire diversi livelli di mascheramento dell'informazione (***data hiding***) è una delle caratteristiche fondamentali della programmazione ad oggetti. Aiuta, infatti, a creare una interfaccia della classe che, nascondendo l'implementazione, mostra l'oggetto definito in maniera da evidenziarne le proprietà e i comportamenti. Mettere assieme in una classe le proprietà e le funzioni è un'altra delle caratteristiche fondamentali della OOP: l'***incapsulamento***. In questo modo ogni oggetto della classe non ha solo caratteristiche ma anche comportamenti esattamente come nella realtà: se si gestisce una biblioteca, un libro, non è solo un oggetto che ha, per esempio, il titolo "Pinocchio" ma è anche oggetto di prestito e restituzione. Una classe contiene tutto il necessario per usare i dati; il programma che usa la classe non ha bisogno di sapere com'è fatta.

Nella OOP, in ragione del mascheramento dei dati, questi normalmente vengono dichiarati nella sezione `private` e si accede ad essi utilizzando dei metodi `public`. Questa, si può dire, è una regola generale: è l'interfaccia della classe che mette a disposizione gli strumenti per accedere ai dati. In tal modo può essere, per vari motivi, modificata la struttura interna dei dati della classe senza che cambi l'uso degli oggetti della classe.

5.6 Costruzione e uso di una classe step by step

Applicare la OOP alla risoluzione di un determinato problema vuol dire innanzi tutto individuare le classi interessate e i metodi che devono avere. Utilizzando una similitudine si può pensare al problema da risolvere come ad una *piece teatrale*: ci sono dei personaggi, ognuno con le proprie caratteristiche e comportamenti, che interagiscono fra loro. Il problema da risolvere è la *piece* da recitare, i personaggi (tratteggiati ognuno dalle proprie caratteristiche) sono le classi e sono interpretati da attori (le istanze della classe: gli oggetti) che, per il fatto di interpretare determinati personaggi, sanno cosa fare e come comportarsi perché ciò fa parte dell'interpretazione di quel determinato personaggio.

Viene proposto, come primo esempio, un semplice programma per conoscere il totale della fattura di cui siano date le righe che la compongono, ognuna individuata da una determinata quantità e dal prezzo unitario dell'oggetto venduto.

Per la risoluzione del problema proposto si può considerare solo la classe riga-fattura. Gli attributi saranno la quantità venduta e il prezzo unitario:

```
namespace fattura{
```

```
  class riga{
    public:

    private:
      int qv;
      float pu;
  };
}
```

per il momento sono stati inseriti, nella parte privata della classe, solo gli attributi. Per quanto riguarda i metodi, intanto bisogna senz'altro prevederne uno per inserire i dati negli attributi privati:

```
namespace fattura{
  class riga{
    public:
      void nuova(int q; float p);
    private:
      int qv;
      float pu;
  };

  // metodo per inserimento di una nuova riga

  void riga::nuova(int q; float p){
    qv = q;
    pu = p;
  };
}
```

Per completare il soddisfacimento delle esigenze del programma da sviluppare, la riga deve essere in condizioni di calcolare, e restituire, il totale:

```
namespace fattura{
  class riga{
    public:
      void nuova(int q; float p);
      float totriga();
    private:
      int qv;
      float pu;
  };

  // metodo per inserimento di una nuova riga

  void riga::nuova(int q; float p){
    qv = q;
    pu = p;
  };

  // metodo per il calcolo del totale della riga

  float riga::totriga(){
    float tr;
    tr = (float) qv*pu;
    return tr;
  };
}
```

A questo punto la definizione della classe, in conseguenza delle richieste del programma da sviluppare, è completa: per il momento si registrerà nel file `c-riga.h` da includere nel sorgente del programma che la dovrà utilizzare.

Prima di procedere oltre è opportuno rispondere ad una eventuale osservazione sulla mancanza, nella definizione della classe, di implementazione di metodi per l'output e il controllo dei dati inseriti. In questo caso, infatti, non è possibile conoscere i dati inseriti che sono conservati in variabili private.

Intanto si può osservare che, per lo sviluppo del programma proposto, non è necessario alcun metodo per l'output dei dati. Inoltre il non prevedere alcun metodo, probabilmente utile per un utilizzo futuro, e in altri contesti, della classe, non toglie generalità; si possono implementare nuovi metodi o, addirittura, modificare quelli esistenti utilizzando l'*ereditarietà*, così come si tratterà in seguito.

```
// Calcolo del totale di una fattura date le righe di vendita

#include <iostream>
using namespace std;

#include "c-riga.h"                                                    /*1*/

main(){
  int qvend,n,i;
  float pzunit,totfat;
  fattura::riga r;                                                     /*2*/

  totfat=0.0;

  // elaborazione righe fattura

  for(i=1;;i++){                                                       /*3*/
    cout << "Riga n. " << i << endl;                                   /*4*/
    cout << "Quantita\' oggetti venduti (0 per finire) ";
    cin >> qvend;

    if(!qvend)                                                         /*5*/
      break;

    cout << "Prezzo unitario ";
    cin >> pzunit;

    r.nuova(qvend,pzunit);                                             /*6*/
    totfat += r.totriga();                                             /*7*/
  };

  // totale cercato

  cout << "\nTotale fattura " << totfat << endl;
}
```

Essendo semplice e leggibile il programma non è stato suddiviso in funzioni.

Con la `1` si include nel file la definizione della classe `riga`. Così da poter, nella `2`, dichiarare una istanza della classe.

Il ciclo `3` è un ciclo senza fine (non c'è infatti alcuna condizione di controllo) da cui si esce (`5`) se si

inserisce un valore nullo nella quantità venduta. Il ciclo `for` è utilizzato per poter far visualizzare, nella `4`, la numerazione delle righe.

Dopo aver acquisito i dati dall'input, nella `6` si richiama il metodo `nuova` per la conservazione dei dati e nella `4` si richiama `totriga` per l'aggiornamento del totale della fattura.

5.7 Dalla struttura alla classe libro

Viene presentato, di seguito, il file `c-libro.h` nel quale è implementata, estendendola rispetto alla presentazione contenuta in un precedente paragrafo, la classe `libro` per la gestione dei prestiti in una biblioteca:

```
namespace biblioteca{                                          /*1*/
  struct datilib{                                              /*2*/
    string titolo;
    string autore;
    string editore;
    float prezzo;
  };

  class libro {                                                /*3*/
  public:
    void setLibro(datilib lset);                               /*4*/
    void getLibro(datilib& lget);                              /*4*/
    bool getInPrestito(){return presente;};                    /*5*/
    bool prestato();                                           /*6*/
    bool restituito();                                         /*6*/
  private:
    datilib libbib;                                            /*7*/
    bool presente;                                             /*8*/
  };

  // Implementazione metodi della classe

  void libro::setLibro(datilib lset){                          /*9*/
    libbib = lset;                                             /*10*/
    presente=true;                                             /*11*/
  }

  void libro::getLibro(datilib& lget){                         /*9*/
    lget = libbib;                                             /*10*/
  }

  // Operazioni di prestito

  bool libro::prestato(){                                      /*9*/
    bool prestitoOk=false;
    if (getInPrestito()){
      presente   = false;
      prestitoOk = true;
    };
    return prestitoOk;
  }

  bool libro::restituito(){                                    /*9*/
    bool ritornatoOk=false;
    if (!getInPrestito()){
      presente    = true;
```

```
        ritornatoOk = true;
      };
      return ritornatoOk;
    }
  }
```

Tutta la definizione della classe è definita nello spazio di nomi `biblioteca` (`1`).

