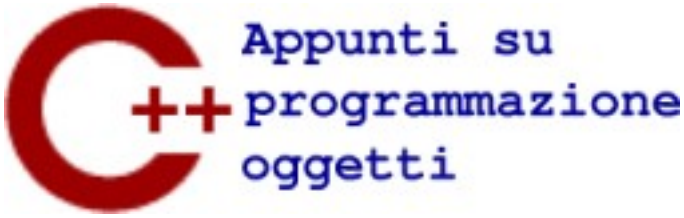


(2015.10)

Indice

0 Guida all'uso: avvertenza importante

Questi appunti sono rivolti a chi non conosce la programmazione e cerca un aiuto per imparare a programmare in C++. Negli appunti vengono esposte le istruzioni del linguaggio in un sottoinsieme significativo che consente, una volta acquisito, di scrivere programmi autonomamente. Appunto perché rivolti a principianti, per questi è importante non solo conoscere le istruzioni ma, sopratutto, sapere *come le istruzioni si mettono assieme* per creare un programma. A tale scopo sono previsti paragrafi che, in punti significativi, spiegano come utilizzare quanto esposto, in precedenza, per creare un programma attraverso un procedimento step-by-step. I paragrafi sono identificabili dall'esplicitazione di tale intenzione nel titolo e sono paragrafi che vanno seguiti e studiati con particolare attenzione magari, dopo aver assimilato la metodologia, applicando il procedimento descritto alla risoluzione di problemi diversi rispetto a quelli presentati come esempio.

Un altro paragrafo particolarmente importante e che si vuole segnalare è quello in cui si parla delle regole di scrittura. Scrivere bene un programma è una competenza fondamentale e non solo perché come diceva qualcuno *i programmi si scrivono principalmente per i programmatori e, solo secondariamente, per essere eseguiti da un computer*; infatti la vita di un programma è fatta principalmente di manutenzione che non può essere svolta in modo efficiente e produttivo se l'algoritmo applicato è poco comprensibile. Il programmatore deve poter capire immediatamente, al limite senza nemmeno leggerlo e ad una occhiata generale, cosa fa il programma e dove. Sforzarsi a rispettare regole precise di scrittura serve a rendere leggibile il programma e aiuta ad avere le idee più chiare su come impostare l'algoritmo risolutivo.

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. La descrizione di 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 la descrizione di 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 questo senso, al di là di una definizione rigorosa, si può definire un 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 pervenire 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 Rappresentazione di algoritmi, istruzioni elementari

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. L'algoritmo evidenzia le azioni che devono essere effettuate sui dati contenuti nelle variabili, qualunque essi siano. È come, per esempio, quando si esprime una proprietà matematica:

```
a + a = 2a
```

Cioè: se si somma un valore qualsiasi con sé stesso si ottiene il doppio del valore stesso.

Quando un esecutore esegue le istruzioni elencate in un algoritmo fornisce un valore per ogni variabile e l'algoritmo produrrà un risultato in relazione alla esecuzione delle istruzioni e ai valori inseriti. Se nell'esempio precedente si assegna il valore 3 alla variabile `a` l'esecuzione dell'algoritmo diventa:

```
3 + 3 = 6
```

Ogni variabile è identificata da un nome che permette di distinguerla dalle altre. In linea teorica il nome potrebbe essere qualsiasi ma per ragioni di chiarezza, di opportunità legate alle regole dei linguaggi di programmazione è bene rispettare alcune regole:

1. nel nome possono essere usati caratteri (dell'alfabeto inglese e, quindi, niente lettere accentate presenti per esempio nella lingua italiana), cifre numeriche e il carattere di sottolineatura (_). Non possono essere usati né spazi né segni di punteggiatura (hanno significati particolari nei linguaggi di programmazione). Ricordare che alcuni linguaggi di programmazione, per esempio il C++ di cui si tratta in questi appunti, distinguono fra maiuscole e minuscole e che, in questi casi, è convenzione usare lettere minuscole e riservare le maiuscole per distinguere le parole se si vuole attribuire il nome ad una variabile in modo che possa essere composto da più parole. Es.: `AltezzaTriangolo`
2. il nome deve essere quanto più possibile esplicativo e, contemporaneamente, il più breve possibile. Es.: potrebbe andare bene `AltTriang` ma va un po' meno bene `a` (poco comprensibile)

A prescindere dal linguaggio di programmazione che tradurrà, perché possa essere eseguito da un computer, un algoritmo, nei tempi sono stati adottati diversi sistemi per rappresentare gli algoritmi in modo che sia semplice comprenderne la logica. In questi appunti si esporranno in parallelo due sistemi: i **diagrammi di Nassi-Schneiderman** (N-S) e la **pseudocodifica** o linguaggio di progetto (L.P.).

Il primo è un sistema grafico che evidenzia chiaramente, a colpo d'occhio, la struttura complessiva dell'algoritmo ed è possibile verificare la correttezza dell'algoritmo utilizzando un programma (vedere riferimenti bibliografici per maggiori informazioni) che permette di esaminare l'effetto passo-passo delle istruzioni che vengono eseguite. È uno strumento, quindi, che fornisce un valido aiuto quando si deve imparare a comporre i primi algoritmi e verificarne la correttezza.

La pseudocodifica è più adatta per la gestione del passaggio algoritmo-programma. Utile sempre quando si ha necessità di *fissare le idee* in fasi delicate della progettazione di un programma.

Le istruzioni elementari che possono essere usate per comporre un algoritmo sono:

- 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.

  ```
  alfa <- beta+gamma*2
  ```

  L'espressione aritmetica viene valutata e il valore risultante viene inserito nella variabile ricevente (nell'esempio `alfa`). Nell'espressione algebrica possono essere usati gli operatori aritmetici `+` (somma), `-` (sottrazione), `*` (prodotto), `/` (divisione). La priorità degli operatori è

quella dell'aritmetica e può essere modificata utilizzando le parentesi tonde.

```
alfa <- ((beta+gamma)*2+delta)/10
```

Ogni nuova assegnazione alla stessa variabile distrugge il valore conservato in precedenza per fare posto al nuovo.

- L'**istruzione di input** fa in modo che l'algoritmo, durante la sua esecuzione, possa ricevere 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.

```
leggi omega
```

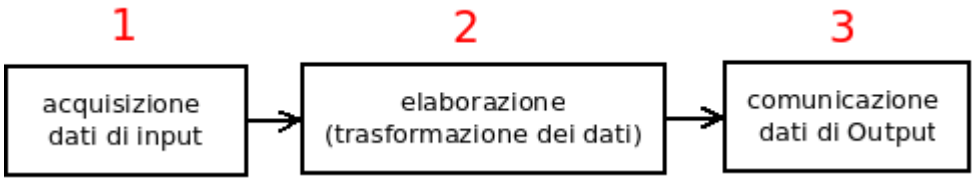
- L'**istruzione di output** fa in modo che l'algoritmo comunichi all'esterno i risultati della propria elaborazione. Nel caso di un elaboratore viene inviato su una unità di output (per esempio il video) il valore contenuto in una determinata variabile o una sequenza di caratteri (una stringa) da stampare così come è.

```
scrivi delta
scrivi "buon giorno"
```

1.3 Le strutture di controllo

Un computer permette di specificare tutte le azioni che devono essere svolte per raggiungere un certo obiettivo e avviare l'esecuzione delle azioni senza altro intervento manuale. Si possono quindi descrivere tutte le azioni da svolgere in un blocco (algoritmo) e sottomettere per l'esecuzione tale blocco, opportunamente tradotto, a un computer.

In generale si può affermare che, per svolgere una qualsiasi elaborazione, è necessario *attraversare* tre fasi:



È necessario introdurre, oltre alle istruzioni elementari, delle leggi di composizione delle 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**. L'algoritmo specifica *cosa fare* in generale. Durante l'esecuzione verranno forniti i dati e l'algoritmo produrrà dei risultati di conseguenza ai dati inseriti e alle elaborazioni specificate.

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:** 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.

N-S

L.P.

```
CalcoloAreaRettangolo

INIZIO
  leggi base
  leggi altezza

  area <- base*altezza

  scrivi area
FINE
```

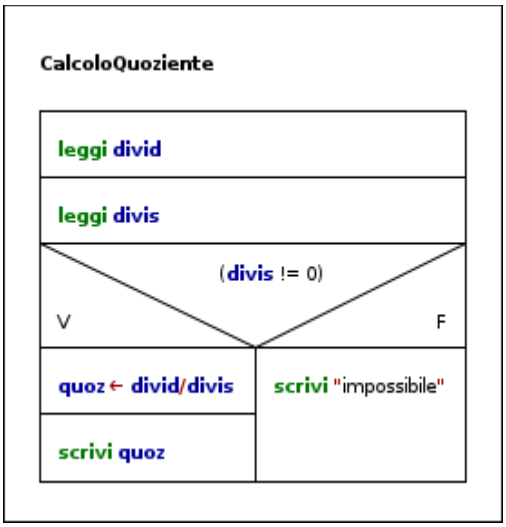
Nel diagramma N-S ogni singola istruzione viene inserita all'interno di un rettangolo (simbolo dell'istruzione generica) e la pila, dall'alto in basso, specifica l'ordine con cui le istruzioni vanno eseguite.

Anche nel L.P. l'ordine di esecuzione è dall'alto in basso e le istruzioni sono inserite dentro il blocco `INIZIO-FINE`.

L'istruzione del calcolo dell'area ha senso solo dopo l'acquisizione del valore delle dimensioni del rettangolo. In questo senso l'effetto prodotto dall'istruzione dipende dal contesto in cui opera (lo **stato del sistema**, inteso come insieme delle variabili e dei valori in esse contenuti)

- **la selezione:** questa struttura permette di scegliere tra due alternative la sequenza di esecuzione. È la struttura che permette, per esempio, di risolvere in modo completo il problema del calcolo del quoziente fra due numeri. Se il divisore è nullo la divisione non è definita e, al momento della stesura dell'algoritmo, si opera con variabili: i valori verranno specificati al momento dell'esecuzione e non si può prevedere ora se esiste la possibilità di calcolare realmente il quoziente.

N-S



L.P.

```
CalcoloQuoziente

INIZIO
  leggi divid
  leggi divis

  if (divis != 0)
    quoz <- divid/divis
    scrivi quoz
  else
    scrivi “impossibile”
  end-if
FINE
```

Dopo la sequenza che specifica l'acquisizione dei valori da associare alle variabili `divid` e

`divis` le elaborazioni successive non possono avere un unico svolgimento, dipendendo questo dal valore acquisito nella variabile `divis`.

La condizione espressa nella struttura `if` permette la scelta, in relazione al valore di verità o falsità della condizione specificata, della elaborazione da svolgere. La sequenza contenuta nella parte `else` (F) potrebbe mancare se si volesse soltanto un risultato laddove possibile: in tale caso se la condizione risultasse non verificata, non si effettuerebbe alcuna elaborazione. L'esecuzione di questo algoritmo porterà 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 specificare i casi possibili, elencare le istruzioni da eseguire e fornire gli elementi (la condizione da verificare) per scegliere la corretta sequenza di elaborazione.

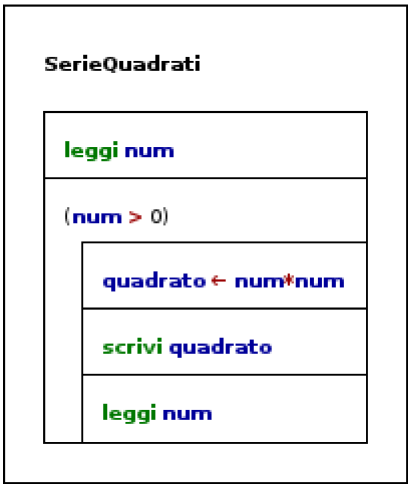
La `end-if` nel L.P. indica il punto dove termina la differenza di elaborazione e l'algoritmo può proseguire con le prossime istruzioni che, nel caso di esempio, non esistono. L'algoritmo dell'esempio prevede una sequenza composta dalle due istruzioni di input e dalla selezione.

In N-S si può notare visivamente che le due strade sono alternative. In L.P. tale caratteristica è evidenziata dal *rientro* delle istruzioni rispetto al bordo in cui è allineata l'istruzione precedente.

Nella condizione da testare, fra le parentesi, possono essere utilizzati gli operatori di confronto: `==` (uguale), `!=` (diverso), `<` e `<=` (minore, minore o uguale), `>` e `>=` (maggiore, maggiore o uguale). Il confronto può essere fatto fra: due variabili, una variabile e un valore, una variabile e il risultato di una espressione algebrica che produce un valore.

- **l'iterazione:** la struttura iterativa permette di ripetere più volte la stessa sequenza di istruzioni mentre continua a essere verificata una determinata condizione (ciclo WHILE). 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 stampa su video 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):

N-S



L.P.

```
SerieQuadrati

INIZIO
  leggi num

  while(num > 0)
    quadrato <- num*num
    scrivi quadrato

    leggi num
  end-while

FINE
```

Nel ***corpo*** della struttura iterativa (la parte compresa fra `while` e `end-while`) 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 istruzioni siano sempre le stesse, cambiano perché cambiano i valori). La condizione `num>0` viene chiamata ***condizione di controllo del ciclo*** e specifica i valori per cui l'elaborazione ha senso (il valore introdotto da input è 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. Se il valore introdotto nella variabile `num` è nullo o negativo il ciclo termina e su tale numero non viene effettuata alcuna elaborazione. Il valore introdotto per far terminare il ciclo è talvolta chiamato *valore sentinella*.

Le regole per la composizione della condizione di controllo del ciclo coincidono con quelle esaminate in precedenza per la condizione di una selezione.

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
while elementi non finiti
  Elabora elemento
  Considera prossimo elemento
end-while
```

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. Ogni istruzione, laddove prevista, può essere una qualunque delle strutture di controllo. Quando si parla di istruzione non si intende quindi, necessariamente, una singola istruzione ma una struttura di controllo che può contenere, al limite, una sola istruzione.

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 la merce che intende acquistare. 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) in cui il valore contenuto viene aggiornato 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 totale dei prezzi dei prodotti che lui acquista)
2. si aggiornerà per ogni prodotto esaminato (ogni nuovo prezzo acquisito verrà cumulato ai precedenti)
3. terminato 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 quello già presente 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
```

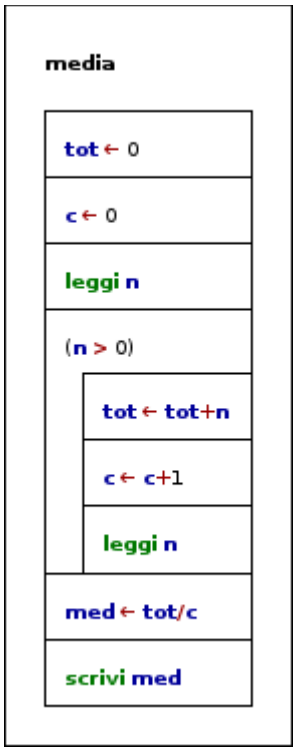
L'inizializzazione serve sia a dare senso all'istruzione di aggiornamento (cosa significherebbe la frase: *aggiorna il valore esistente con il nuovo valore* se non si fosse sicuri di aver assegnato un valore iniziale da cui far partire l'aggiornamento?), sia a fare in modo che l'accumulatore stesso contenga 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 sono stati tutti introdotti, 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 accumulatori e contatori, viene presentata la risoluzione del seguente problema: *data una sequenza di numeri positivi, se ne vuole calcolare la media aritmetica*.

N-S



L.P.

```
Media

INIZIO
  tot <- 0
  c <- 0

  leggi n

  while(n > 0)
    tot <- tot+n
    c <- c+1

    leggi n
  end-while

  med <- tot/c
  scrivi med
FINE
```

Coerentemente con quanto visto nello schema generale dell'utilizzo dei totalizzatori, `tot` 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. Le stesse considerazioni valgono per il contatore `c`, azzerato perché, per il momento non sono stati inseriti numeri. Saranno aggiornati, ambedue, all'interno del ciclo.

Dopo la conclusione del ciclo si possono utilizzare totalizzatore e contatore in quanto i contenuti sono coerenti con i motivi della loro esistenza: `tot` deve accumulare tutti i valori che devono essere contati da `c`, 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 (ciclo FOR). 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 delle istruzioni inserite dentro il ciclo ha termine.

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

```
Ricevi Quantità elementi da elaborare

for contatore da 1 to quantità elementi da elaborare
  Ricevi elemento
  elabora elemento
end-for
```

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

Ciclo while

```
Considera primo elemento

while elementi non finiti
  Elabora elemento
  Considera prossimo elemento
end-while
```

Ciclo for

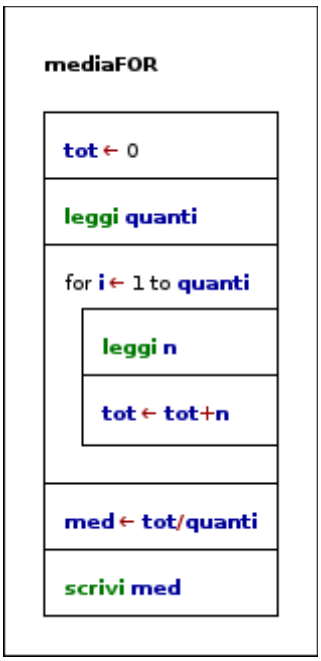
```
Ricevi Quantità elementi da elaborare

for contatore da 1 to quantità elementi
  Ricevi elemento
  elabora elemento
end-for
```

Nel ciclo WHILE 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 (la sentinella): 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 sentinella*). Il conteggio degli elementi elaborati, se necessario, può essere effettuato utilizzando un contatore.

Nel ciclo FOR 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 aggiuntivo.

Come esempio si riporta l'algoritmo del calcolo della media risolto, questa volta, per una quantità `quanti` di numeri.

N-S



L.P.

```
MediaFOR

INIZIO
  tot <-0

  leggi quanti

  for i<-1 to quanti
    leggi n

    tot <- tot+n
  end-for

  med <- tot/quanti
  scrivi med
FINE
```

Rispetto alla versione precedente (ciclo WHILE) qui: non è necessario acquisire l'elemento sentinella, si conosce il numero delle iterazioni da effettuare. Non è necessario utilizzare un contatore a parte utile nel calcolo della media (c'è già `quanti`).

1.6 Applicare le strutture di controllo: esempio step-by-step

Come applicazione della costruzione di un algoritmo con l'aiuto degli schemi esaminati precedentemente si consideri il seguente problema:

Si richiede di calcolare il totale di una fattura conoscendo i dati delle righe che ne fanno parte. Ogni riga è composta dalla quantità degli oggetti dello stesso tipo venduti e dal prezzo unitario dell'oggetto.

Considerando le tre parti in cui un algoritmo si può sempre pensare suddiviso: acquisizione dati in input, elaborazione dei dati, output dei risultati ottenuti, un approccio alla stesura dell'algoritmo risolutivo del problema proposto potrebbe comprendere i seguenti passaggi:

- ***Stabilire chiaramente quali sono i dati di input e i dati di output***. Il problema proposto richiede il calcolo del totale di una fattura (l'output) conoscendo i dati delle righe (input). Ogni riga è composta da quantità oggetti venduti e prezzo unitario che sono i dati che identificano la riga stessa. Le variabili potrebbero essere: `qven` (quantità venduta), `pzun` (prezzo unitario) per gli input e `totfat` (totale fattura) per l'output.
- ***Riconoscere il tipo di algoritmo***. Dall'esame del problema proposto si deduce che si richiede una elaborazione ciclica: ci sono infatti un'insieme di elementi (le righe della fattura) su ognuno dei quali va applicata la stessa elaborazione (cosa fare per ogni riga?): il prodotto della quantità venduta e del prezzo unitario (il totale della riga, della vendita del singolo oggetto) e l'accumulo del totale così ottenuto assieme ai totali delle altre righe. Il testo del problema non fornisce informazioni sulla quantità di righe da elaborare.

