Università degli Studi dell'Aquila
Facoltà di Ingegneria

Corso di Fondamenti di Informatica

Il linguaggio di programmazione C++

Ing. Emanuele Panizzi

A.A. 2000/2001

1 Strumenti di Base

1.1 Struttura di un programma C++

Ogni programma C++ ha una parte principale, chiamata *main*, che contiene le istruzioni da eseguire. Le istruzioni devono essere racchiuse all'interno delle parentesi graffe[1].

```
main()
{}
```

In questo paragrafo realizziamo un programma molto semplice il cui scopo è scrivere sul monitor la frase "Hello World!".
Per fare ciò, dobbiamo inserire all'interno delle parentesi graffe la seguente istruzione:

```
cout << "Hello World!";
```

dove:

- `cout` significa "consolle output" e indica il monitor
- la frase `Hello World!`, detta *stringa*, è contenuta tra virgolette
- l'operatore '<<' è usato per inviare la stringa a `cout`
- il punto e virgola indica la fine dell'istruzione

Inoltre dobbiamo aggiungere, fuori dal main, la seguente direttiva:

```
#include <iostream.h>
```

Il significato di questa direttiva verrà approfondito nei capitoli seguenti. Qui basti dire che essa permette di effettuare scritture sul monitor e letture dalla tastiera.
Il programma completo quindi è:

[1] Le parentesi tonde indicano che si tratta di una funzione, vedi capitolo 3.

```
#include <iostream.h>

main()
{ cout << “Hello World!”;
}
```

1.2 Compilazione

Ogni programma, per poter essere eseguito dal calcolatore, deve essere scritto in un file mediante un editor e poi compilato, cioè tradotto in un formato numerico eseguibile dal processore.
La traduzione viene effettuata da un programma detto compilatore, il quale legge il file contenente il programma da tradurre (detto *programma sorgente*) e produce un nuovo file contenente il programma tradotto in formato numerico (detto *programma eseguibile*).
Ad esempio possiamo scrivere il programma del paragrafo precedente in un file e salvarlo con il nome `helloworld.cpp`; a questo punto è possibile compilarlo creando il file eseguibile `helloworld.exe`.

Alcuni compilatori sono integrati in un ambiente di sviluppo, cioè in un software che comprende un editor mediante il quale è possibile scrivere il file sorgente e salvarlo su disco, per poi compilarlo e infine eseguirlo.
In appendice 7.1 è mostrato come compilare un programma C++ su alcuni dei più comuni compilatori.

1.3 Tipi, variabili ed espressioni

Il linguaggio C++ consente di manipolare dati. I dati possono essere letti dalla tastiera, mostrati sullo schermo, mantenuti nella memoria del calcolatore e utilizzati all’interno di espressioni.
In C++ esistono diversi *tipi* di dato, ognuno dei quali ha un nome. Ne elenchiamo qui i principali:

- `int` i numeri interi
- `float` i numeri decimali (numeri *floating point*)
- `char` i caratteri
- `bool` il tipo booleano

Nei prossimi capitoli mostreremo come sia possibile definire dei nuovi tipi di dato.

Per poter memorizzare ed utilizzare un dato è necessario effettuare una dichiarazione, che specifichi il *tipo* e il nome del dato da utilizzare nel programma. Il nome è formato da uno o più caratteri alfanumerici, ma il primo carattere deve essere una lettera.
Un esempio di dichiarazione è il seguente:

```
int a = 12;
```

In questa dichiarazione il nome a indica un dato di tipo intero; il valore iniziale di a è 12, ma in seguito tale valore potrà essere modificato. Per questa ragione si dice che a è una *variabile*.

La memoria del calcolatore è organizzata in *locazioni*, ognuna delle quali può contenere un dato. Le locazioni sono numerate, quindi ogni locazione è identificata da un numero detto *indirizzo*. Il compilatore associa ogni variabile ad una locazione di memoria. Il contenuto della locazione di memoria è detto *valore della variabile*, mentre l'indirizzo della locazione è detto anche *indirizzo della variabile*.

Analizziamo ora alcuni esempi di dichiarazione di variabile.

- Le due dichiarazioni seguenti sono dichiarazioni *senza inizializzazione*: alle due variabili non viene assegnato un valore iniziale. Il valore delle variabili, finché non sarà modificato, è quello lasciato dai programmi precedenti nelle rispettive locazioni di memoria.

  ```
  int b;
  int c;
  ```

- La riga seguente dichiara tre variabili di tipo `float`, senza inizializzazione:

  ```
  float lunghezza, larghezza, altezza;
  ```

- La seguente espressione dichiara due variabili di tipo carattere, inizializzando soltanto la prima con il carattere `f`:

  ```
  char k1 = 'f', k2;
  ```

- Nella dichiarazione seguente `ris` è una variabile booleana, cioè una variabile che può assumere solo il valore `true` o il valore `false` (vero o falso).

  ```
  bool ris = true;
  ```

Per assegnare un nuovo valore a una variabile si usa l'operatore '=', detto operatore di assegnazione. A sinistra dell'operatore di assegnazione si specifica la variabile da modificare, mentre a destra si indica un'espressione il cui risultato sarà memorizzato nella variabile. Ad esempio, mediante l'istruzione seguente assegniamo un nuovo valore alla variabile a. Un eventuale vecchio valore di a viene sovrascritto, cioè viene perso.

```
a = 20;
```

Nell'esempio precedente l'espressione a destra dell'uguale è molto semplice (è costituita solamente dalla costante intera 20), mentre nella seconda riga dell'esempio seguente si

usa un'espressione più complessa che contiene l'operatore '+' per effettuare la somma tra i due valori delle variabili a e b. Il risultato della somma, 30, viene passato all'operatore di assegnazione che lo memorizza nella variabile c.

```
b = 10;
c = a + b;
```

In C++ è possibile effettuare concatenazioni di assegnazioni:

```
a = b = c;
```

in cui il valore dell'espressione più a destra, in questo caso il valore di c, viene propagato verso sinistra. A causa di questa istruzione le tre variabili a, b e c avranno tutte valore uguale a 30. In particolare, il primo operatore '=' da destra assegna a b il valore di c e poi *passa* questo valore verso sinistra. Il valore viene quindi utilizzato dall'altro operatore '=', che lo assegna ad a e a sua volta lo passa verso sinistra. A questo punto il valore passato viene eliminato, non essendovi alcun altro operatore che lo utilizzi. In effetti tutte le espressioni C++ passano un valore verso sinistra, anche se non sempre il valore viene utilizzato.

Le quattro operazioni, sia su dati di tipo `int` che di tipo `float`, si eseguono mediante gli operatori '+', '-', '*', '/'. Analizzando l'esempio seguente, in cui la variabile r assume il valore 2.25, si può notare l'ordine di precedenza fra gli operatori.

```
float r, s, t;
s = 1.5;
t = 2.0;
r = s * t – s / t;
```

Scriviamo adesso un programma che effettua la somma di due numeri interi. In particolare, il programma

- dichiara tre variabili di tipo intero x, y, e z
- richiede di inserire due numeri interi da tastiera e li memorizza in x e y
- somma tali numeri e memorizza il risultato nella variabile z
- scrive sul monitor il valore di z

In questo programma, oltre agli operatori già analizzati, utilizziamo l'operatore '>>' per inviare un dato dalla tastiera (indicata da `cin`, consolle input) a una variabile. Per richiedere più dati e inviarli a più variabili si possono concatenare più operatori '>>', come mostrato nel programma.

```
#include <iostream.h>

main()
{ int x, y, z;
  cin >> x >> y;
  z = x + y;
  cout << z;
}
```

Un'altra categoria di operatori è quella degli operatori relazionali, che confrontano valori e restituiscono un risultato di tipo booleano. Ad esempio, l'operatore relazionale '>' (operatore di maggioranza) confronta il valore dei suoi operandi e restituisce `true` se il primo è maggiore del secondo, oppure `false` in caso contrario. Considerando che nell'esempio precedente la variabile `t` vale 2.0 e la variabile `s` vale 1.5, l'espressione

```
t > s;
```

restituisce `true`.
Nell'espressione:

```
ris = t > s;
```

l'operatore '>' confronta il valore di `t` con il valore di `s` e passa il risultato all'operatore '=', che lo assegna alla variabile `ris`. Quindi la variabile `ris` contiene adesso il valore `true`.

Gli operatori relazionali sono i seguenti:

- `>` maggiore
- `<` minore
- `>=` maggiore o uguale
- `<=` minore o uguale
- `==` uguale (verifica di uguaglianza)
- `!=` diverso

Infine passiamo ad analizzare gli operatori logici, usati nell'ambito delle espressioni booleane.

- `&&` and
  il risultato è `true` se entrambi gli operandi sono `true`
- `||` or
  il risultato è `true` se almeno uno dei due operandi è `true`
- `!` not
  il risultato è la negazione dell'operando, cioè se un operando `ris` è `true`, allora `!ris` è `false` e viceversa.

Ad esempio, nel seguente frammento di programma si assegna alla variabile `ris` il valore `false`:

```
bool ris1 = true, ris2 = false, ris3 = true;
ris = !(ris1 || ris2) && ris3;
```

Un'ultimo esempio di assegnazione che vale la pena menzionare è l'incremento del valore di una variabile.

```
a = a + 5;
```

Questa istruzione assegna un nuovo valore ad a. Il nuovo valore è il risultato della somma del vecchio valore di a con il valore 5. Poiché, nel nostro precedente esempio, a valeva 30, il nuovo valore è adesso 35.
Allo stesso modo, data una variabile

```
int i;
```

l'istruzione

```
i = i + 1;
```

incrementa di 1 il valore di i. Quando l'incremento è unitario si può utilizzare anche l'operatore '++' nel modo seguente:

```
i++;
```

Questa istruzione e quella precedente sono equivalenti.
Si noti che l'operatore '++', utilizzato come sopra, passa verso sinistra il valore che la variabile aveva prima dell'incremento. Allora l'espressione seguente assegna il valore 36 ad a e il valore 35 a b:

```
b = a++;
```

1.4 Array

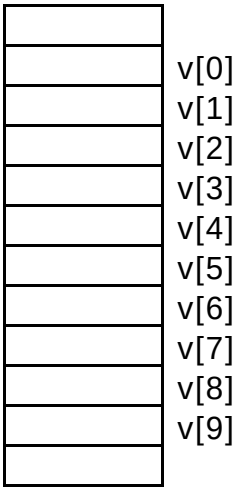
Quando è necessario manipolare collezioni di dati omogenei, è opportuno ordinarli in una struttura, detta *array*. Un array consente di accedere ai dati in esso contenuti specificando un indice che esprime la posizione del dato cercato all'interno della struttura.

Per dichiarare un array è necessario specificare il tipo dei dati in esso contenuti, il nome dell'array e le dimensioni. Ad esempio, mediante la dichiarazione:

```
float v[10];
```

il compilatore costruisce un array di `float`, chiamato `v`, e gli assegna 10 locazioni di memoria consecutive. Il primo elemento dell'array ha indice 0, e si indica con `v[0]`, mentre l'ultimo ha indice 9, e si indica con `v[9]`. Questa dichiarazione è senza inizializzazione, quindi il contenuto degli elementi dell'array `v` non è definito.
Le istruzioni seguenti assegnano i valori 12.0, 5.0 e 60.0 rispettivamente al primo, al secondo e al terzo elemento dell'array `v`:

```
v[0] = 12.0;
```



```
v[1] = 5.0;
v[2] = v[0] * v[1];
```

Un vantaggio legato all'uso degli array consiste nel poter utilizzare come indice una espressione intera. Ad esempio, data una variabile intera `i`, si può scrivere:

```
v[i] = 30.0;
```

In molti casi il valore dell'espressione usata come indice non è definito finché il programma non viene eseguito, quindi la scelta dell'elemento a cui accedere viene fatta solamente a *run-time*, e non a *compile-time*. Ad esempio nel programma seguente il compilatore non può stabilire in quale elemento dell'array verrà memorizzato il valore 12.5. Solo durante l'esecuzione del programma l'utente inserirà da tastiera il valore di `i`, decidendo così l'elemento dell'array da utilizzare.

```
main()
{ int i;
  float v[10];
  cin >> i;
  v[i] = 12.5;
}
```

Questo costituisce un vantaggio poiché dà una maggiore flessibilità nell'accesso e nella modifica dei dati utilizzati dal programma.

Una dichiarazione di array con inizializzazione si effettua indicando i valori degli elementi tra parentesi graffe separati da virgole. Ad esempio:

```
float u[3] = {1.50, 3.15, 2.92};
```

È possibile dichiarare array multidimensionali, cioè in cui sono necessari più indici per identificare un elemento. Ad esempio, possiamo dichiarare un array bidimensionale di interi di dimensioni 3x5 come segue:

```
int w[3][5];
```

In modo simile all'array monodimensionale, possiamo accedere agli elementi di questo array specificando i due indici:

```
w[0][0] = 100;
w[2][4] = 200;
```

Gli array monodimensionali sono anche chiamati *vettori* e gli array multidimensionali *matrici*.

1.5 Strutture di controllo

Le strutture di controllo sono delle istruzioni che permettono di eseguire più di una volta, o di non eseguire affatto, una parte del programma. Per fare ciò, queste istruzioni *valutano una espressione,* anche detta *condizione*, che restituisce in generale un risultato di tipo booleano.
È possibile condizionare l'esecuzione di una porzione di programma mediante le istruzioni `if-else` e `switch`, mentre è possibile eseguire più volte una parte del programma effettuando un ciclo iterativo con le istruzioni `while`, `for` e `do-while`.
Tratteremo esclusivamente le istruzioni `if-else`, `while` e `for`.

1.5.1 If-else

L'istruzione `if-else` ha il seguente formato:

```
if (espressione)
  istruzione;
else
  istruzione;
```

L'`if-else` valuta l'espressione e:

- se il risultato è `true`, esegue esclusivamente la prima istruzione (quella del *ramo if*)
- se il risultato è `false`, esegue esclusivamente la seconda istruzione (quella del *ramo else*)

Il programma prosegue poi con l'istruzione successiva all'`if-else`.

È importante sottolineare che, quando un ramo `if` o un ramo `else` è composto da più di un'istruzione, è necessario racchiudere tutte le istruzioni che ne fanno parte all'interno di una coppia di parentesi graffe, in modo da formare un *blocco*.
In un'istruzione `if-else` il ramo `else` può anche essere omesso. In tal caso, se la condizione è vera viene eseguito il blocco `if`, mentre se la condizione è falsa si procede direttamente alla prima istruzione successiva.

Vediamo ora come il programma seguente usa l'`if-else` per valutare quale sia il maggiore tra due numeri letti dalla tastiera.

```
// max2.cpp
#include <iostream.h>

main()
{ int a,b;
  cin >> a >> b;
  if (a > b)
    cout << "max(a,b)=" << a;
  else
    cout << "max(a,b)=" << b;
```

```
}
```

Se, per esempio, i numeri inseriti da tastiera sono rispettivamente 13 e 7, allora la condizione è vera, cioè a è maggiore di b, e viene eseguita solo l'istruzione del ramo `if` che mostra in output la stringa:

```
max(a,b)=13
```

In questo caso notiamo che non viene eseguita l'istruzione di output posta nel ramo `else`. Se invece vengono inseriti i numeri 3 e 5, allora viene eseguita l'istruzione del ramo else e non quella del ramo if, e la stringa mostrata in output è:

```
max(a,b)=5
```

Infine, si noti come il programma, nel caso i due numeri siano uguali, esegua il ramo `else`.

La prima riga del programma, quella che inizia con una doppia barra '`//`', è un *commento*. I commenti sono delle indicazioni utili per chi legge il programma, ma vengono ignorati dal compilatore. I commenti in C++ sono preceduti dalla doppia barra, e il compilatore considera commento tutto ciò che segue la doppia barra fino alla fine della linea. Il commento di questo esempio indica il nome del programma, `max2.cpp`.

Il programma successivo fa un uso più compilcato delle strutture `if-else` per distinguere tre casi diversi:

- a è maggiore di b
- a è minore di b
- a e b sono uguali

Per fare ciò il programma esegue due `if` annidati, cioè due `if`, uno dei quali è contenuto nel ramo if o nel ramo else dell'altro.

```
// max2plus.cpp

#include <iostream.h>

main()
{ int a,b;
  cout << "inserire a e b:" << endl;
  cin >> a >> b;
  if (a != b)
  { cout << "max(" << a << "," << b << ")=";
    if (a > b)
      cout << a;
    else
      cout << b;
  }
  else
    cout << "a e b sono uguali";
```

```
  cout << endl;
}
```

Se a e b sono diversi si entra nel ramo `if` dell'istruzione `if-else` più esterna. All'interno di esso si trova un'altra istruzione `if-else` che distingue i due casi `a > b` e `a < b`.
Nella tabella seguente sono mostrati gli output corrispondenti ad alcuni input esemplificativi. Le stringhe si formano sul monitor come concatenazione delle diverse istruzioni di output. In particolare se a e b sono diversi viene stampata prima la parte di stringa fino al simbolo '=', poi il valore della variabile a o della variabile b e infine un simbolo di fine linea, `endl`, che non è visibile e serve a far iniziare un eventuale output successivo dall'inizio della riga seguente.

a	b	output
13	7	`max(13,7)=13`
3	5	`max(3,5)=5`
2	2	`a e b sono uguali`

1.5.2 While

Il ciclo `while` è formato da una espressione e da un'istruzione:

```
while (espressione)
  istruzione;
```

In ogni iterazione del ciclo viene valutata l'espressione e, se il risultato è `true`, viene eseguita l'istruzione. Fatto ciò si ritorna a valutare l'espressione e così via. Se, ad un certo punto, la valutazione dell'espressione dà risultato `false`, si *esce dal ciclo*, cioè si esegue la prima istruzione successiva del programma. Notare che l'istruzione di un ciclo `while` potrebbe non essere mai eseguita se l'espressione dà risultato `false` già alla prima valutazione.
Il *corpo del ciclo* può essere composto da più istruzioni. In questo caso è necessario racchiuderle tra parentesi graffe a formare un blocco.

Per illustrare il ciclo while realizziamo un programma che calcola il massimo comun divisore (M.C.D.) tra due numeri. Per fare ciò, dati due numeri interi positivi *m* ed *n* diversi fra loro, sottraiamo il più piccolo al più grande. Ripetiamo la sottrazione finché i due numeri diventano uguali. Il numero ottenuto è il M.C.D.
Ad esempio, dati i numeri 48 e 18 effettuiamo le sottrazioni seguenti.

m	n	
48	18	sottraiamo n ad m
30	18	sottraiamo n ad m
12	18	sottraiamo m ad n

12 6 sottraiamo n ad m
6 6 il Massimo Comun Divisore è 6

Il programma utilizza un ciclo iterativo che termina quando i due numeri diventano uguali. Il numero di iterazioni, ovviamente, non è noto a compile-time, poiché dipende dai numeri m ed n inseriti da tastiera a runtime.

```
// mcd_i.cpp

#include <iostream.h>

main()
{ int m,n;
  cin >> m >> n;
  while (m != n)
  { if (m > n)
      m = m – n;
    else
      n = n – m;
  }
  cout << “M.C.D. = “ << m << endl;
}
```

Mostriamo adesso un altro esempio di uso del while. Questo programma ha lo scopo di calcolare il fattoriale di un numero *n* letto in input, cioè il prodotto `1*2*...*n`.

```
//fatt.cpp

#include <iostream.h>

main()
{ int i = 1, f = 1, n;
  cout << “Fattoriale iterativo. Inserire n: “;
  cin >> n;
  while (i <= n)
    f = f * i++;
  cout << “fattoriale = “ << f << endl;
}
```

Il ciclo compie `n` iterazioni, e la variabile `i` varia da 1 a `n`. Il corpo del ciclo è composto da un’unica istruzione che assegna ad `f` il risultato del prodotto di `f * i`. Notare che `i` viene incrementata ma il valore passato all’operatore ‘`*`’ è quello precedente all’incremento.

1.5.3 For

Il ciclo `for` ha la seguente sintassi:

```
for (espr1; espr2; espr3)
```

```
        istruzione;
```

dove:

- `espr1` è un'espressione che viene eseguita una volta per tutte prima di iniziare il ciclo
- `espr2` è l'espressione che costituisce la condizione del ciclo e viene valutata all'inizio di ogni iterazione; si esegue l'iterazione se il risultato di `espr2` è `true`, altrimenti si esce dal ciclo
- `espr3` è un'espressione che viene valutata alla fine di ogni iterazione, dopo aver eseguito l'istruzione

Il ciclo `for` può sempre essere sostituito da un ciclo `while` come segue:

`for (espr1; espr2; espr3)`
`  istruzione;`
	`espr1;`
	`while (espr2)`
	`{ istruzione;`
	`  espr3;`
	`}`

Mostriamo il ciclo for mediante il programma seguente in cui si effettua la somma di dieci numeri interi letti da tastiera.

```
// somma.cpp

#include <iostream.h>

main()
{ int i,n,somma = 0;
  cout << "Inserire 10 numeri da sommare:" << endl;
  for (i = 0; i < 10; i++)
  { cin >> n;
    somma = somma + n;
  }
  cout << "somma = " << somma << endl;
}
```

Il ciclo viene eseguito 10 volte; alla prima iterazione la variabile `i` vale 0 (è stata posta a zero mediante la `espr1` del ciclo `for`); all'ultima iterazione vale 9 (è stata incrementata alla fine di ogni iterazione mediante `espr3`); alla fine dell'ultima iterazione la variabile `i` viene ancora incrementata di uno, quindi passa al valore 10, e quindi l'`espr2` restituisce `false` e il ciclo termina.
Nel corpo del ciclo viene letto uno dei 10 numeri e il suo valore viene aggiunto al valore della variabile `somma` che viene usata come *accumulatore*.

Mostriamo un altro esempio di uso del ciclo for per calcolare il massimo tra otto numeri interi.

```
// max8.cpp
```

```
#include <iostream.h>

main()
{ cout << “Massimo tra 8 numeri interi” << endl;
  int n,max;
  cout << “Inserire gli 8 numeri” << endl;
  cin >> max;
  for (int i = 1; i < 8; i++)
  { cin >> n;
    if (n > max)
      max = n;
  }
  cout << “max=” << max << endl;
}
```

Questo programma funziona mantenendo un massimo corrente, cioè un massimo valido finché non viene inserito un numero maggiore. Il primo degli otto numeri è per definizione il massimo corrente all'inizio (ancora non sono stati inseriti altri numeri) e quindi viene memorizzato direttamente nella variabile `max`. I numeri successivi vengono inseriti uno alla volta nelle 7 iterazioni[2] del ciclo e, quando uno di essi è maggiore del massimo corrente, viene aggiornato il massimo corrente. La variabile `max` alla fine del ciclo contiene il massimo degli otto numeri letti.