Nella `2` è definita una struttura di tipo `libro`: l'insieme di dati che compongono un libro, l'anagrafica del libro.

Nella `3`, invece, il libro diventa una entità su cui si svolgono determinate operazioni e non solo un insieme di dati. La modifica sostanziale è indicata anche dai nomi utilizzati per le variabili: da `datilib` (dati del libro) a `libbib` (libro in una biblioteca). D'altra parte anche nel mondo reale il modo di agire è abbastanza simile: si registrano i dati del libro e quindi si inserisce il libro nelle disponibilità della biblioteca.

Per poter assolvere ai compiti richiesti da un libro in una biblioteca è necessario, oltre che definire i dati che compongono il libro dal punto di vista dell'utilizzatore della classe (`7`), anche dati (`8`) gestiti internamente alla classe stessa. In ogni caso tutti i dati, al solito, sono definiti nella parte `private` della classe e sono gestiti dai metodi pubblici definiti nella stessa.

Fra i metodi pubblici sono definiti alcuni di tipo `set` e `get` (`4` e `5`), rispettivamente, per l'inserimento e il reperimento dei dati degli oggetti della classe, oltre che (`6`) metodi specifici degli oggetti che interessano. La funzione membro `5` è una funzione inline.

Nell'implementazione delle funzioni (`9`) è utilizzato l'operatore di visibilità `::` per poter fare riferimento alle funzioni definite come membri della classe.

Le operazioni sul libro da gestire nella biblioteca prevedono, fra le altre, la conservazione o l'estrazione dei dati del libro (`10`) ed, eventualmente nel caso dell'inserimento di un nuovo libro (il metodo `setlibro`) anche il settaggio della variabile `presente` (`11`) ad indicare la disponibilità, al prestito, del libro stesso.

I metodi `6` implementano le operazioni ammissibili per i libri. Come si nota nelle definizioni, si tratta, sostanzialmente, di modificare l'attributo `presente`.

5.8 Utilizzo della classe: vettori di oggetti

Il programma proposto, dopo aver caricato in un vettore i libri della biblioteca, visualizza un menù per mezzo del quale si possono gestire alcune operazioni sui libri:

```
// PRESTITI1: gestione prestiti libri di una biblioteca versione 1

#include <iostream>
#include <string>
#include <vector>
using namespace std;

#include "c-libro.h"                                              /*1*/
namespace biblioteca{                                             /*2*/
  void insertLibri(vector<libro>& l);
  void prestito(vector<libro>& l);
  void restituzione(vector<libro>& l);
  void info(vector<libro> l);
}
```

```
main(){
  vector<biblioteca::libro> biblio;                                    /*3*/
  int tipoop;

 // Gestione prestiti libri

  biblioteca::insertLibri(biblio);                                     /*4*/

  for(;;){
    cout << "\n1 - prestito";
    cout << "\n2 - restituzione";
    cout << "\n3 - info libro";
    cout << "\n0 - fine";
    cout << "\nOperazione ? ";
    cin >> tipoop;
    if(!tipoop) break;

    switch (tipoop){
     case 1:
       biblioteca::prestito(biblio);
       break;
     case 2:
       biblioteca::restituzione(biblio);
       break;
     case 3:
       biblioteca::info(biblio);
       break;
    default:
      cout << "Inserire 1,2,3 o 0";
   }
 }
}

// Registrazione libri nella biblioteca

void biblioteca::insertLibri(vector<libro>& l){                        /*5*/
  biblioteca::libro libtemp;
  biblioteca::datilib l1;
  int n,i;

  cout << "Quanti libri? ";
  cin >> n;
  cin.ignore();
  for (i=0;i<n;i++){
   cout << "Libro in posizione n. " << i << endl;
   cout << "Titolo  ->";
   getline(cin,l1.titolo);
   cout << "Autore  ->";
   getline(cin,l1.autore);
   cout << "Editore ->";
   getline(cin,l1.editore);
   cout << "Prezzo  ->";
   cin >> l1.prezzo;
   cin.ignore();

   libtemp.setLibro(l1);                                               /*6*/
   l.push_back(libtemp);                                               /*7*/
  }
```

```
}

// Prestito di un libro

void biblioteca::prestito(vector<libro>& l){
  int quale,quanti;

  quanti = l.size()-1;                                          /*8*/
  cout << "\nQuale libro (0," << quanti << ")? ";
  cin >> quale;
  quale = (quale<0 || quale>quanti) ? quanti:quale;             /*9*/

  if (!l[quale].prestato())                                     /*10*/
    cout << "\nLibro gia\' prestato";
  else
    cout << "\nPrestito effettuato";
}

// Restituzione di un libro prestato

void biblioteca::restituzione(vector<libro>& l){
  int quale,quanti;

  quanti = l.size()-1;                                          /*8*/
  cout << "\nQuale libro (0," << quanti << ")? ";
  cin >> quale;
  quale = (quale<0 || quale>quanti) ? quanti:quale;             /*9*/

  if (!l[quale].restituito())                                   /*10*/
    cout << "\nLibro gia\' presente";
  else
    cout << "\nRestituzione effettuata";
}

void biblioteca::info(vector<libro> l){
  biblioteca::datilib l1;
  int quale,quanti;

  quanti = l.size()-1;                                          /*8*/
  cout << "\nQuale libro (0," << quanti << ")? ";
  cin >> quale;
  quale = (quale<0 || quale>quanti) ? quanti:quale;             /*9*/

  l[quale].getLibro(l1);                                        /*11*/
  cout << l1.titolo << " " << l1.autore << " "
       << l1.editore << " " << l1.prezzo;                       /*12*/
  if(l[quale].getInPrestito())
    cout << "\nLibro presente";
  else
    cout << "\nLibro in prestito";
}
```

Per mezzo della inclusione in `1` possono essere dichiarati, nel programma, sia oggetti di tipo `libro` che strutture di tipo `datilib`.

Nella `2` viene espanso lo spazio di nomi biblioteca, già definito in `c-libro.h`, in modo da contenere anche le funzioni utilizzate nel programma. Poteva essere usato anche un altro nome, ma si è fatta la scelta di inserire le funzioni nello stesso spazio di `biblioteca` in modo da contenere tutto quello di

cui si ha bisogno.

Nel `main` viene dichiarato un vettore per contenere i libri della biblioteca (`3`), caricato dalla chiamata `4`. Subito dopo viene presentato un menù per mezzo del quale si può scegliere l'operazione da effettuare.

La funzione definita in `5`, si occupa di popolare il vettore. Dopo aver acquisito i dati in una struttura temporanea, genera un oggetto di tipo `libro` (`6`) e lo invia al vettore (`7`).

A questa, come alle altre funzioni, è passato come parametro un vettore di tipo `libro`: non è stato necessario specificare altro perché, già nel nome della funzione, è utilizzato l'operatore di visibilità per lo spazio dei nomi `biblioteca` e il tipo `libro` è definito nello stesso spazio.