Ora si è in condizione di affermare, in base alle caratteristiche che deve avere, che l'algoritmo risolutivo è del tipo:

```
Considera primo elemento
while elementi non finiti
  Elabora elemento
  Considera prossimo elemento
end-while
```

Non è infatti un ciclo FOR perché non si conosce la quantità di righe di cui è composta la fattura.

Adesso è necessario adattare lo schema generale al caso in esame, rispondendo alle domande:

- *Chi sono gli elementi da elaborare?* Le righe della fattura. Ogni elemento da elaborare è una riga composta da `qven` e `pzun`.
- *Cosa vuol dire Considera elemento?* Ovvero: come si conoscono i dati di ogni riga? Acquisendoli dall'input.
- *Come si capisce che le righe della fattura sono finite?* Non c'è nella definizione del problema alcuna indicazione sull'elemento sentinella, ma in questo caso si può considerare una riga con quantità di oggetti venduti con valore nullo o negativo.
- *Che elaborazione effettuare sulla riga?* Come evidenziato in precedenza si tratta di calcolare il totale della riga. Totale che potrebbe essere conservato nella variabile `totr` (variabile temporanea che conserva un risultato intermedio della elaborazione).

Il totale della fattura deriva dall'accumulo dei totali di tutte le righe della fattura. Si tratta quindi di un totalizzatore:

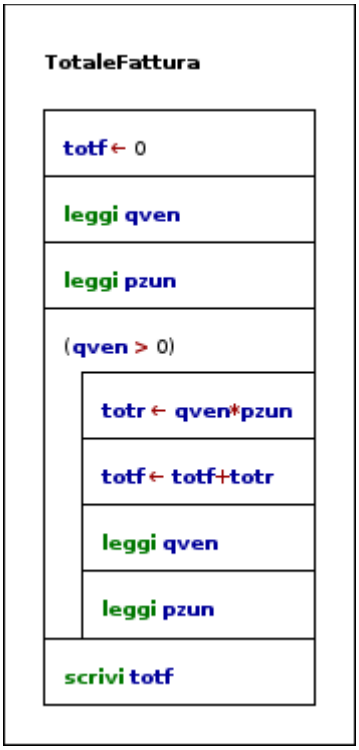
```
Inizializzazione totalizzatore
Inizio ciclo aggiornamento totalizzatore
  ...
  Aggiornamento totalizzatore
Fine ciclo
Uso del totalizzatore
```

Anche in questo caso è necessario rispondere ad alcune domande:

- *Cosa vuol dire Inizializza totalizzatore?* Il totale della fattura accumula i totali delle righe e quindi il suo valore iniziale sarà nullo.
- *Cosa vuol dire Aggiornamento totale fattura?* Aggiungere il totale della riga.
- *Cosa vuol dire Uso del totale della fattura?* Il problema richiede di conoscere in output tale totale.

A questo punto sono stati acquisiti tutti gli elementi che permettono, personalizzando gli schemi generali, di scrivere l'algoritmo risolutivo:

N-S



L.P.

```
TotaleFattura

INIZIO
  totf <- 0

  leggi qven
  leggi pzun
  while(qven > 0)
    totr <- qven*pzun
    totf <- totf+totr

    leggi qven
    leggi pzun
  end-while

  scrivi totf
FINE
```

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 indipendenti dalla macchina, possono essere eseguiti su qualsiasi elaboratore a patto che esista il corrispondente compilatore che ne permetta la traduzione. Si caratterizzano per essere orientati a specifiche aree applicative. Questi linguaggi vengono anche detti di terza generazione.

Per ultimi in ordine di tempo sono arrivati i linguaggi di 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.

L'obiettivo alla base della progettazione del linguaggio C era quello di creare un linguaggio potente e che potesse sfruttare le caratteristiche hardware: un linguaggio che avesse la potenza dell'assembly, che intendeva sostituire, ma allo stesso tempo la facilità d'uso di un linguaggio di terza generazione. Il linguaggio C è stato definito a volte linguaggio di seconda generazione e mezza per esprimere queste caratteristiche. Da allora ad oggi il C ha subito trasformazioni soprattutto in conseguenza della estensione *object-oriented* generando 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.

Questi appunti fanno riferimento alla standardizzazione del linguaggio conosciuta come C++98. Qualche esempio utilizza caratteristiche introdotte successivamente nella revisione C++11.

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 alto 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 a meno che non si intenda utilizzare un nome composto, per esempio, da due parole e in questi casi non essendo possibile separare le parole con il carattere spazio (che per il linguaggio è un separatore) è uso comune scrivere le due parole unite ma con la prima lettera maiuscola come per es. `TotaleFattura`.

Qualsiasi programma in C++ segue lo schema:

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

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

```
  return 0;
}
```

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 `int main()`.

Il significato esatto dell'ultima istruzione del programma (`return 0`) verrà chiarito meglio in seguito. Per il momento si può dire che se l'esecuzione del programma avviene senza interruzioni, e che quindi non ci sono errori, questa sarà l'ultima istruzione eseguita e si potrà essere sicuri che il programma è stato eseguito nel modo previsto.

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 indispensabili 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 Codifica di un programma con struttura sequenziale

Per poter presentare le prime istruzioni di un programma in C++, si propone la codifica di un semplice programma che, dopo aver richiesto le misure della base e dell'altezza, calcola l'area di un rettangolo. Partendo dall'algoritmo risolutivo scritto, come in precedenza in *pseudocodifica*, si

esamineranno le esigenze della codifica.

```
CalcoloAreaRettangolo
INIZIO
  leggi base
  leggi altezza

  area <- base*altezza

  scrivi area
FINE
```

Di seguito viene presentata una proposta di codifica:

```
// Calcolo area rettangolo

#include <iostream>
using namespace std;

int main()
{
  int base,altezza,area;                                          /*1*/

  // descrizione programma e input dimensioni rettangolo

  cout << "Calcolo AREA RETTANGOLO \n \n";                        /*2*/

  cout << "Valore base e altezza separati da spazio: ";
  cin >> base >> altezza;

  // elaborazione

  area = base*altezza;                                            /*3*/

  // output area rettangolo

  cout << "\nBase: " << base << " Altezza: " << altezza << endl;   /*4*/
  cout << "Area: " << area << endl;

  return 0;
}
```

Dopo la dichiarazione delle variabili in 1, il programma è sviluppato in 3 parti che prevedono:

- a cominciare dalla riga 2, la presentazione del programma e l'acquisizione dei dati di input. Rispetto all'algoritmo in pseudocodifica che evidenzia le istruzioni per la risoluzione del problema proposto, qui è presente una istruzione che consente di stampare su video una breve frase che spiega la funzione del programma stesso. Gli input, inoltre, sono preceduti da un *prompt*, una *stringa di invito* (breve frase che chiarisce all'operatore che esegue il programma, il senso delle richieste del programma). Si tenga presente che durante l'esecuzione del programma vengono visualizzate solo le frasi che sono oggetto di istruzioni di output: se non ci fosse una breve presentazione, l'operatore che esegue il programma non conoscerebbe l'elaborazione effettuata e, inoltre, non saprebbe come e cosa rispondere agli input richiesti. Le righe di commento sono uno strumento utile per il programmatore per riconoscere immediatamente le parti che compongono il programma e ricordare l'algoritmo utilizzato, ma sono inutili per l'operatore che esegue il programma e che vede solo gli output

previsti che quindi si devono prendere carico di rendere facilmente fruibili le elaborazioni.

Una unica istruzione provvede a ricevere l'input delle due variabili.

- dalla riga `3` comincia la parte delle elaborazioni che, nell'esempio proposto, è costituita da una sola istruzione. L'operazione di assegnamento è effettuata dall'operatore `=`. L'espressione algebrica a destra rispetta le regole già espresse in precedenza
- dalla riga `4` vengono presentati gli output del programma. Anche qui i valori di output sono preceduti da brevi frasi esplicative.

La forma grafica data al programma è, teoricamente e dal punto di vista del compilatore, 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. La codifica di un programma, qualunque sia il linguaggio di programmazione che si utilizza, è sempre criptica. D'altra parte nello sviluppo del software capita spesso di dover riprendere il codice per poter correggere errori, aggiungere o modificare funzionalità, ecc... Se la scrittura del codice fosse libera ci sarebbero problemi di comprensione che allungherebbero i tempi occorrenti per le modifiche. Per poter diminuire i costi di manutenzione del software, sono state progettate delle regole per la scrittura di un *buon* programma. In un paragrafo successivo si esporranno alcune regole minime, ormai universalmente accettate, da applicare alla scrittura di un programma.

2.5 Variabili ed assegnamenti

Nel programma di esempio subito dopo `int main()` è presente una riga per le dichiarazioni delle variabili intere necessarie:

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

La parola chiave `int` specifica che gli identificatori che la seguono si riferiscono a variabili che possono conservare numeri interi; 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. Ogni dichiarazione può riferirsi ad una o a più variabili dello stesso tipo.

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).

Quando una variabile viene dichiarata assume un ***valore casuale*** compreso nel range dei valori ammessi (una variabile di tipo `int`, per quanto osservato prima, fra -2147483648 e +2147483647).

Tra le operazioni permesse fra variabili di tipo `int` vi sono: la somma (`+`), la sottrazione (`-`), il prodotto (`*`), la divisione (`/`), l'operatore modulo (`%`) che restituisce il resto della divisione intera fra due `int`. È opportuno osservare che la divisione fra due `int` produce sempre un `int`: il valore del quoziente è troncato alla parte intera.

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

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

assegna alla variabile `area` il valore ottenuto dal calcolo specificato sostituendo il valore casuale contenuto nella variabile, introdotto nel momento della dichiarazione e privo di interesse. L'operatore `=` è quindi l'operatore di assegnamento (quello che in L.P. viene identificato da `<-`). Qualche volta potrà servire inserire in una variabile un valore costante e, in questo caso, l'istruzione potrebbe, per esempio, essere:

```
base = 3;
```

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 avessero riguardato un quadrato, si sarebbe potuto scrivere:

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

In questo caso, in sede di esecuzione, l'ordine delle operazioni procederebbe da destra verso sinistra: prima verrebbe assegnato il valore `5` alla variabile `altezza` e quindi, il risultato dell'assegnazione (cioè `5`), verrebbe assegnato alla variabile `base`.

Le dichiarazioni delle variabili dello stesso tipo possono essere scritte in sequenza separate da una virgola o anche dichiarate, con il proprio tipo, ognuna separatamente dalle altre:

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

Se le variabili non sono auto-documentate è opportuno aggiungere commenti per specificarne il significato:

```
int base, altezza,   // dimensioni del rettangolo
    area;            // area del rettangolo
```

La dichiarazione termina nella seconda riga ma le variabili sono distribuite in due righe in modo da aggiungere i commenti esplicativi.

Per quanto riguarda la lunghezza del nome di una variabile occorre tenere presente che soltanto i primi 32 caratteri sono significativi, anche se nelle versioni del C meno recenti questo limite scende a 8 caratteri. Non bisogna 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. Nel nome di una variabile non possono esserci inoltre caratteri particolari (spazi, segni di punteggiatura). In definitiva si può utilizzare una qualsiasi combinazione che preveda caratteri dell'alfabeto inglese minuscoli e maiuscoli, cifre numeriche ed eventualmente trattino-basso _.

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**.

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.

2.6 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 lo 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;

int main()
{
  cout << "Prima riga ";                                          /*1*/
  cout << "seguito della riga" << endl;                           /*2*/
  cout << "Nuova riga molto lunga"
          " la riga continua ancora su video"
          "\n questo viene stampato su una nuova riga"
       << endl;                                                   /*3*/

  return 0;
}
```

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
 questo viene stampato su una nuova riga
```

Nella `1` si incanala (per mezzo dell'*operatore di inserimento* `<<`) 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 poiché 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: `questo viene ...` verrà visualizzata la riga successiva a quella che visualizza `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

L'esecuzione della istruzione di output presente nel programma di esempio:

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

produrrebbe, nel caso fossero stati introdotti i valori `3` e `7` rispettivamente per base e altezza:

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

il primo output, come previsto dal codice di controllo presente verrà stampato una riga sotto l'ultimo output precedente.

2.7 Lo stream di input

Per rendere il programma del calcolo dell'area di un rettangolo più generale, si permette 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 >> altezza;
```

L'esecuzione di questa istruzione fa sì che il sistema attenda da parte dell'utente l'immissione di due numeri separati da uno spazio e che li vada a conservare nelle variabili specificate.

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 due dati e conservali nelle variabili specificate. Come già evidenziato in precedenza la definizione del canale di input si trova, come quella del canale di output, nella libreria `iostream`.

Essendo presente una stringa di invito prima della istruzione di input, quello che l'utente vedrà visualizzato in fase di esecuzione del programma sarà

```
Valore base e altezza separati da uno spazio: _
```

In questo istante si attende che nel canale di input sia disponibile un valore da estrarre. Se l'utente digita i valori `10` e `13` separati da uno spazio e seguiti da *Invio*

```
Valore base e altezza separati da uno spazio: 10 13
```

i dati verranno assegnati, rispettivamente, alle variabili `base` e `altezza`.

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

```
Calcolo AREA RETTANGOLO

Valore base e altezza separati da uno spazio: 10 13

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

Gli input, nell'esempio, sono raggruppati per rendere evidente alla lettura che si tratta della base e dell'altezza dello stesso rettangolo: i due dati introdotti riguardano le dimensioni di un rettangolo anche se, materialmente, si tratta di due valori.

Si potevano acquisire i due valori in maniera indipendente:

```
...
cout << “Valore base :”;
cin >> base;
cout << “Valore altezza :”;
cin >> altezza;
...
```

Se la codifica fosse stata fatta in questo altro modo, il sistema si sarebbe aspettato, di conseguenza ad ogni istruzione di estrazione dallo stream `cin`, un solo dato seguito da *Invio*. Il dato sarebbe stato conservato, di conseguenza, nella variabile specificata.

2.8 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. Data la semplicità ne viene proposta direttamente la codifica.

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

int main()
{
  const int limite=100;                                    /*1*/
```

```
  int i;

  // valore da elaborare

  cout << “Introdurre un valore intero “;
  cin >> i;

  // verifica valore

  if (i<limite)                                                    /*2*/
    cout << "numero introdotto minore di " << limite << endl;      /*3*/

  return 0;
}
```

Nel programma proposto è presente la costante `100` che potrebbe 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 evidenziato 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 fosse presente qualche istruzione che comportasse la modifica, per esempio nel caso proposto, di `limite`, il compilatore genererebbe 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` o `0` (falso) se `i` è maggiore o uguale a `100`: il risultato è dunque un numero intero che viene assegnato alla variabile `a`.

Una ulteriore conseguenza dei valori booleani è che chiedersi se il valore di `a` è diverso da zero è lo stesso che chiedersi 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 si sarebbe 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 nella condizione dell'istruzione `if` :

Operatore	***Esempio***	***Risultato***
`!`	`!a`	(NOT logico) `1` se a è `0`, 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>=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";
```

2.9 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 leggibilità 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.

2.10 L'operatore ?

L'operatore di assegnazione condizionata `?` ha la seguente sintassi:

```
espr1 ? espr2 : espr3
```

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

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

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.

2.11 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 (operatori doppi).

```
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'assegnazione di un nuovo valore, come ci si potrebbe attendere dall'utilizzo del simbolo `=`.

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

Nel seguito di questi appunti si converrà, come comune, 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.

2.12 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 attuale 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 attuale 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.

2.13 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;

int main()
{
  int vn,           // valore della sequenza da elaborare
      conta,somma;  // conteggio e somma numeri pari

  // 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;

  return 0;
}
```

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

Nella `2` 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 `%` per il calcolo del resto della divisione intera fra `vn` e `2` e viene controllato se tale resto è nullo in modo che, nelle `4`, si possano aggiornare il contatore e l'accumulatore.

2.14 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 di elaborazione dei numeri pari di una sequenza, potrebbe essere codificato:

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

int main()
{
  int vn,           // valore della sequenza da elaborare
      conta,somma;  // conteggio e somma numeri pari
  int qn,           // quantità numeri da elaborare
      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=0;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;

  return 0;
}
```

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.

Il programma per prima cosa assegna il valore `0` alla variabile `i` (la prima espressione del `for`), si controlla se il valore di `i` è inferiore 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.

Il ciclo viene eseguito `qn` volte e il contatore assume valori `0, 1, 2, ..., (qn-1)`. Si sarebbe potuto inizializzare il contatore ad `1` e terminare il ciclo con il controllo `i<=qn`, ma qui si è preferito adottare una convenzione, comune nelle applicazioni informatiche, di iniziare i conteggi dal valore `0`.

Questo modo di agire del ciclo `for` è quello comune a tutti i cicli di questo tipo (cicli a contatore) 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 (quella che utilizza il ciclo `while`), nella parte di codice comprendente il 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, mancando la terza espressione del `for`, viene solamente 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.

2.15 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, può essere 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, ancora una volta, 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 compresi altri blocchi `for`, `while` o `do...while`. I cicli possono cioè essere *annidati*.

2.16 Software Engineering: lo stile di scrittura

Un punto fondamentale nella scrittura di un programma è lo stile di scrittura del codice. Seguire delle regole per la scrittura del programma ne facilita la comprensione e la manutenzione e anche, in definitiva, la scrittura stessa. Il software necessita di manutenzione per la correzione degli errori riscontrati durante l'uso, per l'aggiunta di nuove funzionalità, per la modifica delle funzionalità e i costi di manutenzione possono essere più alti dei costi di sviluppo. Un buon stile di scrittura dei programmi fa parte integrante della documentazione e i programmi hanno necessità di essere compresi.

Gli esempi riportati in questi appunti sono scritti secondo alcune regole basilari, di uso ormai universale, in accordo con lo stile utilizzato dai Bell Laboratories.

- Un buon stile di scrittura utilizza ***righe vuote*** e ***commenti*** per rendere il codice più chiaro e comprensibile. Il commento spiega in breve il significato delle righe di codice che lo

seguono. Ogni nuovo blocco che svolge una nuova funzionalità o che elabora un dato intermedio va preceduto da un commento esplicativo. Ogni riga di commento è preceduta e seguita da una riga vuota per renderla più individuabile. Allo stesso modo ogni struttura di controllo (`if`, `while` o `for`).

- Una variabile ha un ***nome*** mnemonico che ricorda il suo uso in modo da agevolare la comprensibilità del codice.
- L'uso di una corretta ***indentazione*** (incolonnamento delle righe di codice) è di fondamentale importanza non solo per la comprensione del codice ma anche per evitare errori comuni. Negli esempi di questi appunti è utilizzata una indentazione di 2 caratteri. Per sintetizzare le regole per l'incolonnamento delle righe di codice:
  - ogni riga di codice è allineata, in generale, con la riga precedente
  - se nella riga precedente si trova una parantesi graffa aperta, la riga va indentata verso destra di tanti spazi quanti se ne sono scelti per la indentazione. Negli esempi 2 spazi
  - il margine di inizio della riga si sposta verso sinistra quando si deve *chiudere* un blocco: si deve inserire una parantesi graffa chiusa o si scrive l'istruzione successiva dopo l'unica istruzione compresa in una struttura condizionale o ciclo.
- L'uso di una corretta indentazione permette di evitare errori frequenti di dimenticanza della chiusura (parantesi graffa chiusa) di una struttura specie quando ci sono più strutture nidificate (una dentro l'altra).

Le convenzioni per l'uso delle parentesi graffe e l'indentazione delle righe di codice si possono così sintetizzare:

```
int main()
{
  // codice del programma
}
```

le parentesi che racchiudono il `main` occupano una riga a parte ciascuna.