Notare che l'`espr1` contiene una dichiarazione di variabile con inizializzazione. La variabile dichiarata esisterà solamente fino alla fine del ciclo.

Mostriamo un altro esempio di uso di cicli for per effettuare la trasposizione di una matrice. Data una matrice di 9 elementi organizzati in 3 righe per 3 colonne, copiamo i suoi elementi in una nuova matrice scambiando le righe con le colonne. Al termine mostriamo in output le due matrici.

```
//trasp.cpp

#include <iostream.h>

main()
{ int m[3][3], n[3][3];
  int i, j;

  // inizializzazione della matrice m
  for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
      cin >> m[i][j];

  // trasposizione di m in n
  for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
      n[j][i] = m[i][j];
```

[2] il numero di iterazioni è pari a 7 (`i` varia da 1 a 7)

```
  // output
  cout << “Matrice m” << endl;
  for (i = 0; i < 3; i++)
  { for (j = 0; j < 3; j++)
      cout << m[i][j] << '\t';
    cout << endl;
  }
  cout << endl << “Matrice n” << endl;
  for (i = 0; i < 3; i++)
  { for (j = 0; j < 3; j++)
      cout << n[i][j] << '\t';
    cout << endl;
  }
  cout << endl;
}
```

Un output di esempio del programma è il seguente:

```
Matrice m
10   20   30
40   50   60
70   80   90

Matrice n
10   40   70
20   50   80
30   60   90
```

Notare l’uso dell’`endl` e del carattere *arresto di tabulazione* ‘`\t`’ per incolonnare gli elementi delle matrici.

1.5.4 Altri due esempi

Due esempi ulteriori a proposito delle strutture di controllo del flusso del programma sono i seguenti.

Nel primo si effettua una ricerca esaustiva di un elemento in un array di 10 numeri interi. Notare l’uso dell’operatore logico ‘&&’ (and, vedi paragrafo 1.3) .

```
// ric.cpp

#include <iostream.h>

main()
{ float v[10];
  // inizializzazione array
  for (int i = 0; i < 10; i++)
    cin >> v[i];
```

```
  //ricerca elemento
  cout << “Inserire il numero da cercare: “;
  cin >> f;
  cout << endl;
  int j = 0;
  while (j < 10 && v[j] != f)
    j++;
  if (j < 10)
    cout << “Trovato in posizione “ << j << endl;
  else
    cout << “Non trovato.” << endl;
}
```

Il secondo esempio è il gioco “Hi-Lo Game” in cui l’utente pensa un numero tra 0 e 1000 e il programma prova a indovinarlo mediante tentativi successivi. Il programma ad ogni tentativo calcola un numero e lo mostra in output chiedendo all’utente se è troppo alto, troppo basso o corretto. L’utente inserisce il numero 1 se il tentativo è superiore al numero pensato, -1 se è inferiore, e 0 quando il numero pensato viene indovinato.

```
// hilo.cpp
// hi-lo game

#include <iostream.h>

main()
{ int max = 1000, min = 0, n, r = 1, tentativi = 0;
  while (r != 0)
  { n = (max + min) / 2;
    cout << n << endl;
    tentativi++;
    cin >> r;
    if (r == 1)
      max = n;
    if (r == -1)
      min = n;
  }
  cout << tentativi << " tentativi." << endl;
}
```

1.6 Puntatori

Come già accennato nel paragrafo 1.3, la memoria del calcolatore è organizzata in *locazioni* in ognuna delle quali può essere memorizzato un dato. Le locazioni sono identificate da un numero detto *indirizzo*. Il compilatore associerà ogni variabile ad una locazione di memoria e quindi ogni variabile avrà un suo indirizzo.

Data una variabile `a`, è possibile conoscere il suo indirizzo mediante l'operatore '`&`', scrivendo: `&a`.
È possibile stampare in output l'indirizzo di una variabile, ma è conveniente trasformarlo in `int`, altrimenti verrà stampato in formato esadecimale. La trasformazione, detta *cast*, si ottiene anteponendo '`(int)`' al dato da stampare. Il seguente programma stampa in output gli indirizzi delle variabili dichiarate.

```
#include <iostream.h>

main()
{ int a,b;
  float c;
  char d;
  bool e;
  cout << (int)&a << endl << (int)&b << endl;
  cout << (int)&c << endl << (int)&d << endl << (int)&e << endl;
}
```

Si può memorizzare l'indirizzo di una variabile in un'altra variabile di tipo speciale, detta *puntatore*, atta a contenere indirizzi di memoria. Per dichiarare un puntatore è necessario specificare il tipo di dato a cui il puntatore può *puntare*.
Ad esempio

```
int* p;
```

è una dichiarazione di un puntatore che potrà contenere indirizzi di variabili `int`.
Allora il seguente frammento di programma:

```
int a;
int* p = &a;
```

dichiara una variabile intera `a` e una variabile *puntatore a intero* `p` inizializzata con l'indirizzo della variabile `a`.

Supponiamo di dichiarare in un programma tre variabili puntatore ad intero[3]:

```
int* p1;
int* p2;
int* p3;
```

e di dichiarare anche delle variabili e un array `int`:

```
int i = 100, j = 200, v[5] = {5, 10, 15, 20, 25};
```

Le seguenti due istruzioni permettono di stampare il valore di `i`:

[3] È consigliabile dichiarare i puntatori separatamente l'uno dall'altro. Infatti la dichiarazione di puntatori sulla stessa linea può dar luogo a confusione. Notare che: `int* p1, p2, p3;` dichiara in effetti un puntatore a `int` (`p1`) e due variabili `int` (`p2` e `p3`), mentre la dichiarazione corretta su una linea sola è:
`int* p1, * p2, * p3;`

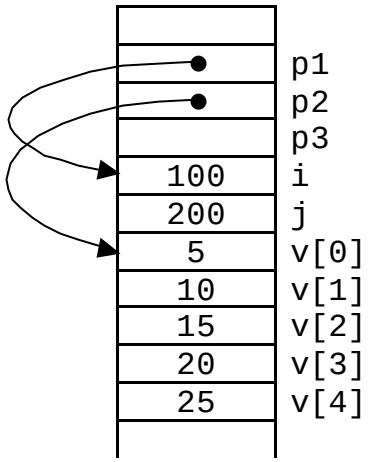
```
p1 = &i;                 // assegna a p1 l'indirizzo di i
cout << *p1 << endl;      // stampa 100, il valore della variabile
                          // puntata da p1 (cioe' i)
```

L'operatore '*' è usato per accedere alla variabile puntata. Notare che lo stesso simbolo '*' è usato per l'operatore di moltiplicazione, per la deferenziazione (accesso alla variabile puntata) e per la dichiarazione di variabili puntatore (come in `int* p1`), ma non c'è mai ambiguità in quanto è possibile distinguere un operatore dall'altro dal contesto in cui si trova.

In modo simile a quanto fatto sopra possiamo accedere agli elementi dell'array:

```
p2 = &v[2];              // assegna a p2 l'indirizzo del terzo
                         // elemento dell'array v
cout << *p2 << endl;     // stampa 15, il valore di v[2]
p2 = &v[0];              // assegna a p2 l'indirizzo di v[0]
cout << *p2 << endl;     // stampa 5, il valore di v[0]
```

In figura vediamo un possibile *layout* di memoria in cui sono rappresentati i valori dei puntatori con una freccia che indica la variabile puntata.



In C++ il nome di un array è il puntatore alla prima locazione dell'array stesso; cioè l'indirizzo iniziale dell'array (che coincide con l'indirizzo del suo primo elemento) è indicato dal nome dell'array senza parentesi quadre.
Le due righe di codice seguenti hanno l'effetto di stampare il valore 5, e sono equivalenti alle ultime due righe mostrate sopra:

```
p2 = v;                  // equivalente a p2 = &v[0];
cout << *p2 << endl;     // stampa 5
```

In C++ vi è una stretta relazione tra puntatori e nomi di array, e un puntatore può essere utilizzato come nome di un array, come mostrato nel frammento di codice seguente:

```
cout << p2[2] << endl;  // stampa 15, il valore di v[2]
```

In effetti p2[2] è il valore della locazione di memoria che si trova all'indirizzo ottenuto sommando 2 al contenuto di p2, e questo lo si può scrivere anche così:

```
cout << *(p2 + 2) << endl;   // stampa 15
```

Per assegnare dei valori alle variabili puntate si usa l'operatore '*', come mostrato nel frammento seguente in cui si assegna il valore 1000 a `i` e a `v[0]`, il valore 200 a `v[4]` e si incrementa di 200 `v[3]`:

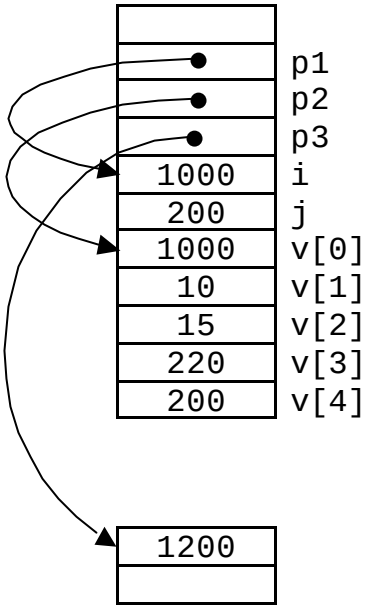
```
*p1 = 1000;
*p2 = *p1;
p2[4] = j;
*(p2 + 3) = p2[3] + j;
```

1.7 Memoria dinamica

È possibile richiedere la creazione di una variabile o di un array a run time, usando l'operatore `new` e specificando il tipo della variabile e il numero di elementi per un array. Ad esempio:

```
p3 = new int;
```

crea una nuova variabile `int` in una locazione libera di una parte della memoria detta *memoria dinamica* o *heap* o *free store* e assegna al puntatore `p3` l'indirizzo di tale locazione. Infatti l'operatore `new` restituisce l'indirizzo della locazione di memoria in cui viene *allocata* la nuova variabile.



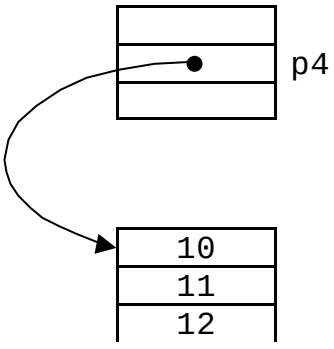
A questo punto è possibile utilizzare la variabile puntata da p3 mediante l'operatore di deferenziazione '*', come nel frammento seguente, nel quale si assegna il valore 1200 alla nuova variabile e poi lo si stampa:

```
*p3 = i + j;
cout << *p3 << endl;
```

Il layout della memoria a run time è mostrato nella figura a lato.

In maniera simile è possibile creare dinamicamente un array, come mostra il seguente esempio e la relativa figura:

```
int* p4 = new int[3];

p4[0] = 10;
p4[1] = 11;
p4[2] = 12;
```



Infine, mediante l'operatore `delete`, è possibile liberare la memoria dinamica utilizzata. Ad esempio, le due istruzioni seguenti

```
delete p3;
```

```
delete [] p4;
```

rilasciano la memoria utilizzata dalle variabili puntate dai puntatori `p3` e `p4` (per eliminare la memoria dinamica utilizzata da un array è necessario usare le parentesi quadre dopo il `delete`, senza specificare le dimensioni dell'array).

Notare che l'operatore `delete`:

- dichiara libera la zona di memoria utilizzata dall'*oggetto* puntato; tale zona di memoria dinamica (una o più locazioni) potrà essere utilizzata da un successivo operatore `new`
- non cancella il contenuto delle locazioni di memoria rilasciate
- non cancella il contenuto della variabile puntatore

Per questo è possibile commettere errori continuando ad utilizzare il puntatore dopo aver effettuato `delete`. Ad esempio il seguente codice provoca un errore del quale non ci si rende conto immediatamente:

```
int* p5 = new int;
*p5 = 20;
delete p5;              // dichiara libera la locazione puntata
int* p6 = new int[3];   // supponiamo che la new assegni a p6 la
                        // locazione che si è appena liberata e
                        // le due successive
p6[0] = 40;
cout << *p5 << endl;    // p5 punta ancora quella locazione;
                        // viene stampato il valore 40
*p5 = 60;               // modifica il valore della locazione
cout << p6[0] << endl;  // stampa 60; il valore di p6[0] è
                        // cambiato inaspettatamente
```

Per evitare questo tipo di errori, è buona norma azzerare il puntatore dopo la `delete`, assegnandogli il valore predefinito `NULL`, come segue:

```
delete p5;
p5 = NULL;
```

L'uso del free store permette di ottimizzare l'uso della memoria allocando il numero di locazioni strettamente necessario e liberandole quando non sono più utilizzate. Inoltre il numero di locazioni da allocare può essere determinato a run time a seconda delle reali esigenze del programma.

L'esempio seguente alloca, inizializza e poi stampa in output un vettore dinamico la cui lunghezza è scelta a run time dall'utente.

```
// dynvec.cpp

#include <iostream.h>

main()
{ cout << "Inserire la lunghezza del vettore: ";
  int i;
```

```
  cin >> i;
  // allocazione
  int* v = new int[i];
  // inizializzazione
  for (int j = 0; j < i; j++)
    v[j] = i * j;
  // stampa
  for (int j = 0; j < i; j++)
    cout << “v[“ << j << “] = ” << v[j] << endl;
  delete [] v;
  v = NULL;
}
```

Il programma seguente invece utilizza array, puntatori e memoria dinamica per manipolare delle stringhe di caratteri.
Le stringhe possono essere realizzate in C++ come array di char. Ogni elemento dell'array è un carattere della stringa. Una stringa deve essere conclusa con un carattere speciale detto *terminatore di stringa*, che si indica con '\0' e corrisponde al carattere ASCII 0. L'operatore '<<' è in grado di inviare a cout una stringa della quale si fornisce il puntatore al primo carattere. La stringa verrà stampata carattere dopo carattere finché non si incontra il terminatore.

In questo esempio si creano due stringhe contenenti le parole "prima" e "seconda" e poi si costruisce una terza stringa – realizzata mediante un array dinamico – nella quale si inserisce la parola "primaseconda". Le variabili `len1` e `len2` vengono utilizzate per calcolare e memorizzare le lunghezze (in inglese *length*) delle due stringhe.

```
// str.cpp

#include <iostream.h>

main()
{ int len1 = 0, len2 = 0;
  char s1[] = {'p', 'r', 'i', 'm', 'a', '\0'}  ;
  char s2[] = {'s','e','c', 'o', 'n', 'd', 'a', '\0'};
  char *s3;
  while (s1[len1] != '\0')
    len1++;
  while (s2[len2] != '\0')
    len2++;
  cout << "len1 = " << len1 << "\tlen2 = " << len2 << endl;
  cout << "stringa 1 = " << s1 << endl;
  cout << "stringa 2 = " << s2 << endl;
  s3 = new char[len1+len2+1];
  int i;
  for (i = 0; i < len1; i++)
    s3[i] = s1[i];
  for (i = 0; i < len2; i++)
    s3[len1+i] = s2[i];
  s3[len1+len2] = '\0';
  cout << s3 << endl;
```

```
}
```

L'output di questo programma è il seguente:

```
len1 = 5  len2 = 7
stringa 1 = prima
stringa 2 = seconda
primaseconda
```

1.8 Riferimenti

Un *riferimento* è un nome alternativo per una variabile. Ad esempio possiamo dichiarare un riferimento y per una variabile x come segue:

```
int x;
int& y = x;
```

La dichiarazione di un riferimento è sempre una dichiarazione con inizializzazione.

Dopo la dichiarazione, x e y rappresentano la stessa variabile, come mostrato nel seguente frammento di codice:

```
x = 10;
y++;
cout << x << endl;          // stampa 11
```

I riferimenti sono utilizzati prevalentemente nel passaggio di parametri nelle funzioni, come mostrato nel paragrafo 3.2.

2 Modularizzazione

2.1 Paradigma client – server

Quando si vuole realizzare un'applicazione reale, che coinvolga una discreta quantità di dati ed esegua su di essi molte operazioni diverse, ognuna delle quali sia di una certa complessità, uno dei problemi con cui ci si scontra è la gestibilità di tutto il sistema.
In particolare, gli obiettivi da raggiungere nella realizzazione di un programma reale sono:

- poter progettare in dettaglio le diverse funzioni del sistema mantenendo però una visione generale del progetto
- realizzare il progetto nel linguaggio di programmazione scelto, esprimendosi nella maniera più diretta possibile
- poter realizzare il progetto in più persone, senza che le interazioni necessarie tra le diverse persone crescano esageratamente, allungando eccessivamente i tempi di realizzazione
- poter collaudare il sistema effettuando delle prove ridotte, ad esempio sulle singole funzioni
- poter modificare alcune parti del sistema secondo le necessità di manutenzione o di aggiornamento che si presentano, senza dover ripensare tutto
- assicurarsi che il funzionamento del programma sia comprensibile da altri o dai suoi stessi programmatori anche dopo mesi o anni

Per raggiungere questi obiettivi è opportuno adottare una metodologia adeguata ed è importante dotarsi di strumenti adeguati, ad esempio di un linguaggio di programmazione che offra delle strutture di controllo del programma e dei dati che permettano di implementare praticamente la metodologia adottata.

Un primo importante criterio da seguire nella progettazione e nella realizzazione di un programma è quello della *decomposizione*.

La soluzione di un problema o la manipolazione di informazioni atta a raggiungere un dato obiettivo è generalmente costituita da più funzioni diverse che cooperano sinergicamente. È importante che il progetto sia decomposto nelle diverse parti che rappresentano tali funzioni e che il programma rispecchi questa divisione mediante le strutture più idonee fornite dal linguaggio.
Ad esempio, è facile capire che un programma vero, anche di piccole dimensioni ma che non sia un semplice programma d'esempio, non può essere realizzato come un unico main. Se così fosse, non si riuscirebbe forse a raggiungere nessuno degli obiettivi citati sopra.
Si pensi infatti che la lunghezza di un programma C++ realistico di complessità medio-bassa è dell'ordine delle 10000 linee di codice, mentre un programma complesso raggiunge anche le centinaia di migliaia o il milione di linee.

Per poter avere una visione generale del sistema è poi importante *astrarre* dai dettagli delle singole funzioni, e vedere ogni parte in cui il sistema è stato diviso come una scatola nera. *L'astrazione* svolge un ruolo fondamentale durante la fase di progettazione, perché permette di concentrarsi sulle caratteristiche salienti del problema e di concepire le parti essenziali del progetto, e basarsi su di esse per creare il quadro d'insieme del sistema.

Un modello utilizzato largamente nella progettazione di sistemi informatici di qualsiasi livello è il paradigma *client – server*. Tale modello si basa sull'assunto che, se si decompone un programma nelle varie funzioni che realizzano l'obiettivo generale, ogni funzione è realizzata da una parte specifica del programma, e i risultati del lavoro di questa parte sono usati da una o più parti differenti del programma che ne hanno bisogno per realizzare i loro rispettivi compiti.
Pensando una funzione come un *servizio*, si chiama *server* la parte del programma che lo realizza e lo offre, e *client* ognuna delle parti del programma che ne usufruiscono.

Ad esempio, se un programma deve effettuare il controllo di un testo alla ricerca di eventuali errori di ortografia, ci sarà una parte del programma la cui funzione è quella di gestire il vocabolario della lingua in cui il testo è scritto. Il servizio fornito da questa parte del programma consiste nel cercare una parola nel vocabolario e, in caso che non esista, cercare le parole ortograficamente simili ad essa, le parole "vicine". Un'altra parte del programma scandisce il testo e, per ogni parola, richiede il servizio del vocabolario. Se il vocabolario indica che la parola non esiste e fornisce un insieme di parole vicine, questo insieme sarà proposto all'utente per scegliere la parola corretta. In questo esempio il vocabolario è il server, mentre l'altra parte è il client.

Il modello client – server si adatta bene sia ad un singolo programma che a un insieme di programmi che cooperano, che possono anche trovarsi su calcolatori diversi connessi in rete. Un esempio classico è l'accesso a una pagina internet: la pagina che abbiamo richiesto viene inviata in rete da un calcolatore sul quale gira un programma server, e viene mostrata sul nostro schermo dal browser, cioè il programma client che gira sul nostro calcolatore.

Molto spesso una parte di programma è allo stesso tempo server e client: infatti, per fornire il servizio, deve avvalersi – come client – dei servizi offerti da altre parti del programma.

Quindi la *modularizzazione*, cioè la divisione del programma in moduli che svolgono le diverse funzioni, è un aspetto fondamentale della progettazione. Nel seguito di questo capitolo analizziamo i criteri su cui basarsi per effettuare correttamente l'astrazione e la decomposizione nella fase di progetto, e mostriamo una panoramica degli strumenti che il C++ offre per realizzare la modularizzazione e per implementare il paradigma client – server.

2.2 Coesione, interfacciamento, accoppiamento e information hiding

Per poter effettuare correttamente la decomposizione del programma in moduli indipendenti, è interessante analizzare quali sono le caratteristiche salienti che un modulo deve avere e quindi desumere da queste i criteri che guidano la modularizzazione.

Un modulo è una parte di programma che svolge una precisa funzione e che quindi offre dei servizi (come server) agli altri moduli del programma. Per fare questo il modulo si avvale dei servizi di altri moduli nei confronti dei quali è client.
L'insieme dei servizi offerti dal modulo e le modalità mediante le quali tali servizi possono essere ottenuti ed utilizzati da altri moduli costituiscono *l'interfaccia* del modulo.

Quattro criteri possono guidare nella modularizzazione: la coesione, l'interfacciamento, l'accoppiamento e l'information hiding [CLNS]. Questi criteri si riferiscono alle caratteristiche interne del modulo e alle sue interazioni con gli altri moduli del programma.