Le operazioni previste dal programma possono essere effettuate solo su libri presenti nella biblioteca. Il rintracciamento del libro è realizzato specificandone la posizione. Per non permettere l'inserimento di posizioni illegittime, dopo aver calcolato la posizione dell'ultimo libro inserito (`8`), un input fuori da tale limite o con posizione negativa, viene, per semplicità, ricondotto all'ultimo inserito (`9`).

Il prestito o la restituzione del libro sono effettuati richiamando i relativi metodi applicati al libro cercato (`10`). I dati del libro vengono recuperati chiamando uno dei metodi `get` (`11`). Viene utilizzata una variabile `l1` di tipo `datilib` di cui successivamente (`12`) vengono visualizzati i membri.

6 Il paradigma ad oggetti

6.1 Classe aggregato

Nell'ultimo esempio l'*oggetto libro* interagiva con una tabella contenente i libri della biblioteca. In pratica la logica di scrittura del programma restava sempre la stessa e i nuovi elementi introdotti (classi ed oggetti) venivano inseriti in quel contesto. Tutto ciò limita fortemente i vantaggi dell'uso di questi nuovi strumenti.

L'esempio proposto aveva principalmente valenza didattica: si trattava di rendere più *soft* il passaggio ad una nuova concezione di progettazione e sviluppo dei programmi. In realtà il vantaggio maggiore degli oggetti, e la straordinaria diffusione che hanno avuto, è giustificato dal fatto che l'uso di queste nuove tecniche rende più semplice la scrittura dei programmi avvicinandola al nostro modo di risolvere i problemi: il mondo che ci circonda è fatto di oggetti che hanno delle proprie caratteristiche (comportamenti) e che interagiscono tra di loro scambiandosi ***messaggi*** (come li chiama la OOP). L'oggetto che riceve il messaggio si comporta in maniera coerente con le azioni legate al messaggio.

Nella vita reale, un dipendente ha dei compiti specifici da assolvere e che mette in atto in conseguenza a direttive (i messaggi) che riceve. Se poi il dipendente è inserito in un sistema complesso (per esempio è inserito in un contesto produttivo), interagisce con altri dipendenti che hanno, anche loro, compiti specifici. In pratica per la risoluzione di un problema bisogna pensare di far lavorare insieme oggetti che interagiscono fra di loro scambiandosi messaggi: è quello che viene chiamato paradigma della programmazione ad oggetti.

"Il paradigma della programmazione ad oggetti è il seguente: si determini quali classi si desiderano; si fornisca un insieme completo delle operazioni di ogni classe; si renda esplicito ciò che hanno in comune con l'eredità" (B.Stroustrup)

Si cercherà di adottare un simile approccio alla risoluzione del problema affrontato nell'esempio proposto della gestione dei prestiti di una biblioteca (dell'ereditarietà degli oggetti se ne parlerà più avanti).

Si era già definita la classe `libro` (la definizione è contenuta nel file `c-libro.h`), ora sarà necessario definire la classe `libreria`: **classe aggregato** di oggetti di tipo `libro`.

Anche per la libreria si possono fare osservazioni simili a quelle fatte allorquando si è trattato di definire la classe `libro`. Una libreria è non solo un contenitore di libri ma anche un gestore degli stessi (non può essere solo una specie di buco nero dove far scomparire i libri). Deve poter permettere operazioni minime di manutenzione: aggiungere un nuovo libro, aggiornare i dati registrati, estrarre i dati di un libro, fornire informazioni sulla quantità di libri esistenti. Si potrebbero aggiungere anche tante altre funzioni che deve assolvere una libreria (per esempio eliminazione di un libro non più presente) ma, nell'esempio proposto, ci si limiterà per semplicità a implementare nella classe solo i metodi descritti. D'altra parte è importante osservare che in questo modo non si toglie generalità: esiste infatti nella programmazione ad oggetti, e questo è uno dei vantaggi notevoli di questo tipo di approccio alla scrittura di programmi come d'altra parte si è già avuto modo di evidenziare, una proprietà degli oggetti (l'***ereditarietà***) che permette di ampliare le funzionalità di una classe in maniera semplice ed efficiente e, se necessario, modificare le funzionalità esistenti.

```
#ifndef C_LIBRERIA                                                  /*1*/
#define C_LIBRERIA                                                  /*1*/

#include <vector>
using namespace std;

#include "c-libro.h"                                                /*2*/

namespace biblioteca{
  class libreria{
  public:
    int aggiungi(libro lib);                                        /*3*/
    bool estrai(int quale,libro& lib);                              /*3*/
    bool aggiorna(int quale,libro lib);                             /*3*/
    int dotazione(){return bib.size();};                            /*3*/
  private:
    vector<libro> bib;                                              /*4*/
  };

  // Aggiunge un libro

  int libreria::aggiungi(libro lib){                                /*5*/
    bib.push_back(lib);
    return dotazione()-1;
  };

  // Estrae le informazioni su un libro

  bool libreria::estrai(int quale,libro& lib){                      /*6*/
    bool estrattoOK=false;

    if(quale>=0 && quale<dotazione()){                              /*7*/
      lib = bib[quale];
      estrattoOK=true;
    }

    return estrattoOK;
  };

  // Aggiorna i dati di un libro esistente

  bool libreria::aggiorna(int quale,libro lib){                     /*8*/
    bool eseguito=false;

    if(quale>=0 && quale<dotazione()){
      bib[quale]=lib;                                               /*9*/
      eseguito = true;
    }
    return eseguito;
  };
}

#endif                                                              /*1*/
```

Le tre righe `1`, due all'inizio del file di definizione della classe, una alla fine sono direttive al pre-compilatore. Capiterà spesso, da ora in poi, di avere necessità di includere più volte, in più file di codice, le classi che servono. Nell'esempio proposto il file `c-libro.h` viene incluso nella definizione della nuova classe, che è appunto un aggregato di libri, ma anche nel programma di

gestione della biblioteca che necessita di definire oggetti di quella classe. Il compilatore si troverebbe, in questo caso, una duplicazione di definizioni e non potrebbe assolvere alla propria funzione. Per evitare il rischio di duplicati, nei file di intestazioni delle classi, si aggiungono direttive che dicono al compilatore che se non è definita una certa variabile, nel caso in esame `C_LIBRERIA`, allora si definisce la variabile (seconda riga) e si prosegue. Se invece la variabile esiste (il file è già stato incluso), si passa alla fine della condizione (ultima riga), evitando una definizione duplicata.

Anche nel file di definizione della classe libro si dovranno aggiungere righe simili, per esempio, per definire la variabile `C_LIBRO`.

L'inclusione della `2` è qui necessaria per poter definire oggetti della classe `libro`.

I metodi pubblici della classe, definiti in `3`, permettono le operazioni minime che servono per la gestione dei prestiti per come è stata enunciata. L'aggregato di libri (la `libreria`) è definito nella `4` come vettore di tipo `libro`, ma di questo chi usa la classe, non ha alcuna percezione: esistono solo i metodi pubblici per gestire gli oggetti della classe.

Il metodo `aggiungi` della `5` inserisce un nuovo libro nella libreria e restituisce, qualora serva, la posizione in cui è stato inserito.

Il metodo `estrai`, definito in `6`, restituisce le informazioni su un libro di cui viene specificata la posizione. Se la posizione è fuori dall'intervallo permesso (controllo in `7`), restituisce un valore booleano `false`.

il metodo `aggiorna` della `8` mette un libro, ricevuto come parametro, nella libreria in una posizione che è ricevuta anch'essa come parametro. L'assegnazione alla posizione è effettuata dalla `9`. Il metodo restituisce `false` se l'aggiornamento non è stato possibile (posizione fuori dai limiti ammessi).