```
while(...){
  // codice contenuto nel ciclo
}
// codice fuori dal ciclo
```

il blocco che racchiude il codice contenuto in un ciclo (`while` o `for`) comincia dalla riga dell'istruzione ciclica e si chiude con la parentesi che occupa da sola una riga.

```
if(...){
  // codice per condizione vera
}
else {
  // codice per condizione falsa
}
// altro codice
```

se nella struttura condizionale non è presente il blocco per la condizione falsa, si codifica come la struttura ciclica. Se presente il blocco per condizione falsa questo va trattato come se fosse una struttura a parte.

In sintesi la codifica di una struttura, qualora questa comprenda un blocco di codice, prevede la parentesi di apertura nella stessa riga del codice di inizio della struttura e la parentesi di chiusura da sola in una riga. Le righe che contengono le istruzioni che dipendono dalla struttura vanno indentate perché in tal modo, anche visivamente, ne diventa chiara la dipendenza.

2.17 Dall'algoritmo al codice: procedimento step-by-step

Per sintetizzare quanto esposto in questa parte si propone la codifica dell'algoritmo, già sviluppato, per il calcolo del totale di una fattura conoscendo quantità oggetti venduti e prezzo unitario di ogni oggetto presente in ogni riga che la compone. Ora, però, si vedrà il procedimento per ottenere il programma completo partendo dalla definizione del problema.

Come primo passo si può codificare la struttura generale del programma con la dichiarazione delle variabili:

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

int main()
{
  int qven, pzun;  // q. venduta, prezzo unitario singola vendita
  int totf;        // totale fattura

  return 0;
}
```

Per la dichiarazione delle variabili sono state utilizzate 2 righe in modo da distinguere, visivamente, le variabili in input e output.

Si aggiunge ora una descrizione del programma. Non si tratta di spiegare cosa fa il programma in maniera estesa (la lettura da uno schermo di un computer è difficoltosa e fa perdere tempo) ma di inserire frasi non ridondanti che siano più brevi possibili, eliminando parole che non ne aumentano la comprensibilità (per esempio eliminare gli articoli), ma anche che siano quanto più possibile chiarificatrici:

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

int main()
{
  int qven, pzun;  // q. venduta, prezzo unitario singola vendita
  int totf;        // totale fattura

  cout << “Calcolo totale fattura dai dati delle righe” << endl;

  return 0;
}
```

la codifica dell'algoritmo prevede righe di commento che evidenziano le varie fasi dell'elaborazione e spiegano in ***cosa*** consiste l'elaborazione, laddove il codice spiega ***come*** devono essere trattati i dati per ottenere quella elaborazione.

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

int main()
```

```
{
  int qven, pzun;  // q. venduta, prezzo unitario singola vendita
  int totf;        // totale fattura

  cout << "Calcolo totale fattura dai dati delle righe" << endl;

  // inizializzazione totale fattura

  // prima riga fattura

  // elaborazione righe

  while(qven > 0){

    // calcolo totale riga

    // aggiornamento totale fattura

    // altra riga

  };

  // output totale

  return 0;
}
```

La prima parte del programma prevede una sequenza comprendente l'inizializzazione del totale della fattura (totalizzatore) e l'input del primo elemento dell'elaborazione ciclica (la prima riga della fattura). Subito dopo è necessario inserire un ciclo per l'elaborazione ciclica delle righe. A questo punto, per come richiesto nella condizione di controllo del ciclo, è necessario stabilire la terminazione dell'elaborazione. Poiché nel testo del problema non è specificata la fine delle righe da prendere in considerazione si può assumere l'acquisizione di valori che fanno perdere di validità, per esempio, alla quantità di oggetti venduti: non ha senso infatti considerare per tale quantità valori negativi o il valore nullo. Un discorso simile si sarebbe potuto fare considerando invece il prezzo unitario. La sequenza delle operazioni da eseguire termina con l'output del totale della fattura.

A questo punto possono essere dettagliate le varie fasi: viene stabilito il ***come*** si effettua l'elaborazione prendendo in considerazione la codifica, ogni volta, del frammento di codice sottostante la singola riga di commento. Si isolano mentalmente le singole operazioni da effettuare:

```
// prima riga fattura

cout << "Dati riga" << endl
     << "quant. venduta e prezzo unitario separati da spazio" << endl
     << "(<=0 per finire) ";
cin  >> qven >> pzun;
```

Il codice completo è riportato di seguito dove si è aggiunta anche una variabile temporanea di lavoro necessaria per conservare il totale della singola riga della fattura necessario per l'aggiornamento del totale della fattura:

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

int main()
```

```
{
  int qven, pzun;  // q. venduta, prezzo unitario singola vendita
  int totf;        // totale fattura
  int totr;        // totale riga

  cout << "Calcolo totale fattura dai dati delle righe" << endl;

  // inizializzazione totale fattura

  totf = 0;

  // prima riga fattura

  cout << "Dati riga" << endl
       << "quant. venduta e prezzo unitario separati da spazio" << endl
       << "(<=0 per finire) ";
  cin  >> qven >> pzun;

  // elaborazione righe

  while(qven > 0){

    // calcolo totale riga

    totr = qven*pzun;

    // aggiornamento totale fattura

    totf += totr;

    // altra riga

    cout << "Dati riga" << endl
         << "quant. venduta e prezzo unitario separati da spazio" << endl
         << "(<=0 per finire) ";
    cin  >> qven >> pzun;
  };

  // output totale

  cout << "Totale fattura " << totf << endl;

  return 0;
}
```

Forse l'uso dei commenti potrebbe sembrare eccessivo ma ciò dipende solo dalla banalità, dal punto di vista della quantità di codice, dell'algoritmo codificato.

Leggendo solo le righe di commento si capisce immediatamente ***cosa fa*** il programma e ***dove***. Ciò risulta indispensabile per individuare i punti di intervento per la manutenzione.

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;

int 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;

  return 0;
}
```

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 questo codice perché è fondamentale, nella scrittura di un programma qualsiasi, 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 (in realtà sono 32 ma un bit è riservato al segno), può contenere solo numeri interi e i valori ammissibili variano tra i margini riportati (***range di rappresentatività***). Se si applica il modificatore `short`, l'occupazione di memoria cambia e di conseguenza anche il range 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, in quanto 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 ancora 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 registrazione dei valori simbolici `true` e `false`.

```
...
bool test;
```

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

3.2 Il cast

Quando si dichiarano variabili di tipo diverso è necessario fare attenzione a che non si mischino all'interno della stessa espressione algebrica, o assegnazione, valori e/o variabili di tipo diverso. Il compilatore segnala errore generico *type mismatch* (non corrispondenza di tipo) quando si tenta di forzare un valore di un tipo in una variabile dichiarata di tipo incompatibile con quel valore. In alcuni casi il compilatore effettua una forzatura, per esempio, se si prova ad inserire un valore `float` in una variabile di tipo `int`: chiaramente la parte decimale verrà troncata.

A volte, invece, 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***.

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

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

  uno = 1;
  due = 2;
  tre = uno/due;
  cout << tre << endl;

  return 0;
}
```

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*. Il programma, si scriverebbe così:

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

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

  uno = 1;
```

```
  due = 2;
  tre = static_cast<float> (uno)/due;
  cout << tre << endl;

  return 0;
}
```

In questo caso il risultato dell'esecuzione del programma coincide con l'attesa. Infatti il quoziente viene calcolato come `float` (il contenuto della variabile `uno` viene forzato `float`) e quindi, dopo, assegnato alla variabile `tre`.

In definitiva utilizzando l'operatore unario `static_cast<float>` applicato alla variabile `uno` si forza, nel calcolo dell'espressione, il suo valore ad essere di un tipo specifico (nell'esempio una divisione intera viene forzata a fornire un risultato di tipo virgola mobile essendo presente un valore di tale tipo).

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;

int main()
{
  char min,mai;
  const int scarto=32;                                              /*1*/

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

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

  return 0;
}
```

Nella riga con etichetta `1` è definita la costante `scarto`. Il valore `32` dipende dal fatto che, nel codice ASCII utilizzato nella rappresentazione dei caratteri nella memoria di un computer, 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 se il carattere immesso rientra nei limiti 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 numerica:

```
if (min>=97 && min<=122)
```

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).

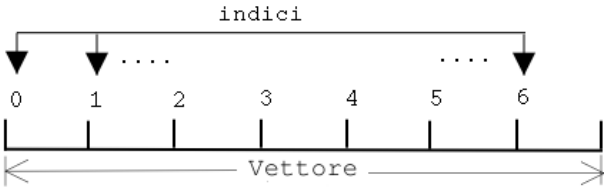
3.3 Introduzione a classi e oggetti: i vettori

Il **vettore** è la prima struttura di dati (*container*, contenitore) trattata in Informatica. Si parla di struttura dati quando ci si riferisce ad un insieme di dati organizzati secondo una determinata legge. Una struttura dati è un insieme dove, come nell'accezione matematica del termine, è definita una legge di appartenenza e sui cui elementi possono essere svolte determinate operazioni: creazione della struttura, inserimento/eliminazione di un elemento, ricerca di un elemento, selezione di un sottoinsieme.

Il vettore è la struttura dati più comunemente usata ed è quella che sta a fondamento di quasi tutte le altre.

Un vettore 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 al vettore come ad un insieme di cassetti numerati: un modo per poter accedere al contenuto di un cassetto è quello di specificare il posto in cui si trova l'insieme e il numero associato all'elemento (l'**indice**) che ne indica la posizione relativa rispetto al punto iniziale della struttura. Il primo elemento del vettore ha indice 0.



Nelle variabili semplici per accedere al valore contenuto in esse è necessario specificare il nome e, inoltre, una variabile che dovrà conservare un valore diverso avrà un nome diverso. Nel vettore esiste un nome che però stavolta identifica l'intera struttura.

Gli elementi del vettore vengono allocati in posizioni di memoria adiacenti.

C++, nel caso di strutture (container) 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 facilitandone l'elaborazione.

La definizione di vettore è contenuta nella libreria `vector` che quindi va inclusa nel codice 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).

Le differenze sostanziali fra la dichiarazione di una variabile di tipo elementare, per esempio una variabile di tipo `int`, e un oggetto, per esempio, della classe `vector` si possono sintetizzare:

- Una dichiarazione del tipo:

```
int a;
```

comporta, in ragione del tipo, la definizione di alcune operazioni (le quattro dell'aritmetica elementare e l'operazione modulo) che possono essere applicate al dato contenuto nella variabile che, di fatti, è un *contenitore per dati* su cui si vogliono eseguire quelle operazioni.

- Una dichiarazione del tipo:

```
vector<int> v;
```

dichiara un oggetto della classe `vector` che contiene numeri interi (il tipo è specificato fra le parentesi angolari `<>`) come elementi, ma `v` essendo un oggetto (una *istanza*) della classe `vector` ha tutti i comportamenti definiti nella classe. Non è possibile gestire direttamente gli elementi del vettore perché con un oggetto si interagisce utilizzando i ***metodi*** definiti per quella classe. Il metodo è una competenza, una funzionalità che mette a disposizione una determinata elaborazione sui dati del container. Il metodo si richiama per l'oggetto (si manda un messaggio all'oggetto) e questo reagirà secondo quanto previsto dal metodo stesso cioè effettuerà sugli elementi l'elaborazione descritta.

I metodi più comuni della classe `vector` che verranno utilizzati negli esempi di questi appunti sono sintetizzati nella tabella:

Metodo (classe vector)	***Comportamento***
`empty()`	Restituisce un valore booleano (*true* o *false*) indicante se il vettore è vuoto o contiene almeno un elemento
`size()`	Restituisce la quantità di elementi contenuti nel vettore
`begin()`	Restituisce la posizione in memoria del primo elemento del vettore
`end()`	Restituisce la posizione in memoria successiva all'ultimo elemento del vettore
`at()`	Permette di accedere all'elemento del vettore che si trova in una certa posizione. È necessario specificare l'indice come numero intero dentro le parentesi (parametro).
`push_back()`	Permette di inserire, in coda nel vettore, un nuovo elemento. L'elemento va specificato come parametro fra parentesi e deve essere del tipo previsto nella dichiarazione del vettore
`insert()`	Permette l'inserimento di un elemento in una posizione qualsiasi del vettore. Fra parentesi vanno specificati due variabili/valori (parametri) separati dalla virgola: il puntatore alla posizione di inserimento, l'elemento da inserire
`erase()`	Permette l'eliminazione dalla struttura di un elemento. Richiede come parametro un puntatore all'elemento.

Un metodo si applica all'oggetto specificandolo dopo il nome dell'oggetto e, separato da questo dall'operatore di appartenenza (il punto).

Per esempio:

```
...
vector<int> v;
...
cout << v.size();
```

```
...
```

la riga di codice produce la visualizzazione della quantità di elementi contenuti nel vettore `v`.

3.4 Iteratori

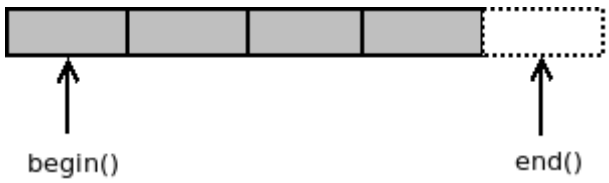
L'iteratore è un oggetto che consente l'accesso ai singoli elementi di un contenitore di oggetti, per esempio un vettore. Per mezzo di un iteratore si può effettuare la scansione lineare di un vettore, ovvero scorrere uno dopo l'altro tutti gli elementi contenuti in un vettore allo scopo, per esempio, di effettuare determinate elaborazioni. Un iteratore è un ***puntatore*** all'elemento (cioè fornisce una indicazione su dove si trova in memoria centrale l'elemento) e può essere aggiornato secondo le regole dell'aritmetica in modo da puntare all'elemento successivo della sequenza comunque siano composti gli elementi della sequenza.

Come elementare esempio di uso degli iteratori si propone un frammento di codice in cui, utilizzando una scansione sequenziale di un vettore di numeri interi, viene raddoppiato il valore di ciascun elemento contenuto in esso:

```
vector<int> numeri;                                          /*1*/
vector<int>::iterator it;                                    /*2*/
...
for(it=numeri.begin();it!=numeri.end();it++)                 /*3*/
  *it=*it*2;                                                 /*4*/
```

Nella `1` viene dichiarato un vettore di tipo `int`. Nella `2` si dichiara un iteratore in grado di scorrere gli elementi di un vettore del tipo specificato. Se ci fosse stato un vettore di tipo `float`, si sarebbe dovuto definire un iteratore opportuno: `vector<float>::iterator it`.

Il ciclo della `3` utilizza l'iteratore in accordo alle stesse regole dei numeri interi: viene assunto come valore iniziale dell'iteratore la posizione del primo elemento del vettore (`numeri.begin()`), si passa poi al prossimo elemento (`it++`) fino a quando il valore dell'iteratore rappresenti una posizione valida diversa, cioè, rispetto a quella restituita dal metodo `end()`. Passando al prossimo elemento, infatti, l'iteratore punta al successivo elemento fino a raggiungere il valore restituito dal metodo.



L'accesso all'elemento (`4`) è ottenuto ***deferenziando*** l'iteratore mediante l'operatore `*` (`*it` ovvero l'elemento puntato da `it`). In questo modo si può operare con l'elemento per esempio raddoppiandone il valore come nel codice proposto.

Si può accedere al singolo elemento anche utilizzando il metodo `at()`. Se per esempio si vuole raddoppiare l'elemento di posizione 4, l'operazione può essere codificata:

```
numeri.at(4)=numeri.at(4)*2;
```

In questo caso viene utilizzata come parametro del metodo la posizione relativa dell'elemento. È necessario che nella posizione specificata ci sia l'elemento. L'indice deve indicare un elemento esistente.

In conclusione si può sintetizzare:

- l'iteratore si usa quando si tratta di effettuare accesso sequenziale agli elementi dell'insieme
- il metodo `at()` si usa se si necessita di accesso diretto a singoli elementi.

Il metodo `at` prevede come parametro un numero intero (la posizione dell'elemento) e non può essere quindi utilizzato un iteratore. È possibile tuttavia ricavare la posizione relativa di un elemento all'interno del vettore, conoscendo il valore dell'iteratore che punta all'elemento:

```
vector<int> numeri;
vector<int>::iterator it;
int posRel;
...
for(it=numeri.begin();it!=numeri.end();it++){
  posRel = it-numeri.begin();                                    /*1*/
  cout << "posizione elemento " << posRel;
```

La posizione dell'elemento è ricavata (`1`) sottraendo dal valore attuale dell'iteratore (la posizione in memoria dell'elemento attuale) la posizione del primo elemento del vettore. La sottrazione fra due iteratori produce un numero intero perché, come si è già fatto notare, gli iteratori seguono le regole dell'aritmetica. Il risultato della differenza è un numero che rappresenta di quante posizioni è spostato l'elemento attuale rispetto alla posizione del primo elemento del vettore.

3.5 Elaborazioni di base sui vettori. Un esempio step by step

L'obiettivo di questo paragrafo è fornire frammenti di elaborazione comuni per i vettori. Tali frammenti sono da intendersi come i mattoncini elementari da usare, assemblando assieme quelli che servono, nello sviluppo di un generico programma che utilizza vettori.

- **Input di un vettore.** Si tratta di un frammento di codice che consente di popolare un vettore (contenente per esempio numeri interi positivi):

```
// Popolamento di un vettore

vector<int> numeri;  // vettore in cui inserire elementi
int temp;            // variabile temporanea per valori da inserire

// Valore in variabile temporanea

cin >> temp;
while(temp > 0){

  // Inserimento nel vettore

  numeri.push_back(temp);                                        /*1*/

  // Prossimo valore

  cin >> temp;
}
```

  L'inserimento viene effettuato richiamando per il vettore il metodo `push_back` (`1`) con il valore conservato in una variabile temporanea del tipo uguale al tipo degli elementi che dovrà contenere il vettore.

- **Output di un vettore.** Si tratta del frammento duale rispetto al precedente:

```
// Output di un vettore
```

```
vector<int>::iterator it;  // iteratore per scansione lineare vettore
...
cout << “Elementi del vettore” << endl;

for(it=numeri.begin();it!=numeri.end();it++)                          /*1*/
  cout << *it << “\t”;                                                /*2*/
cout << endl;
```

Viene effettuata una *scansione lineare* del vettore (1): si esaminano tutti gli elementi del vettore. A parte il fatto che qui si effettua l'output degli elementi contenuti nel vettore (2) accessibili deferenziando l'iteratore, il codice è quello della scansione lineare esaminato in precedenza.

- **Ricerca di un elemento in un vettore**. Si tratta di sapere se un elemento fa parte di un insieme. Si suppone che, in accordo alle proprietà di un insieme, gli elementi siano diversi fra di loro:

```
// Ricerca

vector<int>::iterator it,   // iteratore per scansione
                      pos;  // iteratore per elemento trovato       /*1*/
int cosa;  // valore da cercare nell'insieme
...
pos = numeri.end();                                                 /*2*/

// scansione lineare dell'insieme

for(it=numeri.begin();it!=numeri.end();it++){

  // verifica corrispondenza elemento attuale con valore cercato

  if(*it==cosa){
    pos = it;  // conserva puntatore all'elemento                   /*3*/
    break;     // forza l'uscita dal ciclo                          /*4*/
  }
}

// risultati ricerca

if(pos!=numeri.end())                                               /*5*/
  cout << "elemento trovato in posizione "
       << pos-numeri.begin() << endl;                               /*6*/
else
  cout << "elemento non trovato" << endl;
```

L'algoritmo proposto fa uso di un iteratore (1) per fornire informazioni sull'esito della ricerca. L'iteratore viene inizializzato (2) ad una posizione inesistente, il valore viene modificato (3) inserendone la posizione se durante la scansione del vettore si trova un elemento uguale al valore cercato. L'istruzione `break` della 4 forza l'uscita anticipata dal ciclo: è infatti inutile continuare nella scansione degli elementi successivi del vettore essendo già stata verificata la corrispondenza. Il risultato della ricerca può essere conosciuto effettuando un test sul valore dell'iteratore: se è stato modificato rispetto al valore iniziale (5) punterà ad un elemento del vettore la cui posizione relativa si può calcolare mediante la sottrazione 6.

- **Ricerca di un sottoinsieme (selezione)**. La selezione consiste nel ricopiare in un nuovo vettore tutti gli elementi del vettore iniziale che soddisfano ad una determinata condizione:

```
// Selezione numeri pari

vector<int> pari;
...