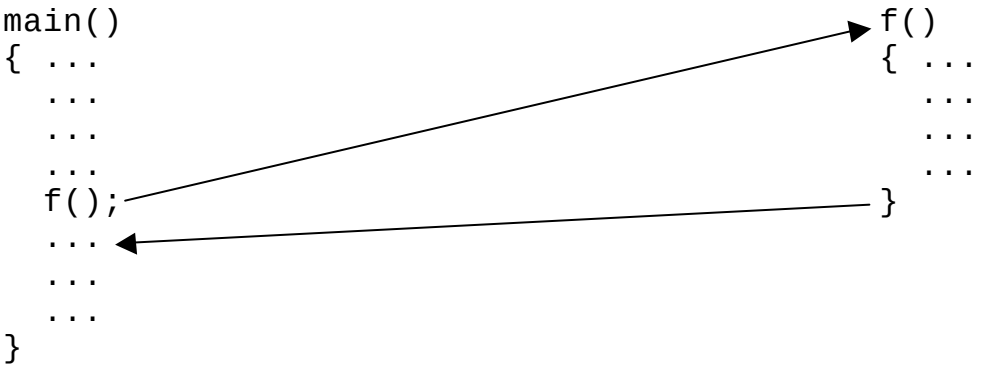
- Coesione. Il servizio offerto da un modulo può comprendere più funzioni, ma è opportuno che queste siano fortemente correlate tra loro, cioè che si riferiscano ad uno stesso aspetto del problema da risolvere. Il fatto di racchiudere in un modulo delle funzioni omogenee fra loro rende il modulo più distinto ed indipendente dagli altri, quindi migliora la modularizzazione.
- Interfacciamento. Per permettere ai moduli client di usufruire dei servizi forniti dal server, è necessario specificare chiaramente il modo di accedere ai servizi. L'interfaccia specifica in che modo il modulo è in relazione con gli altri moduli del programma. Un interfaccia ben definita migliora la qualità della modularizzazione, perchè delinea chiaramente la frontiera di separazione tra i moduli.
- Accoppiamento. L'accoppiamento indica il grado di interazione non esplicita fra un modulo e gli altri. È importante che l'accoppiamento sia tenuto basso, per ridurre al minimo le dipendenze nascoste tra il modulo e il resto del programma. In presenza di tali dipendenze, infatti, un cambiamento in una parte del programma esterna al modulo potrebbe provocare degli errori di funzionamento non voluti all'interno del modulo. Inoltre, un modulo dipendente dal resto del programma in maniera non esplicita è molto difficilmente estraibile dal programma e riusabile in altri contesti.

- Information hiding. L'ultimo parametro indica quanto sia importante nascondere (hiding, dall'inglese: nascondere) all'esterno i dettagli del funzionamento del modulo. Cioè, è importante evitare che i moduli esterni possano accedere ai dati o ai servizi del modulo senza utilizzare l'interfaccia. Questo da un lato previene l'accoppiamento e dall'altro aumenta la possibilità di modificare il funzionamento interno del modulo lasciando invariata solamente l'interfaccia, ad esempio per aumentare l'efficienza del modulo o per correggere eventuali malfunzionamenti.

Per una buona modularizzazione è quindi importante mantenere alte la coesione, l'interfacciamento e l'information hiding e tenere più basso possibile l'accoppiamento.

2.3 Modularizzazione mediante funzioni

Una *funzione* in C++ è una porzione di programma che effettua un determinato compito e che può essere chiamata ed eseguita una o più volte durante lo svolgimento del programma principale.



Un compito tipico, molto semplice, di una funzione è quello di calcolare un valore a partire da un dato di ingresso.

Ad esempio è possibile scrivere un programma che, mediante una funzione, calcola il cubo di un numero intero.
La funzione viene *chiamata* dall'interno del main, e le viene passato dal main un parametro, il numero di cui calcolare il cubo. L'esecuzione del main viene interrotta e viene eseguita la sequenza di istruzioni che compongono la funzione. Al termine di queste istruzioni il risultato (il cubo del numero dato) viene restituito al main che può usarlo nelle sue istruzioni successive.

In questo caso la funzione è server e il main è client. L'interfaccia della funzione è costituita dal suo nome, dal tipo degli *argomenti* o *parametri*, (i dati di ingresso e uscita, in questo caso due numeri interi) e dalle modalità con cui gli argomenti devono esserle passati.

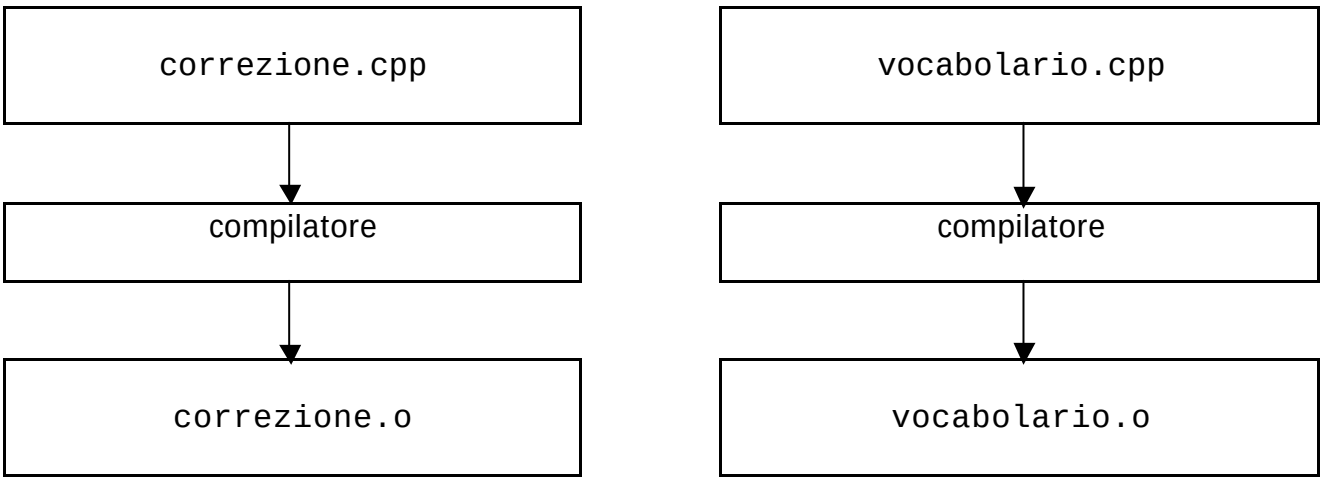
I criteri esposti nel paragrafo precedente, applicati alle funzioni C++, suggeriscono che la funzione esegua un solo compito chiaro e preciso (per mantenere alta la coesione), che la sua interfaccia epliciti chiaramente i parametri di ingresso e di ritorno (vedi capitolo 3) e

che la funzione, al fine minimizzare l'accoppiamento, usi solo i parametri definiti nell'interfaccia per comunicare con il resto del programma (e non, ad esempio, variabili *globali* accessibili anche dal main, vedi paragrafo 3.4.1). Infine l'information hiding è garantita a patto di usare all'interno della funzione variabili *locali* non accessibili dall'esterno (vedi paragrafo 3.4.1).

2.4 Modularizzazione mediante file

Appena un programma C++ cresce di dimensioni e di complessità, è opportuno separare il suo codice in più file.

Riprendendo l'esempio del programma per la correzione degli errori ortografici, possiamo supporre di realizzare il servizio vocabolario mediante due funzioni C++, una delle quali si occupa di cercare una parola data, mentre l'altra si occupa di cercare parole simili a quella data. Queste due funzioni potrebbero essere scritte in un file a se stante, che quindi realizzerebbe il modulo relativo al vocabolario. Chiamiamo questo file `vocabolario.cpp`. In un altro file potremmo invece scrivere il main e eventuali altre funzioni non direttamente connesse alla gestione del vocabolario. Chiamiamo questo file `correzione.cpp`.
Il compilatore allora viene utilizzato due volte, per compilare separatamente ognuno dei due file. Nel file `correzione.cpp` devono essere dichiarate, cioè riportate, le interfacce delle due funzioni contenute nell'altro file; infatti è necessario che il compilatore conosca tali interfacce durante la compilazione di `correzione.cpp` per poter compilare correttamente le chiamate alle funzioni effettuate dall'interno del main.
Il compilatore produce due file *oggetto* (`correzione.o` e `vocabolario.o`) che poi devono essere collegati insieme per formare il programma eseguibile (come spiegato nel paragrafo successivo).

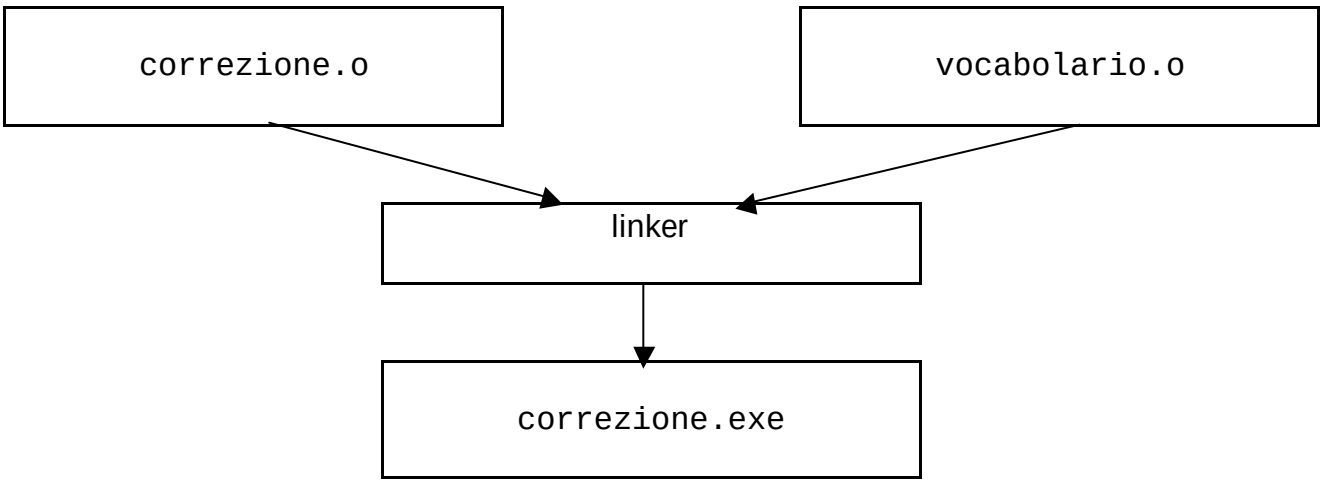


In un file è possibile dichiarare delle variabili globali utilizzabili da tutte le funzioni, per mantenere dei dati comuni (vedi paragrafo 3.4.1). L'accesso a queste variabili può essere negato alle funzioni esterne al file dichiarando le variabili come *statiche*. Questa tecnica corrisponde ad effettuare information hiding nella modularizzazione mediante file.

L'interfaccia del modulo è data dall'interfaccia delle funzioni contenuto in esso e dalle eventuali variabili globali non statiche. La coesione si ottiene racchiudendo nel modulo tutto e solo ciò che riguarda un dato servizio, mentre l'accoppiamento si minimizza limitando il numero di variabili globali non statiche allo stretto necessario.

2.5 Linking

Il linking è l'operazione di collegamento di più file oggetto per formare un unico file eseguibile. Il linker controlla che tutte le le funzioni dichiarate in un file siano definite (cioè siano presenti) in uno e uno solo degli altri file. Infatti mentre il compilatore ha solamente bisogno di sapere che una funzione esiste e di conoscere la sua interfaccia per poter compilare le istruzioni di chiamata della funzione, il linker deve sapere in quale file oggetto è definita la funzione per poter collegare l'istruzione di chiamata ad essa con le istruzioni le corpo delle funzione che devono essere eseguite a runtime.
Il linker infine ha il compito di stabilire definitivamente le locazioni di memoria nelle quali devono essere allocate le variabili e di sostituire questi indirizzi agli indirizzi provvisori usati dal compilatore nei file oggetto.



Al programma eseguibile si dà generalmente lo stesso nome del file che contiene il main, anche se è possibile scegliere un altro nome.

2.6 Modularizzazione mediante tipo astratto

Un tipo astratto di dato è un insieme di funzioni e di domini (cioè di tipi) su cui queste funzioni agiscono. Lo scopo di un tipo astratto è quello di permettere di rappresentare un nuovo tipo, per manipolare in un programma oggetti più complessi.

Ad esempio si potrebbe definire il tipo astratto "punto", che rappresenta i punti dello spazio tridimensionale. Ogni oggetto di questo tipo rappresenta un punto, identificato dalle tre coordinate (il tipo delle coordinate è uno dei domini del tipo astratto) e si può

interagire con esso mediante delle funzioni (ad esempio si può definire una funzione che calcola la distanza del punto dall'origine degli assi).

In C++ è possibile definire un nuovo tipo astratto mediante il costrutto `class` (vedi cap. 4) specificando i dati necessari per rappresentare un'istanza del tipo (detta oggetto) e le funzioni utilizzabili per manipolare il tipo. Un tipo astratto in effetti rappresenta una classe di oggetti aventi caratteristiche comuni che possono essere utilizzati in modo uniforme mediante le funzioni.

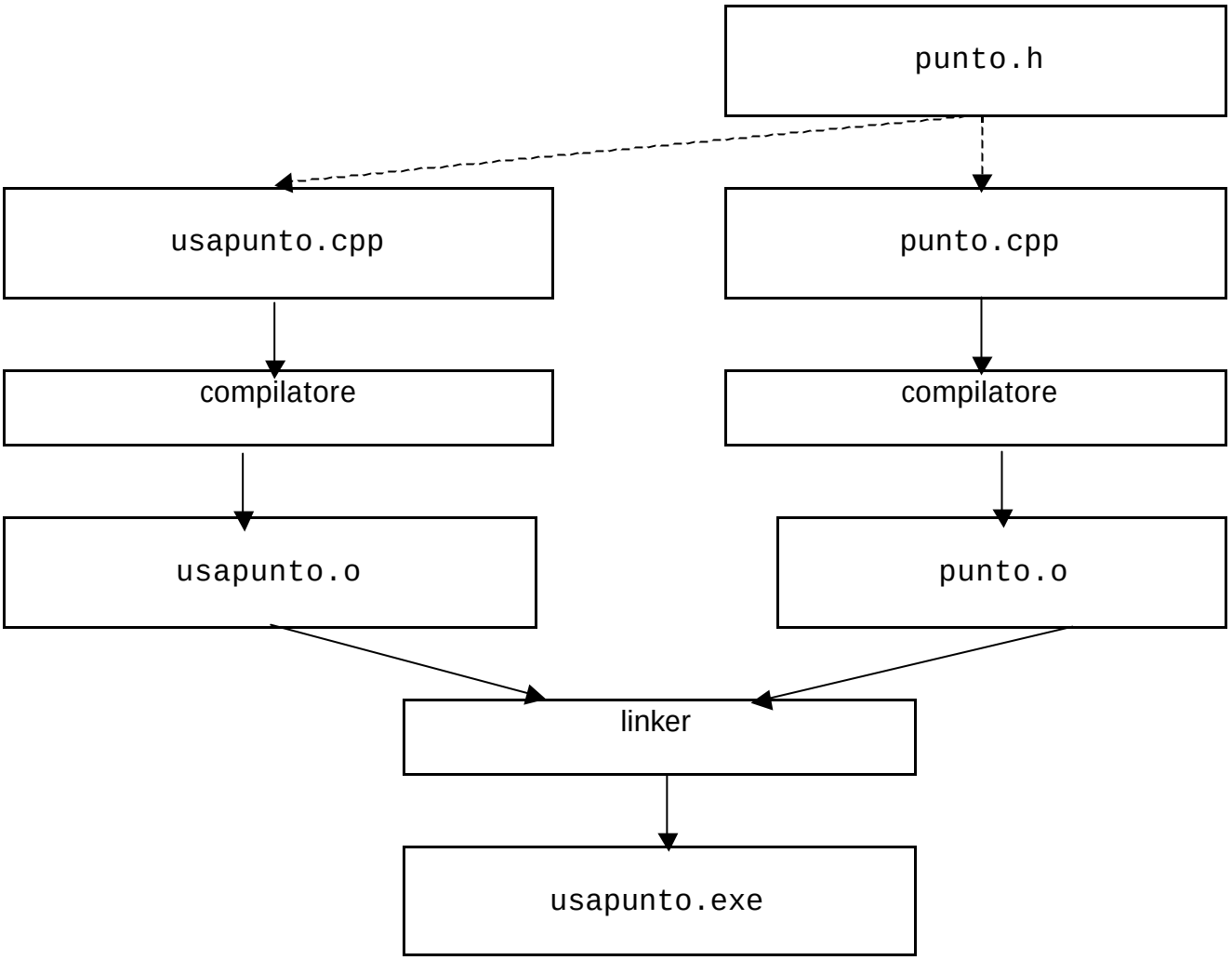
La modularizzazione per tipo astratto avviene incapsulando in una classe tutto ciò che riguarda il nuovo tipo (domini e funzioni) e permettendo al resto del programma di creare nuovi oggetti della classe e di utilizzarli manipolandoli mediante le funzioni della classe.
La coesione è ottenuta progettando la classe in modo che contenga tutto e solo ciò che riguarda un certo tipo di dato; l'interfaccia del modulo è data dalle funzioni della classe; l'accoppiamento può essere minimizzato facendo in modo che gli oggetti del tipo non condividano dati con il resto del programma, e infine l'information hiding si ottiene impedendo che il main o le funzioni esterne alla classe possano accedere a dati o funzioni della classe che non fanno parte dell'interfaccia.

È possibile e consigliabile suddividere la definizione della classe in due parti: da un lato la dichiarazione nella quale sono specificati solamente i dati della classe (i domini del tipo astratto) e l'interfaccia delle funzioni. Dall'altro la definizione delle funzioni, cioè la specifica delle istruzioni che le compongono. Per fare questo si deve inserire la dichiarazione in un file con estensione "`.h`" e la definizione in un file con estensione "`.cpp`". Il file `.h` (header) deve essere incluso sia dal file `.cpp` che dagli altri file che utilizzano la classe. In questo modo il compilatore conosce l'interfaccia della classe durante la compilazione di ognuno dei file che la utilizzano.

Ad esempio possiamo realizzare un programma che effettua operazioni su punti mediante tre file: `punto.h` e `punto.cpp` descrivono la classe, mentre `usapunto.cpp` contiene il main che usa gli oggetti della classe `punto`. La direttiva che si usa per includere un file è la `#include`, nel nostro esempio sia usapunto.cpp che punto.cpp contengono la direttiva:

```
#include "punto.h"
```

Nella figura seguente l'inclusione è rappresentata mediante frecce tratteggiate.



2.7 Modularizzazione mediante namespace

I namespace servono a raggruppare logicamente più definizioni di variabili e di funzioni che sono relative allo stesso servizio. I namespace permettono di specificare una interfaccia per il servizio e permettono di incapsulare in un unico spazio le variabili e le funzioni non locali che sono correlate fra loro. I namespace non saranno trattati nel seguito del testo. Per una trattazione specifica si rimanda a [Stroustrup].

3 Funzioni

3.1 Funzioni in C++

Una *funzione* è una porzione di programma che effettua un determinato compito e che può essere chiamata ed eseguita una o più volte durante lo svolgimento del programma principale. Un compito tipico, molto semplice, di una funzione è quello di calcolare un valore a partire da un dato di ingresso. Ad esempio è possibile scrivere un programma che, mediante una funzione, calcola il cubo di un numero intero:

```
// cubo.cpp

#include <iostream.h>

int cubo(int i)
{ int c = i * i * i;
  return c;
}

main()
{ int a,b;
  cin >> a;
  b = cubo(a);
  cout << b << endl;
}
```

La definizione della funzione comprende l'*intestazione* (la prima riga) e il *corpo* (le istruzioni racchiuse tra parentesi graffe). L'intestazione specifica tre informazioni:

- il tipo del valore restituito dalla funzione (anche detto tipo di ritorno): in questo esempio, `int`
- il nome della funzione: in questo esempio, `cubo`
- i *parametri formali* passati alla funzione, cioè i dati di ingresso sui quali la funzione lavora, racchiusi tra parentesi; in questo esempio l'unico parametro formale è un intero, `int i`

Nel `main()` la funzione viene chiamata specificandone il nome e mettendo tra parentesi il *parametro attuale*, cioè il dato di ingresso da passarle. Il valore del parametro attuale viene copiato nel parametro formale prima di iniziare ad eseguire il corpo della funzione (questo modo di passare i parametri si chiama *passaggio per valore*, vedi 3.2).

La funzione è definita nel programma prima del `main()`, ma il programma inizia sempre dal `main()`. Infatti il main è una funzione speciale, in quanto è quella che viene chiamata dal sistema operativo per lanciare il programma. D'altra parte, il compilatore, per poter compilare le istruzioni del main nelle quali si fa riferimento a una funzione, deve essere a conoscenza dell'esistenza della funzione, cioè deve avere già incontrato la dichiarazione.

Il `main()` di questo programma legge da tastiera un numero intero, lo memorizza in a e chiama la funzione `cubo()` specificando a come parametro attuale. A questo punto l'esecuziuone del `main()` viene temporaneamente interrotta, il valore del parametro attuale a viene copiato nel parametro formale `i`, e si passa all'esecuzione del corpo della funzione. La funzione calcola il cubo moltiplicando `i` per se stessa due volte e memorizza il risultato nella variabile `c`. Poi, mediante l'istruzione `return`, il risultato viene restituito al chiamante, cioè al `main()`. L'istruzione return causa anche la terminazione della funzione e quindi si riprende l'esecuzione del `main()` che, mediante l'operatore '=', assegna a b il valore restituito dalla funzione e poi lo stampa. A questo punto il programma termina.

Notare che la variabile `i` e la variabile c vengono create quando la funzione viene chiamata e vengono eliminate al termine della funzione. Si dice che il loro *ciclo di vita* termina quando termina la funzione (vedi paragrafo 3.4.2).