6.2 Interazione fra oggetti

A questo punto si può riscrivere il programma per la gestione dei prestiti della biblioteca, nel quale interagiranno oggetti di tipo `libro` ed oggetti di tipo `libreria`, cominciando dalla funzione `main`:

```
// PRESTITI2: gestione prestiti libri di una biblioteca versione 2

#include <iostream>
#include <string>
using namespace std;

#include "c-libro.h"                                          /*1*/
#include "c-libreria.h"                                       /*1*/

namespace biblioteca{
  void insertLibri(libreria& l);                              /*2*/
  void prestito(libreria& l);                                 /*2*/
  void restituzione(libreria& l);                             /*2*/
  void info(libreria l);                                      /*2*/
}

main(){
  biblioteca::libreria biblio;                                /*3*/
  int tipoop;
```

```
 // Gestione prestiti libri

  biblioteca::insertLibri(biblio);

  for(;;){
    cout << "\n1 - prestito";
    cout << "\n2 - restituzione";
    cout << "\n3 - info libro";
    cout << "\n0 - fine";
    cout << "\nOperazione ? ";
    cin >> tipoop;
    if(!tipoop) break;

    switch (tipoop){
     case 1:
       biblioteca::prestito(biblio);
       break;
     case 2:
       biblioteca::restituzione(biblio);
       break;
     case 3:
       biblioteca::info(biblio);
       break;
    default:
      cout << "Inserire 1,2,3 o 0";
   }
 }
}
```

Per mezzo delle `1` si includono nel file le definizioni delle classi. Si noti che la classe contenuta in `c-libro.h` è inclusa anche nella definizione della classe contenuta in `c-libreria.h`. La definizione sarebbe duplicata se non fosse condizionata all'esistenza della variabile `C_LIBRO` che, quando richiamata da `libreria.h`, è già definita.

Le funzioni il cui prototipo è in `2`, richiedono come parametro un oggetto di tipo `libreria` che è definito nello stesso spazio di nomi e, quindi, conosciuto. Cosa non valida nella dichiarazione dell'oggetto in `3` che ha necessità di specificare l'operatore di visibilità.

Dichiarazione della `3` a parte, il resto della funzione `main` coincide con quello della stessa funzione in `PRESTITI1`. D'altra parte il `main` si occupa soltanto di presentare il menù di scelta e richiamare le funzioni di gestione.

```
// Registrazione libri nella biblioteca

void biblioteca::insertLibri(libreria& l){                          /*1*/
  biblioteca::libro libtemp;
  biblioteca::datilib l1;
  int n,i;

  cout << "Quanti libri? ";
  cin >> n;
  cin.ignore();
  for (i=0;i<n;i++){
   cout << "Libro in posizione n. " << i << endl;
   cout << "Titolo  ->";
   getline(cin,l1.titolo);
   cout << "Autore  ->";
   getline(cin,l1.autore);
```

```
    cout << "Editore ->";
    getline(cin,l1.editore);
    cout << "Prezzo  ->";
    cin >> l1.prezzo;
    cin.ignore();

    libtemp.setLibro(l1);
    l.aggiungi(libtemp);                                          /*2*/
  }
}
```

La funzione di inserimento dei libri è quasi del tutto coincidente con la corrispondente della versione precedente del programma di gestione della biblioteca. L'unica differenza è che, stavolta, viene passato (1) un parametro di tipo `libreria` e l'inserimento del libro viene effettuato nella 2 inviando il messaggio `aggiungi` alla libreria e passandogli il libro da aggiungere.

```
// Prestito di un libro

void biblioteca::prestito(libreria& l){                           /*1*/
  int quale,quanti;
  biblioteca::libro lib;

  quanti = l.dotazione()-1;                                       /*2*/
  cout << "\nQuale libro (0," << quanti << ")? ";
  cin >> quale;

  if (l.estrai(quale,lib)){                                       /*3*/
    if(lib.prestato()){                                           /*4*/
      l.aggiorna(quale,lib);                                      /*5*/
      cout << "\nPrestito registrato" << endl;
    }else
      cout << "\nLibro non presente" << endl;
  }else
    cout << "\nLibro inesistente" << endl;
}

// Restituzione di un libro prestato

void biblioteca::restituzione(libreria& l){                       /*1*/
  int quale,quanti;
  biblioteca::libro lib;

  quanti = l.dotazione()-1;                                       /*2*/
  cout << "\nQuale libro (0," << quanti << ")? ";
  cin >> quale;

  if (l.estrai(quale,lib)){                                       /*3*/
    if(lib.restituito()){                                         /*4*/
      l.aggiorna(quale,lib);                                      /*5*/
      cout << "\nRestituzione registrata" << endl;
    }else
      cout << "\nLibro non in prestito" << endl;
  }else
    cout << "\nLibro inesistente" << endl;
}
```

Le funzioni per le operazioni di prestito e restituzione ricevono come parametro un oggetto di tipo `libreria` su cui operare (1), calcolano la posizione dell'ultimo libro inserito (2) chiamando il

metodo `dotazione` della libreria, estraggono dalla libreria il libro desiderato se esiste, verificando il valore ritornato dal metodo (`3`), effettuano l'operazione richiesta sul libro (`4`), aggiornano (`5`), infine, i dati del libro nella libreria (ripongono il libro nella libreria).

```
// Informazioni su libro

void biblioteca::info(libreria l){                                    /*1*/
  biblioteca::libro lib;
  biblioteca::datilib l1;
  int quale,quanti;

  quanti = l.dotazione()-1;
  cout << "\nQuale libro (0," << quanti << ")? ";
  cin >> quale;

  if(l.estrai(quale,lib)){                                            /*2*/
    lib.getLibro(l1);                                                 /*3*/
    cout << l1.titolo << " " << l1.autore << " "
         << l1.editore << " " << l1.prezzo << endl;
    if(lib.getInPrestito())
      cout << "\nLibro presente";
    else
      cout << "\nLibro in prestito";
  }else
    cout << "Libro inesistente" << endl;
}
```

La funzione di visualizzazione dei dati di un libro, estrae (`2`), dalla libreria passata come parametro (`1`), il libro interessato, legge i suoi dati (`3`) visualizza i membri della struttura.

6.3 Ereditarietà: da libro a libSocio

Una delle proprietà più potenti delle classi è la possibilità di definire una nuova classe (classe discendente, ***subclass***) a partire da una classe esistente (classe antenata, ***superclass***). La classe discendente ha tutte le proprietà della classe genitrice e, in più, può avere nuove proprietà o metodi specifici per la nuova classe o, addirittura, può ridefinire i metodi della classe genitrice. Questo meccanismo è quello a cui si fa riferimento quando si parla di ereditarietà.

Questo esposto brevemente è un meccanismo che permette, a partire da una classe generica, di derivare, mediante ereditarietà, classi via via sempre più specializzate: la classe figlia è una specializzazione della classe padre, i comportamenti generali della classe derivata sono quelli della classe genitrice a cui si aggiungono i comportamenti tipici degli oggetti della classe derivata stessa. Per portare un esempio intuitivo si potrebbe dire che la zanzara è un insetto (con tutte le caratteristiche tipiche degli insetti) che punge e succhia il sangue (caratteristiche tipiche della zanzara).