// Scansione lineare dell'insieme

for(it=numeri.begin();it!=numeri.end();it++){

  // verifica se elemento da inserire in selezione

  if(!(*it%2))                                                    /*1*/
    pari.push_back(*it);
}

// Verifica se insieme vuoto

if(pari.empty())                                                  /*2*/
  cout << “Non ci sono numeri pari” << endl;
```

  Il frammento di programma differisce da quello della ricerca solo per il fatto che qui la verifica della condizione (`1`) comporta l'inserimento dell'elemento nella selezione. Utilizzando il metodo `empty()` del nuovo vettore (`2`) si può stabilire se il sottoinsieme è vuoto.

Il prossimo programma è un esempio di come i frammenti presentati possono essere messi assieme per risolvere un problema con l'utilizzo di vettori. Poiché un vettore sostanzialmente è un insieme si potrebbe pensare alla costruzione dell'insieme intersezione di due insiemi i cui elementi sono, per esempio, interi.

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

int main()
{
  vector <int> insiemeA, insiemeB, intersezione;
  vector<int>::iterator it1;

  cout << "Intersezione fra due insiemi di interi" << endl;

  // Popolamento primo insieme                                    /*1*/

  // Popolamento secondo insieme                                  /*1*/

  // Scansione lineare primo insieme

  for(it1=insiemeA.begin();it1!=insiemeA.end();it1++){

    // cerca l'elemento nel secondo insieme

    // se presente inserire in insieme intersezione

  }
```

```
  // Output intersezione

  return 0;
}
```

Nelle 1 basta adattare il frammento di codice dell'inserimento di elementi in un vettore. Qui si suppone che l'utilizzatore del programma stia attento, poiché si tratta di insiemi, ad inserire valori distinti per gli elementi di ciascun insieme.

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

int main()
{
  vector <int> insiemeA, insiemeB, intersezione;
  vector<int>::iterator it1;
  int temp;     // valore temporaneo da inserire come elemento      /*1*/

  cout << "Intersezione fra due insiemi di interi" << endl;

  // Popolamento primo insieme                                      /*1*/

  cout << "primo insieme" << endl;
  cout << "Elemento (0 per finire) ";
  cin >> temp;
  while(temp){

    // Inserimento nel vettore

    insiemeA.push_back(temp);

    // Prossimo valore

    cout << "Elemento (0 per finire) ";
    cin >> temp;
  }

  // Popolamento secondo insieme                                    /*1*/

  cout << "secondo insieme" << endl;
  cout << "Elemento (0 per finire) ";
  cin >> temp;
  while(temp){

    // Inserimento nel vettore

    insiemeB.push_back(temp);

    // Prossimo valore

    cout << "Elemento (0 per finire) ";
    cin >> temp;
  }

  // Scansione lineare primo insieme

  for(it1=insiemeA.begin();it1!=insiemeA.end();it1++){
```

```
    // cerca l'elemento nel secondo insieme                          /*2*/

    // se presente inserire in insieme intersezione                  /*3*/

  }

  // Output intersezione

  return 0;
}
```

Dichiarata una nuova variabile per valore temporaneo si possono aggiungere i due cicli per popolare i due insiemi (1).

Per popolare l'insieme intersezione è necessario trovare gli elementi in comune fra i due insiemi. Il problema si può risolvere (2) cercando ogni elemento del primo insieme (per mezzo di una scansione che permette di esaminare, in sequenza, gli elementi uno per volta) nel secondo insieme. Si tratta di adattare e inserire il frammento di codice della ricerca esaminato in precedenza. Se l'elemento si trova vuol dire che è un elemento dell'insieme intersezione (3).

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

int main()
{
  vector <int> insiemeA, insiemeB, intersezione;
  vector<int>::iterator it1, it2,  // iter. per scansione vettori
                        pos;       // puntatore elemento comune
  int temp;     // valore temporaneo da inserire come elemento

  cout << "Intersezione fra due insiemi di interi" << endl;

  // Popolamento primo insieme

  cout << "primo insieme" << endl;
  cout << "Elemento (0 per finire) ";
  cin >> temp;
  while(temp){

    // Inserimento nel vettore

    insiemeA.push_back(temp);

    // Prossimo valore

    cout << "Elemento (0 per finire) ";
    cin >> temp;
  }

  // Popolamento secondo insieme

  cout << "secondo insieme" << endl;
  cout << "Elemento (0 per finire) ";
  cin >> temp;
  while(temp){
```

```
    // Inserimento nel vettore

    insiemeB.push_back(temp);

    // Prossimo valore

    cout << "Elemento (0 per finire) ";
    cin >> temp;
  }

  // Scansione lineare primo insieme

  for(it1=insiemeA.begin();it1!=insiemeA.end();it1++){

    // cerca l'elemento nel secondo insieme

    pos=insiemeB.end();                                          /*1*/
    for(it2=insiemeB.begin();it2!=insiemeB.end();it2++){
      if(*it2==*it1){    // verifica se elementi dei due vettori uguali
        pos=it2;
        break;
      }
    }

    // se presente inserire in insieme intersezione

    if(pos!=insiemeB.end())
      intersezione.push_back(*it1);
  }

  // Output intersezione                                         /*2*/

  if(intersezione.empty())                                       /*3*/
    cout << "Insieme intersezione vuoto" << endl;
  else {
    cout << "Insieme intersezione" << endl;
    for(it1=intersezione.begin();it1!=intersezione.end();it1++)
      cout << *it1 << "\t";
    cout << endl;
  }

  return 0;
}
```

Dalla `1` viene inserito il codice per la ricerca nel secondo vettore dell'elemento del primo (`*it1`).

Con l'inserimento del frammento dell'output del vettore che contiene l'insieme intersezione (`2`) il programma è completo. L'insieme intersezione potrebbe essere vuoto ed è quindi necessario controllare tale evenienza (`3`).

3.6 Utilizzo metodi della classe vector

Per mostrare le applicazioni di altri metodi della classe `vector`, viene proposto un programma che, acquisito un vettore, per esempio di numeri interi, ordinato in modo crescente e un numero intero, lo inserisca all'interno del vettore, nel posto che gli compete. Il programma permette anche, alla fine, l'eliminazione di un elemento, di cui si specifica la posizione, dalla struttura.

```
#include <iostream>
```

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

int main()
{
  vector<int> numeri;
  vector<int>::iterator it,   // iter per scansione vettore
                        pos;  // iter posizione inserimento
  const int qelem = 10;  // quantità elementi contenuti nel vettore
  int i,     // contatore ciclo
      temp;  // var. temporanea per inserimento in vettore
  int cosa,  // elemento da inserire
      p;     // pos relativa elem da eliminare

  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<qelem; i++){
    cout << "elemento " << i << " -->";
    cin  >> temp;

    // verifica ordinamento

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

  cout << "Elemento da inserire ";
  cin >> cosa;

  // Ricerca posizione

  pos=numeri.end();
  for(it=numeri.begin(); it!=numeri.end(); it++){
    if(cosa<(*it)){                                                  /*4*/
      pos = it;
      break;
    }
  }

  // Inserimento nella posizione trovata

  if(pos!=numeri.end())                                              /*5*/
    numeri.insert(pos,cosa);                                         /*6*/
  else
    numeri.push_back(cosa);                                          /*7*/

  // Vettore con nuovo elemento inserito

  cout << "\nNuovo vettore " << endl;
  for(it=numeri.begin(); it!=numeri.end(); it++)                     /*8*/
```

```
    cout << *it << "\t";
  cout << endl;

  // Eliminazione di un elemento dal vettore

  cout << "Da quale posizione togliere? ";
  cin >> p;                                                    /*9*/
  numeri.erase(numeri.begin()+p);                              /*10*/

  // Nuova visualizzazione

  cout << "\nNuovo vettore " << endl;
  for(it=numeri.begin(); it!=numeri.end(); it++)               /*8*/
    cout << *it << "\t";
  cout << endl;

  return 0;
}
```

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, per l'operatore OR (`||`), basta che la prima condizione sia verificata per non procedere oltre. Viene utilizzato il metodo `at` per accedere all'elemento precedente.

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. In generale non è una buona prassi modificare, all'interno di un ciclo `for`, la variabile che controlla il ciclo (può essere fonte di errori difficili da rintracciare) ma qui la variabile è utilizzata solo per visualizzare l'ordine di inserimento dell'elemento. Una soluzione migliore potrebbe essere utilizzare un ciclo `while`.

Per l'inserimento degli elementi nel vettore questa volta è utilizzato un ciclo `for` trattandosi di inserire 10 elementi. Per il resto il ciclo di input è identico al ciclo `while` trattato in precedenza.

Anche la ricerca della posizione di inserimento è formalmente identica a quella esaminata in esempi precedenti salvo il fatto (`4`) che qui si cerca il primo elemento del vettore che abbia valore maggiore dell'elemento da inserire. Il posto, nel vettore, che il nuovo elemento prenderà sarà appunto questo. Se è stato trovato un elemento del vettore maggiore del valore da inserire (`5`) si inserisce l'elemento in tale posizione (`6`). Il metodo `insert` richiede come primo parametro un iteratore alla posizione di inserimento che, in questo caso, sarà la posizione del primo elemento del vettore con valore maggiore. Se non esiste alcun elemento del vettore maggiore, il nuovo valore verrà accodato (`7`) a quelli già presenti.

Anche il metodo `erase` per l'eliminazione di un elemento dal vettore, richiede un iteratore che punti a tale elemento. Dall'input (`9`) si riceve la posizione relativa (lo ***scostamento***): basta (`10`) sommare tale scostamento al valore del puntatore al primo elemento del vettore.

I cicli delle `8` si occupano di visualizzare gli effetti, nel vettore, delle operazioni svolte.

3.7 La classe string

Le variabili di tipo `char` consentono di conservare un singolo carattere. Se si voglio conservare stringhe, 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.

La stringa può essere considerata come un vettore di `char` e, da questo punto di vista, un oggetto di questa classe dispone di tutti i metodi esposti in precedenza per un vettore generico come, ad esempio, cancellazione di caratteri al suo interno, possibilità di accesso ai singoli caratteri che ne fanno parte. Un oggetto di tipo `string` possiede inoltre metodi specifici utili se, appunto, il vettore non è un vettore generico ma una stringa:

Metodo (classe string)	***Comportamento***
`length()`	Restituisce un numero intero che rappresenta la quantità di caratteri contenuti nella stringa
`empty()`	Restituisce valore booleano indicante se la stringa è vuota (valore *true*) o contiene almeno un carattere (valore *false*)
`begin()`	Restituisce un iteratore alla posizione in memoria del primo carattere della stringa
`end()`	Restituisce un iteratore alla posizione successiva all'ultimo carattere della stringa
`at()`	Permette di accedere al carattere che si trova in una certa posizione. È necessario specificare l'indice come numero intero dentro le parentesi.
`insert()`	Permette l'inserimento di una stringa in una posizione qualsiasi di un altra stringa. Fra parentesi vanno specificati due parametri separati dalla virgola: la posizione di inserimento, la stringa da inserire
`erase()`	Permette l'eliminazione di una sottostringa da una stringa. Vanno specificati come parametri la posizione a partire dalla quale eliminare caratteri e la quantità dei caratteri da eliminare
`find()`	Restituisce un numero intero che indica la posizione in cui è stato trovato il carattere specificato. Se la ricerca ha esito negativo il metodo restituisce il valore -1. Il primo parametro, obbligatorio, indica il carattere da cercare all'interno della stringa. Il secondo parametro, opzionale, è un numero intero che indica la posizione all'interno della stringa a partire dalla quale iniziare la ricerca. Se il parametro non viene specificato, viene assunta per default la posizione iniziale della stringa (0)
`substr()`	Restituisce una sottostringa estratta. I parametri specificano la posizione a partire dalla quale estrarre caratteri e la quantità di caratteri da estrarre. I caratteri non vengono eliminati dalla stringa ma solo ricopiati nella nuova stringa

Le posizioni cui si fa riferimento nei metodi sono contate a partire da 0 (posizione del primo carattere della stringa).

Per agevolare alcune elaborazioni presenti anche in esempi di questi appunti è opportuno aggiungere, fra le funzionalità disponibili per gli oggetti `string`:

`stoi()` `stof() stod()`	**Funzioni** introdotte nella revisione C++11 per convertire una stringa e che restituiscono, rispettivamente, un intero, un float o un double. **Non sono metodi** e la stringa da convertire si inserisce come parametro. Si usano specificando direttamente il nome della funzione e il parametro.

Per oggetti della classe `string` è definito anche l'operatore `+` di concatenamento stringhe.

```
...
string s1, s2, s3;
s1 = “buon “;
s2 = “giorno”;
s3 = s1 + s2;
...
```

Dopo le operazioni indicate `s3` conterrà la stringa “`buon giorno`”.

3.8 Stringhe: esempi di utilizzo dei metodi della classe

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;

int main()
{
  string dove;                                                       /*2*/
  char cosa;      // carattere da cercare
  int volte,pos;  // ricorrenze, posizione carattere nella stringa
  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";
  getline(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 finché 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;

    return 0;
}
```

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

L'input di una stringa (`3`) poteva essere effettuato come per le variabili di tipo elementare utilizzando l'operatore di estrazione dal canale `cin`. Questo però va bene se nella stringa fornita in input non si inseriscono caratteri speciali come 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. Utilizzando la funzione `getline` può essere acquisita una stringa contenete anche caratteri speciali come lo spazio o i caratteri di punteggiatura fino (per default) al carattere *Invio* che chiude l'input della stringa. La funzione si utilizza specificando fra parentesi almeno due parametri: il canale da dove leggere e la variabile destinata a contenere i caratteri provenienti dal canale. Il terzo parametro, opzionale e non utilizzato negli esempi, permette di specificare il carattere che si intende come terminatore della stringa. Se non specificato tale carattere è il carattere *Invio*.

L'assegnazione di un valore ad una stringa viene effettuata per mezzo del solito operatore `=`, è solo necessario racchiudere la stringa fra doppi apici (`dove = "stringa di prova";`). 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.

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

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

int main()
{
  string dove,      // stringa da elaborare
         metti;     // parola da inserire
  int inizio,fine,  // posizioni spazi delimitanti parola da sostituire
      quanti;       // quanti caratteri eliminare

  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;

  return 0;
}
```

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 il metodo `insert` della `5`, richiamato per `dove`. Come parametri si specificano la posizione iniziale di inserimento 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.

Come ultimo esempio, che illustra l'uso del metodo per la conversione di una stringa in valore numerico, si mostra un programma che calcola la media aritmetica delle valutazioni che un allievo ha ottenuto in una materia che viene fornita come input. Le valutazioni di tutte le materie vengono fornite come stringhe contenenti materia e voto separati dalla virgola. Questo programma assumerà un interesse particolare per un tipo di elaborazione che verrà trattata più avanti (file CSV).

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

int main()
{
  string matinp,          // materia interessata
         matvoto, temp,   // valutazione ottenuta dall'allievo
         mat;             // solo materia
```

```
  int cont,pos;          // conta voti
  float vot,somma;       // voto numerico, somma voti
  float media;

  cout << "Calcola media voti di una materia, "
          "da stringhe con materia e voto" << endl;

  cout << "Quale materia? ";
  getline(cin,matinp);                                          /*1*/

  cont=0;
  somma=0.0;
  cout << "Materia e voto separati da virgola (invio per finire) ";
  getline(cin,matvoto);                                         /*2*/

  while(!(matvoto.empty())){
    temp=matvoto;                                               /*3*/
    pos=temp.find(',');                                         /*4*/

    // estrae materia e verifica se interessa

    mat=temp.substr(0,pos);                                     /*5*/
    temp=temp.erase(0,pos+1);                                   /*6*/
    if(mat==matinp){

      // trasforma voto da stringa a float ed elabora

      vot = stof(temp);                                         /*7*/
      cont++;
      somma += vot;
    }
    cout << "Materia e voto separati da virgola (invio per finire) ";
    getline(cin,matvoto);                                       /*2*/

  }

  if(cont){
    media = somma/cont;
    cout << "media dei voti nella materia " << media << endl;
  }
  else
    cout << "Non ci sono valutazioni per quella materia" << endl;

  return 0;
}
```

Nella 1 si acquisisce la materia di cui si vuole calcolare la media delle valutazioni.

Nelle 2 si acquisiscono le stringhe delle valutazioni finché la stringa stessa non risulti vuota. La stringa viene ricopiata in un'altra (3) per poterla elaborare senza che le modifiche comportino un mutamento di quello che è stato acquisito in input. Questa è una pratica comune: gli input non si modificano mai perché rappresentano, ai fini di qualsiasi tipo di esigenza, il punto di partenza della elaborazione. Se serve, come nel caso proposto, modificare il valore di un input, questo si ricopia in una nuova variabile.

Per ogni valutazione innanzi tutto si ricerca (4) la posizione della virgola separatrice, si ricopia (5) in una nuova stringa tutta la sottostringa che rappresenta la materia: la quantità di caratteri è lo

stesso numero della posizione della virgola (le posizioni si contano da 0) e si cancella (`6`) la parte estratta oltre che la virgola separatrice. Questa scelta è dettata dalla motivazione di rendere più semplice l'elaborazione della suddivisione delle parti logiche della stringa, principalmente quando le parti sono tante.

L'istruzione principale del programma è la `7`. Tolta la parte della materia, con la `6`, nella stringa `temp` è rimasto solo il voto ma non è immediatamente utilizzabile perché in formato stringa. È necessario trasformarlo nel tipo opportuno. Se era necessaria una conversione in valore di tipo `int` occorreva utilizzare `stoi` al posto di `stof`.