Analizziamo adesso un altro esempio: vogliamo scrivere un programma che legga da tastiera una parola e che converta in maiuscolo tutti i caratteri alfabetici minuscoli contenuti in essa. Per fare ciò scandiamo la parola e passiamo un carattere alla volta ad una funzione che, se il carattere è minuscolo, ci restituisce il corrispondente carattere maiuscolo, altrimenti ci restituisce il carattere stesso.
Nel corpo della funzione sfruttiamo il fatto che i caratteri sono codificati secondo il codice ASCII. Nella tabella ASCII i caratteri maiuscoli e i caratteri minuscoli hanno codici consecutivi tra loro e sono ordinati alfabeticamente. Pertanto la distanza tra un carattere minuscolo e il corrispondente carattere maiuscolo è costante. Ad esempio il carattere 'A' ha codice ASCII 65, e il carattere 'a' ha codice 97, quindi la loro distanza è 32. Anche tra 'B' (ASCII 66) e 'b' (ASCII 98) c'è distanza 32, e così via per tutte le lettere fino a 'Z' e 'z'. Quindi per ottenere il codice di un carattere minuscolo è sufficiente sottrarre la distanza 32 ('a'-'A') al codice del corrispondente carattere minuscolo.

```
// upcase.cpp

#include <iostream.h>

char up(char c)
{ if (c >= 'a' && c <= 'z')
    c = c – ('a' - 'A');
  return c;
}

main()
{ int i = 0;
  char s[16];
  cout << "Conversione in maiuscole." << endl;
```

```
  cout << “Inserire parola (max 15 caratteri): ”;
  cin >> s;
  while (s[i] != ‘\0’)
  { s[i] = up(s[i]);
    i++;
  }
  cout << s << endl;
}
```

In questo esempio la funzione `up()` viene chiamata tante volte quanti sono i caratteri della parola `s`. Il parametro attuale è `s[i]`, con `i` che varia di volta in volta, mentre il parametro formale è `c`. Ogni volta che la funzione inizia viene creata una nuova variabile c che viene distrutta quando la funzione termina.

L’esempio seguente è un programma che ribalta un array monodimensionale, cioè scambia il posto degli elementi in modo che il primo venga messo in ultima posizione, il secondo in penultima e così via. Questo programma utilizza tre funzioni diverse per realizzare tre diversi compiti: l’inizializzazione dell’array (funzione `init()`), il ribaltamento dell’array (funzione `revvect()`) e la stampa dell’array (funzione `printout()`).
In questo esempio ci sono due novità:

- l’uso di parametri di tipo puntatore (usati per passare l’indirizzo del vettore); questo argomento verrà approfondito in 3.3;
- l’uso della parola chiave `void` per indicare che una funzione non restituisce alcun valore (per esempio, dato che l’unico scopo della funzione `print()` è quello di stampare su `cout` gli elementi dell’array, la funzione non restituisce alcun valore al `main()`e quindi il suo tipo di ritorno è dichiarato `void`);

```
// revvect.cpp

#include <iostream.h>

void print(int* v, int n)
{ cout << endl;
  for (int i = 0; i < n; i++)
    cout << “v[“ << i << “]=” << v[i] << endl;
}

int* init(int n)
{ int* v = new int[n];
  for (int i = 0; i < n; i++)
    v[i] = (n – i) * 10;
  return v;
}

void reverse(int* v, int n)
{ int x;
  for (int i = 0; i < n/2; i++)
  { x = v[i];
```

```
    v[i] = v[n-1-i];
    v[n-1-i] = x;
  }
}

main()
{ cout << "Ribaltamento array monodimensionale." << endl;
  cout << "Inserire la dimensione dell'array: ";
  int dim;
  cin >> dim;
  int* arr = init(dim);
  print(arr,dim);
  reverse(arr,dim);
  print(arr,dim);
}
```

3.2 Passaggio di parametri per valore e per riferimento

Negli esempi del paragrafo precedente abbiamo utilizzato sempre il passaggio di parametri per valore. Questo tipo di passaggio di parametri è chiamato così poiché durante la chiamata della funzione viene copiato il valore del parametro attuale nel parametro formale. Quando la funzione inizia, parametro attuale e parametro formale hanno lo stesso valore: da questo momento in poi, il parametro formale (che è una variabile a se stante) può essere modificato senza che questa modifica si ripercuota sul parametro attuale. Non ci sono variabili in comune tra il `main()` e la funzione, il parametro formale è una variabile a se stante. Notare che se la funzione modificasse il parametro formale, questa modifica non potrebbe influire sul valore del parametro attuale.

L'altro tipo di passaggio di parametri del C++ è il *passaggio per riferimento*, in cui non viene effettuata alcuna copia. Infatti in questo caso il parametro formale è un *alias*, cioè un nome alternativo, del parametro attuale.
Nel passaggio di parametri per riferimento ogni modifica apportata al parametro formale è a tutti gli effetti una modifica al parametro attuale, in quanto si tratta della stessa variabile.
Per specificare un passaggio di parametri per riferimento si usa il simbolo '&', ad esempio `int& i`.

L'esempio seguente mostra il diverso comportamento di due funzioni che usano i due tipi di passaggio di parametri. Per evidenziare il fatto che la prima usa il passaggio per valore e la seconda il passaggio per riferimento, abbiamo chiamato le funzioni "pv" e "pr".

```
#include <iostream.h>

void pv(int x)          // passaggio per valore: COPIA
{ x = x + 1;
  cout << x << endl;    // x == 11
}

void pr(int& y)         // passaggio per riferimento: ALIAS
```

```
{ y = y + 1;
  cout << y << endl;     // y == 11
}

main()
{ int i = 10;
  pv(i);
  cout << i << endl;     // i == 10
  pr(i);
  cout << i << endl;     // i == 11
}
```

All'inizio del `main()` `i` vale 10. Nella chiamata a `pv()` il valore 10 viene copiato in `x`, `x` viene incrementata a 11 e viene stampato 11. Si ritorna al `main()` e si stampa `i` che vale ancora 10. Poi viene fatta la chiamata a `pr()`. Qui `y` vale 10, ma non è una variabile indipendente, bensì un alias di `i`. Allora se `y` viene incrementata di 1, anche `i` risulta incrementata di 1. Perciò all'interno della funzione `pr()` viene stampato 11, poi si torna al `main()` e si stampa 11, il nuovo valore di `i`.

3.3 Passaggio di puntatori

Il passaggio di un puntatore può essere effettuato per valore o per riferimento. Il primo è a tutti gli effetti un passaggio di parametro per valore, ma il valore passato è un indirizzo, che viene utilizzato per accedere all'oggetto puntato, come nell'esempio seguente:

```
void f(int* p)
{ *p = *p + 1;            // modifica l'oggetto puntato
  cout << *p << endl;     // *p == 11
}

main()
{ int i = 10;
  f(&i);
  cout << i << endl;      // i == 11
}
```

Il parametro attuale di questo esempio è l'indirizzo di `i`. La funzione `f()` non modifica il parametro formale `p` (e anche se lo facesse questo non si potrebbe ripercuotere sul parametro attuale poiché si tratta di un passaggio per valore), ma modifica l'oggetto puntato da `p`, cioè la variabile `i` stessa.

Il passaggio per puntatore è molto usato nel caso in cui la funzione deve operare su un array, come visto nel paragrafo 3.1, e come mostrato nell'esempio seguente.

```
int lunghezza(char* s)
{ int i = 0;
  while (s[i] != '\0')
    i++;
  return i;
}
```

```
main()
{ char* stringa = “C++”;     // stringa punta ad un array di char
  int lung = lunghezza(stringa);
  cout << lung << endl;      // stampa 3
}
```

Il passaggio di un puntatore per riferimento può essere usato per modificare il contenuto di un puntatore, ad esempio:

```
void alloca(int*& p, int n)
{ p = new int[n];
}

main()
{ int i;
  cin >> i;
  int* v;
  alloca(v,i);
  for (int j = 0; j < i; j++)
    cin >> v[j];
}
```

In questo caso, `p` è un alias di `v`, e quindi l’assegnazione a `p` modifica in effetti il valore di `v`. Dopo l’esecuzione della funzione, `v` punta ad un vettore di `i` interi, che viene inizializzato mediante il ciclo `for`.

3.4 Visibilità e ciclo di vita delle variabili

3.4.1 Visibilità

La *visibilità* o *scope* di una variabile indica la parte del programma dove la variabile è visibile, cioè la parte di programma che può accedere alla variabile per leggerla o per modificarla.
La visibilità di una variabile in C++ è limitata all’interno del blocco (coppia di parentesi graffe) in cui è stata dichiarata.

Ad esempio, una variabile dichiarata all’interno del `main()` è visibile solo nel `main()` mentre non è visibile dall’interno delle funzioni:

```
#include <iostream.h>

void f()
{ cout << i << endl;  // ERRORE, i non è visibile
}

main()
{ int i = 100;        // i è visibile da qui ...
```

```
  f();
}                       // ... a qui
```

Questo implica che in una funzione può essere dichiarata una variabile (o un parametro formale) che ha lo stesso nome di una variabile del main(). Ovviamente si tratterà di due variabili diverse e ognuna di esse sarà visibile nel blocco in cui è stata dichiarata:

```
#include <iostream.h>

void f(int i)
{ int j;
  j = i * 2;            // sono le i e j dichiarate dentro f()
}

main()
{ int i,j;
  i = 12;               // sono le i e j dichiarate
  j = 34;               // nel main()
  f(i);                 // è la i del main()
}
```

Attenzione: il parametro attuale e il parametro formale usati nell'esempio sono sempre due variabili diverse anche se hanno lo stesso nome.

In C++ è possibile dichiarare variabili al di fuori di ogni funzione. Queste variabili sono *globali*, cioè visibili da ogni funzione del programma, incluso ovviamente il `main()`.

```
#include <iostream.h>

int a = 0;

void f(int i)
{ a = a + i;
}

main()
{ cout << a << endl;              // 0
  for (int i = 0; i < 10; i++)
    f(i);
  cout << a << endl;              // 45
}
```

La possibilità di dichiarare variabili globali deve essere usata con cautela poiché va contro la modularizzazione e nasconde eventuali errori.

3.4.2 Ciclo di vita

Il *ciclo di vita* di una variabile è il periodo compreso tra il momento in cui la variabile viene creata (cioè in cui le viene associata una locazione di memoria) e il momento in cui viene eliminata (e la locazione di memoria viene resa libera e riutilizzabile).

In generale una variabile ha un ciclo di vita che inizia con la dichiarazione e termina con la parentesi graffa di chiusura del blocco in cui è dichiarata. Per esempio, una variabile dichiarata all'inizio di una funzione termina alla fine della funzione stessa, una variabile dichiarata all'interno di un `if` o di un `while` termina alla fine del corpo del `if` o del `while`:

```
#include <iostream.h>

main()
{ int i = 5,j;
  if (i < 10)
  { int k;
    k = i * 2;
    j = k / 3;
  }
  cout << j << endl;  // 3
  cout << k << endl;  // ERRORE, k non è più viva
}
```

Nell'esempio seguente si mostra che una variabile locale di una funzione termina il suo ciclo di vita con la fine della funzione stessa. Se la funzione viene richiamata nel seguito del programma, la variabile locale verrà ri-dichiarata e ri-allocata ex-novo.
Nel `main()` dell'esempio si chiama due volte una funzione `f()`, nella quale esiste una variabile locale `x`.
Nella prima chiamata (o attivazione) della funzione, viene creata la `x`, viene stampata, poi viene incrementata di 10 e poi viene stampata di nuovo.
Dopo la prima chiamata di `f()` si effettua una chiamata a `g()`: è probabile che una delle variabili locali di `g()` occupi la locazione che prima era della `x`, modificandone il valore.
Nella seconda chiamata di `f()`, viene creata una nuova `x`, che molto probabilmente avrà un valore diverso da quello che aveva alla fine della chiamata precedente. Questo può dipendere da vari fattori:

- la nuova `x` potrebbe essere allocata in una locazione diversa da quella della vecchia `x`
- la nuova `x` potrebbe essere allocata nella locazione della vecchia `x`, ma tra una chiamata e l'altra della funzione `f()` la locazione potrebbe essere stata utilizzata da una variabile di un'altra funzione, e quindi modificata
- il compilatore potrebbe effettuare un'inizializzazione a 0 delle variabili `int` locali alle funzioni: in questo caso la `x` verrebbe comunque posta a zero all'inizio della nuova chiamata, perdendo qualsiasi valore precedente

```
#include <iostream.h>

void f()
```

```
{ int x;
  cout << "f(): all'inizio: " << x << endl;
  x = x + 10;
  cout << "f(): alla fine:" << x << endl;
}

void g(int a)
{ int b;
  b = a;
}

main()
{ f();
  g(100);
  f();
}
```

Una variabile all'interno di una funzione può essere dichiarata `static` per indicare che non deve essere distrutta alla fine della attivazione della funzione ma deve essere lasciata viva e riutilizzabile dalle attivazioni successive:

```
#include <iostream.h>

int accumula(int i)
{ static int a = 0;
  a = a + i;
  return a;
}

main()
{ int n;
  for (int i = 0; i < 10; i++)
  { cin >> n;
    accumula(n);
  }
  cout << "Totale = " << accumula(0) << endl;
}
```

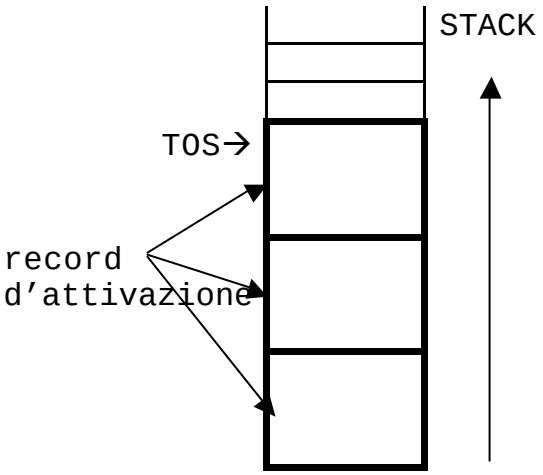
In questo esempio la funzione ha lo scopo di accumulare nella variabile statica a la somma dei valori che le vengono passati nelle dieci chiamate.
Alla prima chiamata la variabile `a` viene inizializzata con il valore 0. Ad ogni chiamata la funzione somma il valore di `i` alla variabile `a`. Al termine viene stampato il valore finale che è la somma dei 10 valori inseriti. La variabile statica viene distrutta soltanto quando il programma termina.
La variabile statica dichiarata all'interno di una funzione ha ciclo di vita uguale a quello di una variabile globale (dall'inizio al termine del programma) ma non è visibile al di fuori della funzione, quindi è migliore ai fini della modularizzazione.

Il ciclo di vita è un concetto distinto dalla visibilità, infatti:

- il fatto che una variabile sia viva in un dato istante non implica che sia visibile dalla porzione di programma che è in esecuzione in quell'istante (si pensi a una variabile globale e una variabile dichiarata in una funzione aventi lo stesso nome: all'interno della funzione sono vive entrambe, ma solo quella locale è visibile; un'altro esempio è l'uso di variabili `static`)
- una variabile locale di una funzione può esistere in più istanze, perché la funzione può essere richiamata più volte (vedi paragrafo 3.5): durante l'esecuzione della funzione tutte le istanze sono vive, ma una sola è visibile.

3.5 Stack e record d'attivazione

Le variabili dichiarate all'interno di una funzione vengono allocate in un area di memoria particolare detta stack (in italiano: pila). Ogni volta che viene chiamata una funzione, viene riservato sullo stack un *record d'attivazione*, cioè un gruppo di locazioni di memoria contigue che contengono le variabili e i parametri dichiarati nella funzione più altri dati necessari per gestire il ritorno dalla funzione.

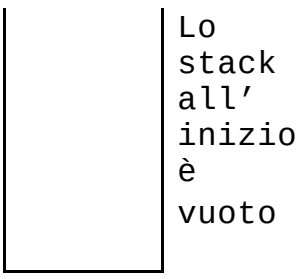


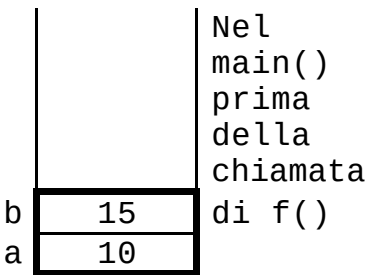
L'area di memoria utilizzata si chiama stack proprio perché è gestita a pila, secondo la logica Last In First Out. I record di attivazione vengono inseriti nello stack uno sopra l'altro. L'ultimo record d'attivazione inserito nello stack è il primo ad essere eliminato, e non è possibile eliminare un record d'attivazione se non è quello *affiorante*, cioè quello che si trova sulla cima della pila, sul *Top Of Stack* (TOS).

Il primo record d'attivazione posto nello stack all'inizio di qualsiasi programma è quello relativo al `main()`.

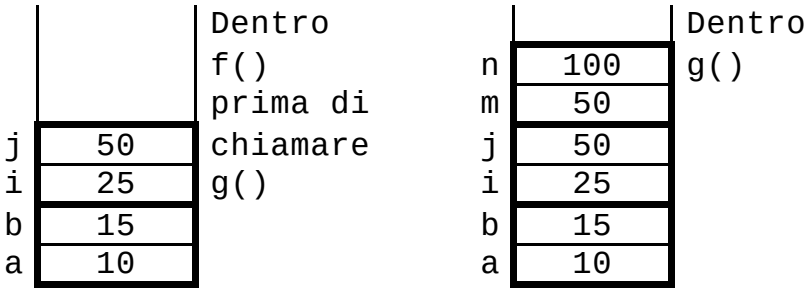
Poi, se il `main()` chiama una funzione `f()` come nell'esempio e nelle figure seguenti, verrà posto sullo stack il record d'attivazione di `f()`. Se `f()` a sua volta chiama una funzione `g()` verrà posto sullo stack il record d'attivazione di `g()`.
I record d'attivazione saranno eliminati dallo stack quando le funzioni terminano, quindi nell'ordine inverso a quello con cui sono stati inseriti. Cioè l'ultimo record d'attivazione inserito (quello di `g()`) sarà eliminato per primo, poi sarà eliminato quello di `f()` e poi quello del `main()`.

```
void g(int m)
{ int n;
  n = m * 2;
}

void f(int i)
{ int j;
  j = i * 2;
```





```
  g(j);
}

main()
{ int a,b;
  a = 10;
  b = 15;
  f(a+b);
}
```



3.6 Ricorsione

- induzione + applicazione princ. ind. per funz ricorsive
- progetto funzioni C++ ricorsive ("la fun già esiste")
- come viene eseguito un programma ricorsivo (esempio stack)

3.6.1 Principio di induzione

Il principio di induzione dice che, dato l'insieme N dei numeri naturali, se:

1. una certa proprietà P vale per il numero 0;
2. per ogni numero i, se P vale per i allora P vale per il successore di i, cioè $i+1$;

allora la proprietà P vale per tutti i naturali.

Le affermazioni 1. e 2. sono dette rispettivamente *passo base dell'induzione* e *passo induttivo*.

Mediante il principio d'induzione possiamo dimostrare che una proprietà è valida per tutti i numeri naturali. Ad esempio, consideriamo la proprietà seguente, dove n è un numero appartenente ad N:

$$P(n) = 0+1+2+...+n = n(n+1)/2$$

Passo base: $P(0) = 0 = 0(0+1)/2$

Passo induttivo: assumiamo $P(i)$ vero, allora:

$$P(i+1) = 0+1+2+...+i+(i+1) = (0+1+2+...+i)+(i+1) = P(i) + (i+1) = i(i+1)/2 + (i+1) = (i(i+1) + 2(i+1))/2 = (i+1)(i+2)/2$$

Quindi $P(i+1)$ è vero.

Quindi $P(n)$ vale per ogni n appartenente ad N.