La classe `libro`, utilizzata precedentemente, definisce i comportamenti generici di un libro, contenuto in una libreria, che può essere prestato. Se però si pensa ad una biblioteca con soci iscritti che usufruiscono dei servizi, l'operazione di prestito, definita nella classe `libro`, è generica. Non basta, infatti, dire che il libro è in prestito, occorrerebbe, per esempio, avere anche informazioni sul socio che lo ha preso in prestito e l'operazione, per esempio, di prestito riguarda anche la registrazione dei dati del socio che ha preso in prestito il libro.

```
// Definizione di classe derivata
```

```
#ifndef C_LIBSOCIO
#define C_LIBSOCIO

#include "c-libro.h"

namespace biblioteca{
  struct datisoc{                                                    /*1*/
    string nome;
    string cognome;
    string via;
  };

  class libSocio : public libro {                                    /*2*/
  public:
    libSocio();                                                      /*3*/
    void getSocio(datisoc& sget);                                    /*4*/
    bool prestato(datisoc s1);                                       /*5*/
    bool restituito();                                               /*5*/
  private:
    void putSocio(datisoc s1);                                       /*6*/
    datisoc socbib;
  };

  // Inizializzazione dati socio

  libSocio::libSocio(){
    socbib.nome=" ", socbib.cognome=" ", socbib.via=" ";
  }

  // Dati del socio

  void libSocio::getSocio(datisoc& sget){
    sget = socbib;
  }

  // Inserisce i dati del socio

  void libSocio::putSocio(datisoc s1){
    socbib = s1;
  }

  // Prestito del libro al socio

  bool libSocio::prestato(datisoc s1){                               /*7*/
    bool prestitoOK=false;

    if(libro::prestato()){                                           /*8*/
      putSocio(s1);                                                  /*9*/
      prestitoOK=true;
    }
    return prestitoOK;
  }

  // Restituzione libro dal socio

  bool libSocio::restituito(){                                       /*7*/
    bool ritornatoOK=false;
    datisoc s1;
```

```
    if(libro::restituito()){                                    /*8*/
      s1.nome=" ", s1.cognome=" ", s1.via=" ";
      putSocio(s1);                                             /*9*/
      ritornatoOK=true;
    }
    return ritornatoOK;
  }
}

#endif
```

Nella 1, similmente al caso della classe `libro`, si definisce una struttura che contiene i dati di interesse del socio.

Nella 2 si evidenzia l'inizio di una gerarchia di classi: la nuova classe `libSocio` è un discendente di `libro` e ne eredita tutti i metodi. La sintassi del C++ prevede l'uso dell'operatore `:` fra il nome attribuito alla classe discendente e quello della classe genitore. Il qualificatore `public` associato a `libro`, assicura che i metodi pubblici di `libro` saranno pubblici anche per `libSocio`. I qualificatori ammessi sono: `public`, `protected`, `private`. Se si fosse specificato `private` i metodi ereditati sarebbero stati pubblici per `libro` ma non per `libSocio`: per esempio `getLibro()` sarebbe disponibile per un oggetto della classe `libro` ma non per un oggetto della `libSocio`. Il qualificatore `public` garantisce, nell'esempio proposto, l'accessibilità del metodo `getLibro()` anche da parte degli oggetti della classe `libSocio`.

Nella parte pubblica vengono definiti i metodi specifici della classe (4), in aggiunta a quelli ereditati da `libro`. Una proprietà notevole dell'ereditarietà è quella evidenziata nelle righe `5`. La classe discendente ha la possibilità di ridefinire i metodi ereditati dalla classe base (***overload***) per poterli adattare alla propria specificità.

Nella parte `private` della classe vengono definiti i dati membri della nuova classe, accessibili dai metodi della classe. Per la proprietà del mascheramento dei dati (incapsulamento), le funzioni membro di `libSocio` non hanno accesso diretto ai dati membri di `libro` (`libbib`, `presente`) essendo questi dichiarati nella sezione `private` della classe base. L'unico modo per accedere a tali dati è quello di utilizzare i metodi ereditati. Se si voleva avere la possibilità, per la classe `libSocio`, di accedere direttamente a `libbib`, `presente`, bastava dichiararli nella sezione `protected` della classe `libro`.

Fra i metodi presenti si nota nella `3` la presenza di un metodo particolare: il ***costruttore*** della classe. Il costruttore è una funzione che ha lo stesso nome della classe e non ritorna alcun parametro, nemmeno `void`. Al costruttore possono invece essere passati parametri, nel modo consueto delle convenzioni di passaggio di parametri ad una funzione. Il costruttore ha questo nome perché è la funzione che viene richiamata in automatico (non può essere richiamata esplicitamente) quando si instanzia un oggetto della classe (quando cioè si dichiara una variabile di tipo `libSocio`). Nella funzione vengono scritte le istruzioni che devono essere eseguite nel momento in cui viene allocato spazio in memoria per una istanza della classe: tipicamente le inizializzazioni dei dati contenuti nella parte `private`. Simmetricamente può esistere anche il ***distruttore*** della classe, una funzione che viene eseguita in automatico quando cessa la visibilità dell'oggetto della classe e quindi, come il costruttore, non può essere richiamata esplicitamente. Nel caso di allocazione dinamica di memoria, il distruttore, potrebbe essere utile per recuperare la memoria utilizzata. Il nome del distruttore della classe `libSocio`, qualora esistesse, sarebbe `~libSocio()`. Il distruttore è una funzione che ha lo

stesso nome della classe, preceduto dal carattere ~.

Nell'esempio proposto il costruttore si rende necessario per inizializzare i dati del socio a cui è stato effettuato il prestito e che, quando si inserisce un libro nella biblioteca, ancora non esiste.

Il metodo `putSocio`, come da `6`, non è accessibile da parte degli oggetti della classe. È richiamato solo da un altro metodo della classe.

Le `7` ridefiniscono due metodi già definiti nella classe `libro`, in modo da adattarsi alla nuova classe. Tutte le volte che si richiamerà uno dei metodi applicati ad un oggetto della classe `libSocio`, verrà fatto riferimento a tali funzioni: le nuove definizioni mascherano le definizione di `prestato()` e `restituito()` ereditate da `libro`. Le istruzioni delle righe `8` ordinano di eseguire il metodo relativo per come era stato definito in `libro`: è questo quello a cui si fa riferimento quando si dice che una classe discendente è una specializzazione della classe base. In pratica qui si dice: occorre eseguire queste nuove istruzioni (quelle definite a cominciare dalle `7`) in aggiunta a quelle che si eseguivano prima (righe `8`). I nuovi metodi si occupano di conservare (`9`) gli opportuni valori nelle variabili private del socio. Agli aggiornamenti dei dati del libro, ci pensa il metodo richiamato nelle `8`.

Le regole adottate per la derivazione di `libSocio` da `libro` possono essere estese al caso generale:

- La classe `C2` eredita dalla classe `C1` quando la classe figlia `C2` è una specializzazione della classe padre `C1`. `libro` è un libro generico della biblioteca, `libSocio` è un libro al quale è collegato un socio.
- La classe `C2` eredita dalla classe `C1` se **`C2` IS-A `C1`** (C2 è un C1). `libSocio` è un `libro` con ulteriori attributi, metodi aggiunti e ridefiniti.