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

Se si vogliono limitare le istruzioni da eseguire a quelle specificate nella `case` che verifica il valore cercato, occorre inserire l'istruzione `break` che effettua un salto e 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;

int main()
{
  string espress;               // stringa con espressione algebrica
  string::iterator it;          // iter per scansione stringa       /*1*/
  int pargraf,parquad,partond;  // contatori vari tipi parentesi
```

```
    // acquisizione espressione come stringa

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

    // ricerca parentesi nella espressione

    pargraf=parquad=partond=0;

    for(it=espress.begin();it!=espress.end();it++) {                    /*2*/

      switch(*it) {                                                     /*3*/
      case '{':; case '}':                                              /*4*/
        pargraf++;
        break;                                                          /*5*/
      case '[':; case ']':                                              /*4*/
        parquad++;
        break;                                                          /*5*/
      case '(':; case ')':                                              /*4*/
        partond++;
        break;                                                          /*5*/
      }
    }

    // presentazione dei risultati

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

    return 0;
  }
```

Nella `1` si dichiara un iteratore per la scansione sequenziale della stringa. La dichiarazione e il funzionamento di tale iteratore è identico a quello utilizzato in precedenza per la scansione di un vettore.

Il ciclo che comincia da `2` si occupa di effettuare la *scansione lineare* della stringa: verranno esaminati tutti i caratteri che la compongono. La stringa è trattata come un vettore di caratteri.

Nella riga con etichetta `3` viene specificata l'espressione da valutare: `*it` cioè il carattere dell'espressione algebrica che si ottiene deferenziando l'iteratore.

Nelle righe con etichetta `4` 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 `5` 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.

3.10 Vettori di stringhe

Il primo programma proposto si occupa di verificare se alcune parole acquisite da input, sono contenute in un dizionario. Il dizionario 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;

int main()
{
  vector<string> vocab;                                              /*1*/
  vector<string>::iterator it, pos;  // iter per scansione e ricerca
  string stinp,    // var per l'input nel vettore
         parcerc;  // parola da cercare nel vocabolario
  int i;
  bool continua;

  // Acquisisce parole da inserire nel dizionario

  continua=true;                                                     /*2*/
  for(i=0;continua;i++) {
    cout << "\nParola " << i << " (Invio per finire) " ;
    getline(cin,stinp);

    if(!stinp.empty())                                               /*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.empty()) {                                         /*5*/


    // Cerca la parola

    pos=vocab.end();
    for (it=vocab.begin();it!=vocab.end();it++) {                    /*6*/
      if (*it==parcerc) {                                            /*7*/
        pos=it;
        break;                                                       /*8*/

      }
    }

    if (pos!=vocab.end())
      cout << "\nParola trovata, posizione " << pos-vocab.begin();
    else
      cout << "\nParola non trovata";
```

```
    // Prossima parola da cercare

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

  return 0;
}
```

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 e le inserisce in un vettore:

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

int main()
{
  string testo,estratta;  // testo da elaborare, parola estratta
  vector<string> token;   // parole estratte
  vector<string>::iterator it;
  int inizio,fine,        // pos spazi delimitanti parola
      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'è 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(it=token.begin();it!=token.end();it++)
    cout << *it << endl;

  return 0;
}
```

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 fine della 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 per la ricerca, il punto finale dell'elaborazione 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 sotto-problemi 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 sotto-problemi in modo analogo.

In definitiva si comincia specificando la sequenza degli stati di avanzamento per la soluzione del problema proposto anche se, in questa prima fase, possono mancare i dettagli realizzativi: si presuppone infatti che tali dettagli esistano già. Il programma, in questa prima stesura, conterrà, oltre alle solite strutture di controllo, anche operazioni complesse che dovranno poi essere ulteriormente specificate. Passando poi 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).

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 schermata di un video in modo da coprire, con lo sguardo, l'intero o quasi listato e limitare il singolo segmento a poche azioni di modo che sia più semplice isolare eventuali errori. Inoltre il sottoprogramma deve essere isolato dal contesto in cui opera, cioè 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, effettuando il sottoprogramma una singola 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).

4.2 Comunicazioni fra sottoprogrammi

Seguendo il procedimento per scomposizioni successive si arriva alla fine ad un programma principale che coordina le azioni di una serie di sottoprogrammi affinché si arrivi al risultato atteso. Il programma principale *chiama* in un certo ordine i sottoprogrammi (lancia l'esecuzione delle istruzioni che ne fanno parte); ogni sottoprogramma oltre che *chiamato* può anche essere il *chiamante* di un ulteriore sottoprogramma. 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) dal sottoprogramma chiamante e gli ritorna i

propri output (intesi come somma delle informazioni prodotte al suo interno). Questi, a loro volta, potrebbero costituire gli input per un ulteriore sottoprogramma.

Per garantire quanto più possibile la portabilità e l'indipendenza dei sottoprogrammi, i linguaggi strutturati distinguono 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 al chiamante.

4.3 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 e questo 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++, per risolvere problemi di questo genere, rende disponibili i *namespace*, che delimitano la visibilità dei nomi in essi dichiarati.

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

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

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

  return 0;
}
```

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, nel `main`, 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*/

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

  return 0;
}
```

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. In linea generale può essere conveniente utilizzare una istruzione `using namespace` per le librerie standard del linguaggio ma specificare l'operatore di visibilità in tutti gli altri casi.

Se lo stesso namespace è definito più volte, ogni nuova definizione espande quella precedente: tutte le variabili sono definite all'interno di un unico namespace anche se dichiarate in tempi diversi.

4.4 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 quando questa ultima ha completato le sue elaborazioni, un valore che potrà essere assegnato ad una variabile e per questo le funzioni vengono utilizzate principalmente a destra del segno di assegnamento.
- 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.

4.5 Le funzioni in C++. Istruzione return

Nel linguaggio C++ tutti i sotto-programmi sono funzioni. Per poter simulare le procedure che non ritornano alcun valore è disponibile 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:

- Il **prototipo** della funzione. È uso comune dichiararlo all'inizio del programma, prima della definizione della funzione `main`, o in ogni caso prima della funzione che effettua la chiamata. La struttura generale del prototipo di una funzione è:

  ```
  tipo-ritornato nome-funzione(dichiarazione tipi di parametri);
  ```

  Il prototipo rappresenta l'interfaccia della funzione: quanti parametri necessitano alla funzione per espletare i propri compiti e il loro tipo. Specifica come si usa la funzione: il tipo ritornato fornisce indicazione sul tipo di variabile a cui dovrà essere assegnato il valore ritornato dalla funzione, la quantità con la specifica dei tipi indica quante variabili/valori, e di che tipo, devono essere specificate come parametri per poter usare in modo corretto la funzione.

  I prototipi sono stati introdotti per permettere al compilatore di effettuare un controllo sulla quantità e 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 (problema che riguarda le variabili locali della funzione): sono indispensabili solo la quantità e il tipo.

  Nella costruzione di programmi complessi capita di utilizzare molte funzioni. In questo caso le funzioni possono essere raggruppate in **librerie** e i rispettivi prototipi raggruppati nei file 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.

- 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 coerentemente con 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 dove è evidenziato anche lo stile di scrittura:

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

La prima riga della definizione differisce dal prototipo per il fatto che, qui, sono presenti le dichiarazioni delle variabili locali e, inoltre, la riga non termina con il punto e virgola ma è seguita da un blocco che racchiude le istruzioni contenute nella funzione. La dichiarazione dei parametri segue, in linea di massima, le regole della definizione delle variabili utilizzata negli altri esempi trattati in precedenza, tranne per:

- i parametri sono distinti dalla virgola
- per ogni tipo si può dichiarare una sola variabile. Non è possibile con un unico tipo dichiarare più parametri, è necessario ripetere la dichiarazione di tipo
- se il parametro è passato per valore la sua dichiarazione rispecchia quella della dichiarazione generica di una variabile. Se il parametro è passato per riferimento il tipo va seguito dal carattere `&` come nell'esempio:

```
int funzione(int primo, float& secondo)
{
  ...
};
```

Le definizioni di funzioni potrebbero essere scritte in qualsiasi punto del programma: verranno mandate in esecuzione in seguito alla chiamata e quindi non avrebbe alcuna importanza il posto fisico dove sono allocate. Sono però comuni delle convenzioni di scrittura secondo le quali le definizioni delle funzioni sono codificate dopo il `main`.

Fra le istruzioni contenute nella definizione della funzione particolare importanza assume l'istruzione `return` utilizzata per ritornare al chiamante il 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
```

Se la funzione ritorna un tipo `void` l'istruzione `return` manca.

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` (che è un nome riservato) è eseguita per prima all'avvio del programma, in qualunque posto sia messa, e le funzioni sono eseguite nell'ordine in cui sono chiamate.

Di conseguenza a quanto richiesto dalla sintassi si può osservare, inoltre, che `main` è una funzione che ritorna un valore intero. Tutto ciò fa parte di una concezione delle funzioni ereditata dal linguaggio C: una funzione ritorna sempre un valore che indica la conclusione corretta dell'elaborazione (valore 0 o altro valore se si sono riscontrati problemi). E questo è anche il motivo per cui in C tutti i sotto-programmi sono funzioni.

4.6 Il metodo top-down: un esempio step by step

Come applicazione della metodologia top-down nello sviluppo di un programma viene riproposto l'algoritmo di selezione: dato un vettore di interi non nulli, si vuole generare il sottoinsieme dei pari. Ora il problema sarà risolto applicando il paradigma funzionale.

Il primo passo da compiere è stabilire, nella funzione `main`, quali sono le fasi che portano alla soluzione del problema:

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

// prototitpi

int main()
{
  vector<int> numeri,  // insieme da elaborare                        /*2*/
              pari;    // sottoinsieme dei pari

  cout << "Estrazione di sottoinsieme:" << endl
       << "data una sequenza di interi positivi" << endl
       << "comunica il sottoinsieme dei pari" << endl;

  // input numeri da elaborare

  // estrazione sottoinsieme dei pari

  // output sottoinsieme pari

  return 0;
}
```

In questa prima stesura, a parte le inclusioni che servono (`1`) e la dichiarazione delle variabili necessarie alla risoluzione del problema (`2`) che in questo esempio sono due oggetti della classe `vector`, sono presenti soltanto righe di commento utilizzate per stabilire ***quali*** cose fare e in ***quale*** ordine.

Stabilita, per grandi linee, la sequenza delle funzioni da svolgere, si passa a chiarire meglio la prima funzione: quella che si deve occupare dell'input dei numeri da elaborare. In questa sede è importante solo stabilire **cosa** la funzione deve fare e **come** si deve utilizzare.

- **Cosa fa la funzione**: la funzione, che potrà avere nome `inputNumeri`, dovrà occuparsi dell'input da tastiera dei numeri da elaborare.
- **Come si utilizza la funzione**: la funzione non ha bisogno di input (non gli serve niente per potere assolvere ai propri compiti). Fornisce, in output (dati che rende disponibili come risultati della propria elaborazione), il vettore caricato con i dati acquisiti da tastiera.

In conseguenza, a quanto osservato, si potrà aggiungere al programma il prototipo e la chiamata alla funzione:

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

// prototitpi
```

```
namespace selez{
  vector<int> inputNumeri();                                    /*1*/
};

int main()
{
  vector<int> numeri,  // insieme da elaborare
              pari;    // sottoinsieme dei pari

  cout << "Estrazione di sottoinsieme:" << endl
       << "data una sequenza di interi positivi" << endl
       << "comunica il sottoinsieme dei pari" << endl;

  // input numeri da elaborare

  numeri = selez::inputNumeri();                                /*2*/

  // estrazione sottoinsieme dei pari

  // output sottoinsieme pari

  return 0;
}
```

Nel prototipo (1) è evidenziato il fatto che la funzione ritorna un oggetto di tipo `vector<int>`. Il prototipo è inserito nel namespace `selez`. Nella chiamata alla funzione (2), `main` assegna il valore di ritorno della funzione a `numeri` e qui si ritroveranno i valori acquisiti dalla funzione. La chiamata è effettuata applicando l'operatore di visibilità per specificare il namespace in cui è valida la definizione della funzione.

Con lo stesso procedimento si stabiliscono i compiti e le *modalità d'uso* delle rimanenti due funzioni:

- La funzione che si deve occupare dell'estrazione del sottoinsieme di pari, e che potrà avere nome `estraiPari`, ha necessità di conoscere il vettore da cui estrarre il sottoinsieme (per la funzione è un input e quindi sarà un parametro passato per valore) e produrrà (output per la funzione e, quindi, valore di ritorno della funzione) il vettore sottoinsieme.
- La funzione che dovrà occuparsi dell'output su video del sottoinsieme, e che potrà avere nome `outputPari`, dovrà conoscere (input, quindi parametro passato per valore) il vettore da visualizzare. Non produce dati da tornare a `main`. Fra l'altro la chiamata a questa funzione è pure l'ultima istruzione che viene eseguita. Non saranno necessari parametri in uscita.

Il programma, con i prototipi e le chiamate alle funzioni, sarà:

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

// prototitpi

namespace selez{
  vector<int> inputNumeri();
  vector<int> estraiPari(vector<int>);
  void outputPari(vector<int>);
};
```

```
int main()
{
  vector<int> numeri,  // insieme da elaborare
              pari;    // sottoinsieme dei pari

  cout << "Estrazione di sottoinsieme:" << endl
       << "data una sequenza di interi positivi" << endl
       << "comunica il sottoinsieme dei pari" << endl;

  // input numeri da elaborare

  numeri = selez::inputNumeri();                                    /*1*/

  // estrazione sottoinsieme dei pari

  pari = selez::estraiPari(numeri);                                 /*2*/

  // output sottoinsieme pari

  selez::outputPari(pari);                                          /*3*/

  return 0;
}
```

Il vettore `numeri` viene acquisito in conseguenza alla chiamata della prima funzione (`1`). La funzione acquisisce un vettore generico di interi che, in virtù della `1`, diventa `numeri`.

La funzione `estraiPari` genera il sottoinsieme dei pari contenuti nel vettore passato come parametro. La chiamata `2` passa il parametro `numeri` e quindi tale funzione estrae i pari da quel vettore. Il risultato è depositato in `pari`.

La funzione `outputPari` in realtà stampa su video tutti gli elementi del vettore di interi che viene passato come parametro. La chiamata `3` fa in modo che l'output riguardi il vettore generato in seguito alla chiamata `2` cioè il vettore dei numeri pari.

Il programma è formalmente completo: sono specificate le funzioni che assolve e la sequenza delle fasi di elaborazione. Naturalmente per poterlo compilare e, successivamente, eseguire è necessario aggiungere il codice delle funzioni. Finora si è specificato **cosa** le funzioni fanno, ora dovrà essere specificato **come** lo fanno, ovvero si dovrà aggiungere la definizione delle funzioni.

```
...
// prototitpi

namespace selez{
  vector<int> inputNumeri();
  vector<int> estraiPari(vector<int>);
  void outputPari(vector<int>);
};

int main()
{
  ...
}

// input numeri da elaborare
```

```
vector<int> selez::inputNumeri()
{
  vector<int> num;                                                    /*3*/
  int temp;

  cout << "Numero da elaborare (<=0 per finire) ";
  cin >> temp;

  while(temp>0){
    num.push_back(temp);

    cout << "Numero da elaborare (<=0 per finire) ";
    cin >> temp;
  }

  return num;                                                         /*4*/
}

// estrazione sottoinsieme pari

vector<int> selez::estraiPari(vector<int> num)                        /*3*/
{
  vector<int> numpari;                                                /*3*/
  vector<int>::iterator it;

  for(it=num.begin();it!=num.end();it++){
    if(!(*it%2))                                                      /*1*/
      numpari.push_back(*it);                                         /*2*/
  }

  return numpari;
}

// output sottoinsieme pari

void selez::outputPari(vector<int> numpari)                           /*3*/
{
  vector<int>::iterator it;

  if(pari.empty())
    cout << “Non ci sono numeri pari” << endl;
  else{
    cout << “Numeri pari trovati” << endl;
    for(it=numpari.begin();it!=numpari.end();it++)
      cout << *it << “\t”;
    cout << endl;
  }
}
```

La funzione che si occupa dell'input del vettore dichiara (nella prima riga con etichetta `3`) una variabile locale il cui valore verrà ritornato al programma chiamante per mezzo della `4`. La variabile, al termine della funzione, cessa la sua visibilità e il chiamante non potrebbe conoscere il suo valore.

La funzione che genera il sottoinsieme dei pari, esamina il vettore da elaborare e se trova un elemento pari (controllo in `1`), lo carica nel vettore dei pari (`2`).

Il fatto che, in diverse funzioni, vengano dichiarate variabili con lo stesso nome (`3`) è voluto e

dovuto solo a motivi di chiarezza e leggibilità: in realtà si tratta di variabili diverse (sono dichiarazioni locali alle singole funzioni). Per esempio la funzione `estraiPari` genera un vettore contenente i numeri pari presenti in un vettore passato alla funzione come parametro, la funzione `outputPari`, in realtà, stampa su video il contenuto di un vettore che riceve come parametro: è la successione delle chiamate alle funzioni stabilita nel `main` e l'uso corretto dei parametri che fa in modo che la funzione di estrazione dei pari li estragga dal vettore acquisito in precedenza e che la funzione di output mandi in stampa il risultato delle operazioni della funzione di estrazione dei pari.

La funzione `estraiPari` genera il vettore dei pari che viene tornato al chiamante facendo in modo che quest'ultimo sia a conoscenza dei risultati dell'elaborazione e ciò è il modo corretto di risolvere il problema trattandosi di un unico valore da calcolare, tuttavia per evidenziare le diversità con il caso del passaggio di parametri per riferimento si propone il confronto con la funzione scritta utilizzando il passaggio per riferimento:

Valore ritornato

```
vector<int> selez::estraiPari(vector<int>
num)
{
  vector<int> numpari;
  vector<int>::iterator it;

  for(it=num.begin();it!=num.end();it++){
    if(!(*it%2))
     numpari.push_back(*it);
  }
  return numpari;
}
```

Passaggio per riferimento

```
void selez::estraiPari(vector<int> num,
vector<int>& numpari)
{

  vector<int>::iterator it;

  for(it=num.begin();it!=num.end();it++){
    if(!(*it%2))
     numpari.push_back(*it);
  }

}
```

Nella funzione a destra è stato aggiunto un parametro passato per riferimento (evidenziato da `&`) che conterrà il vettore dei pari. La funzione ritorna un `void` e opera su `numpari` che è un nome locale ma fa riferimento alla stessa zona di memoria del parametro che viene passato all'atto della chiamata. Non esiste la riga di codice con `return` perché il valore calcolato è già nel parametro e, inoltre, la funzione torna un tipo indefinito (`void`).

Anche se la funzione, dal punto di vista del risultato ottenuto, potrebbe essere scritta in uno qualsiasi dei due modi presentati, il formalismo corretto è il primo essendoci un solo valore da comunicare con l'esterno (la funzione ha un solo output). Il secondo formalismo (passaggi per riferimento e tipo tornato `void`) si adotterà solo in presenza di più output da fornire o di nessuno.

5 Strutture e tabelle

5.1 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 nel blocco 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;
  };
}

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

  cout << "Esempio di uso di una struttura in C++" << endl;

  // inserimento dati del dipendente
```

```
    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*/

    // stampa dati dipendente

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

    return 0;
}
```

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.

Sia gli input che gli output vengono effettuati sui membri della struttura poiché gli operatori di inserimento e di estrazione sono definiti, per default, per i tipi elementari. Si vedrà in seguito come ridefinire tali operatori in modo da potersi applicare, per esempio, ad una struttura come se fosse un tipo elementare.

5.2 Tabelle: vettori di strutture

Una tabella (vettore di strutture) è costituita da una successione di righe ognuna composta da una ***chiave*** che la identifica in modo univoco e dal valore associato ad essa che può essere un ***record*** costituito da due o più ***campi***. Anche un vettore di tipi elementari, per esempio di `int`, può definirsi tabella solo che, in questo caso, l'elemento esistente per ogni riga e associato alla chiave ha un solo campo.

Sulle tabelle sono comuni due tipi di algoritmi già definiti in precedenza per un vettore generico: la ricerca (data una tabella, cercare al suo interno un determinato record), la selezione (data una tabella, costruirne una nuova che contiene i record della prima tabella che soddisfano a determinati requisiti).