Si può applicare il principio di induzione anche alla progettazione di algoritmi. Ad esempio, si può definire la somma tra due numeri naturali mediante la seguente funzione matematica:

$$Somma(x,y) = \begin{cases} x & se\ y = 0 \\ succ(Somma(x,pred(y)) & se\ y > 0 \end{cases}$$

dove *succ* e *pred* sono due funzioni che restituiscono rispettivamente il successore e il predecessore di un numero naturale.
In questo modo si definisce la funzione *Somma* mediante se stessa, in maniera *ricorsiva*. L'applicazione del principio di induzione è evidente dal fatto nella definizione di questa funzione abbiamo individuato:

- un passo base (caso $y=0$) in cui il valore della funzione è determinato direttamente (è pari a x)
- un passo induttivo, o anche *passo ricorsivo* (caso $y>0$), in cui il valore della funzione è determinato dal valore della stessa funzione applicata al predecessore

Un altro esempio è la funzione *Molt*, che effettua la moltiplicazione tra due numeri naturali:

$$Molt(x,y) = \begin{cases} 0 & se\ y = 0 \\ Somma(x,Molt(x,pred(y))) & se\ y > 0 \end{cases}$$

Infine la funzione *Esp* calcola il numero ottenuto elevando x alla y-esima potenza:

$$Esp(x,y) = \begin{cases} 1 & se\ y = 0 \\ Molt(x,Esp(x,pred(y))) & se\ y > 0 \end{cases}$$

3.6.2 Funzioni ricorsive in C++

Anche in C++ è possibile definire una funzione ricorsivamente. Una funzione ricorsiva, per risolvere un problema, si basa sulla soluzione di un problema avente le stesse caratteristiche ma minore dimensione o complessità. Per ottenere la soluzione di questo problema la funzione ricorsiva richiama se stessa.
Ovviamente la ricorsione non può essere infinita, cioè la funzione non può richiamare se stessa indefinitamente, ma deve esserci una dimensione base del problema che possa essere risolta in maniera diretta. Questo caso, detto *passo base della ricorsione*, è il corrispondente del passo base dell'induzione visto sopra. L'altro caso, detto passo ricorsivo, corrisponde all'applicazione del passo induttivo, e consiste nell'invocazione della funzione stessa con parametri opportuni.
Facendo uso delle due funzioni C++ `Succ()` e `Pred()`:

```
int Succ(int i)
{ return i+1; }

int Pred(int i)
{ return i-1; }
```

possiamo definire le funzioni ricorsive `Somma()`, `Molt()` e `Esp()` come segue:

```
int Somma(int x, int y)
{ if (y == 0)
    return x;                          // passo base
  else
    return Succ(Somma(x,Pred(y)));     // passo ricorsivo
}

int Molt(int x, int y)
{ if (y == 0)
    return 0;                          // passo base
  else
    return Somma(x,Molt(x,Pred(y))); // passo ricorsivo
}

int Esp(int x, int y)
{ if (y == 0)
    return 1;                          // passo base
  else
    return Molt(x,Esp(x,Pred(y)));     // passo ricorsivo
}
```

Tre accorgimenti possono aiutare nella progettazione di una funzione ricorsiva:

1. individuare il passo base
2. individuare il passo ricorsivo
3. ragionare come se la funzione esistesse già, e quindi utilizzarla all'interno del passo ricorsivo

Spesso utilizzeremo funzioni ricorsive nei capitoli successivi. Riportiamo nel seguito un altro esempio che può aiutare a comprendere più a fondo il meccanismo di progettazione di una funzione ricorsiva. L'esempio si riferisce al calcolo del fattoriale di un numero, (come in 1.5.2), ma questa volta effettuato ricorsivamente.

```
// fatt_r.cpp
// calcolo del fattoriale effettuato mediante funzione ricorsiva

int fatt(int n)
{ int f;
  if (n == 1)           // passo base
    f = 1;
  else                  // passo ricorsivo
    f = n * fatt(n-1);  // si usa fatt() come se già fosse pronta
  return f;
}

main()
{ int i = 4, j;
  j = fatt(i);
}
```

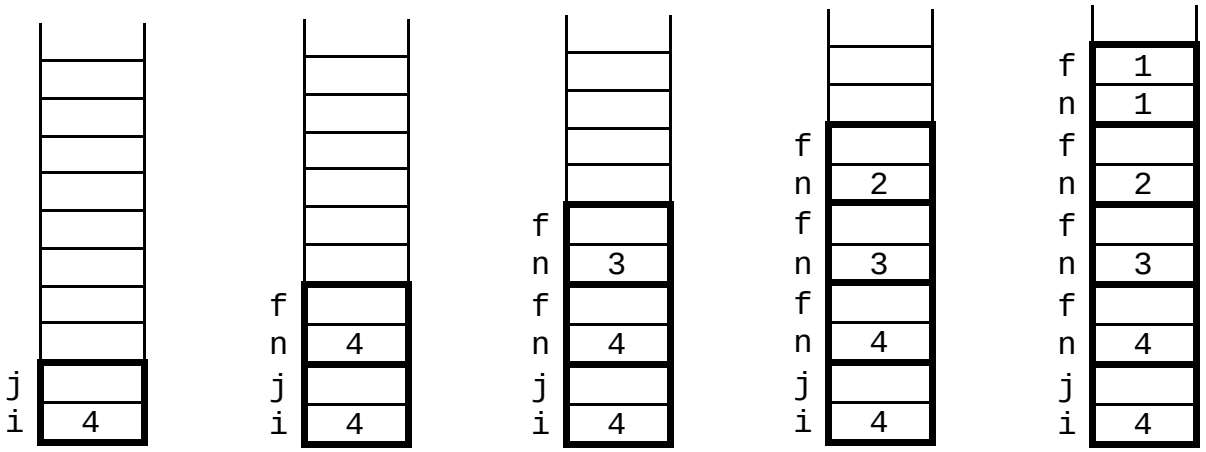
3.6.3 Meccanismo di funzionamento

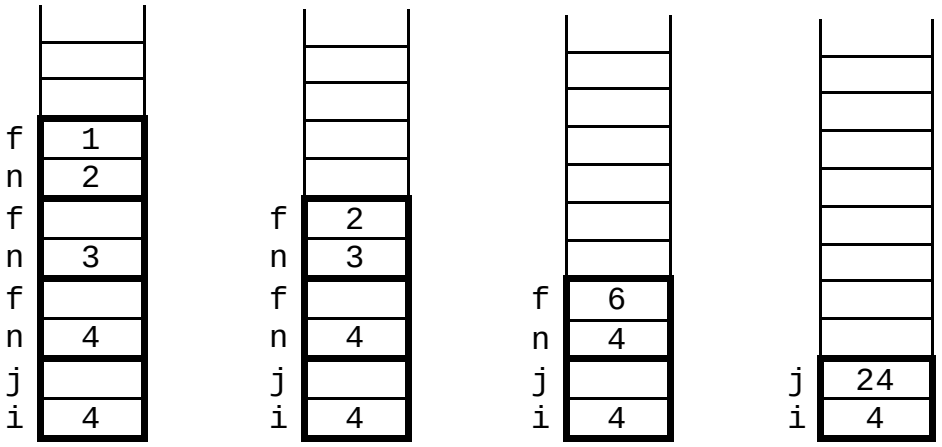
Una funzione ricorsiva chiama se stessa come se chiamasse un'altra qualsiasi funzione.
Nel momento della chiamata viene allocato sullo stack un nuovo record di attivazione per la nuova attivazione della funzione. I record di attivazione per le diverse chiamate hanno lo stesso layout, cioè contengono le stesse variabili, ma ognuno di essi si riferisce ad un'attivazione diversa, e quindi i valori contenuti nelle locazioni di memoria sono diversi.

Durante l'esecuzione di una funzione ricorsiva, quindi, sono presenti nello stack tante istanze di ogni variabile e di ogni parametro dichiarato nella funzione quante sono le attivazioni della funzione in quell'istante. L'attivazione in esecuzione ad un dato istante è quella più recente, cioè l'ultima che è stata chiamata, e il suo record di attivazione si trova sulla cima dello stack.

Le attivazioni della funzione termineranno nell'ordine inverso in cui sono state chiamate, dando luogo alla eliminazione dei rispettivi record di attivazione dallo stack secondo la logica Last In First Out.

Vediamo, come esempio, il meccanismo dettagliato di attivazione della funzione fattoriale definita nel paragrafo 3.6.2, nel caso del calcolo del fattoriale di 4. Mostriamo nelle figure seguenti la situazione dello stack ogni volta che la funzione viene chiamata o ritorna. La prima figura si riferisce alla situazione dello stack durante l'esecuzione del `main()` prima della prima chiamata di `f()`, mentre l'ultima si riferisce alla situazione dello stack nel `main()` prima della fine del programma.




f 1
n 2
f
n 3
f
n 4
j
i 4
f 2
n 3
f
n 4
j
i 4
f 6
n 4
j
i 4
j 24
i 4

4 Classi

Come abbiamo visto nel paragrafo 2.6, la modularizzazione per tipo astratto si basa sulla possibilità di definire in un modulo un nuovo tipo di dato, nascondendo all'esterno tutte le proprietà e i dettagli che riguardano il modo di rappresentare il tipo.
Negli altri moduli si possono creare e usare delle *istanze* del tipo. Le operazioni che si possono effettuare su di esse corrispondono alle funzioni specificate nel tipo astratto.
Per definire un tipo astratto è necessario specificare:

- un insieme di domini, cioè di tipi
- un insieme di costanti, eventualmente vuoto
- un insieme di funzioni definite sui domini

Di domini ne esiste almeno uno, detto dominio di interesse, che corrisponde al tipo astratto stesso, cioè raccoglie tutto il tipo astratto.

Per esempio, il tipo astratto "booleano" può essere definito come segue:

Tipo Astratto Booleano

Domini:	*Booleano (dominio di interesse)*
Costanti:	*vero, falso*
Funzioni:	*and: (Booleano × Booleano)→ Booleano*
	or: (Booleano × Booleano)→ Booleano
	not: (Booleano)→ Booleano

In questo tipo astratto c'è un solo dominio (il dominio di interesse), due costanti e tre funzioni. Le due funzioni *and* e *or* sono definite sul prodotto cartesiano *Booleano* × *Booleano* e hanno *Booleano* come codominio. Quindi richiedono due argomenti di tipo *Booleano* e restituiscono un risultato *Booleano*. La funzione *not* richiede invece un solo argomento. Le operazioni svolte dalle tre funzioni non sono qui specificate poiché ben note.

Come ulteriore esempio, il tipo astratto "coppia di numeri interi" può essere definito nel seguente modo:

Tipo Astratto Coppia

Domini:	*Coppia (dominio di interesse) , Intero*
Funzioni:	*Crea: (Intero × Intero)→ Coppia*
	Primo: (Coppia)→ Intero
	Secondo: (Coppia)→ Intero
	Somma: (Coppia × Coppia)→ Coppia

In questo tipo astratto l'insieme delle costanti è vuoto. La prima funzione, *Crea*, è definita sul prodotto cartesiano *Intero* × *Intero* e ha come codominio *Coppia*. Questa funzione crea una coppia formata da due numeri interi. Le seguenti due funzioni, estraggono il primo o il secondo intero da una coppia data. L'ultima funzione somma due coppie ad esempio sommando gli interi corrispondenti. È possibile descrivere più rigorosamente, anche se discorsivamente, le precondizioni e le postcondizioni di ogni funzione (quindi ciò che la funzione fa) nella specifica del tipo astratto (vedi [clns]).

4.1 Classi e oggetti

In C++ i tipi astratti e le loro istanze si possono realizzare rispettivamente mediante le *classi* e gli *oggetti*.

La definizione di una classe descrive il tipo astratto. Una volta definita una classe si possono creare degli oggetti che corrispondono alle istanze del tipo astratto.

Ad esempio, possiamo realizzare in C++ una classe `Coppia`[4] che rappresenta il tipo astratto *Coppia*:

```
class Coppia                                    1
{private:                                       2
  int i,j;                                      3
 public:                                        4
  Coppia(int, int);                             5
  int Primo();                                  6
  int Secondo();                                7
  Coppia Somma(Coppia);                         8
};                                              9
```

Potremo nel seguito creare oggetti di questa classe effettuando una dichiarazione:

```
Coppia c1(10,20), c2(1,3), c3(0,0);
```

[4] La definizione di questa classe verrà spiegata in dettaglio nel seguito.

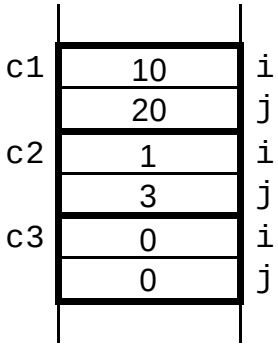
Questa dichiarazione è analoga a una dichiarazione di variabile per un tipo predefinito (quale, ad esempio, `int i1, i2, i3;`). Dopo questa dichiarazione esisteranno tre oggetti di tipo `Coppia` (chiamati `c1`, `c2` e `c3` e inizializzati con i valori (10,20), (1,3) e (0,0)) che potremo utilizzare nel programma alla stessa stregua di tre variabili di un tipo predefinito. In effetti `c1`, `c2` e `c3` sono delle variabili.

4.2 Campi e funzioni proprie

La definizione della classe `Coppia` è costituita:

- dalla parola chiave `class` seguita dal nome della classe (vedi la riga 1 della definizione)
- dalla dichiarazione `private:` (vedi paragrafo 4.4)
- dalla dichiarazione delle variabili `i` e `j`, dette *campi*
- dalla dichiarazione `public:` (vedi paragrafo 4.4)
- dalla dichiarazione delle funzioni: `Coppia()`, `Primo()`, `Secondo()` e `Somma()`
- dal punto e virgola dopo la graffa chiusa che termina la definizione, riga 9 (è uno dei due casi in cui è necessario il punto e virgola dopo una graffa – l'altro è la inizializzazione di un array)

I due campi sono replicati in ogni oggetto della classe, e quindi ogni oggetto potrà avere dei valori diversi in `i` e `j`. Infatti, un oggetto della classe ha una struttura formata da due parti, che rappresentano i due campi. Un oggetto `Coppia` occupa due locazioni di memoria.
Nella figura a lato rappresentiamo il layout di memoria relativo ai tre oggetti `c1`, `c2` e `c3`, evidenziando i due campi in ciascun oggetto.



Le funzioni, invece, sono comuni a tutti gli oggetti della classe e il loro scopo è quello di manipolare gli oggetti della classe. Non possono essere chiamate se non per manipolare un oggetto della classe, e per questo sono dette *funzioni proprie* della classe. Le altre funzioni del programma si dicono funzioni *esterne* rispetto alla classe.

Le funzioni proprie corrispondono una ad una alle funzioni del tipo astratto, come mostra la seguente tabella:

funzione propria della classe `Coppia`	funzione del tipo astratto *Coppia*
`Coppia()`	*Crea*
`Primo()`	*Primo*
`Secondo()`	*Secondo*
`Somma()`	*Somma*

Le funzioni proprie sono solamente dichiarate all'interno della definizione della classe. Nel seguito del programma dovremo definirle, cioè dovremo specificarne il corpo. Nella dichiarazione delle funzioni non abbiamo specificato il nome dei parametri formali, ma solo

il loro tipo. Il nome lo specificheremo nella definizione, dove sarà necessario per poter utilizzare i parametri.

4.3 Costruttore

La prima funzione propria, `Coppia()`, corrisponde alla funzione *Crea*. In C++ una funzione che crea un nuovo oggetto della classe è una funzione particolare detta *costruttore*, e deve avere lo stesso nome della classe.

Il costruttore accetta zero o più argomenti (nel nostro caso ne accetta due, di tipo intero), ma non può avere un tipo di ritorno (per questa ragione il tipo di ritorno non va specificato affatto nella dichiarazione del costruttore, come mostrato nella riga 5 della definizione della classe).

Quando nel programma si dichiara un oggetto della classe, viene automaticamente chiamato il costruttore che provvede a costruire l'oggetto inizializzando i suoi campi nel modo corretto.

Una classe deve sempre avere almeno un costruttore. Se definiamo una classe senza costruttore, il C++ genera automaticamente un costruttore *di default* senza argomenti, che viene chiamato quando dichiariamo un oggetto. Possiamo definire più di un costruttore, a patto che gli argomenti dei vari costruttori siano diversi tra loro (vedi paragrafo 4.8).

4.4 Parte pubblica e parte privata

Le dichiarazioni `private:` e `public:` che abbiamo utilizzato nella definizione della classe `Coppia` servono a gestire l'information hiding.

Tutto ciò che è dichiarato dopo la parola chiave `private:` (nel nostro caso i due campi `i` e `j`) è accessibile solo dalle funzioni proprie della classe.

Tutto ciò che è dichiarato dopo la parola chiave `public:` (nel nostro caso le quattro funzioni) è accessibile anche dal resto del programma.

La parte `public` costituisce l'interfaccia della classe. Nel nostro esempio l'interfaccia è costituita solo dalle quattro funzioni, e quindi si può interagire con un oggetto `Coppia` solo mediante esse. Ad esempio non è possibile leggere i valori dei due campi direttamente, ma lo si può fare soltanto usando le funzioni `Primo()` e `Secondo()`.

4.5 Oggetto d'invocazione

Scriviamo adesso un main che utilizza oggetti `Coppia`. Supponiamo di aver scritto la dichiarazione della classe `Coppia` (esattamente come appare a pag. 52) in un file chiamato `coppia.h` che includeremo dal file contenente il main.

```
// coppia.h

class Coppia
{private:
  int i,j;
 public:
  Coppia(int, int);
  int Primo();
  int Secondo();
  Coppia Somma(Coppia);
};
```

Notare che possiamo scrivere il main anche prima di aver definito le funzioni proprie della classe. Infatti conosciamo già l'interfaccia di tutte le funzioni della parte pubblica, e questo è sufficiente per poterle usare nel programma.

```
// usocoppia.cpp
#include "coppia.h"
#include <iostream.h>

main()
{ Coppia c1(10,20), c2(1,3), c3(0,0);
  int x;
  x = c1.Primo();
  cout << x << endl;                                    // 10
  cout << c1.Primo() << "\t" << c2.Primo() << endl;   // 10   1
  c3 = c1.Somma(c2);
  cout << c3.Primo() << "\t" << c3.Secondo() << endl; // 11   23
}
```

Nella prima riga del main dichiariamo tre variabili `Coppia`, cioè tre oggetti.

Nelle due righe seguenti dichiariamo una variabile intera e le assegniamo il valore del primo intero della coppia `c1` (cioè 10). Per fare questo usiamo la funzione `Primo()` applicata all'oggetto `c1`. La sintassi del C++ richiede che il nome della funzione sia preceduto da un punto e dal nome dell'oggetto cui la funzione si riferisce. Tale oggetto è chiamato *oggetto di invocazione*.

Stampando il contenuto di `x`, otterremo il valore 10.

Alla quinta riga stampiamo i valori dei primi elementi di `c1` e di `c2`, chiamando due volte la funzione `Primo()` e usando come oggetto di invocazione rispettivamente `c1` e `c2`. I risultati restituiti dalla `Primo()` vengono passati a `cout`.

Nella penultima riga del main chiamiamo la funzione `Somma()`. Questa funzione accetta un solo parametro di tipo `Coppia`, lo somma all'oggetto di invocazione (sommando gli elementi corrispondenti, vedi paragrafo successivo) e restituisce un oggetto `Coppia` contenente il risultato. Nella chiamata noi utilizziamo come oggetto di invocazione `c1` e come parametro attuale `c2`. L'oggetto restituito viene quindi copiato nella variabile `c3` dall'operatore '='.

Tutte le funzioni proprie di una classe hanno quindi un parametro sottinteso dello stesso tipo della classe: l'oggetto di invocazione. Pertanto se una funzione di un tipo astratto accetta *n* argomenti del dominio di interesse, la corrispondente funzione della classe accetterà gli *n* argomenti così suddivisi:

- *1* come oggetto di invocazione
- *n-1* come parametri formali

La funzione può leggere e modificare l'oggetto di invocazione, accedendo a tutti i campi pubblici e privati. Da questo punto di vista l'oggetto di invocazione equivale quindi a un oggetto della classe passato per riferimento. Si dice che una funzione fa *side-effect* quando modifica l'oggetto di invocazione. Cioè la funzione, oltre a restituire eventualmente un valore, produce un effetto collaterale che consiste nella modifica dell'oggetto di invocazione.

4.5.1 Puntatore `this`

Nel record di attivazione di una funzione propria viene memorizzato anche un puntatore all'oggetto di invocazione; tale puntatore può essere utilizzato all'interno della funzione, e il suo nome (uguale per tutte le funzioni) è `this`.

4.6 Definizione delle funzioni proprie

A questo punto dobbiamo definire le funzioni proprie della classe, cioè scrivere il corpo di tali funzioni. Scriviamo la definizione delle funzioni in un file separato, chiamato `coppia.cpp`. Questo file deve includere `coppia.h` affinché il compilatore conosca l'interfaccia delle funzioni che dobbiamo definire e possa fare una verifica di consistenza tra dichiarazioni e definizioni.

Nell'intestazione di ogni funzione dobbiamo specificare anche il nome della classe alla quale la funzione appartiene, facendo uso dell'operatore '::'.

```
NomeClasse::NomeFunzione
```

Infatti, in programmi più complessi, è possibile che due classi abbiano due funzioni proprie con lo stesso nome, ed è quindi necessario distinguerle.

```
// coppia.cpp

#include “coppia.h”

Coppia::Coppia(int a, int b)
{ i = a;
  j = b;
}

int Coppia::Primo()
{ return i; }

int Coppia::Secondo()
{ return j; }

Coppia Coppia::Somma(Coppia c)
{ Coppia ris(0,0);
  ris.i = i + c.i;
  ris.j = j + c.j;
  return ris;
}
```

Analizziamo le definizioni delle quattro funzioni:

- Il costruttore riceve in ingresso due parametri interi `a` e `b`. Questi due parametri vengono usati per inizializzare i due campi `i` e `j` dell’oggetto che stiamo costruendo. Quindi, a seguito della dichiarazione `Coppia c1(10,20)` viene chiamato il costruttore, i valori 10 e 20 vengono copiati nei parametri formali del costruttore e da qui vengono copiati nei due campi del nuovo oggetto.

- La funzione `Primo()` non ha parametri formali. Però, essendo una funzione propria della classe, viene sempre invocata specificando un oggetto di invocazione. Una funzione propria può accedere a un campo dell’oggetto di invocazione direttamente specificando il nome del campo. Infatti la funzione `Primo()` accede al valore del campo `i` e restituisce tale valore al chiamante mediante l’istruzione `return`.

- La funzione `Secondo()` è analoga alla funzione `Primo()`.

- La funzione `Somma()` ha lo scopo di sommare i campi corrispondenti di due oggetti `Coppia`. Gli oggetti considerati sono:
  - l’oggetto di invocazione
  - l’oggetto c ricevuto come parametro formale

  La funzione dichiara, nella prima riga, un nuovo oggetto `Coppia` da utilizzare per memorizzare il risultato della somma. Questo oggetto viene chiamato `ris` ed ha,

all'inizio, entrambi i campi uguali a zero. Nella riga successiva viene assegnato al campo `i` di `ris` il risultato della somma del campo `i` dell'oggetto di invocazione e del campo `i` dell'oggetto `c`. Per indicare un campo di un oggetto diverso dall'oggetto di invocazione è necessario specificare il nome dell'oggetto seguito da un punto e poi dal nome del campo: ad esempio, `ris.i` oppure `c.i`.
Similmente, nella riga successiva, vengono sommati i campi `j`, e infine l'oggetto `ris` viene restituito al chiamante mediante la `return`.

Dopo aver scritto questo file abbiamo tutto il necessario per compilare il programma.
A seconda del compilatore che usiamo, dobbiamo creare un progetto e effettuare un build, oppure compilare separatamente i due file `.cpp` e poi linkarli per ottenere `usocoppia.exe` (vedi appendice 7.1).

4.7 Oggetto d'invocazione costante e parametri costanti

È consigliabile dichiarare esplicitamente se una funzione propria non modifica l'oggetto di invocazione. Infatti, in presenza di tale di tale dichiarazione, il compilatore effettua un controllo all'interno del corpo della funzione e segnala come errori eventuali modifiche all'oggetto di invocazione fatte accidentalmente. Inoltre tale dichiarazione fornisce un'informazione utile a chi deve sviluppare moduli che utilizzano la classe.

Infine tale dichiarazione è necessaria quando si vogliono utilizzare oggetti dichiarati costanti, argomento che non tratteremo; ulteriori dettagli cfr., ad esempio, [Stroustrup].