6.4 Rivisitazione del programma di gestione prestiti

Nel programma di gestione dei prestiti, per come presentato fino ad ora, si interagisce con oggetti della classe libro, che reagivano in un certo modo, definito dai metodi, ai messaggi inviati. Per esempio, volendo effettuare un prestito, si invia il messaggio `prestato()` ad un oggetto della classe e questo, risponde modificando un indicatore (il valore conservato nella variabile privata `presente`).

Nella programmazione ad oggetti, così come con gli oggetti della vita reale quando si interagisce con essi, se si desidera un effetto diverso basta utilizzare un oggetto diverso. Per esempio se si utilizza una matita per scrivere, qualora si volesse un segno più duraturo, occorrerebbe utilizzare una penna. Parafrasando il modo di esprimersi della OOP, si direbbe che il messaggio inviato all'oggetto resta sempre lo stesso (si sta scrivendo); è l'oggetto che reagisce in modo diverso. Questo è quello che la OOP chiama ***polimorfismo***: caratteristica che consente alle istanze di classi differenti di reagire allo stesso messaggio (una chiamata di funzione) in maniere diverse.

```
// PRESTITI3: gestione prestiti libri di una biblioteca versione 3

#include <iostream>
#include <string>
using namespace std;

#include "c-libsocio.h"
#include "c-libreria3.h"                                        /*1*/
```

```
namespace biblioteca{
  void insertLibri(libreria& l);
  void prestito(libreria& l);
  void restituzione(libreria& l);
  void info(libreria l);
}

main(){
  biblioteca::libreria biblio;                                    /*2*/
  int tipoop;
  ...
}
```

Nella nuova revisione del programma di gestione prestiti della biblioteca, viene incluso un nuovo file con una nuova classe (1). Il file `c-libreria3.h`, di cui è superfluo riportare il listato, è ottenuto, per il momento, dal file `c-libreria.h` cambiando tutte le ricorrenze di `libro` con `libSocio`. In tal modo la classe `libreria` diventa un aggregato di oggetti di tipo `libSocio`. Si esaminerà più avanti un sistema più elegante, e corretto formalmente, di adattare la classe aggregato al nuovo tipo di oggetti.

Considerate le variazioni riportate non è necessario apportare alcuna modifica alle dichiarazioni di oggetti della classe, come in 2, perché la definizione della classe è la stessa ma fa, ora, riferimento ad oggetti diversi.

```
// Registrazione libri nella biblioteca

void biblioteca::insertLibri(libreria& l){
  biblioteca::libSocio libtemp;                                   /*1*/
  biblioteca::datilib ltemp;
  ...
   libtemp.setLibro(ltemp);                                       /*2*/
   l.aggiungi(libtemp);
  }
}
```

Anche la funzione di inserimento libri non subisce alcuna variazione rispetto alla versione precedente perché in sede di inserimento libri, il socio non interviene. Nella 2 si richiama il metodo per l'inserimento del libro, valido per oggetti della classe `libro`, ma anche per oggetti della classe `libSocio` poiché quest'ultima eredita i metodi della classe genitrice `libro`. Si fa notare che i dati del socio sono inizializzati a stringhe nulle e ciò in virtù della definizione `1` e del fatto che è definito, nella classe, un costruttore che provvede a ciò già a partire dall'esistenza di un oggetto della classe, cioè dalla `1`.

```
// Prestito di un libro

void biblioteca::prestito(libreria& l){
  int quale,quanti;
  biblioteca::libSocio lib;
  biblioteca::datilib ltemp;
  biblioteca::datisoc stemp;

  quanti = l.dotazione()-1;
  cout << "\nQuale libro (0," << quanti << ")? ";
  cin >> quale;

  if (l.estrai(quale,lib)){
```

```
    lib.getLibro(ltemp);                                          /*1*/
    cout << ltemp.titolo << " " << ltemp.autore << " "
         << ltemp.editore << " " << ltemp.prezzo << endl;
    cin.ignore();
    cout << "Nome socio :";
    getline(cin,stemp.nome);                                      /*2*/
    cout << "Cognome socio :";
    getline(cin,stemp.cognome);                                   /*2*/
    cout << "Recapito socio :";
    getline(cin,stemp.via);                                       /*2*/
    if(lib.prestato(stemp)){                                      /*3*/
      l.aggiorna(quale,lib);
      cout << "\nPrestito registrato" << endl;
    }else
      cout << "\nLibro non presente" << endl;
  }else
    cout << "\nLibro inesistente" << endl;
}
```

La funzione che gestisce il prestito di un libro è quella che subisce le maggiori modifiche. Stavolta, infatti, dopo essersi accertati della esistenza del libro richiesto e aver estratto il libro dalla biblioteca (`1`), si acquisiscono, per mezzo delle `2`, i dati del socio e si invia il messaggio `prestato` (`3`) ad un oggetto della classe `libSocio`. L'oggetto risponde, per come ridefinito il metodo, modificando il flag `presente` del libro e aggiungendo i dati del socio.

```
// Restituzione di un libro prestato

void biblioteca::restituzione(libreria& l){
  biblioteca::libSocio lib;                                       /*1*/
  ...
    if(lib.restituito()){                                         /*2*/
  ...
}
```

La funzione che gestisce la restituzione del libro, non necessita di alcuna modifica. È il metodo richiamato in `2`, ora inviato ad oggetti di tipo diverso (`1`), che si occupa dei nuovi compiti.

```
// Informazioni su libro

void biblioteca::info(libreria l){
  biblioteca::libSocio lib;
  biblioteca::datilib ltemp;
  biblioteca::datisoc stemp;
  int quale,quanti;

  quanti = l.dotazione()-1;
  cout << "\nQuale libro (0," << quanti << ")? ";
  cin >> quale;

  if(l.estrai(quale,lib)){
    lib.getLibro(ltemp);                                          /*1*/
    lib.getSocio(stemp);                                          /*2*/
    cout << "Libro" << endl;
    cout << ltemp.titolo << " " << ltemp.autore << " "
         << ltemp.editore << " " << ltemp.prezzo << endl;
    cout << "Socio" << endl;
    cout << stemp.nome << " " << stemp.cognome << " "
         << stemp.via << endl;
```

```
    }else
      cout << "Libro inesistente" << endl;
  }
```

La funzione di visualizzazione, oltre che dei dati del libro (1), si deve occupare pure di estrarre i dati del socio (2).

6.5 Le classi modello: i template

Nella stesura dell'ultima versione del programma di gestione prestiti, si è avuta l'esigenza di adattare la classe `libreria` in modo da contenere oggetti di tipo `libSocio` e non più di tipo `libro` per come era stata definita originariamente. Il problema è stato risolto sostituendo brutalmente, nella classe, le ricorrenze di `libro` con `libSocio`. È però importante osservare che si trattava, come anche evidenziato, di una soluzione temporanea. La biblioteca potrebbe avere necessità di gestire oltre che libri anche, per esempio, riviste o altro da poter prestare ai propri soci.

In pratica si tratterebbe di avere la possibilità di gestire nello stesso modo cose diverse: `aggiungi`, `estrai`, `aggiorna`, vanno bene perché soddisfano le necessità di gestione dei servizi della biblioteca, ma, come nel mondo reale, questi servizi dovrebbero poter essere applicabili a tutti gli oggetti disponibili nella biblioteca.