Il primo programma proposto (esempio di ricerca), 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, ne visualizza i suoi dati. Nel listato seguente viene presentato il `main` e i prototipi delle funzioni utilizzate:

```
#include <iostream>
```

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

namespace ditta{                                                    /*1*/
  struct dip{
    string reparto;
    string cognome;
    string nome;
    float stipendio;
  };

  vector<dip> insertDip();
  void insertCogNom(string&, string&);
  vector<dip>::iterator cerca(vector<dip>&, string, string);        /*2*/
}

int main()
{
  vector<ditta::dip> libropaga;  // elenco dipendenti               /*3*/
  string cogncerca,nomcerca;  // cognome e nome da cercare
  vector<ditta::dip>::iterator pos;  // iter dip

  // Inserimento dati dei dipendenti,

  libropaga = ditta::insertDip();                                   /*4*/

  // cognome e nome dip da cercare

  ditta::insertCogNom(cogncerca,nomcerca);                          /*4*/

  // Ricerca dip

  pos = ditta::cerca(libropaga,cogncerca,nomcerca);                 /*5*/

  // Visualizza dati dip

  if(pos!=libropaga.end()){                                         /*6*/
    cout << "\nReparto di appartenenza " << pos->reparto << endl;   /*7*/
    cout << "Stipendio " << pos->stipendio << endl;                 /*7*/
  }
  else
    cout << "Non risulta come dipendente" << endl;

  return 0;
}
```

In `1` è definito uno spazio di nomi in cui dichiarare la struttura con i dati di interesse del dipendente e i prototipi delle funzioni.

Nella `3` viene dichiarato un vettore di tipo `dip`, tipo a sua volta definito in `ditta`.

Nelle `4` si chiamano le funzioni per l'input della tabella dei dipendenti e dei dati della persona da cercare.

Nella `5` si richiama la funzione di ricerca del dipendente. La funzione restituisce un iteratore che punta al record del dipendente all'interno della tabella. Come si nota dalla `2`, alla funzione viene passato un riferimento alla tabella dei dipendenti. In realtà per la funzione di ricerca la tabella è un

input e quindi sarebbe sufficiente passare il parametro per valore, ma è previsto il ritorno di un iteratore e se la tabella fosse ricopiata in un'altra posizione (passaggio per valore), il puntatore indicherebbe la posizione nel vettore locale: informazione inutilizzabile dal chiamante.

Nelle `7` si stampano le informazioni richieste se il dipendente esiste (`6`). La deferenziazione dell'iteratore (`*pos`) produrrebbe un oggetto di tipo `dip` che non può essere usato con l'operatore di inserimento: è necessario deferenziare i singoli componenti utilizzando l'operatore freccia (`->`).

Per completare il programma si riportano le definizioni delle funzioni:

```
// Inserimento dipendenti

vector<ditta::dip> ditta::insertDip()
{
  vector<ditta::dip> d;
  ditta::dip tempdip;
  int i;

  cout << "Inserimento dati dei dipendenti" << endl;

  for(i=0;;i++){                                                    /*1*/
    cout << "\nDip " << i << endl;
    cout << "Reparto ";
    getline(cin,tempdip.reparto);

    if(tempdip.reparto.empty())                                     /*2*/
      break;

    cout << "Cognome ";
    getline(cin,tempdip.cognome);                                   /*3*/
    cout << "Nome ";
    getline(cin,tempdip.nome);                                      /*3*/
    cout << "Stipendio ";
    cin >> tempdip.stipendio;                                       /*3*/
    cin.ignore();                                                   /*4*/

    d.push_back(tempdip);                                           /*5*/

  }
  return d;
}

// Cognome e nome dip da cercare

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

// Ricerca

vector<ditta::dip>::iterator
ditta::cerca(vector<ditta::dip>& d,string c,string n)
{
```

```
  vector<ditta::dip>::iterator it, p;

  p=d.end();                                                        /*6*/

  for(it=d.begin();it!=d.end();it++){
    if(it->cognome==c && it->nome==n){                              /*7*/
      p=it;                                                         /*8*/
      break;                                                        /*9*/
    }
  }

  return p;
}
```

Il ciclo che inizia in `1` acquisisce i dati dei dipendenti finché non si digita un *Invio* a vuoto sul campo reparto (`2`). Il ciclo non ha una condizione di uscita. L'uscita dal ciclo è effettuata dalla istruzione `break` eseguita di conseguenza ad un input vuoto su reparto.

Nella funzione di inserimento dei dati dei dipendenti, prima vengono conservati gli input in una struttura temporanea (`3`) e, quindi, l'elemento viene inserito nel vettore (`5`). La `4` svuota il buffer di tastiera del carattere *Invio*, lasciato dall'input numerico precedente, al fine di consentire il prossimo `getline`. Tale operazione è indispensabile quando sono mischiati input che utilizzano `cin` e `getline`: dopo l'input di un valore effettuato con `cin` e se poi devono essere effettuati input con la funzione `getline` è necessario svuotare il buffer dal carattere *Invio* perché, in caso contrario, la funzione trovando l'*Invio* considererebbe il prossimo input già effettuato e lo salterebbe.

La funzione di ricerca inizializza un iteratore (`6`) e se, poi, il cognome e nome passati come parametri hanno gli stessi valori dei rispettivi della riga della tabella considerata (`7`), la posizione viene conservata (`8`) e si forza una uscita anticipata dal ciclo (`9`).

Il prossimo programma proposto effettua una selezione. Acquisita la tabella dei dipendenti e un reparto, viene generato e 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.

```
...
namespace ditta{
  ...
  string insertRep();
  vector<dip> selezione(vector<dip>, string);
  void stampa(vector<dip>);
}
...
int main()
{
  vector<ditta::dip> libropaga,
                     dipRep; // dipendenti del reparto             /*1*/
  string repcerca;           // reparto oggetto della selezione    /*1*/
  ...
  repcerca = ditta::insertRep();                                   /*2*/
  ...
  dipRep = ditta::selezione(libropaga,repcerca);                   /*3*/
  ...
  ditta::stampa(dipRep);                                           /*4*/
```

```
  return 0;
}
...
// Inserimento reparto da selezionare

string ditta::insertRep()
{
  string r;

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

vector<ditta::dip>
ditta::selezione(vector<ditta::dip> d1,string r)
{
  vector<ditta::dip> d2;
  vector<ditta::dip>::iterator it;

  for(it=d1.begin();it!=d1.end();it++){
    if(it->reparto==r)                                          /*5*/
      d2.push_back(*it);                                        /*6*/
  }

  return d2;
}

// Stampa dipendenti del reparto cercato

void ditta::stampa(vector<ditta::dip> d2)
{
  vector<ditta::dip>::iterator it;

  if(!d2.empty()){                                              /*7*/
    cout << "\nDipendenti che lavorano nel reparto" << endl;
    for(it=d2.begin();it<d2.end();it++)
      cout << it->cognome   << " "                              /*8*/
           << it->nome      << " "                              /*8*/
           << it->stipendio << endl;                            /*8*/
  }
  else
    cout << "Non risultano dipendenti nel reparto" << endl;
}
```

Nello spazio di nomi `ditta` la funzione `insertCogNom` è sostituita dalla `insertRep`, `cerca` è sostituita da `selezione` e viene aggiunta la funzione `stampa` per la stampa della tabella dei dipendenti del reparto. Nelle dichiarazioni di variabili della `1` si aggiungono: un nuovo vettore `dipRep` e la stringa `repcerca`.

La chiamata alla funzione di inserimento del reparto, della `2`, ritorna un dato di tipo `string`.

La `3`, rispetto al programma precedente dove è usata una funzione di ricerca, è modificata in modo da chiamare la nuova funzione. In `4` si stampa l'elenco dei dipendenti del reparto prodotto dalla funzione richiamata in precedenza (`3`).

La funzione di selezione, dopo aver verificato nella `5` che il reparto è quello interessato, invia

l'elemento al nuovo vettore (`6`): un elemento di tipo `dip`.

Se la selezione non è vuota (`7`) le `8` stampano, per ogni elemento, i singoli membri della struttura non potendosi applicare, come già osservato, l'operatore di inserimento a oggetti di tipo `dip`.

6 Il paradigma ad oggetti

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

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 viene effettuato dentro la struttura stessa. La funzione è inserita ed elabora dati della struttura e non richiede che i dati, se appartenenti alla stessa struttura, siano passati come parametri.

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

cout << "Titolo :" << lib1.titolo << endl;
if(!lib1.prestitoOk())                                         /*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 `prestitoOk` 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 prestitoOk()
  {
    bool esito=false;
    if (presente){
      presente   = false;
      esito = true;
    };
    return esito;
  };
};
```

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 nelle classi: se, infatti, non si specifica altrimenti, la visibilità è limitata solo all'interno della classe, per esempio la funzione `prestitoOk` può accedere a `presente` ma ad essa non si può accedere come componente di una variabile di tipo `libro`.

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 (*attributi*) e le funzioni (*metodi*) 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 i dati e tutto il necessario per usare i dati; il programma che usa la classe non ha bisogno di sapere com'è fatta ma soltanto ha necessità di sapere quale sono le procedure definite nella classe e come si richiamano.

Nella OOP, in ragione del mascheramento dei dati, questi normalmente vengono dichiarati nella sezione `protected` (scelta effettuata in questi appunti per il motivo di rendere disponibili i dati alle classi discendenti), o nella sezione `private`, e si accede ad essi utilizzando i metodi presenti nella parte `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.

6.2 OOP: progetto e uso di classi step by step. Costruttori

Applicare la OOP alla risoluzione di un determinato problema vuol dire innanzi tutto individuare le classi interessate (quali sono i dati) e i metodi che devono avere (che operazioni si richiedono sui dati).

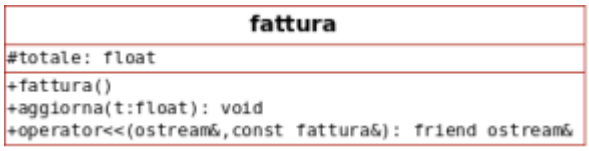
Viene ora riproposto, come primo esempio, il programma per *conoscere* il **totale** di una **fattura** di cui *siano date* le **righe** che *la compongono*, ognuna individuata da una determinata **quantità** e dal **prezzo unitario** dell'oggetto venduto. Questa volta il problema viene affrontato utilizzando il paradigma OOP.

Le domande da porsi per affrontare il problema sono due: quali sono i dati coinvolti, che tipo di operazioni sono richieste per quei dati ovvero stabilire le classi con i metodi. Un semplice sistema suggerito da testi che si occupano di OOD (*Object Oriented Design*) per il riconoscimento delle classi consiste nell'evidenziare, nella descrizione del problema da risolvere, i nomi e i verbi. Nell'esempio proposto i nomi sono segnati in **grassetto** e i verbi in *italico*. Le classi e gli attributi vanno ricercati fra i nomi e i metodi fra i verbi. Nell'esempio è chiaro che il totale è un attributo di fattura e quantità e prezzo unitario sono attributi della riga. È possibile quindi riconoscere due classi:

- La fattura individuata, dal punto di vista dei dati, dal totale. Operazioni richieste: inizializzazione totale, aggiornamento del totale, output del totale.

  La rappresentazione di una classe può essere espressa con il linguaggio grafico UML

(Unified Modeling Language), progettato per usi diversi fra cui anche quello di rappresentare le classi e che ne rende immediatamente visibili le caratteristiche:



Oltre alla parte sovrastante dove ha sede il nome della classe, il box rimanente è suddiviso in due parti: nella parte superiore sono riportati gli attributi e nella parte inferiore i metodi. La visibilità è evidenziata con un segno che precede il nome (`+` equivale a `public`, `#` a `protected`, `-` a `private`). I nomi delle variabili sono seguiti dal tipo, i metodi (con gli eventuali parametri scritti in modo da seguire le stesse convenzioni degli attributi) dal tipo ritornato.

Il codice della classe fattura, inserito nel namespace `elabfat`, sarà:

```
namespace elabfat{
  class fattura{
  public:
    fattura(): totale(0.0) {};                                    /*1*/
    void aggiorna(float t){totale +=t;};                          /*2*/
    friend ostream& operator<<(ostream&, const fattura&);         /*3*/
  protected:
    float totale;                                                 /*4*/
  };
  ostream& operator<<(ostream& output, const fattura& f)          /*5*/
  {
    output << "Totale fattura: " << f.totale << endl;
    return output;
  };
};
```

La classe ha solo una variabile definita nella parte protetta (`4`).

Il primo metodo definito in `1` è particolare: il ***costruttore***. Si tratta di un metodo che ha lo stesso nome della classe e che non restituisce alcun valore nemmeno `void`. Il metodo non può essere richiamato esplicitamente, ma viene richiamato in automatico quando si istanzia un oggetto della classe. In pratica quando si definisce una variabile di tipo `fattura` viene inizializzata la variabile privata: definire una nuova fattura vuol dire, in automatico, azzerarne il totale. Il costruttore non deve essere per forza definito. Accanto al costruttore potrebbe essere necessario definire anche un ***distruttore***: metodo che viene richiamato in automatico quando cessa la visibilità dell'oggetto. Il distruttore quando c'è ha lo stesso nome della classe come il costruttore ma preceduto da ~. Se ci fosse per la classe `fattura` avrebbe nome `~fattura()`. Il distruttore diventa necessario, per esempio, quando c'è una allocazione dinamica della memoria e bisogna liberare e recuperare la memoria occupata. Sia al costruttore che al distruttore possono essere passati parametri.

Nel costruttore è uso comune, invece della solita sintassi, inizializzare, quando si tratta di variabili di tipi elementari, i valori base utilizzando la sintassi della `1`. Se ci fossero state più variabili, sarebbero state separate dalla virgola.

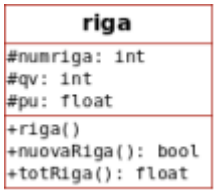
Il metodo `2` aggiorna il totale della fattura con il parametro passato. La definizione della funzione, composta da un unica istruzione è riportata direttamente (***funzioni inline***). La

differenza fra le funzioni inline e le altre in cui viene distinto il prototipo dalla definizione consiste nel modo con cui vengono trattate dal compilatore. Le funzioni inline vengono tradotte come *macroistruzioni*: il codice viene inserito tutte le volte che la funzione viene richiamata. La definizione codificata a parte comporta invece la presenza di un unico codice: tutte le volte che la funzione è richiamata viene effettuato un salto all'unico codice esistente. In definitiva le funzioni inline vengono utilizzate se comprendono poco codice (sono più veloci ma occupano spazio: tre chiamate, per es., equivalgono a tre duplicazioni di codice), le funzioni con definizione a parte quando ne fanno parte parecchie righe di codice.

La `3` **ridefinisce l'operatore <<** per oggetti della classe `fattura`. L'operatore, così come definito nella libreria `iostream` e come altre volte fatto notare, consente l'output di stringhe e di variabili di tipi elementari. La ridefinizione è una caratteristica importante della OOP (**overloading**) che permette la personalizzazione, in questo caso, dell'operatore di inserimento in modo che possa essere usato per oggetti del tipo `fattura`. Dal punto di vista sintattico è necessario passare alla funzione un parametro di tipo `ostream` e uno del tipo l'oggetto per il quale si scrive il codice per la ridefinizione. La parola chiave `friend` permette alla funzione, anche se non è un metodo della classe, l'accesso alla parte privata o protetta.

Il codice della ridefinizione dell'operatore di inserimento (`5`) istruisce su come deve essere applicato tale operatore agli oggetti della classe: tutte le volte che viene usato l'operatore di inserimento verranno eseguite queste operazioni.

- La riga individuata dalla quantità venduta e dal prezzo unitario. Operazioni: generazione di una nuova riga, calcolo totale riga.



Il codice sarà:

```
namespace elabfat{                                                  /*1*/
  class riga{
  public:
    riga(): numriga(0) {};                                          /*2*/
    bool nuovaRiga();
    float totRiga();
  protected:
    int numriga;
    int qv;
    float pu;
  };

  // metodo per inserimento di una nuova riga

  bool riga::nuovaRiga()                                            /*3*/
  {
    int q;
    float p;
    bool OK=true;                                                   /*4*/
```

```
    cout << "Riga n." << ++numriga << endl;
    cout << "Quantita\'_venduta prezzo_unitario (0 0 per finire) ";
    cin >> q >> p;

    if(q<=0 || p<=0.0){                                          /*5*/
      numriga--;
      OK = false;
    }
    else{
      qv = q;
      pu = p;
    }

    return OK;
  };

  // metodo per il calcolo del totale della riga

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

Anche la definizione di questa classe è contenuta (`1`) nello stesso namespace di quella precedente: il namespace viene espanso in modo da comprendere le nuove definizioni.

Il metodo `2` (costruttore) inizializza una variabile che conterà le righe che verranno considerate facenti parte della fattura.

Il metodo `3` fa esistere una nuova riga (`4`) se (`5`) entrambi i valori di quantità venduta e prezzo unitario sono accettabili.

La definizione dei metodi utilizza l'operatore di visibilità `::` per l'accesso agli elementi della classe.

A questo punto la definizione delle classi, in conseguenza delle richieste del programma da sviluppare, è completa: il codice di ogni classe si registra in un file. Il file `c_fattura` contiene la definizione della classe `fattura`, il file `c_riga` quello della classe `riga`. I file saranno inclusi nel sorgente del programma che dovrà utilizzare le classi in esse definite.

Prima di procedere oltre è opportuno rispondere ad una eventuale osservazione sulla mancanza, nella definizione della classe, di implementazione di eventuali altri metodi per elaborazioni diverse che potrebbero essere necessario effettuare. In questi casi, 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 altro metodo. 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.

Il `main`, in pratica, si dovrà occupare soltanto di fare interagire oggetti della classe `riga` con oggetti della classe `fattura` come anche risulta evidente dal diagramma UML:

Il simbolo del rombo pieno dalla fattura alla riga si legge come: *è composto da*. Nell'esempio è quindi evidenziato che un oggetto della classe `fattura` è composta da uno o più oggetti della classe `riga`. Le righe hanno senso se fanno parte di una fattura.