Per effettuare questa dichiarazione si deve usare la parola chiave `const` sia nella definizione della classe che nella definizione della funzione, come segue:

- nella definizione della classe:
```
class Coppia
{ // ...
  int Primo() const;
  // ...
};
```

- nella definizione della funzione:
```
int Coppia::Primo() const
{ // ...
}
```

Anche i parametri formali passati per riferimento possono essere dichiarati costanti, cioè non modificabili. Questo permette di effettuare un passaggio di parametri più efficiente (poiché non richiede la copia del valore dal parametro attuale a quello formale) evitando il rischio di modifiche non volute del parametro attuale[5]. In questo caso la parola chiave

[5] Questa dichiarazione può essere applicata sia a funzioni proprie di una classe che a funzioni esterne.

const deve essere specificata – sia nella dichiarazione (nella definizione della classe) che nella definizione della funzione – prima del tipo del parametro. Ad esempio, possiamo modificare la funzione Somma() dichiarando come costanti sia l'oggetto di invocazione che il parametro:

- nella definizione della classe:
```
class Coppia
{ // ...
  Coppia Somma(const Coppia&) const;
  // ...
};
```

- nella definizione della funzione:
```
Coppia Coppia::Somma(const Coppia& c) const
{ // ...
}
```

Infine, anche il valore di ritorno può essere passato per riferimento. È necessario, in questo caso, prestare attenzione al fatto che l'oggetto passato per riferimento sia vivo quando verrà utilizzato dal chiamante. Persentiamo nel seguito due esempi relativi a due funzioni esterne a qualsiasi classe, anche se lo stesso discorso è valido per le funzioni proprie di una classe.

Il primo esempio è un buon esempio di uso di questa dichiarazione, mentre il secondo esempio, volutamente sbagliato, mostra un errore tipico che si commette nel passaggio del valore di ritorno per riferimento.

```
// esempio corretto

int& f(int i)
{ static int a = 0;  // a rimane viva fino alla fine del programma
  a = a + i;
  return a;
}

main()
{ int x;
  cin >> x;
  while ( x != 0)
  { int y = f(x); // ok, il valore di a viene assegnato ad y
    cout << x << "\t" << y << endl;
    cin >> x;
  }
}


// esempio SBAGLIATO

int& f(int i)
{ int a = 0;  // a rimane viva SOLO fino alla fine della f()
```

```
  a = a + i;
  return a;
}

main()
{ int x;
  cin >> x;
  while ( x != 0)
  { int y = f(x); // ERRORE! l'oggetto di cui è stato restituito
                  // il riferimento (cioè a) non è più vivo
    cout << x << "\t" << y << endl;
    cin >> x;
  }
}
```

4.8 Overloading

Come già accennato nel paragrafo 4.3, ogni classe può avere più di un costruttore. Per poter distinguere due costruttori è necessario che questi si differenzino nel numero o nel tipo dei parametri. Cioè è possibile che due costruttori:

- abbiano un numero diverso di parametri formali, oppure
- abbiano lo stesso numero di parametri formali, ma almeno uno dei parametri di un costruttore sia diverso da quello corrispondente dell'altro costruttore

Per esempio possiamo inserire nella classe `Coppia` due nuovi costruttori: uno senza argomenti, che inizializzi a zero i due campi della coppia, e l'altro con un solo argomento di tipo `int`, che inizializzi i due campi della coppia usando lo stesso valore. La definizione della classe diventa pertanto:

```
// coppia.h

class Coppia
{private:
  int i,j;
 public:
  Coppia();
  Coppia(int);
  Coppia(int, int);
  int Primo() const;
  int Secondo() const;
  Coppia Somma(Coppia) const;
};
```

Le definizioni dei due nuovi costruttori (nel file `coppia.cpp`) sono le seguenti:

```
Coppia::Coppia()
{ i = j = 0; }

Coppia::Coppia(int a)
{ i = j = a; }
```

La possibilità di definire più funzioni che abbiano lo stesso nome si chiama *overloading*. L'overloading consiste nel sovraccaricare una funzione con più implementazioni diverse. Le funzioni hanno lo stesso nome (quindi è come se si trattasse di una sola funzione con più implementazioni) e lo stesso tipo di ritorno, ma diverso numero o tipo dei parametri. A seconda del numero e del tipo dei parametri attuali nella chiamata, viene scelta dal compilatore la funzione da chiamare. L'overloading può essere fatto sia su funzioni proprie che su funzioni esterne.

4.9 Due esempi: la classe Punto e la classe Complesso

In questo paragrafo realizziamo due classi usando le tecniche e strutture del linguaggio mostrate finora. Le spiegazioni sono volutamente poche, ma si possono trovare nei paragrafi precedenti gli argomenti relativi alle strutture utilizzate.

4.9.1 Classe Punto

La prima classe è la classe `Punto`, i cui oggetti rappresentano dei punti nello spazio tridimensionale, identificati dalle coordinate x, y e z. È possibile specificare le coordinate di un punto al momento della dichiarazione, oppure effettuare una dichiarazione senza argomenti che inizializza le tre coordinate al valore 0.0. Possiamo chiedere se un punto si trovi nell'origine degli assi cartesiani mediante la funzione `EstOrigine()`, possiamo valutare la distanza fra due punti con la funzione `Distanza()`, possiamo ottenere il punto opposto rispetto all'origine degli assi mediante la funzione `Speculare()`e possiamo modificare un oggetto cambiando di segno le sue coordinate mediante la funzione `RendiSpeculare()` che fa side effect sull'oggetto di invocazione (vedi paragrafo 4.5); notare che la `RendiSpeculare()` non può essere dichiarata `const`. Infine, considerando i punti come l'estremo di un vettore avente l'altro estremo nell'origine, possiamo effettuare somme vettoriali con la funzione `Somma()`.
La classe è definita nel file `punto.h`, mentre le sue funzioni proprie sono definite in `punto.cpp`. Il file `usapunto.cpp` contiene un main che usa le funzioni della classe. Il file `math.h`, contenente la libreria di funzioni matematiche del C++, è incluso da `punto.cpp` per poter usare la funzione `sqrt()` che calcola la radice quadrata del suo argomento.

```
// punto.h

#include <iostream.h>

class Punto
{private:
  float x,y,z;
 public:
  Punto();
  Punto(float,float,float);
  bool EstOrigine() const;
  float Distanza(Punto) const;
  Punto Speculare() const;
```

```
  Punto Somma(const Punto&) const;
  void RendiSpeculare();
};


// punto.cpp

#include "punto.h"
#include <math.h>

Punto::Punto()
{ x = y = z = 0.0; }

Punto::Punto(float a,float b,float c)
{ x = a;
  y = b;
  z = c;
}

bool Punto::EstOrigine() const
{ return x == 0.0 && y == 0.0 && z == 0.0;
}

float Punto::Distanza(Punto p) const
{ float dx = x - p.x;
  float dy = y - p.y;
  float dz = z - p.z;
  return sqrt(dx*dx + dy*dy + dz*dz);
}

Punto Punto::Speculare() const
{ return Punto(-x,-y,-z);
}

Punto Punto::Somma(const Punto& p) const
{ return Punto(x+p.x,y+p.y,z+p.z);
}

void Punto::RendiSpeculare()
{ x = -x;
  y = -y;
  z = -z;
}


// usapunto.cpp

#include <iostream.h>
#include "punto.h"

main()
{ Punto p1(0.0,0.0,0.0),p2(1.0,2.0,3.0),p3(9.0,0.0,0.0);
```

```
  if (p1.EstOrigine())
    cout << "ok" << endl;
  else
    cout << "nok" << endl;
  float d = p2.Distanza(p3);
  float e = p1.Distanza(p3);
  cout << e << endl << d << endl;
  Punto p4, p5;
  p4 = p2.Speculare();
  p5 = p2.Somma(p4);
  float dist = p1.Distanza(p5);
  cout << "deve venire 0:  " << dist << endl;
}
```

4.9.2 La classe Complesso

La classe Complesso definisce un nuovo tipo di dato che rappresenta i numeri complessi[6]. Ogni numero complesso è composto da una parte reale e una immaginaria, rappresentate da due float. Il costruttore della classe permette di creare un numero complesso a partire da due reali; le altre funzioni proprie permettono di conoscere la parte reale e la parte immaginaria di un complesso (le funzioni Re() e Im()), di calcolare il modulo di un complesso (Modulo()), di calcolare il complesso coniugato (il complesso avente parte immaginaria di segno opposto a quella del complesso dato) (Coniugato()) e di effettuare la somma di due numeri complessi.

```
// complesso.h

class Complesso
{private:
  float re, im;
 public:
  Complesso();
  Complesso(float,float);
  float Re() const;
  float Im() const;
  float Modulo() const;
  Complesso Coniugato() const;
  Complesso Somma(const Complesso&) const;
};

// complesso.cpp

#include <math.h>

Complesso::Complesso()
```

[6] Nelle librerie del C++ una classe simile già esiste. Questa la definiamo a scopo puramente didattico.

```
{ re = im = 0.0; }

Complesso::Complesso(float r, float i)
{ re = r;
  im = i;
}

float Complesso::Re() const
{ return re; }

float Complesso::Im() const
{ return im; }

float Complesso::Modulo() const
{ return sqrt(re*re + im*im); }

Complesso Complesso::Coniugato() const
{ return Complesso(re,-im); }

Complesso Complesso::Somma(const Complesso& z) const
{ return Complesso(re+z.re,im+z.im); }


// usacomplesso.cpp

#include “complesso.h”

main()
{ Complesso a;
  Complesso b(12.3,15.4), c(13.4,16.7);
  float r;
  r = b.Modulo();
  b = c.Coniugato();
  r = b.Re();
  a = b.Somma(c);
}
```

4.10 Overloading degli operatori

La funzione `Somma()` dichiarata nelle classi del paragrafo precedente è di utilizzo piuttosto innaturale. Infatti, per esempio, per sommare due oggetti `Complesso` dobbiamo usare la seguente notazione:

```
c1 = c2.Somma(c3);
```

mentre per sommare due numeri float possiamo usare l’operatore C++ ‘+’, che consente una scrittura più diretta e più naturale. L’operatore ‘+’ corrisponde a una funzione, chiamata `operator+()`, che può essere *overload-ata*.

Infatti è possibile definire una versione della funzione `operator+()` che accetti come argomenti due oggetti della classe `Complesso`, in modo da poter scrivere:

```
c1 = c2 + c3;
```

Questa funzione verrà chiamata quando ci si trova in un'espressione contenente il simbolo '+' con a sinistra e a destra due operandi di tipo `Complesso`. La funzione accetterà l'operando sinistro come oggetto di invocazione e l'operando destro come parametro formale. Inoltre restituirà un oggetto di tipo `Complesso` i cui campi corrispondono alla somma dei rispettivi campi dei due oggetti.
Nella definizione della classe dobbiamo dichiarare la funzione:

```
Complesso operator+(Complesso) const; // in complesso.h
```

Inoltre dobbiamo definire la funzione nel file `complesso.cpp`:

```
// in complesso.cpp

Complesso Complesso::operator+(const Complesso& p) const
{ return Complesso(x+p.x,y+p.y,z+p.z);
}
```

L'operatore '+' può essere invocato con la sintassi seguente:

```
Complesso c,d,e;
c = d + e;
```

oppure con la sintassi – meno intuitiva ma corretta:

```
c = d.operator+(e);
```

Così come l'operatore '+' possiamo definire uno degli altri operatori quali '–', '*', '/', '[]', '++', '<<' etc.

Ad esempio, l'operatore '++'[7] può essere definito come segue:

```
Complesso operator++();                // in complesso.h
```

[7] Attenzione: l'operatore qui definito è l'operatore '++' prefisso (che restituisce il valore dell'operando dopo l'incremento) la cui sintassi è la seguente:

```
++x;
```

Per definire invece l'operatore prefisso (che restituisce il valore che l'operando aveva prima dell'incremento, con sintassi tipo `x++`) è necessario usare un operando in più, fittizio, di tipo `int`, affinché il compilatore possa fare distinzione:

```
Complesso operator++(int)             // in complesso.h
Complesso Complesso::operator++(int)  // in complesso.cpp
{ Complesso ris(re,im);
  re = re + 1.0;
  return ris;
}
```

```
Complesso Complesso::operator++()     // in complesso.cpp
{ re = re + 1.0;
  return *this;
}
```

In questo caso l'unico operando viene passato alla funzione come oggetto di invocazione, e, dato un complesso:

```
Complesso c1;
```

la sintassi:

```
c1++;
```

corrisponde a:

```
c1.operator++();
```

Come ulteriore esempio, l'operatore '[]' può essere usato per estrarre la prima, la seconda o la terza coordinata di un punto nella classe `Punto`, come segue:

```
float operator[](int) const;          // in punto.h

float Punto::operator[](int i) const   // in punto.cpp
{ if (i == 0)
    return x;
  else if (i == 1)
    return y;
  else
    return z;
}
```

Questo operatore può essere usato come segue:

```
Punto p(10.0,20.0,30.0);
float xx = p[0]; // xx = 10.0
float yy = p[1]; // yy = 20.0
float zz = p[2]; // zz = 30.0
```

ma è corretto anche scrivere:

```
xx = p.operator[](0);
```

in quanto l'oggetto della classe viene comunque passato alla funzione propria come oggetto di invocazione mentre l'intero viene passato come parametro.

Infine possiamo definire l'operatore '<<' per la classe `Complesso`, per inviare un oggetto a `cout` in un formato che possiamo scegliere arbitrariamente. La definizione di questo operatore è più elaborata di quelle precedenti, ma segue uno schema standard che può essere riutilizzato in tutte le classi per le quali abbiamo necessità di effettuare la stampa degli oggetti su `cout`.

Ad esempio, supponiamo di voler stampare un complesso usando la seguente notazione:

(partereale , parteimmaginaria)

Un complesso dovrebbe essere stampato in questo formato quando scriviamo l'espressione:

```
cout << c;
```

Come si vede, l'operando sinistro dell'operatore '<<' non è un oggetto della classe, bensì è `cout`, che è un oggetto della classe `ostream`, definita in `iostream.h`.
Pertanto non possiamo definire `operator<<()` come funzione propria della classe `Complesso`.
Allora dobbiamo definire tale operatore come funzione esterna, avente come primo argomento un `ostream` e come secondo argomento un `Complesso`. Inoltre la funzione deve restituire un oggetto ostream per poter concatenare più operatori (come in: `cout << c << d;`)

```
ostream& operator<<(ostream&,const Complesso&);
```

Questa dichiarazione deve essere fatta nel file `complesso.h` al di fuori della definizione della classe.

Adesso definiamo la funzione, nel file complesso.cpp:

```
ostream& operator<<(ostream& os,const Complesso& p)
{ os << "(" << p.re << "," << p.im << ")";
  return os;
}
```

La funzione usa un oggetto della classe `ostream`, `os`, e gli invia le varie parti che compongono la stampa del complesso nel formato scelto, cioè la parentesi aperta, la parte reale, la virgola, la parte immaginaria e la parentesi chiusa.
I '<<' usati sono altre versioni dell'operatore, definite per i tipi float e stringa.

Si può notare che la funzione deve accedere ai campi re e im dell'oggetto p, cosa impossibile per una funzione esterna, poiché i due campi sono dichiarati nella parte privata della classe. Pertanto è necessario dichiarare questa funzione come *funzione amica* della classe, inserendo la seguente riga all'interno della definizione della classe in `complesso.h`:

```
// complesso.h
```

```
class Complesso
{
  // ...
  friend ostream& operator<<(ostream&,const Complesso&);
};
```

Questa dichiarazione permette alla funzione di accedere alla parte privata della classe come se fosse una funzione propria. La possibilità di dichiarare una funzione come `friend` deve essere usata con attenzione perché va contro l'information hiding e aumenta l'accoppiamento tra moduli.

4.11 Copia nel passaggio per valore e costruttore di copia

Quando un oggetto di una classe viene usato come parametro attuale in un passaggio per valore, il suo valore viene copiato nelle locazioni dell'oggetto che funge da parametro formale. Ad esempio, facendo riferimento alla classe `Complesso` definita nei paragrafi precedenti:

```
#include "complesso.h"

void f(Complesso x)
{
  //  ...
}

main()
{ Complesso c ;
  f(c);
}
```

il valore dei campi di `c` viene copiato nei rispettivi campi di `x`.

Per effettuare la copia entra in gioco una funzione, chiamata automaticamente dal compilatore, detta *costruttore di copia*.
Il costruttore di copia viene usato per costruire il parametro formale inizializzando i campi con i valori dei campi corrispondenti del parametro attuale.

Ogni classe ha per default (cioè in assenza di altra indicazione) un costruttore di copia fornito dal compilatore. È però possibile ridefinire il costruttore di copia se vogliamo che effettui operazioni differenti. Per fare ciò nella classe Complesso dovremmo dichiarare nella definizione una funzione con la seguente intestazione:

```
Complesso(const Complesso&);
```

Come si vede, si tratta di un costruttore avente un solo parametro formale, dello stesso tipo della classe.

Poi dovremmo definire questo costruttore nel file complesso.cpp, specificando nel corpo le operazioni che deve effettuare.

Il formato dell'intestazione del costruttore di copia è sempre lo stesso per tutte le classi.

Per analizzare le motivazioni che possono spingere a ridefinire il costruttore di copia, analizziamo la classe Lista nel prossimo paragrafo.

4.11.1 La classe Lista

Consideriamo una lista di interi, cioè una sequenza come:

7 3 11 4 8 6

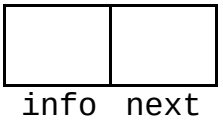
Vogliamo rappresentare una struttura come questa in un programma C++, facendo in modo che la quantità di interi in essa contenuti possa variare durante il corso del programma senza sprecare memoria non utilizzata e senza vincoli sul massimo numero di elementi memorizzabili.

Per fare ciò possiamo memorizzare gli elementi in memoria dinamica, allocando ogni nuovo elemento che dobbiamo inserire nella lista mediante la `new`, e rilasciando lo spazio utilizzato da un elemento – mediante la `delete` – quando questo viene tolto dalla lista.

Realizziamo l'elemento come un oggetto della classe seguente:

```
class Elemento
{public:
  int info;
  Elemento* next;
};
```

Gli oggetti di questa classe hanno due campi: il primo contiene l'informazione, cioè il valore dell'elemento; il secondo è un puntatore che punta all'oggetto che rappresenta l'elemento successivo (in inglese, next) della lista.



La lista dell'esempio iniziale può essere rappresentata allora come una concatenazione di elementi:

Creiamo una nuova classe, che chiamiamo `Lista`, i cui oggetti sono delle liste realizzate nel modo appena descritto.

L'oggetto della classe contiene un solo campo di tipo `Elemento*` che punta al primo elemento della lista. Questo campo contiene il valore NULL quando non ci sono elementi nella lista.

Nella parte pubblica della classe mettiamo un costruttore, una funzione per inserire un elemento in testa alla lista (cioè aggiungendolo a sinistra, davanti al primo elemento corrente), una funzione per eliminare il primo elemento della lista e l'operatore '<<' per stampare la lista.

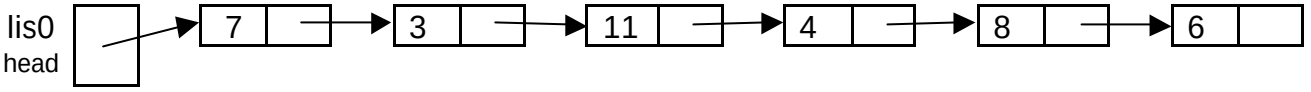
```
class Lista
{private:
  Elemento* head;
 public:
  Lista();
  void Inserisci(int);
  int Estrai();
 friend ostream& operator<<(ostream&, const Lista&);
};
```

Salviamo le due classi precedenti in un file che chiamaiamo `lista.h`, nel quale dobbiamo includere `iostream.h` (per poter dichiarare l'operatore '<<' che usa `ostream`). In un altro file, che chiamiamo `lista.cpp`, definiamo le funzioni della classe `Lista`.

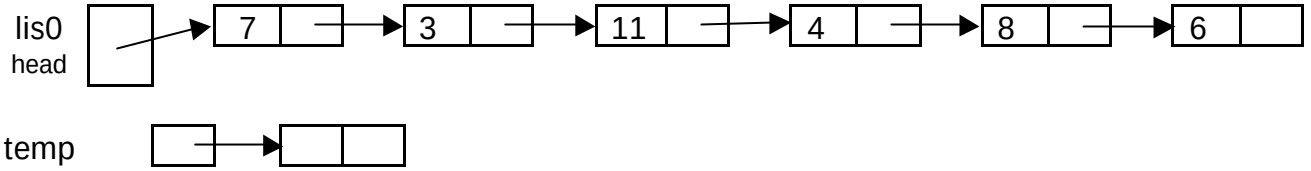
```
// lista.cpp

#include "lista.h"

Lista::Lista()
{ head = NULL; }

void Lista::Inserisci(int i)
{ Elemento* temp = new Elemento; // creiamo un nuovo Elemento in
                                 // memoria dinamica e ne
                                 // assegniamo l'indirizzo a un
                                 // puntatore temporaneo
  temp->info = i;
  temp->next = head;
  head = temp;
}
```

Possiamo mostrare graficamente cosa accade se inseriamo il nuovo elemento 23 alla lista dell'esempio iniziale usando la funzione `Inserisci()`. Nelle figure seguenti è rappresentato a sinistra un oggetto della classe `Lista` che chiamiamo `lis0`; il suo unico campo è il puntatore `head`. A destra di tale oggetto sono rappresentati gli oggetti `Elemento` che sono concatenati tra loro e si trovano in memoria dinamica.

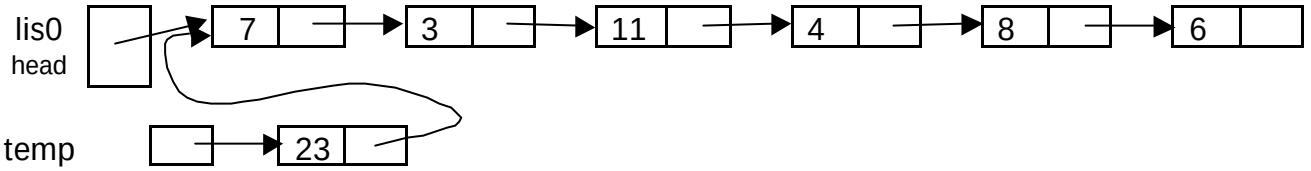
PRIMA DELLA CHIAMATA ALL FUNZIONE `Inserisci()`



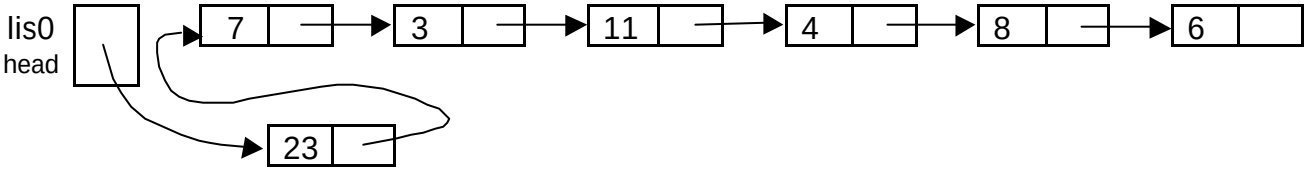
DOPO AVER ESEGUITO LA `new`



DOPO AVER INIZIALIZZATO I CAMPI `info` E `next` DEL NUOVO ELEMENTO



DOPO LA FINE DELLA FUNZIONE `Inserisci()`

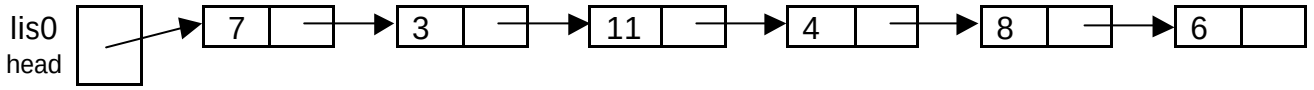


La funzione `Estrai()` procede in modo opposto.