Il linguaggio C++ permette di definire classi che gestiscono elementi generici, caratteristica molto utile per la definizione di classi aggregato di oggetti, per esempio la libreria. Utilizzando questa caratteristica, una volta definita la classe `libreria` si avranno a disposizione i metodi per la gestione di oggetti generici e, se si dichiara che `libreria` contiene oggetti di tipo `libro`, il metodo `estrai` permetterà di estrarre un oggetto di tipo `libro`, se invece contiene oggetti di tipo `libSocio`, il metodo restituirà un oggetto di tipo `libSocio`.

```
// Definizione della classe modello libreria

#ifndef C_LIBRERIA4
#define C_LIBRERIA4

#include <vector>
using namespace std;

namespace biblioteca{
  template <class tipo>                                          /*1*/
  class libreria{
  public:
    int aggiungi(tipo lib);
    bool estrai(int quale,tipo& lib);
    bool aggiorna(int quale,tipo lib);
    int dotazione(){return bib.size();};
  private:
    vector<tipo> bib;                                            /*2*/
  };

  // Aggiunge un libro

  template <class tipo>                                          /*1*/
  int libreria<tipo>::aggiungi(tipo lib){                        /*3*/
    bib.push_back(lib);
    return dotazione()-1;
  };
```

```
  // Estae le informazioni su un libro

  template <class tipo>                                          /*1*/
  bool libreria<tipo>::estrai(int quale,tipo& lib){              /*3*/
    bool estrattoOK=false;

    if(quale>=0 && quale<dotazione()){
      lib = bib[quale];
      estrattoOK=true;
    }

    return estrattoOK;
  };

  // Aggiorna i dati di un libro esistente

  template <class tipo>                                          /*1*/
  bool libreria<tipo>::aggiorna(int quale,tipo lib){             /*3*/
    bool eseguito=false;

    if(quale>=0 && quale<dotazione()){
      bib[quale]=lib;
      eseguito = true;
    }
    return eseguito;
  };
}

#endif
```

Le righe `1` che precedono tutte le definizioni, specificano, appunto, che si tratta di una classe modello. Fra parentesi angolari (<>) è contenuta la parola chiave `class` e il nome, scelto dal programmatore, per identificare il tipo di classe trattato. Si tratta, sostanzialmente, di un *segnaposto* che verrà sostituito dal tipo effettivo quando la classe sarà utilizzata e che permette, alla classe stessa, la possibilità di riferirsi ad oggetti diversi. La `2` definisce un vettore che conterrà elementi di tipo `tipo`.

Nella definizione dei metodi (`3`), il nome della classe è seguito dalle parentesi angolari con il segnaposto.

6.6 Utilizzo dei template

Se si utilizza la nuova `c-libreria4.h` le modifiche da apportare al programma di gestione prestiti sono molto limitate:

```
#include <iostream>
#include <string>
using namespace std;

#include "c-libsocio.h"
#include "c-libreria4.h"                                         /*1*/

namespace biblioteca{
  void insertLibri(libreria<libSocio>& l);                       /*2*/
  void prestito(libreria<libSocio>& l);                          /*2*/
  void restituzione(libreria<libSocio>& l);                      /*2*/
```

```
  void info(libreria<libSocio> l);                                    /*2*/
}

main(){
  biblioteca::libreria<biblioteca::libSocio> biblio;                  /*3*/
...
}
```

Oltre all'inclusione del template (1), l'uso della classe modello richiede il tipo da specificare nel segnaposto (2). Nel caso in esame la libreria gestirà oggetti di tipo `libSocio`.

L'unica differenza fra le 2 e la 3 è relativa all'ambito di visibilità: nelle 2 si è nello spazio `biblioteca` che contiene anche la definizione di `libSocio`, cosa non vera per la 3. È quindi necessario specificare l'operatore di visibilità e lo spazio dei nomi in cui `libSocio` è definito.

Come sicuramente si sarà notato l'inclusione, in molti esempi, della `vector`, ha permesso di utilizzare la struttura vettore. Ogni volta, specificando il tipo nel segnaposto, si è gestito un vettore di stringhe, di libri ecc... In definitiva, come ora dovrebbe essere stato chiarito, si tratta della definizione di una classe template.

Se la biblioteca volesse gestire, utilizzando le stesse funzionalità, le riviste, basterebbe aggiungere la definizione della nuova classe e utilizzare il template in modo corretto:

```
#include <iostream>
#include <string>
using namespace std;

#include "c-libsocio.h"
#include “c-rivista.h”                                                /*1*/
#include "c-libreria4.h"
...
main(){
  biblioteca::libreria<biblioteca::libSocio> lib1;                    /*2*/
  biblioteca::libreria<biblioteca::rivista>  lib2;                    /*3*/
...
}
```

Nell'ipotesi che il file `c-rivista.h` contiene la definizione della nuova classe, la 1 permette l'uso della classe.

Nella 2 si definisce un oggetto `lib1` che è una libreria di oggetti di tipo `libSocio`, laddove nella 3, invece, gli oggetti sono di tipo `rivista`. Per il resto i metodi definiti in `libreria` si applicheranno agli oggetti di un tipo o a quelli di un altro, a seconda se si invierà il messaggio a `lib1` o a `lib2`.

6.7 Interfaccia e implementazione dei metodi di una classe

Finora tutta la definizione della classe è stata inserita in un unico file. In generale, invece, si preferisce scindere l'interfaccia della classe (quello che l'utilizzatore ha necessità di conoscere) dall'implementazione dei metodi della classe. È più opportuno quindi organizzare i sorgenti, per esempio nel caso della classe `libreria` e del programma di gestione di prestiti, nel modo seguente:

```
// file c-libreria4.h con interfaccia della classe ----------------+
                                                                   |
#ifndef C_LIBRERIA4                                                |
#define C_LIBRERIA4                                                |
                                                                   |
```

```
#include <vector>
using namespace std;

namespace biblioteca{
  template <class tipo>
  class libreria{
  public:
    ...
  private:
    ...
  };
}
#endif

// fine c-libreria4.h ------------------------------------------+

// file c-libreria4.cpp con implementazione metodi della classe ---+

#include “c-libreria4.h”
namespace biblioteca{
  ...
  template <class tipo>
  int libreria<tipo>::aggiungi(tipo lib){
  ...
  template <class tipo>
  bool libreria<tipo>::estrai(int quale,tipo& lib){
  ...
  template <class tipo>
  bool libreria<tipo>::aggiorna(int quale,tipo lib){
  ...
}

// fine c-libreria4.cpp ----------------------------------------+

// file prestiti4.cpp con gestione prestiti ----------------------+

...
#include “c-libreria4.h”
...

// fine prestiti4.cpp ------------------------------------------+
```

Occorre, infine, compilare i due file `c-libreria4.cpp` e `prestiti4.cpp` e linkarli assieme affinché tutto funzioni.

7 Dati su memorie di massa

7.1 Input/Output astratto

In ragione della volatilità della memoria centrale è comune l'esigenza di conservare dati su memorie permanenti e di poter avere la possibilità di rileggerli in futuro. Il sistema di I/O fornisce il concetto astratto di canale (lo ***stream***). Con tale termine si intende un dispositivo logico indipendente dalla periferica fisica: chi scrive il programma si dovrà occupare dei dati che transitano per il canale prescindendo dalle specifiche del dispositivo fisico che sta usando (un lettore di dischi magnetici, la tastiera, il video). Il termine ***file*** si riferisce invece ad una astrazione che è applicata a qualsiasi tipo di dispositivo fisico. In poche parole, si potrebbe affermare che il file rappresenta il modo attraverso il quale l'utilizzatore vede sistemati i dati sul dispositivo di I/O, e che il canale, o flusso, è il modo con cui i dati sono accessibili per l'utilizzazione.