```
#include <iostream>
#include “c_fattura”                                          /*1*/
#include “c_riga”                                             /*1*/
using namespace std;

int main()
{
  elabfat::riga r;                                           /*2*/
  elabfat::fattura f;                                        /*2*/

  // elaborazione righe fattura

  while(r.nuovaRiga())                                       /*3*/
    f.aggiorna(r.totRiga());                                 /*4*/

  // stampa fattura

  cout << f;                                                 /*5*/

  return 0;
}
```

Le `1` permettono di includere le definizioni delle classi. I nomi dei file che contengono le definizioni sono racchiusi fra carattere doppio apice alto ad indicare che i file si trovano nello stesso posto fisico dove è salvato il `main`. Le parentesi angolari (`<>`) racchiudono i nomi dei file che si trovano nelle *directory di include* usati dal compilatore e definiti durante l'installazione dello stesso.

Nelle `2` si dichiarano due oggetti appartenenti, rispettivamente, alle classi `riga` e `fattura`.

L'interazione fra i due oggetti si riduce a: controllare se c'è ancora una riga da elaborare (`3`), aggiornare il totale della fattura con il totale della riga (`4`).

la stampa del totale della fattura (`5`) termina il programma.

L'applicazione del paradigma della OOP ha portato, oltre a vantaggi relativi al riutilizzo del codice o maggiore facilità di individuazione di errori, notevoli semplificazioni nella scrittura del programma: prima ci si occupa solamente delle elaborazioni relative ai singoli dati coinvolti nel problema (nel caso proposto la fattura e la riga). Definite le classi il programma che utilizza oggetti delle classi si limita solo all'interazione fra gli stessi: *ogni oggetto*, in quanto istanza di una classe, *sa già come comportarsi*.

6.3 Gestione biblioteca step by step (1): la classe libro

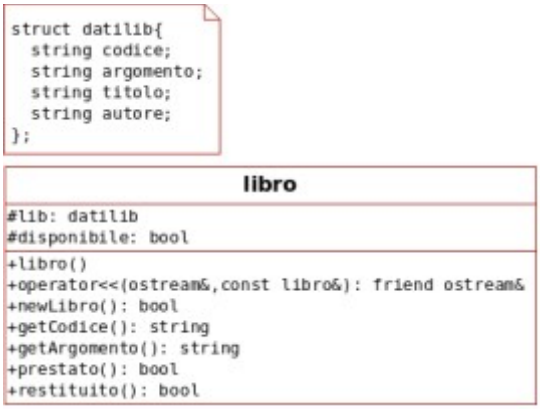
In questo e nei due paragrafi seguenti viene presentato un progetto di una semplice gestione di una biblioteca. L'esempio oltre che essere un caso più complesso di quello presentato in precedenza

mette in evidenza altre semplificazioni nella scrittura di un programma conseguenza di caratteristiche della progettazione OOP.

"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 della gestione dei prestiti di una biblioteca (dell'ereditarietà degli oggetti si tratterà più avanti).

GESTIONE BIBLIOTECA: Le operazioni richieste riguardano la *registrazione del prestito* e della *restituzione* di un **libro**, l'aggiunta di libri alla dotazione della biblioteca, la possibilità di *visualizzare* i libri esistenti e i libri che sono *catalogati secondo un determinato argomento*. Il libro, per semplificare togliendo quelli non influenti per le elaborazioni, può avere come attributi un codice che lo identifica in maniera univoca, l'argomento in cui viene classificato, il titolo e l'autore.



La classe `libro` è contenuta nel file `c_libro`. Agli attributi specificati nel problema è necessario aggiungerne un altro per la registrazione della disponibilità del libro ad essere prestato. Le operazioni da effettuare sul libro prevedono oltre alla generazione del libro stesso, la stampa, la conoscenza del codice identificativo (per le operazioni sul libro) e dell'argomento (per la selezione per argomento):

```
#ifndef C_LIBRO                                                    /*1*/
#define C_LIBRO                                                    /*1*/

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

namespace biblioteca{
  struct datilib{                                                  /*2*/
    string codice;
    string argomento;
    string titolo;
    string autore;
  };

  class libro{
  public:
    libro(): disponibile(true){};                                  /*3*/
    friend ostream& operator<<(ostream&, const libro&);            /*4*/
    bool newLibro();                                               /*5*/
    string getCodice() const {return lib.codice;};                 /*6*/
```

```
    string getArgomento() const {return lib.argomento;};            /*6*/
    bool prestato();                                                 /*7*/
    bool restituito();                                               /*7*/
  protected:
    datilib lib;                                                     /*2*/
    bool disponibile;                                                /*2*/
  };

 // overloading operatore inserimento

  ostream& operator<<(ostream& output, const libro& l1)
  {
    output << l1.lib.codice << " | " << l1.lib.argomento << " | "
           << l1.lib.titolo << " | " << l1.lib.autore;

    if(l1.disponibile)
      output << "\nLibro presente" << endl;
    else
      output << "\nLibro in prestito" << endl;
    return output;
  };

  // dati nuovo libro

  bool libro::newLibro()
  {
    bool esito = true;

    cout << "Dati nuovo libro" << endl;
    cout << "Codice -> ";
    getline(cin,lib.codice);

    if(lib.codice.empty())                                           /*8*/
      esito = false;                                                 /*9*/
    else{
      cout << "Argomento -> ";
      getline(cin,lib.argomento);                                    /*10*/
      cout << "Titolo -> ";
      getline(cin,lib.titolo);                                       /*10*/
      cout << "Autore -> ";
      getline(cin,lib.autore);                                       /*10*/
    };

    return esito;
  };

  // Operazioni di prestito

  bool libro::prestato()
  {
    bool prestatoOk=false;

    if (disponibile){                                                /*11*/
      disponibile = false;                                           /*12*/
      prestatoOk  = true;
    };

    return prestatoOk;
  }
```

```
  bool libro::restituito()
  {
    bool ritornatoOk=false;

    if (!disponibile){                                          /*13*/
      disponibile = true;                                       /*14*/
      ritornatoOk = true;
    };

    return ritornatoOk;
  }

};

#endif                                                          /*1*/
```

Le tre righe 1, due all'inizio del file di definizione della classe, una alla fine, sono direttive al pre-compilatore. Capita spesso di avere necessità di includere più volte, in più file di codice, le classi che servono. Nell'esempio proposto il file `c_libro` viene incluso nella definizione dell'altra classe `c_libreria`, 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_LIBRO`, 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.

I dati del libro sono definiti nella struttura 2. Una variabile di questo tipo, identificante il libro da gestire, è definita nella sezione `protected` della classe assieme alla variabile booleana che indicherà la disponibilità del libro al prestito.

La classe ha un costruttore (3) che inizializza il valore di disponibilità del libro. Un nuovo libro registrato è subito disponibile per operazioni di prestito. Viene inoltre (4) ridefinito l'operatore di inserimento per un oggetto della classe.

Il metodo 5 restituisce informazioni sull'avvenuta generazione di un nuovo oggetto della classe: se il codice identificativo non viene immesso (8) il nuovo oggetto non viene generato (9), in caso contrario il libro può esistere (10).

I due metodi 6, codificati come funzioni inline, restituiscono i dati necessari per le elaborazioni richieste (si ricorda che è necessario utilizzare dei metodi pubblici perché gli oggetti non hanno accesso alle variabili `protected`). Il qualificatore ***const*** alla fine della dichiarazione indica il fatto che i due metodi non modificano i dati della classe. I dati sono infatti accessibili in modo implicito, senza necessità di passaggio di parametri, ai metodi della classe ma se si vuole essere sicuri che un metodo, che non debba modificare gli attributi della classe, non lo faccia anche per errore, è opportuno aggiungere il qualificatore. In questo caso un tentativo errato di modifica degli attributi comporterà un messaggio di errore da parte del compilatore.

I metodi 7 effettuano le due possibili operazioni richiedibili sul libro. Formalmente la codifica dei due metodi è identica: si interroga la disponibilità del libro per l'operazione da effettuare (11 e 13) e, se possibile, si procede (12 e 14). L'unica differenza fra i due metodi consiste nel fatto che l'operazione di prestito richiede la disponibilità del libro (11), la restituzione che si tratti di un libro

prestato (`13`).

6.4 Gestione biblioteca step by step (2): la classe libreria

GESTIONE BIBLIOTECA: Le operazioni richieste riguardano la registrazione del prestito e della restituzione di un libro, l'*aggiunta di libri* alla dotazione della **biblioteca**, la possibilità di *visualizzare* i libri esistenti e i libri che sono *catalogati secondo un determinato argomento*. Il libro, per semplificare togliendo quelli non influenti per le elaborazioni, può avere come attributi un *codice che lo identifica* in maniera univoca, l'argomento in cui viene classificato, il titolo e l'autore.

Dalla definizione del problema si deduce una classe `libreria` il cui codice sarà contenuto nel file `c_libreria`. A differenza della precedente si tratta di una ***classe aggregato***: un contenitore di oggetti di tipo libro a cui si richiedono determinate funzionalità. Le operazioni previste riguardano la possibilità di aggiungere un nuovo libro, rintracciare il libro associato ad un codice specificato, aggiornare i dati del libro in seguito ad una operazione di prestito o restituzione, ottenere l'elenco dei libri in dotazione, selezionare i libri per argomento:

```
#ifndef C_LIBRERIA
#define C_LIBRERIA

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

#include "c_libro"

namespace biblioteca{
  class libreria{
  public:
    libreria() {it=elenco.end();};                                  /*0*/
    bool FindCodice(string);                                        /*1*/
    bool AddLibro(libro);                                           /*2*/
    libro getLibro() const {return *it;};                           /*3*/
    bool aggiorna(libro);                                           /*4*/
    bool IsElenco() const {return (!elenco.empty());};              /*5*/
    void IniziaElenco(){it=elenco.begin();};                        /*6*/
    bool getNextLibro(libro&);                                      /*7*/
    libreria SelezArg(string) const;                                /*8*/
  protected:
    vector<libro> elenco;                                           /*9*/
    vector<libro>::iterator it;                                     /*9*/
  };

  // cerca libro associato a un codice

  bool libreria::FindCodice(string cLib)                            /*10*/
  {
```

```
  bool trovato = false;

  for(it=elenco.begin();it!=elenco.end();it++)                    /*11*/
    if(it->getCodice()==cLib){                                    /*12*/
      trovato = true;                                             /*13*/
      break;
    };

  return trovato;
};

// aggiunge un libro alla libreria

bool libreria::AddLibro (libro l)                                 /*14*/
{
  bool aggiunto=true;

  if(FindCodice(l.getCodice()))                                   /*15*/
    aggiunto = false;
  else
    elenco.push_back(l);

  return aggiunto;
};

// aggiorna i dati del libro attuale

bool libreria::aggiorna(libro l)                                  /*16*/
{
  bool aggOK=true;

  if(it!=elenco.end())                                            /*17*/
    *it=l;
  else
    aggOK=false;

  return aggOK;
};

// passa al prossimo libro

bool libreria::getNextLibro(libro& l)                             /*18*/
{
  bool esiste=false;

  if(it!=elenco.end()){                                           /*19*/
    l = *it;
    esiste = true;
    it++;                                                         /*20*/
  }

  return esiste;
};

// seleziona libri per argomento

libreria libreria::SelezArg (string a) const                      /*21*/
{
  libreria sel;
```

```
      vector<libro>::const_iterator its;

      for(its=elenco.begin();its!=elenco.end();its++)                    /*22*/
        if(its->getArgomento()==a)
          sel.AddLibro(*its);

      return sel;
    };
  };

  #endif
```

Il contenitore di libri, come si deduce dalle `9` è implementato utilizzando un vettore. Fra i dati della classe è definito pure un iteratore per puntare al singolo libro che interesserà le elaborazioni. Il contenitore non è un semplice vettore ma prevede i metodi per la gestione della libreria. La classe ha un costruttore (`0`) che inizializza l'iteratore.

Il metodo `FindCodice` (`1`) si occupa di fornire informazioni sull'esistenza, nella libreria, del codice di un libro. Questa informazione è utile per l'inserimento di un nuovo libro (`2`): se il codice è univoco non possono esserci due codici uguali e quindi, prima di accettare un nuovo codice è necessario verificare che non esista già. Inoltre la biblioteca deve fornire la funzionalità di ricerca di un libro per codice. La (`3`) restituisce il libro associato al valore attuale dell'iteratore.

il metodo `aggiorna` (`4`) registra le operazioni effettuate su un libro (prestito/restituzione) nella libreria.

I metodi `5`, `6`, e `7` sono funzionalità utili per la scansione della libreria. Rispettivamente: sapere se esiste un elenco (`5`), iniziare l'elenco (`6`), accedere al prossimo libro dell'elenco (`7`). Internamente alla classe viene utilizzato un vettore e quindi parecchi metodi di questo gruppo si sono potuti scrivere come funzioni inline perché ci si appoggia ai metodi della classe `vector`.

Il metodo `SelezArg` (`8`) costruisce un nuovo elenco con i libri dell'argomento passato come parametro. Il valore di ritorno è un oggetto della stessa classe e quindi dotato dei metodi della classe: per esempio per scorrere l'elenco. Il metodo avrebbe potuto avere come valore di ritorno un `vector` di tipo `libro` ma, in questo caso, si sarebbe trattato di un insieme di dati e non di libri della biblioteca dotati dei metodi appena descritti.

Tutti i metodi che non modificano in alcun modo i dati della classe sono dichiarati `const`.

L'implementazione del metodo `FindCodice` (`10`) effettua una scansione sequenziale del vettore (`11`) e se il codice del libro ha lo stesso valore del codice passato come parametro (`12`) si modifica il valore logico di ritorno (`13`), si esce forzatamente dal ciclo e, in tal modo, il valore dell'iteratore permette l'accesso all'elemento con quel codice.

il metodo per l'inserimento di un libro in elenco (`14`) ne verifica la possibilità: se il codice associato al libro esiste (`15`) non si procede.

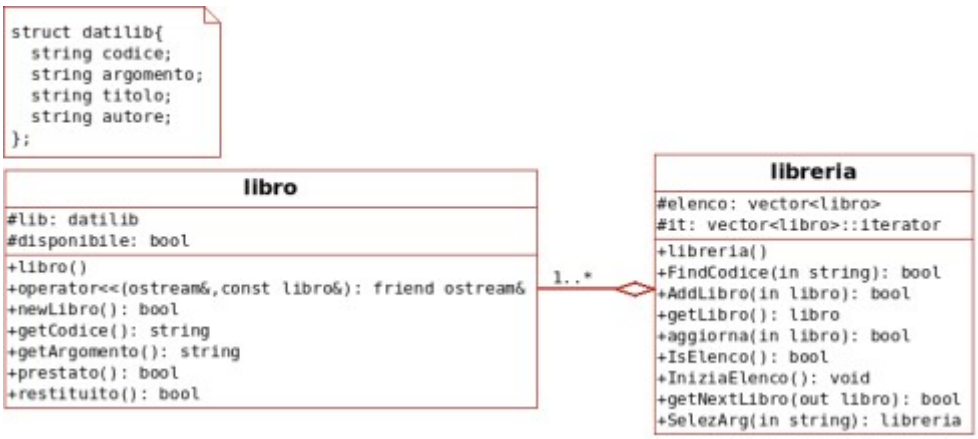
Il metodo per l'aggiornamento dei dati di un libro (`16`) sostituisce il libro puntato dal valore attuale dell'iteratore definito nella classe. Il controllo in `17` serve solo nel caso il metodo sia stato richiamato erroneamente.

Il metodo `getNextLibro` (`18`) restituisce, nel parametro, il libro puntato attualmente se esiste (`19`) e aggiorna successivamente (`20`) l'iteratore al prossimo libro in modo da renderlo disponibile per la prossima lettura.

Il metodo `21`, dal punto di vista dell'algoritmo, è una operazione di selezione commentata anche in altri esempi e non necessita di ulteriori commenti. È necessario aggiungere invece qualcosa riguardante la dichiarazione `const_iterator`. Il metodo è di tipo `const` perché non deve modificare i dati della classe, ma l'uso degli iteratori permetterebbe tali modifiche e il compilatore, in caso di dichiarazione di variabile di tipo `iterator`, segnalerebbe errore nella riga `22` perché l'iteratore scorre il vettore permettendo la modifica dei singoli elementi. Un iteratore definito come `const_iterator` non consente la modifica dell'elemento.

6.5 Gestione biblioteca step by step (3): la funzione main

Definite le classi interessate il `main` si deve occupare dell'interazione fra oggetti di diverse classi. Una volta definiti i metodi le funzioni sono codificate utilizzando un insieme ridotto di righe di codice facili da comprendere: basta avere presenti le definizioni delle classi.



Il rombo vuoto dalla parte della classe `libreria` si legge come: *è un insieme di*. Un oggetto della classe `libreria` è un insieme di uno o più oggetti della classe `libro`. A differenza dell'aggregazione `fattura-riga` di un esempio precedente indicata con il rombo pieno, qui gli oggetti della classe `libro` possono esistere anche senza far parte di una libreria, le righe invece esistevano in quanto facenti parti di una fattura e la cancellazione di una fattura avrebbe comportato la cancellazione delle righe che la componevano.

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

#include "c_libro"
#include "c_libreria"

// prototipi

void InsertLibri(biblioteca::libreria&);
void VisLibri(biblioteca::libreria);
void VisLibroCod(biblioteca::libreria);
void SelezPerArg(biblioteca::libreria);
void Prestito(biblioteca::libreria&);
void Restituzione(biblioteca::libreria&);

int main()
{
  biblioteca::libreria biblio;                                    /*1*/
  int tipoOp;
```

```
  for(;;){                                                          /*2*/
    cout << "Gestione Biblioteca" << endl;
    cout << "1 - Inserisci Libri" << endl;
    cout << "2 - Visualizza Libri" << endl;
    cout << "3 - Cerca Libro per Codice" << endl;
    cout << "4 - Seleziona per Argomento" << endl;
    cout << "5 - Prestito" << endl;
    cout << "6 - Restituzione" << endl;
    cout << "0 - Fine" << endl;
    cout << "Operazione? ";
    cin >> tipoOp;
    cin.ignore();

    if(!tipoOp) break;                                              /*3*/

    switch(tipoOp){
    case 1:
      InsertLibri(biblio);
      break;
    case 2:
      VisLibri(biblio);
      break;
    case 3:
      VisLibroCod(biblio);                                          /*4*/
      break;
    case 4:
      SelezPerArg(biblio);                                          /*4*/
      break;
    case 5:
      Prestito(biblio);                                             /*4*/
      break;
    case 6:
      Restituzione(biblio);                                         /*4*/
      break;
    default:
      cout << "Scelta non ammessa" << endl;
    };
  };

  return 0;
}
```

Il `main` può comprendere la dichiarazione della libreria (`1`) che viene passata come parametro alle varie funzioni di gestione associate alla scelta dell'operatore (`4`) e richiamate all'interno di un ciclo (`2`) che termina (`3`) con l'immissione del valore nullo nella scelta.

```
// (1) Inserimento libri

void InsertLibri(biblioteca::libreria& el)
{
  biblioteca::libro libTemp;

  cout << "Inserimento libri Codice vuoto per finire" << endl;

  while(libTemp.newLibro()){                                        /*1*/
    if(!el.AddLibro(libTemp))                                       /*2*/
      cout << "Codice duplicato ripetere inserimento" << endl;
```

```
  };
};
```

La funzione di inserimento mentre ci sono libri (1) li inserisce nella libreria (2). La scelta di inviare alla funzione la libreria come parametro e non tornare un tipo libreria è dovuta al fatto che, così, si possono inserire nuovi libri in qualsiasi momento. Nel caso si fosse adottata la soluzione di tornare un tipo libreria la libreria sarebbe stata reinizializzata ad ogni richiamo della funzione.

```
// (2) Visualizza libri

void VisLibri(biblioteca::libreria el)
{
  biblioteca::libro temp;

  if(el.IsElenco()){                                                    /*1*/
    el.IniziaElenco();                                                  /*2*/
    cout << "Elenco libri" << endl;

    while(el.getNextLibro(temp))                                        /*3*/
      cout << temp;                                                     /*4*/
  }
  else
    cout << "Non ci sono libri che soddisfano criterio" << endl;
};
```

La funzione si occupa della stampa su video dell'elenco dei libri contenuti nella libreria passata come parametro: possono essere tutti i libri (scelta 2) o quelli che soddisfano a determinati requisiti (scelta 4). Se c'è un elenco (1), lo si inizializza (2) e, mentre ci sono libri in elenco (3), si stampano le informazioni del libro (4).

```
// (3) cerca libro per codice

void VisLibroCod(biblioteca::libreria el)
{
  string cod;

  cout << "Ricerca libro per codice" << endl;

  cout << "Codice (vuoto per finire) ";
  getline(cin,cod);

  while(!cod.empty()){                                                  /*1*/
    if(el.FindCodice(cod))                                              /*2*/
      cout << el.getLibro();                                            /*3*/
    else
      cout << "Codice inesistente" << endl;

    // nuovo codice da cercare

    cout << "Codice (vuoto per finire) ";
    getline(cin,cod);

  };
};
```

Finché l'utilizzatore della funzione non introduce un valore vuoto (1), si ricerca il codice e se esiste un libro ad esso associato (2) lo si visualizza (3).

```
// (4) seleziona elenco per argomento

void SelezPerArg(biblioteca::libreria el)
{
  biblioteca::libreria elSel;
  string argInp;

  cout << "Argomento per la selezione ";
  getline(cin,argInp);

  elSel=el.SelezArg(argInp);                                    /*1*/
  VisLibri(elSel);                                              /*2*/
};
```

La selezione per argomento si limita a richiamare il metodo opportuno (`1`) della classe cui appartiene l'oggetto `el` (`libreria`) e a richiamare la funzione di visualizzazione (`2`) passandogli come parametro l'elenco.

```
// (5) prestito di un libro

void Prestito(biblioteca::libreria& el)
{
  string cod;
  biblioteca::libro l;

  cout << "Codice libro per prestito ";
  getline(cin,cod);

  if(!el.FindCodice(cod))                                       /*1*/
    cout << "Codice inesistente" << endl;
  else{
    l=el.getLibro();                                            /*2*/
    cout << l;

    // verifica se prestito possibile

    if(l.prestato()){                                           /*3*/
      cout << "Operazione effettuata" << endl;
      el.aggiorna(l);
    }
    else
      cout << "Operazione non possibile" << endl;
  };
};

// (6) restituzione di un libro

void Restituzione(biblioteca::libreria& el)
{
  string cod;
  biblioteca::libro l;

  cout << "Codice libro per restituzione ";
  getline(cin,cod);

  if(!el.FindCodice(cod))                                       /*1*/
    cout << "Codice inesistente" << endl;
  else{
```

```
      l=el.getLibro();                                          /*2*/
      cout << l;

      // verifica se prestito possibile

      if(l.restituito()){                                       /*3*/
        cout << "Operazione effettuata" << endl;
        el.aggiorna(l);
      }
      else
        cout << "Operazione non possibile" << endl;
    };
  };
```

Il codice delle due funzioni di prestito/restituzione del libro è quasi identico salvo il fatto che l'operazione è diversa, ma di questo si occupa il metodo opportuno. Dal punto di vista logico l'operazione consiste nel cercare il codice del libro interessato (`1`) e, se trovato, stampare i dati (`2`) ed effettuare l'operazione (`3`) se possibile.

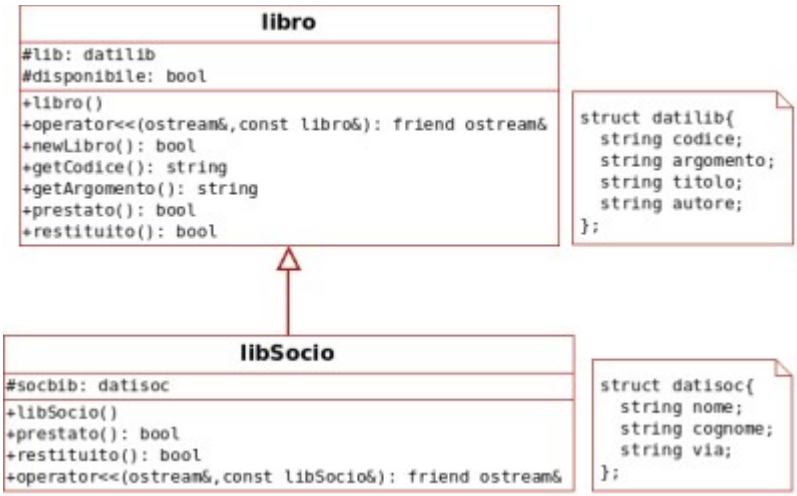
Anche da questo esempio dovrebbe essere evidente la semplificazione del progetto del programma dovuta all'applicazione delle tecniche OOP.

6.6 Ereditarietà: da libro a libSocio

Una delle proprietà più potenti delle classi è la possibilità di definire una nuova classe (classe discendente, classe derivata, ***subclass***) a partire da una classe esistente (classe antenata, classe base, ***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 anche in parte. Questo meccanismo è quello a cui si fa riferimento quando si parla di ereditarietà e permette di ridurre notevolmente il codice da scrivere. Le funzioni permettono di utilizzare codice esistente se il nuovo ambiente in cui si vogliono utilizzare è perfettamente identico a quello in cui si sono sviluppate, l'ereditarietà permette di adattare il codice esistente al nuovo ambiente.

Questo meccanismo 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 una classe discendente dalla classe insetto: 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, dovrebbe riguardare anche la registrazione dei dati del socio che ha preso in prestito il libro.



Nel diagramma UML è evidenziato da una freccia che la classe `libSocio` è discendente da `libro`: è dotata dei metodi pubblici definiti nella classe antenata più quelli definiti o ridefiniti in essa.

Il file `c_libSocio` contiene la definizione della classe:

```
#ifndef C_LIBSOCIO
#define C_LIBSOCIO

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

#include "c_libro"

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

  // overloading operatore estrazione per la struttura

  istream& operator>>(istream& input, datisoc& ds){                  /*2*/
    cout << "Dati socio" << endl;
    cout << "Nome -> ";
    getline(input,ds.nome);
    cout << "Cognome -> ";
    getline(input,ds.cognome);
    cout << "Recapito -> ";
    getline(input,ds.via);
    return input;
  };

  class libSocio : public libro {                                    /*3*/
  public:                                                            /*4*/
    libSocio();                                                      /*5*/
    bool prestato();                                                 /*6*/
    bool restituito();                                               /*6*/
    friend ostream& operator<<(ostream&, const libSocio&);           /*7*/
  protected:
    datisoc socbib;                                                  /*8*/
```

```
};

// overloading operatore inserimento

ostream& operator<<(ostream& output, const libSocio& l1)
{

  output << static_cast<libro> (l1);                                    /*9*/

  if(!(l1.disponibile)){                                                /*10*/
    output << "Prestato a:" << endl
           << l1.socbib.nome << " | " << l1.socbib.cognome << " | "
           << l1.socbib.via << endl;
  }
  return output;
};

// Inizializzazione dati socio

libSocio::libSocio()
{
  socbib.nome=" ", socbib.cognome=" ", socbib.via=" ";                  /*5*/
}

// Prestito del libro al socio

bool libSocio::prestato()                                               /*11*/
{
  bool prestitoOK=false;

  if(libro::prestato()){                                                /*12*/
    cin >> socbib;                                                      /*13*/
    prestitoOK=true;
  }
  return prestitoOK;
}

// Restituzione libro dal socio

bool libSocio::restituito()                                             /*11*/
{
  bool ritornatoOK=false;

  if(libro::restituito()){                                              /*12*/
    socbib.nome=" ", socbib.cognome=" ", socbib.via=" ";                /*13*/
    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. Non ci sarà una classe socio perché non si intende gestire i soci e i dati del socio sono considerati un *corredo* dei dati del libro.

Nella `2` si effettua l'overloading dell'operatore di estrazione per elementi della struttura dei dati del

socio. Il formalismo sintattico è quasi identico a quello utilizzato per l'overloading dell'operatore di inserimento. Si noti che qui, nel passaggio del parametro dell'elemento per cui si sta ridefinendo l'operatore, manca `const` e, d'altra parte, bisogna modificarne il valore. La variabile di tipo `istream` (`input`) prende il posto del flusso `cin` così come utilizzato solitamente.

Nella `3` 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. Lo specificatore di accesso, 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`.

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 `6`. La classe discendente ha la possibilità di ridefinire i metodi ereditati dalla classe base (***overloading dei metodi***) per poterli adattare alla propria specificità. Per gli oggetti della classe è ridefinito l'operatore di inserimento (`7`).

Nella parte `protected` della classe vengono definiti i nuovi dati membri della nuova classe (`8`) in aggiunta a quelli ereditati dalla classe genitrice.

Fra i metodi è presente (`5`) il costruttore che si occupa dell'inizializzazione dei dati del socio. 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.

Nella ridefinizione dell'operatore di inserimento per gli oggetti della classe, oltre a fare riferimento (`10`) a variabili ereditate da `libro`, è presente la riga `9` che in pratica fa riferimento all'overloading dell'operatore per oggetti della classe `libro`. Il casting in pratica si potrebbe tradurre in modo discorsivo come: *per la parte `libro` di `libSocio` stampa come già noto, per la parte nuova si specifica di seguito* (`10` e seguenti).

Le `11` 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 `12` 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 aggiunta alle istruzioni che si eseguivano prima (righe `12`) i nuovi metodi si occupano di conservare (`13`) gli opportuni valori nelle variabili private del socio. Agli aggiornamenti dei dati del libro, ci pensa il metodo richiamato nelle `12`.

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.

6.7 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 protetta `disponibile`).

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.

Se si vuole adattare il programma di gestione della libreria con l'utilizzo di oggetti della nuova classe `libSocio` le modifiche da apportare al codice esistente sono minime:

- La classe `libreria` dovrà fare riferimento ad oggetti `libSocio` e non più `libro`. Si vedrà nel prossimo paragrafo come organizzare la classe `libreria` in modo da non apportare alcuna modifica sia che si tratti di contenere oggetti di tipo `libro` che oggetti di tipo `libSocio`. Ora, temporaneamente, si può adottare un sistema a *forza bruta*: sostituire, nel file `c_libreria`, le ricorrenze di `libro` con `libSocio`.
- Nel programma di gestione, oltre ad includere le nuove classi:

  ```
  #include “c_libSocio”
  #include “c_libreria2”
  ```

  esistono solo due posti dove si fa riferimento ad oggetti della classe `libro`: le operazioni di prestito e restituzione. Basta, anche qui, modificare `libro` in `libSocio`.

6.8 Le classi modello: i template

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`, definiti come metodi nella classe `libreria` 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 contenitori 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 (tutti quelli che la biblioteca potrebbe avere esigenza di gestire) e, se si dichiara che `libreria` contiene oggetti di tipo `libro`, il metodo `getLibro` permetterà di estrarre un oggetto di tipo `libro`, se invece contiene oggetti di tipo `libSocio`, il metodo restituirà un oggetto di tipo `libSocio`.

In UML i template si individuano specificando sulla destra in alto il parametro, con il relativo tipo, utilizzato come segnaposto.

```
// Definizione di una classe modello: libreria

#ifndef C_LIBRERIA3
#define C_LIBRERIA3

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

namespace biblioteca{
  template <class T>                                                    /*1*/
  class libreria{
  public:
    libreria() {it=elenco.end();};
    bool FindCodice(string);
    bool AddLibro(T);
    T getLibro() const {return *it;};
    bool aggiorna(T);
    bool IsElenco() const {return (!elenco.empty());};
    void IniziaElenco(){it=elenco.begin();};
    bool getNextLibro(T&);
    libreria SelezArg(string) const;
  protected:
    vector<T> elenco;                                                   /*2*/
    typename vector<T>::iterator it;                                    /*3*/
  };

  // cerca libro associato a un codice

  template <class T>                                                    /*1*/
  bool libreria<T>::FindCodice(string cLib)                             /*4*/
  {
    bool trovato = false;

    for(it=elenco.begin();it!=elenco.end();it++)
      if(it->getCodice()==cLib){                                        /*5*/
        trovato = true;
        break;
      };

    return trovato;
  };

  // aggiunge un libro alla libreria

  template <class T>                                                    /*1*/
```

```
  bool libreria<T>::AddLibro (T l)                                    /*4*/
  {
    bool aggiunto=true;

    if(FindCodice(l.getCodice()))                                     /*5*/
      aggiunto = false;
    else
      elenco.push_back(l);

    return aggiunto;
  };

  // aggiorna i dati del libro attuale

  template <class T>                                                  /*1*/
  bool libreria<T>::aggiorna(T l)                                     /*4*/
  {
    bool aggOK=true;

    if(it!=elenco.end())
      *it=l;
    else
      aggOK=false;

    return aggOK;
  };

  // passa al prossimo libro

  template <class T>                                                  /*1*/
  bool libreria<T>::getNextLibro(T& l)                                /*4*/
  {
    bool esiste=false;

    if(it!=elenco.end()){
      l = *it;
      esiste = true;
      it++;
    }

   return esiste;
   };

  // seleziona libri per argomento

  template <class T>                                                  /*1*/
  libreria<T> libreria<T>::SelezArg (string a) const                  /*4*/
  {
    libreria<T> sel;
    typename vector<T>::const_iterator its;                           /*3*/

    for(its=elenco.begin();its!=elenco.end();its++)
      if(its->getArgomento()==a)                                      /*5*/
        sel.AddLibro(*its);

    return sel;
  };
};
```

```
#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 `T` 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 `T`. la sintassi della dichiarazione di un iteratore per una classe template prevede lo specificatore `typename` (3).

Nella definizione dei metodi (4), il nome della classe è seguito dalle parentesi angolari con il segnaposto.

La classe contenitore `libreria`, per come definita, vale per qualsiasi tipo di oggetti purché dotati dei metodi `getCodice` e `getArgomento` (5).

6.9 Utilizzo delle classi template

Anche in questo caso, se si utilizza la nuova `c_libreria3`, le modifiche da apportare al programma di gestione prestiti sono molto limitate:

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

#include "c_libsocio"
#include "c_libreria3"                                          /*1*/

// prototipi

void InsertLibri(biblioteca::libreria<biblioteca::libSocio>&);      /*2*/
void VisLibri(biblioteca::libreria<biblioteca::libSocio>);          /*2*/
void VisLibroCod(biblioteca::libreria<biblioteca::libSocio>);       /*2*/
void SelezPerArg(biblioteca::libreria<biblioteca::libSocio>);       /*2*/
void Prestito(biblioteca::libreria<biblioteca::libSocio>&);         /*2*/
void Restituzione(biblioteca::libreria<biblioteca::libSocio>&);     /*2*/

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

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`.

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, la classe `vector` è definita come 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"
#include “c_rivista”                                                    /*1*/
#include "c_libreria3"
...
int main()
{
  biblioteca::libreria<biblioteca::libSocio> lib1;                     /*2*/
  biblioteca::libreria<biblioteca::rivista>  lib2;                     /*3*/
...
}
```

Ammesso che il file `c_rivista` contenga la definizione della nuova classe, la `1` permette l'uso della classe.

Nella `2` si definisce un oggetto `lib1` che è un contenitore 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`.

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 del C++ 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 di elaborazione. 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 conservazione di dati in un file di tipo CSV, e un programma che si occupa della lettura dei dati così conservati.

I file CSV (Comma Separated Values) sono file, di tipo testo, in cui in ogni riga c'è una registrazione, per esempio i dati di un libro. I dati di tipo stringa (titolo, autore, editore) sono racchiusi fra due caratteri *doppio apice*, i dati numerici (prezzo) non ne sono racchiusi. I vari dati del singolo libro sono separati dal carattere *virgola*, da cui il nome del tipo di file. L'importanza dei file di questo tipo dipende dal fatto che è comune nei programmi che si occupano di gestire dati (programmi di database, tabelloni elettronici) la funzionalità di importa/esporta di dati in file formato CSV e quindi si può utilizzare un semplice programma in C++ per leggerli, elaborarli in qualsiasi modo e riscriverli in modo da poterli rileggere con un applicativo qualsiasi che preveda l'importazione di dati da un file CSV. Anche se l'applicativo che si utilizza non prevede l'elaborazione che necessita, questa si può aggiungere scrivendo un opportuno programma in C++.

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;

struct libro {
  string titolo;
  string autore;
  string editore;
  float prezzo;
};

int main()
```

```
{
  ofstream datiLib;                                                    /*2*/
  libro libtemp;
  bool continua=true;

  datiLib.open("libri.txt",ios::app);                                  /*3*/

  while(continua){

    // disponibilità dati da registrare nel file
    // nell'esempio lettura da tastiera

    cout << "Titolo :";
    getline(cin,libtemp.titolo);                                       /*4*/
    if(libtemp.titolo=="")                                             /*5*/
      break;

    cout << "Autore :";
    getline(cin,libtemp.autore);                                       /*4*/
    cout << "Editore :";
    getline(cin,libtemp.editore);                                      /*4*/
    cout << "Prezzo :";
    cin >> libtemp.prezzo;                                             /*4*/
    cin.ignore();

    // registrazione dati nel file

    datiLib << "\"" << libtemp.titolo << "\"" << ",";                  /*6*/
    datiLib <<  "\"" << libtemp.autore << "\"" << ",";                 /*6*/
    datiLib <<  "\"" << libtemp.editore << "\"" << ",";                /*6*/
    datiLib << libtemp.prezzo << endl;                                 /*6*/
  }
  datiLib.close();                                                     /*7*/

  return 0;
}
```

L'inclusione `1` rende disponibile gli 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 `datiLib`. L'utilizzo è identico a quello del flusso `cout` che 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`).

Nella `7` si interrompono i collegamenti con il file esterno. Il metodo `close()` chiude lo stream

`datiLib`. Il metodo è il simmetrico di `open()`.

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>
using namespace std;

struct libro {
  string titolo;
  string autore;
  string editore;
  float prezzo;
};

libro estraeCampi(string);                                          /*1*/

int main()
{
  ifstream datiLib;                                                 /*2*/
  libro temp;
  string riga;

  // legge i dati dei libri conservati in un file CSV

  datiLib.open("libri.txt");                                        /*3*/
  while(getline(datiLib,riga)){                                     /*4*/

    // estrae campi

    temp = estraeCampi(riga);                                       /*5*/

    // utilizzo dati
    // nell'esempio stampa su video

    cout << temp.titolo << " - " << temp.autore << " - "            /*6*/
         << temp.editore << " - " << temp.prezzo << endl;

  };
  datiLib.close();

  return 0;
}

// Estrae i campi da una riga letta dal file CSV

libro estraeCampi(string r)
{
  libro temp;
  int pos;

  // estrazione campi di tipo stringa

  pos = r.find(',');                                                /*7*/
  temp.titolo = r.substr(1,pos-2);                                  /*8*/
  r = r.erase(0,pos+1);                                             /*9*/
```

```
    pos = r.find(',');                                          /*7*/
    temp.autore = r.substr(1,pos-2);                            /*8*/
    r = r.erase(0,pos+1);                                       /*9*/

    pos = r.find(',');                                          /*7*/
    temp.editore = r.substr(1,pos-2);                           /*8*/
    r = r.erase(0,pos+1);                                       /*9*/

    // estrazione prezzo

    temp.prezzo = stof(r);                                      /*10*/

    return temp;
}
```

Il programma utilizza una funzione (`1`) per separare i vari campi dalla riga letta dal file.

Nella `2` viene dichiarata una variabile di tipo `ifstream` (stream di input) a cui si associa il file `libri.txt` (`3`). In questo caso, a differenza del flusso di output, non è necessario specificare la modalità di apertura: in output i dati si possono scrivere in un nuovo file o accodare ad un file esistente, in input si leggono i dati uno di seguito all'altro e non esiste altra alternativa. È possibile tuttavia, anche se superfluo, specificare la modalità `ios::in`.

Il programma, sostanzialmente, legge le righe esistenti nel file CSV finché è possibile (`4`), separa i componenti della riga (`5`) e stampa i dati del libro (`6`).

La funzione per l'estrazione dei singoli dati del libro dalla riga, cerca la posizione della virgola (`7`), estrae la quantità di caratteri, ripulita dai doppi apici iniziali e finali, di cui è composto il singolo dato (`8`) e, per semplificare la prossima estrazione, elimina (`9`) i caratteri estratti compresi doppi apici e virgola separatrice.

Il dato di tipo numerico è ottenuto richiamando la funzione (C++11) `stof` sulla stringa e convertendola in `float` (`10`).

8 Riferimenti bibliografici

Per l'elaborazione di questi appunti sono stati consultati i libri:

- *Linguaggio C* di B.Kernighan e D.Ritchie. Il testo base da cui partire per la conoscenza del linguaggio C. Testo dal cui studio deriva la prima versione di questi appunti.
- *Il linguaggio C++* di B.Stroustrup. Il testo base, opera dello sviluppatore del linguaggio.
- *C++ by Dissection* di I.Pohl. Validissimo testo per l'apprendimento del linguaggio basato su esempi dissezionati e largamente commentati. Il testo è stato anche punto di riferimento per le regole di scrittura adottate anche in questi appunti.
- *C++ the Complete Reference* di H.Schildt e `http://www.cplusplus.com`. Fonti inesauribili a cui attingere per la conoscenza di tutte le caratteristiche del linguaggio con esempi esplicativi di uso.
- *Data Structures Using C++* di D.S.Malik soprattutto per i suggerimenti su OOD

Il programma Structorizer (rilasciato con licenza GPL e sviluppato in Java, quindi, utilizzabile sotto qualsiasi piattaforma per cui esiste una JVM) è di aiuto nella fase iniziale della scrittura degli algoritmi. Permette di inserire facilmente i blocchi del diagramma di Nassi-Schneiderman e, principalmente, ne consente la verifica eseguendone le istruzioni passo-passo e mostrando, in seguito all'esecuzione, l'effetto prodotto:

`http://structorizer.fisch.lu/`