L'associazione di canali alla tastiera per l'input e al video per l'output è curata in automatico dal sistema operativo per consentire il dialogo con il sistema. La tastiera e il video sono cioè le *periferiche di default*: il sistema è già connesso con esse. Per quanto riguarda invece le comunicazioni con altre periferiche è necessario esplicitare l'associazione di canali per tali comunicazioni.

7.2 Esempi di gestione di file di testo su dischi: i file CSV

Come esempi di applicazione della manipolazione di dati su memorie di massa, viene proposto un programma per la creazione di un file di tipo CSV, e un programma che si occupa della lettura dei dati scritti.

I file CSV (Comma Separated Values) sono file, di tipo testo, in cui in ogni riga c'è una registrazione, per esempio un libro. I dati di tipo stringa (titolo, autore, editore) sono racchiusi fra due caratteri *doppio apice*, i dati numerici (prezzo) non sono racchiusi fra due caratteri *doppio apice*. I vari dati del singolo libro sono separati dal carattere *virgola*, da cui il nome del tipo di file. Tutti i programmi che si occupano di gestire dati (programmi di database, tabelloni elettronici) prevedono una esportazione dei dati in file formato CSV.

Il primo programma proposto crea su disco un file che contiene i dati, dei libri, immessi da tastiera:

```
#include <iostream>
#include <string>
#include <fstream>                                                    /*1*/
using namespace std;

namespace biblioteca{
  struct libro {
    string titolo;
    string autore;
    string editore;
    float prezzo;
  };
}

main(){
  ofstream out;                                                       /*2*/
  biblioteca::libro libtemp;
  bool continua=true;
```

```
  out.open("libri.txt",ios::app);                                    /*3*/
  while(continua){
    cout << "Titolo :";
    getline(cin,libtemp.titolo);                                     /*4*/
    if(libtemp.titolo=="")                                           /*5*/
      break;
    out << "\"" << libtemp.titolo << "\"" << ",";                    /*6*/
    cout << "Autore :";
    getline(cin,libtemp.autore);                                     /*4*/
    out <<  "\"" << libtemp.autore << "\"" << ",";                   /*6*/
    cout << "Editore :";
    getline(cin,libtemp.editore);                                    /*4*/
    out <<  "\"" << libtemp.editore << "\"" << ",";                  /*6*/
    cout << "Prezzo :";
    cin >> libtemp.prezzo;                                           /*4*/
    cin.ignore();
    out << libtemp.prezzo << endl;                                   /*6*/
  }
}
```

L'inclusione `1` permette l'utilizzo degli oggetti che permettono la gestione dei flussi su memorie di massa.

Nella `2` si dichiara una variabile (il nome, ovviamente, è a scelta del programmatore) di tipo `ofstream` (flusso di output), laddove, per la `3`, si associa a detta variabile il file su disco `libri.txt`. Il flusso è aperto in modalità append (`ios::app`): se il file esiste, i nuovi inserimenti vengono aggiunti in coda, se non esiste, viene creato. Se invece il flusso è aperto in modalità output (`ios::out`), il file, anche se esistente, viene rigenerato e i dati eventualmente presenti sono persi.

Le `4` permettono di inserire i dati da tastiera. L'inserimento termina (`5`) quando si preme *Invio* a vuoto, in risposta alla richiesta di input del titolo del libro.

Con le `6` i dati vengono inviati al file attraverso il flusso `out`. L'utilizzo è identico a quello del flusso `cout` che, invece, permette l'invio al video. La differenza è data sostanzialmente dalla `3`: il concetto astratto di flusso permette di trattare, allo stesso modo, entità diverse (tastiera,video, file su disco). I singoli dati sono mandati al flusso, ognuno racchiuso da una coppia di caratteri *doppio apice* e separati da *virgole*. L'ultimo (il prezzo) è concluso dal carattere di fine linea (`endl`).

Anche il programma per la lettura dei dati dal file non si differenzia in maniera sostanziale da un qualsiasi programma che legge dati da un flusso legato alla tastiera:

```
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>                                                   /*1*/
using namespace std;

namespace biblioteca{
  struct libro {
    string titolo;
    string autore;
    string editore;
    float prezzo;
  };
}
```

```
main(){
  ifstream in;                                                    /*2*/
  biblioteca::libro libtemp;
  string riga,temp;
  int inizio,fine,lcampo;

  in.open("libri.txt",ios::in);                                   /*3*/
  while(getline(in,riga)){                                        /*4*/
    inizio=1;

    // estrae titolo

    fine=riga.find(',',inizio);                                   /*5*/
    lcampo=(fine-inizio)-1;                                       /*6*/
    libtemp.titolo = riga.substr(inizio,lcampo);                  /*7*/

    // estrae autore

    inizio=fine+2;
    fine=riga.find(',',inizio);                                   /*5*/
    lcampo=(fine-inizio)-1;                                       /*6*/
    libtemp.autore = riga.substr(inizio,lcampo);                  /*7*/

    // estrae editore

    inizio=fine+2;
    fine=riga.find(',',inizio);                                   /*5*/
    lcampo=(fine-inizio)-1;                                       /*6*/
    libtemp.editore = riga.substr(inizio,lcampo);                 /*7*/

    // estrae prezzo

    inizio=fine+1;
    fine=riga.length()+1;
    lcampo=fine-inizio;                                           /*6*/
    temp=riga.substr(inizio,lcampo);                              /*7*/
    istringstream is(temp);                                       /*8*/
    is >> libtemp.prezzo;                                         /*9*/

    // mostra i risultati

    cout << libtemp.titolo << " - " << libtemp.autore << " - "
         << libtemp.editore << " - " << libtemp.prezzo << endl;
  }
}
```

La `1` fornisce una libreria che permette di trattare una stringa come un flusso. Nel programma c'è un esempio di applicazione di detta libreria.

Nella `2` viene dichiarata una variabile che rappresenterà il flusso associato al file `libri.txt` (`3`) aperto in modalità input (`ios::in`).

Il controllo in `4` stabilisce che il ciclo vale fin quando arriva, dal flusso, una riga di testo. La funzione per l'input è sempre la `getline`, ma con la specifica del flusso `in` al posto di `cin`.

L'algoritmo per l'estrazione dei singoli dati dalla stringa, è simile all'algoritmo, esaminato in precedenza, sull'estrazione delle parole da una stringa di testo. In questo caso i vari campi sono suddivisi dalla virgola che, quindi, rappresenta il carattere da cercare (`5`). Successivamente vengono

calcolate (6) le lunghezze delle stringhe dei singoli campi, che vengono estratte dalle 7. Anche l'ultimo campo, il prezzo del libro, viene estratto come stringa.

Nella 8 viene dichiarato un flusso, in input, associato ad una stringa che viene passata al flusso come parametro. In questo modo, si può operare con il flusso in come se, per esempio, fosse il flusso associato alla tastiera, e con la 9, lo si dirige nella variabile desiderata.