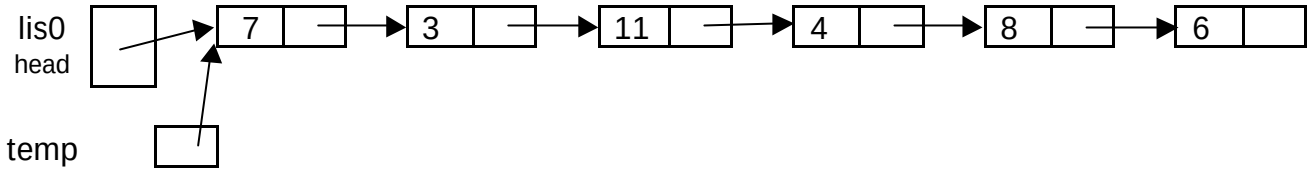
```
int Lista::Estrai()
{ if (head == NULL)
    return 0;              // se la lista è vuota restituiamo
                           // convenzionalmente il valore zero
  Elemento* temp = head;
  head = head->next;
  int i = temp->info;
  delete temp;
  temp = NULL;
  return i;
}
```

Le figure seguenti mostrano il comportamento della `Estrai()` passo per passo.

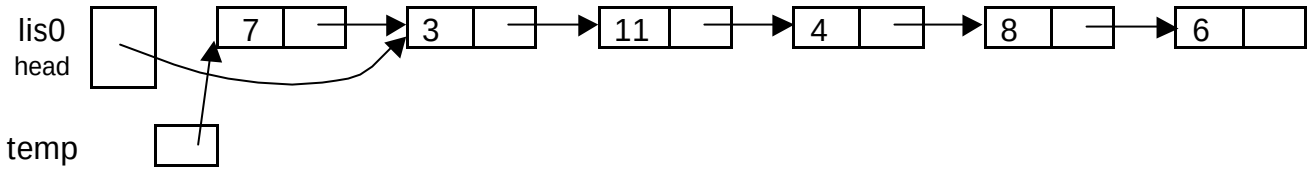
PRIMA DELLA CHIAMATA ALLA FUNZIONE `Estrai()`



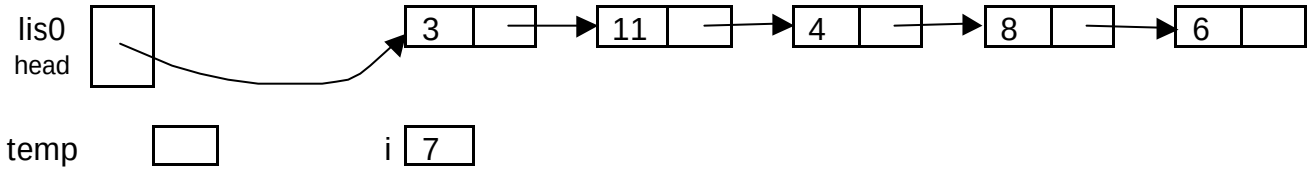
DOPO LA DICHIARAZIONE E L'INIZIALIZZAZIONE DI `temp`



DOPO L'ASSEGNAZIONE `head = head->next`



PRIMA DELLA `return`



La funzione `operator<<()`, infine, stampa tutti gli elementi della lista separati da arresti di tabulazione.

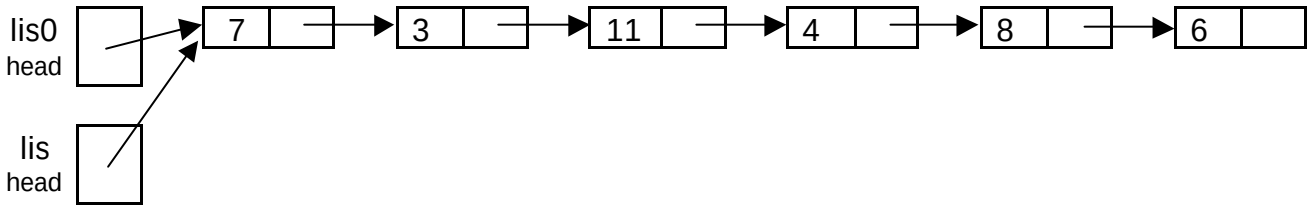
```
ostream& operator<<(ostream& os, const Lista& lis)
{ Elemento* e = lis.head;
  while (e != NULL)
  { os << e->info << "\t";
    e = e->next;
  }
  os << endl;
  return os;
}
```

4.11.2 Il problema dell'interferenza

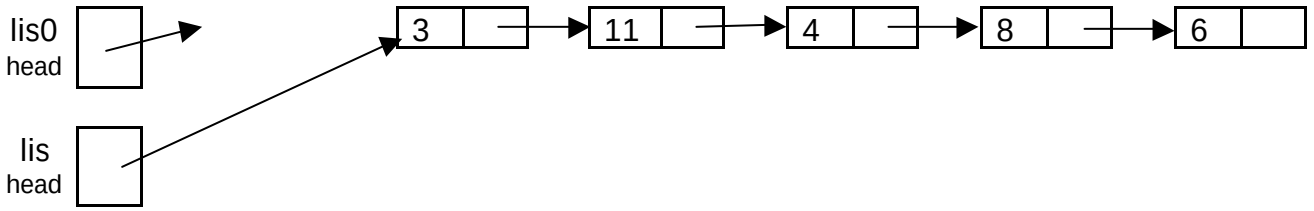
Cosa accade se proviamo a scrivere il seguente codice?

```
// usalista.cpp

#include "lista.h"

void f(Lista lis)    // N.B.: passaggio per valore!!
{ lis.Estrai();
}

main()
{ Lista lis0;
  // ... qui inizializziamo lis0 con gli elementi 7 3 11 4 8 6
  // ... mediante ripetute chiamate alla Inserisci()
  f(lis0);
}
```

Nel main effettuiamo una chiamata alla funzione `f()` passando `lis0` per valore. Trattandosi di un passaggio per valore, ci aspettiamo che l'eliminazione del primo elemento dalla lista `lis` non modifichi la lista `lis0`.
Invece `lis0` viene – erroneamente – modificata. Questo accade perché nella chiamata viene lanciato automaticamente il costruttore di copia di default, il quale copia il valore dell'unico campo dell'oggetto `lis0` nel corrispondente campo dell'oggetto `lis`, creando una *condivisione di memoria* (vedi figura seguente). Cioè, benché i due oggetti `lis` e `lis0` siano distinti, gli elementi da questi puntati in memoria dinamica sono condivisi.



A questo punto la chiamata alla funzione `Estrai()` (che effettua side effect sul suo oggetto di invocazione `lis`) modifica entrambe le liste, producendo l'eliminazione non voluta del primo elemento dalla lista `lis0` e, cosa ancor più grave, lasciando il puntatore `head` di `lis0` "appeso". Cioè, `head` punta adesso a una locazione di memoria dinamica non più valida, e l'oggetto `lis0` è scollegato dai suoi elementi.



Questo fenomeno, detto *interferenza*, si verifica a causa della presenza contemporanea della condivisione di memoria e del side effect sull'oggetto di invocazione.

Per evitare l'interferenza è necessario ridefinire il costruttore di copia in modo che effettui una copia profonda (cioè elemento per elemento) della lista, in modo da eliminare uno dei due fattori che danno luogo all'interferenza, la condivisione di memoria.

4.11.3 Copia profonda

In questo paragrafo modifichiamo la classe `Lista` ridefinendo il costruttore di copia e aggiungendo la funzione `CopiaProfonda()` che il costruttore di copia userà per effettuare la copia di tutti gli elementi così da evitare la condivisione di memoria. La funzione sarà dichiarata nella parte privata poiché vogliamo che sia utilizzata solo dal costruttore di copia e da altre eventuali funzioni proprie, ma non vogliamo che faccia parte dell'interfaccia della classe.

Nel file `lista.h` dichiariamo le due nuove funzioni:

```
// lista.h

// ...

class Lista
{private:
  Elemento* head;
  Elemento* CopiaProfonda(Elemento*);
 public:
  Lista();
  Lista(const Lista&);  // costruttore di copia
  // ...
};
```

Nel file lista.cpp definiamo le due funzioni.
La funzione `CopiaProfonda()` è una funzione ricorsiva che accetta un puntatore al primo elemento della lista originale e restituisce il puntatore al primo elemento della lista copiata. Il passo base consiste nel caso in cui il puntatore d'ingresso è `NULL`: questo accade se la lista è vuota o quando la funzione viene richiamata sull'ultimo elemento della lista. Nel passo di ricorsione viene creato un nuovo `Elemento`, si copia in esso (nel campo `info`) il valore dell'elemento corrispondente della lista originale, si chiama la funzione stessa per copiare la sottolista rimanente e infine si assegna al campo `next` l'indirizzo del primo elemento della sottolista copiata.

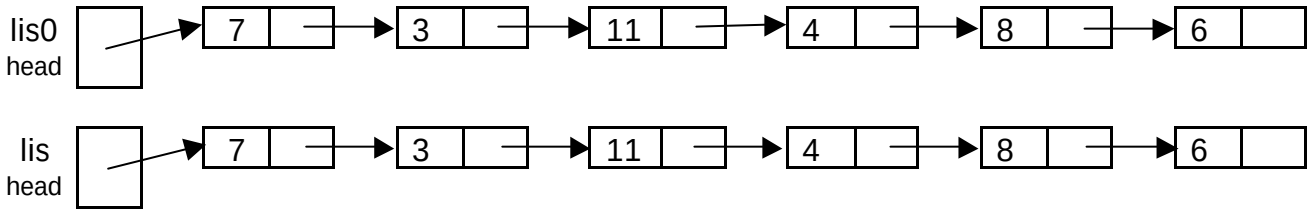
```
// lista.cpp

// ...

Elemento* Lista::CopiaProfonda(Elemento* e)
{ if (e == NULL)
    return NULL;
  Elemento* temp = new Elemento;
```

```
  temp->info = e->info;
  temp->next = CopiaProfonda(e->next);
  return temp;
}

// ...
```

Il costruttore di copia è molto semplice, poiché si basa sulla `CopiaProfonda()`:

```
// ...

Lista::Lista(const Lista& orig)
{ head = CopiaProfonda(orig.head);
}
```

Mediante l'uso di questo costruttore di copia non si verifica più la condivisione di memoria, come mostrato nella figura seguente, e quindi non compare più il problema dell'interferenza.



4.12 Operatore di assegnazione

Anche le assegnazioni tra due liste, come ad esempio

```
lis1 = lis0;
```

possono creare condivisione di memoria. Infatti l'operatore '=' fornito per default dal compilatore effettua una copia campo per campo degli oggetti che riceve come operandi.

Per evitare questo problema, dobbiamo ridefinire anche questo operatore. Definiamo anche la funzione `Cancella()` che useremo per eliminare tutti gli elementi eventualmente presenti nella lista a sinistra dell'uguale, prima di rimpiazzarli con i nuovi elementi. Le intestazioni dell'operatore di assegnazione e della funzione `Cancella()` sono le seguenti:

```
// nel file lista.h
// ...
class Lista
{private:
  // ...
```

```
  void Cancella(Elemento*);
 public:
  // ...
  Lista& operator=(const Lista&);
  // ...
};
```

La definizione, ricorsiva, della funzione `Cancella()` è la seguente:

```
// nel file lista.cpp

void Lista::Cancella(Elemento* e)
{ if (e == NULL)
    return;
  Cancella(e->next);
  delete e;
}
```

Il corpo dell'operatore di assegnazione è composto da quattro passi fondamentali:

1. controllo che l'oggetto di invocazione sia diverso dal parametro formale, per evitare assegnazioni tipo `lis0 = lis0;`
2. cancellazione degli elementi eventualmente collegati al campo head della lista oggetto di invocazione; per questo passo facciamo uso della funzione ricorsiva `Cancella()`;
3. copia profonda degli elementi del parametro formale (mediante la `CopiaProfonda()`);
4. restituzione al chiamante dell'oggetto di invocazione stesso, il cui valore può essere utilizzato, ad esempio, nelle concatenazioni di assegnazioni;

```
Lista& Lista::operator=(const Lista& orig)
{ if (this == &orig)                // primo passo
    return *this;

  Cancella(head);                   // secondo passo
  head = CopiaProfonda(orig.head);  // terzo passo
  return *this;                     // quarto passo
}
```

4.13 Distruttore

Analizziamo la seguente funzione:

```
f(Lista lis)
{ lis.Inserisci(100);
  lis.Inserisci(200);
}
```

La lista, alla fine della funzione, è formata dall'oggetto `lis`, che si trova nel record di attivazione di `f()` nello stack, e dai due elementi, che si trovano in memoria dinamica.

Quando la funzione termina, l'oggetto `lis` viene rimosso dallo stack, ma la memoria dinamica contenente i due elementi non viene rilasciata. Infatti non viene effettuata la `delete` su tali elementi.

Pertanto si verifica un effetto spiacevole, per due motivi:

- la memoria dei due elementi non viene rilasciata
- si perde l'indirizzo del primo dei due elementi, e quindi non sarà più possibile effettuare la delete

Per evitare questo problema è possibile definire una funzione propria della classe, detta distruttore, che effettui la cancellazione delle locazioni di memoria dinamica utilizzate dall'oggetto. Questa funzione verrà chiamata automaticamente dal compilatore quando l'oggetto sta per essere eliminato.

La dichiarazione del distruttore è la seguente:

```
// in lista.h

class Lista
{
  // ...
 public:
  // ...
  ~Lista();
  // ...
};
```

Il distruttore non ha tipo di ritorno e non ha parametri formali.

La definizione può essere basata sulla funzione `Cancella()`, come segue:

```
// in lista.cpp

Lista::~Lista()
{ Cancella(head); }
```

4.14 Riepilogo della gestione della memoria

Vogliamo qui sottolineare il fatto che, in generale, quando gli oggetti di una classe usano la memoria dinamica, risulta necessario definire sia il costruttore di copia che l'operatore di assegnazione che il costruttore.

Riepiloghiamo nel seguito i casi in cui vengono chiamati automaticamente il costruttore di copia, l'operatore di assegnazione e il costruttore:

Costruttore di Copia	Esempio
`C(const C&);`	
Passaggio per valore	`f(lis)`
Restituzione del tipo di ritorno per valore	`Lista g();`
Inizializzazione	`Lista lis1 = lis0;`

Operatore di assegnazione	Esempio
`C& operator=(const C&);`	
Assegnazione	`lis1 = lis0;`

Distruttore	Esempio
`~C();`	

Fine del ciclo di vita dell'oggetto	`void f(Lista lis)`
	`{ ... }` qui l'oggetto muore
	
	`Lista* plis = new Lista;`
	`delete plis;` qui l'oggetto puntato muore

5 Standard library

La Standard Library è una libreria[8] (cioè una raccolta) di componenti elementari che il programmatore può utilizzare per realizzare le strutture dati e gli algoritmi del programma. Tali componenti sono costituiti da classi e funzioni di uso generale e sono dichiarati in alcuni file .h e definiti in altrettanti file oggetto forniti con tutti i più diffusi compilatori C++. Per utilizzarli è necessario includere gli opportuni file .h nei propri programmi, come descritto nel seguito di questo capitolo.

La standard library permette di modularizzare un progetto decomporre il problema in sottoproblemi: la parte di programma che affronta un determinato sottoproblema può essere realizzata mediante la struttura dati e gli algoritmi più opportuni fra quelli forniti. I componenti della standard library possono essere visti come mattoni con cui costruire strutture più complesse nell'ambito del programma.
Inoltre, basandosi sulle funzioni e sulle classi della standard library, si focalizza l'attenzione sul problema generale e si astrae dalla definizione e dalla gestione di dettagli implementativi ottenendo un significativo ausilio nella progettazione dei programmi.

Le classi della standard library descrivono in maniera generale e uniforme alcune strutture dati di base. Il modo per accedere ai dati di una struttura è indipendente dal tipo di struttura. Per accedere agli elementi di una struttura si possono utilizzare gli iteratori, che sono dei metodi astratti di accesso ai dati che estendono il concetto di puntatore.

Le funzioni realizzano degli algoritmi applicabili alle strutture dati definite mediante le classi. Gli algoritmi funzionano su qualsiasi struttura dati della standard library.

[8] Benché la traduzione corretta del termine inglese "library" sia "biblioteca", in informatica viene comunemente tradotto con "libreria".

Le classi e le funzioni della standard library sono in realtà dei modelli di classi e funzioni che possono essere facilmente adattati al problema da trattare istanziando alcuni parametri. La parametricità è gestita mediante l'uso dei *template*, descritti nel paragrafo successivo.

Ricapitolando, la standard library è basata su quattro tipi principali di componenti:

- i CONTENITORI (le strutture dati di base)
- gli ALGORITMI GENERICI (applicabili ai dati contenuti nei contenitori)
- gli ADATTATORI (degli strumenti che modificano i contenitori e gli algoritmi)
- gli ITERATORI (utilizzati per accedere ai singoli dati dei contenitori)

Infine nella standard library sono presenti altre classi di utilità, tra cui quella principale è la classe `string` che permette di manipolare delle stringhe di caratteri in maniera più agevole dei `char*`. Mediante questa classe è possibile ad esempio dichiarare ed utilizzare delle stringhe come segue:

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

main()
{ string s1,s2;
  s1 = "standard";
  s2 = "library";
  string s3 = s1 + " " + s2;      // "standard library"
  if (s3 == "standard library")
    cout << "ok: " << s3 << endl;
  s3+=".";                        // "standard library."
  cout << s3 << endl;
}
```

5.1 Template

Consideriamo la seguente classe:

```
class CoppiaInteroCarattere
{public:
  int primo;
  char secondo;
  CoppiaInteroCarattere(int,char);
};
```

dove il costruttore inizializza i due campi dell'oggetto mediante i valori dei parametri:

```
CoppiaInteroCarattere::CoppiaInteroCarattere(int x, char y)
{ primo = x;
  secondo = y;
}
```

Questa classe ci permette di gestire coppie in cui il primo elemento è un intero e il secondo elemento è un carattere. Se però vogliamo gestire coppie i cui elementi sono di altri tipi, ad esempio bool-float, dobbiamo definire una nuova classe:

```
class CoppiaBoolFloat
{public:
  bool primo;
  float secondo;
  CoppiaBoolFloat(bool,float);
};
```

Mediante un template è possibile invece definire una classe modello, dalla quale il compilatore possa generare automaticamente le classi effettive. Una classe modello deve essere basata su uno o più parametri ai quali viene assegnato un valore al momento della creazione della classe effettiva.

La nostra classe modello, che chiamiamo `Coppia`, ha due parametri che rappresentano rispettivamente i tipi del primo e del secondo campo. Chiamiamo questi parametri `T1` e `T2`. La definizione è la seguente:

```
template<class T1, class T2>
class Coppia
{public:
  T1 primo;
  T2 secondo;
  Coppia(T1,T2);
};
```

Per dichiarare un oggetto della classe `Coppia`, dobbiamo istanziare i due parametri, cioè scegliere i due tipi effettivi da assegnare a `T1` e `T2`, come nei due esempi seguenti:

```
Coppia<int,char> c1(10,'a');
Coppia<bool,float> c2(true,1.23);
```

In queste due dichiarazioni abbiamo dichiarato `c1` come coppia int-char e `c2` come coppia bool-float. Un'altra buona dichiarazione è la seguente:

```
Coppia<Coppia<int,char>,float> c3(c1,0.5);
```

Un altro esempio di uso dei template è nella definizione delle funzioni modello. Consideriamo la funzione seguente:

```
int intmax(int x, int y)
{ if (x < y)
    return y;
  else
    return x;
}
```

Se vogliamo definire delle funzioni che calcolano il massimo fra due numeri, anziché definire una funzione diversa per ogni coppia di tipi dei parametri (es. int-int, float-float etc.) possiamo usare una *template function*, definita come segue:

```
template<class T>
T max(T x, T y)
{ if (x < y)
    return y;
  else
    return x;
}
```

A seconda del tipo dei due parametri attuali verrà scelta automaticamente dal compilatore la funzione appropriata:

```
int u = 3, v = 4, z;
float f = 4.7, g;
z = max(u,v);   // T istanziato con int
g = max(f,3.5); // T istanziato con float
```

È necessario che per il tipo `T` sia definito l'operatore '<'. Questo operatore esiste per i tipi predefiniti, ma dobbiamo definirlo esplicitamente nelle classi create da noi. Ad esempio, se definiamo opportunamente l'operatore '<' per la classe `Coppia`, possiamo cercare il massimo tra due coppie. Dobbiamo dichiarare, all'interno della definizione della classe, la seguente funzione:

```
bool operator<(Coppia);
```

e definirla, ad esempio, come segue:

```
template<class T1, class T2>
bool Coppia<T1,T2>::operator<(Coppia<T1,T2> c)
{ if (primo == c.primo)
    return secondo < c.secondo;
  else
    return primo < c.primo;
}
```

A questo punto possiamo scrivere:

```
Coppia<int,char> c4(20,'b'), c5(0,' ');
c5 = max(c1,c4); // T viene istanziato con Coppia<int,char>
```

Attenzione, se i tipi scelti per i due operandi sono diversi tra loro, e se il compilatore non può convertire implicitamente uno dei due operandi nel tipo dell'altro (ad esempio, un int può essere convertito in un float senza perdita di informazione), allora non può essere istanziato il parametro T e non può essere generata una funzione appropriata:

```
g = max(f,c5);   // ERRORE, incongruenza di tipi, T non istanziato
```

5.2 Complessità

Di solito, quando progettiamo un algoritmo per risolvere un dato problema, possiamo scegliere più strade diverse che portano alla soluzione del problema. Alcune strade però ci permettono di giungere alla soluzione effettuando un numero minore di operazioni, quindi in maniera più efficiente. Un algoritmo che permette di risolvere il problema con meno operazioni ha minore *complessità* degli altri.

Ad esempio consideriamo il problema di verificare se un certo numero intero, ad esempio 12, è presente in una sequenza ordinata di numeri come quella seguente:

1 2 4 7 8 10 12 19 20

Possiamo immaginare diversi algoritmi per fare ciò. Il più semplice è quello di *ricerca esaustiva*, che consiste nel leggere ad uno ad uno i numeri della sequenza finché non troviamo il numero 12 o finché la sequenza non sia finita.

Il numero di passi di questo algoritmo varia a seconda della posizione in cui si trova il numero 12, ma nel caso peggiore dobbiamo leggere tutti i numeri della sequenza. Il caso peggiore può verificarsi se il numero cercato è l'ultimo della sequenza oppure se il numero non è presente. In generale, se la sequenza è composta da n elementi, dobbiamo effettuare n operazioni di confronto, e si dice che la complessità di questo algoritmo è dell'ordine di n, e si scrive $O(n)$.

Possiamo pensare un altro algoritmo che abbia complessità minore di questo e che sfrutti il fatto che i numeri della sequenza sono ordinati in ordine crescente. Questo algoritmo, detto *ricerca binaria*, funziona in questo modo:

- si confronta il numero da cercare con il numero centrale della sequenza: si possono verificare tre casi: *a*) il numero centrale è maggiore del numero cercato; *b*) i due numeri sono uguali; *c*) il numero centrale è minore del numero cercato.
- nel caso *a* si riapplica l'algoritmo alla sottosequenza sinistra, cioè quella che va dal primo numero della sequenza iniziale al numero precedente quello centrale (nel nostro esempio sarebbe quella composta dai numeri 1 2 4 7).
- nel caso *b* il numero è stato trovato e l'algoritmo è terminato.
- nel caso *c* si riapplica l'algoritmo alla sottosequenza destra (nel nostro esempio 10 12 19 20).
- l'algoritmo termina comunque se la sottosequenza scelta non ha elementi. Nel nostro esempio, se cercassimo il numero 25, arriveremmo ad un certo punto a fare un confronto con il numero 20 (l'ultimo della sequenza iniziale), cadremmo nel caso *c* e non avremmo ulteriori elementi a destra di 20 per riapplicare l'algoritmo. In questo caso dichiariamo che il numero non è stato trovato.
- Quando una sequenza è formata da un numero pari di elementi, l'elemento centrale è quello a sinistra dei due centrali (ad esempio, il centrale di 10 12 19 20 è 12) e quando è formata da un solo elemento, quello è anche il numero centrale.

Se applichiamo questo algoritmo al nostro esempio effettuiamo solo due confronti per trovare il numero (prima con il numero centrale, 8, poi col numero centrale della sottosequenza destra, che è proprio il numero 12) quindi cadiamo nel caso *b* e l'algoritmo termina). Siamo stati fortunati. Nel caso peggiore, comunque, il numero di confronti che avremmo dovuto fare non sarebbe stato molto alto: infatti, poiché ad ogni passo dell'algoritmo scartiamo metà della sequenza che stiamo analizzando, il numero totale di confronti è pari al logaritmo in base 2 del numero di elementi, $\log n$.

Si dice che questo algoritmo ha complessità $O(\log n)$ nel caso peggiore.

La complessità di un algoritmo è molto importante poiché al crescere della dimensione del problema (ad esempio del numero n di elementi) un algoritmo con complessità più alta impiega molto più tempo ad essere eseguito al calcolatore.

Per esempio riportiamo nella seguente tabella il tempo impiegato dai nostri due algoritmi nel caso peggiore al variare di n, ipotizzando che il tempo necessario per fare un confronto di due numeri sul calcolatore sia pari a 1 microsecondo (1μs).

n	**ricerca esaustiva**	**ricerca binaria**
10	10 μs	4μs
100	100 μs	7μs
1000	1 ms	10μs
10000	10 ms	14μs
1000000	0.1 s	17μs

Oltre alla complessità del caso peggiore possiamo anche valutare la complessità in altri modi. Elenchiamo nel seguito due diverse possibilità:

- La complessità nel caso medio può essere ottenuta esaminando le probabilità con cui si verificano i vari casi (nel nostro esempio le probabilità che venga cercato il primo elemento della sequenza, o il secondo, … o l'ultimo).
- La complessità ammortata è ottenuta sommando il costo dell'algoritmo per ognuno degli n casi possibili e dividendo il totale per n.

Come esempio di complessità ammortata consideriamo una pila realizzata mediante un array v di quattro elementi e un algoritmo usato per inserire dati nella pila. Quando viene inserito il primo dato, l'algoritmo effettua una semplice scrittura in v[0]. Definiamo costo unitario il costo di questa operazione. Anche l'inserimento del secondo, del terzo e del quarto dato hanno costo unitario. Al momento dell'inserimento del quinto dato, non essendovi più spazio nell'array, l'algoritmo dichiara un nuovo array di grandezza doppia, copia in esso i quattro dati precedenti, inserisce il nuovo dato nel quinto posto e cancella il vecchio array. Supponiamo che questa operazione abbia costo 5, avendo considerato unitario il costo delle operazioni di copiatura e nullo il costo della creazione e cancellazione dei due array. Se poniamo $n = 5$, vediamo che per l'inserimento dei primi $n-1$ dati il costo è unitario, mentre per l'inserimento dell'n-esimo dato il costo è pari a n. Perciò la complessità nel caso peggiore è $O(n)$. La complessità ammortata è data invece da:

$$(1+1+1+1+n)/n = 2n/n = 2$$

cioè è indipendente da *n*, è costante, *O(1)*.

5.3 Contenitori

I contenitori sono le strutture dati di base definite nella standard library. Sono delle classi modello, cioè realizzate mediante template, che permettono di collezionare oggetti di qualsiasi tipo. Il tipo degli oggetti da collezionare viene scelto dall'utente istanziando il template.

Esistono due categorie di contenitori: i contenitori sequenziali e i contenitori ordinati associativi.

Contenitori Sequenziali	Contenitori ordinati associativi
ARRAY	MAP
VECTOR	MULTIMAP
DEQUE	SET
LIST	MULTISET

I contenitori sequenziali organizzano gli elementi in modo lineare. L'ordine degli elementi nel contenitore è quello in cui essi sono stati inseriti. Tra i contenitori sequenziali figurano le liste (LIST), i vettori a lunghezza variabile (VECTOR), le code a doppio ingresso (DEQUE) e gli ARRAY, i quali, pur essendo predefiniti in C++, costituiscono un primo esempio di contenitore.

I contenitori ordinati associativi, al contrario di quelli sequenziali, utilizzano il valore degli oggetti per creare un ordinamento e quindi per permettere un più rapido ritrovamento degli oggetti stessi. Il più importante tra essi è la mappa (MAP) in cui ogni oggetto è associato a un valore detto *chiave*. Data una chiave è possibile trovare l'oggetto associato in maniera molto efficiente. La chiave è di un tipo scelto a piacere; ad esempio mediante una stringa si può associare a ogni oggetto del contenitore un nome, utilizzabile per ritrovare ed eventualmente modificare l'oggetto. Tra i contenitori associativi troviamo anche le MULTIMAP, cioè mappe in cui possono essere presenti due o più chiavi uguali, gli insiemi (SET) e gli insiemi con possibilità di elementi ripetuti (MULTISET). I SET e i MULTISET possono anche essere visti come casi degeneri di MAP e MULTIMAP, nei quali l'unica cosa che viene memorizzata nel contenitore è il valore della chiave.

Analizziamo, come esempio iniziale, il contenitore sequenziale LIST. Questo contenitore permette di creare una lista di oggetti di un dato tipo. L'inserimento di un nuovo elemento nella lista sarà effettuato sempre in tempo costante (cioè *O(1)*), sia che venga inserito all'inizio, sia alla fine, sia in un punto interno della lista. La ricerca di un elemento o l'accesso ad un elemento presente nella lista richiede invece sempre un tempo lineare,

O(n), poiché richiede, nel caso peggiore, l'attraversamento di tutti gli elementi che lo precedono.

La dichiarazione di una lista si effettua istanziando un template. Supponiamo ad esempio di voler dichiarare una lista di interi:

```
#include <list>          // header file da includere
using namespace std;      // dichiarazione necessaria poiché
                          // la classe list è definita all'interno
                         // del namespace std

main()
{ list<int> lis;         // dichiarazione di una lista di interi
  lis.push_front(10);    // inserimento di un elemento all'inizio
  lis.push_back(20);     // inserimento di un elemento alla fine
  cout << lis.front();   // accesso al primo elemento e stampa
  cout << endl;
  cout << lis.back();    // accesso all'ultimo elemento e stampa
  cout << endl;
  cout << lis.size();    // stampa il numero di elementi di lis
  cout << endl;
  lis.back() = 30;       // modifica dell'ultimo elemento[9]
  cout << lis.back();
  cout << endl;
  lis.pop_front();       // cancellazione dell'elemento iniziale
  lis.pop_back();        // cancellazione dell'elemento finale
  if (lis.empty())       // true se la lista è vuota
    cout << "ok";        // deve stampare "ok"!
  cout << endl;
}
```

Per accedere agli elementi di un contenitore, oltre alle funzioni mostrate nell'esempio (che sono definite anche per altri contenitori, come sarà mostrato nel seguito), esistono gli *iteratori*, descritti nel paragrafo successivo.

5.4 Iteratori

Gli iteratori sono degli oggetti che vengono utilizzati per scorrere gli elementi di un contenitore e accedere a quelli che interessano. Le caratteristiche principali di un iteratore sono le seguenti:

- dato un iteratore, è possibile dereferenziarlo, accedendo all'oggetto a cui l'iteratore punta

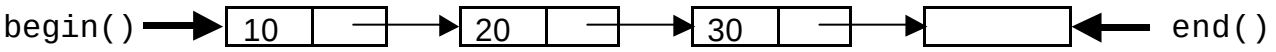
[9] Le funzioni `front()` e `back()` restituiscono per riferimento il tipo degli oggetti contenuti nella lista (ad esempio una usuale dichiarazione è `T& front();`), quindi possono essere usate a sinistra di un'assegnazione per modificare l'elemento iniziale o finale della lista.

- dato un iteratore, è possibile incrementarlo mediante l'operatore '++' per ottenere l'iteratore all'oggetto successivo contenuto nel contenitore
- dati due iteratori, è possibile confrontarli con l'operatore '==' per decidere se puntano allo stesso oggetto

Da queste caratteristiche emerge chiaramente che i puntatori sono dei buoni iteratori per il contenitore ARRAY. Ad esempio, dato un array di interi, i puntatori a intero (`int*`) corrispondono alle caratteristiche elencate.
Come per un array di interi esiste il tipo `int*` che ne costituisce l'iteratore, così per ogni contenitore esiste una classe che ne costituisce l'iteratore. Ad esempio, il classe degli iteratori per il contenitore `list<int>` è la classe `list<int>::iterator`, cioè una classe chiamata `iterator` e definita all'interno della classe `list`.

Notare che non esiste un iteratore nullo, analogo al puntatore NULL. Esiste però un iteratore che indica la fine del contenitore: ogni iteratore può essere confrontato con questo per determinare se punta ad un oggetto interno al contenitore o no, cioè per determinare se è valido. Questo iteratore, restituito dalla funzione pubblica `end()` presente in ogni classe contenitore, punta ad un elemento fittizio successivo all'ultimo elemento del contenitore.

Infine, tutte le classi contenitore mettono a disposizione, nella parte pubblica, la funzione `begin()` che restituisce l'iteratore al primo elemento del contenitore.



Nell'esempio seguente vediamo come si dichiara un iteratore e come sia possibile scorrere la sequenza di elementi di un contenitore accedendo a ciascuno di essi mediante la dereferenziazione dell'iteratore.

```
// scorrimento.cpp

#include <iostream.h>
#include <list>
#include <iterator>     // header file da includere

using namespace std;

main()
{ // dichiarazione e inizializzazione di una lista
  list<int> lis;
  lis.push_back(1);
  lis.push_back(3);
  lis.push_back(5);
  lis.push_back(7);
  lis.push_back(9);
  lis.push_back(11);

  // dichiarazione di un iteratore per una lista di int
  list<int>::iterator i;
```

```
  // stampa degli elementi della list
  for (i = lis.begin(); i != lis.end(); i++)
    cout << *i << endl;
}
```

Si noti che:

- `list<int>::iterator` è il tipo dell'iteratore. È necessario utilizzare per ogni contenitore un iteratore del tipo appropriato.
- Nel ciclo for inizializziamo l'iteratore mediante la funzione `begin()`, che restituisce l'iteratore al primo elemento della lista. Quindi alla prima iterazione `i` punterà al primo elemento di `lis`.
- La condizione di uscita del for consiste nel confronto dell'iteratore `i` con il valore restituito dalla funzione `end()`. Questa funzione restituisce l'iteratore ad un elemento fittizio successivo all'ultimo elemento, cioè indica la fine della lista. Quindi prima di uscire dal for si visiteranno tutti gli elementi di `lis`.
- La terza espressione del for è l'incremento dell'iteratore, ottenuto mediante l'operatore '`++`'.
- Questo esempio si può utilizzare per scorrere gli elementi di qualsiasi contenitore (cambiando il tipo del contenitore e il tipo dell'iteratore coerentemente)

Gli iteratori permettono agli algoritmi di accedere in maniera uniforme agli elementi di qualsiasi contenitore, e quindi permettono di progettare algoritmi generici, che possono operare su qualsiasi contenitore.

Un algoritmo generico effettua delle operazioni su una sequenza di elementi consecutivi contenuti in un contenitore. Tale sequenza deve essere indicata mediante due iteratori che puntano rispettivamente al primo elemento e all'elemento successivo all'ultimo. Quando la sequenza comprende tutti gli elementi del contenitore, si utilizzano le funzioni `begin()` e `end()` per ottenere i due iteratori. Se invece si tratta di una sottosequenza, è necessario individuare i due iteratori, come mostrato nell'esempio seguente.

Mostriamo come effettuare il ribaltamento di una sottosequenza mediante l'algoritmo generico `reverse()`.
Considerando la stessa sequenza dell'esempio precedente (1 3 5 7 9 11), vogliamo ottenere la nuova sequenza 7 5 3 1 9 11 in cui abbiamo ribaltato i primi quattro elementi.

Usiamo come iteratori che identificano la sottosequenza:

- `lis.begin()`
- l'iteratore che punta all'elemento di valore 9, il primo successivo alla sottosequenza

```
// sottosequenza.cpp

#include <iostream.h>
#include <list>
#include <iterator>
```

```
using namespace std;

main()
{ list<int> lis;
  lis.push_back(1);
  lis.push_back(3);
  lis.push_back(5);
  lis.push_back(7);
  lis.push_back(9);
  lis.push_back(11);

  list<int>::iterator i;

  // individuazione dell'iteratore che punta all'elemento
  // _successivo_ all'ultimo della sottosequenza che vogliamo
  // considerare
  for (i = lis.begin(); *i != 9; i++); // corpo del ciclo: VUOTO
  // adesso i punta all'elemento 9

  reverse(lis.begin(),i); // ribaltamento sottosequenza 1 3 5 7

  // stampa degli elementi della lista: 7 5 3 1 9 11
  for (i = lis.begin(); i != lis.end(); i++)
    cout << *i << endl;
}
```

Gli algoritmi generici saranno mostrati in maggiore dettaglio nel paragrafo 5.6.

5.5 Caratteristiche dei contenitori

5.5.1 Array

Sono gli array del C++, che sono assimilati a contenitore della standard library perché si possono applicare ad essi tutti gli algoritmi generici.

Caratteristiche:

- random access: il costo di un accesso in lettura o scrittura ad un elemento è costante (*O(1)*) e indipendente dalla dimensione dell'array e dalla posizione dell'elemento
- lunghezza fissa: una volta dichiarato un array, la sua dimensione è fissata definitivamente

5.5.2 Vector

Sono dei vettori monodimensionali che, a differenza degli array, hanno lunghezza variabile.

Caratteristiche:

- random access
- lunghezza variabile
- inserimenti e cancellazioni alla fine del vettore effettuati in tempo ammortato costante
- sono possibili inserimenti e cancellazioni in altre posizioni, ma richiedono lo spostamento a destra o a sinistra degli elementi successivi, e quindi un tempo lineare ($O(n)$).

Esempio di uso delle funzioni principali:

```
#include <vector>                    file da includere
using namespace std;
...
vector<int> v;                       dichiarazione
v.push_back(100);                    inserimento alla fine
v.push_back(200);
v.push_back(300);
cout << v[0] << endl;                accesso e stampa
v[0]++;                              modifica di un elemento esistente
v.pop_back();                        cancellazione dell'elemento finale
                                     dichiarazione e inizializzazione iteratore
vector<int>::iterator i = v.begin();
i++;                                 incremento iteratore
v.insert(i,400);                     inserimento in mezzo
v.erase(i);                          cancellazione in mezzo
v[0] = 300;                          ERRORE: inserimento di elementi nuovi
                                     solo con push_back() o insert()
for (i = v.begin(); i != v.end(); i++)
  cout << *i << endl;                scorrimento e stampa di tutti gli elementi
```

5.5.3 Deque

Sono delle code a doppio ingresso. Si possono cioè inserire elementi sia in testa che in coda, mediante appositi metodi, in maniera efficiente.

Caratteristiche:

- coda a doppio ingresso
- random access
- lunghezza variabile

- inserimenti e cancellazioni all'inizio e alla fine della deque effettuati in tempo ammortato costante
- sono possibili inserimenti e cancellazioni in altre posizioni, ma richiedono lo spostamento a destra o a sinistra degli elementi successivi, e quindi un tempo lineare ($O(n)$).

Esempio di uso delle funzioni principali:

`#include <deque>`	file da **includere**
`using namespace std;`	
`...`	
`deque<char> d;`	**dichiarazione**
`d.push_back('z');`	**inserimento** alla fine
`d.push_front('a');`	**inserimento** all'inizio
`cout << d[0] << d[1] << endl;`	**accesso** e stampa ("az")
`d[0] = 'A';`	**modifica** di un elemento esistente
`d.pop_back();`	**cancellazione** dell'elemento finale
`d.pop_front();`	**cancellazione** dell'elemento iniziale
`deque<char>::iterator i = ...;`	dichiarazione e inizializzazione **iteratore**
`d.insert(i,'X');`	**inserimento** in mezzo
`d.erase(i);`	**cancellazione** in mezzo
`d[2] = 'x';`	**ERRORE**: inserimento di elementi nuovi solo con `push_back()`, `push_front()` o `insert()`
`for (i = d.begin(); i != d.end(); i++)` `  *i = 'x';`	**scorrimento** di tutti gli elementi e loro sostituzione con 'x'

5.5.4 List

Sono delle liste doppiamente collegate. Cioè ogni elemento è connesso con l'elemento precedente e con quello successivo.

Caratteristiche:

- accesso lineare agli elementi (complessità $O(n)$ nel caso peggiore)
- lunghezza variabile
- inserimento e cancellazione in tempo costante ($O(1)$) in qualunque punto

Esempio di uso delle funzioni principali:

`#include <list>`	file da **includere**
`using namespace std;`	
`...`	
`list<float> lis;`	**dichiarazione**

```
lis.push_front(1.0);                  inserimento all'inizio
lis.push_back(2.0);                   inserimento alla fine
cout << lis.front();                  accesso al primo elemento e stampa
cout << lis.back();                   accesso all'ultimo elemento e stampa
list<float>::iterator i;              iteratore
i = lis.begin();                      i  punta al primo elemento (1.0)
i++;                                  i punta all'elemento 2.0
*i = 3.0;                             modifica di un elemento esistente
float temp[4] = {1.1,1.2,1.3,1.4};
lis.insert(i,&temp[0],&temp[4]);      inserimento di una sequenza
d.pop_back();                         cancellazione dell'elemento finale
d.pop_front();                        cancellazione dell'elemento iniziale
d.erase(i);                           cancellazione in mezzo
```

Notare l'inserimento di una sequenza: i numeri contenuti nell'array `temp` e indicati dai due iteratori (puntatori per un array) `&temp[0]` e `&temp[4]` (quest'ultimo punta all'elemento successivo a quello finale) vengono inseriti nella lista nell'ordine in cui si trovano e prima dell'elemento puntato da `i`.

5.5.5 Map

Una map è un contenitore associativo in cui sono memorizzate coppie di valori di tue tipi `K` e `T`. Il tipo `K` è detto chiave. Una map permette di trovare, in modo efficiente, il valore dell'elemento `T` associato a una data chiave `K`.

Caratteristiche:

- permette di trovare l'oggetto `T` associato a una chiave `K`
- senza ripetizioni: non possono esserci due chiavi uguali
- accesso in tempo logaritmico (*O(log n)*)

Esempio di uso delle funzioni principali:

```
#include <map>                        file da includere
#include <string>
using namespace std;
...
map<string,int> m;                    dichiarazione
map<string,int,less<string> > m;      dichiarazione se il compilatore non
                                      accetta gli argomenti di default
                                      nei template[10]
```

[10] Affinché gli oggetti in una map siano ordinati, è necessaria una funzione di ordinamento che, date due chiavi, sappia distinguere qual'è la minore. Questa funzione è per default l'operatore '<', ma deve essere specificata come parametro del template nei compilatori che non accettano argomenti default nei template.

```
m[“Rossi”] = 30;
```
inserimento mediante `operator[]()`

```
m[“Bianchi”] = 27;
```
inserimento mediante `operator[]()`

```
mm.insert(pair<string,int>("Verdi",18));
```
inserimento mediante `insert()` e costruttore della classe `pair` che forma la coppia da inserire nella map

```
cout << m.size() << endl;
```
stampa il **numero di elementi** di `m` (3)

```
cout << m[“Rossi”] << endl;
```
accesso e stampa del valore 30

```
cout << m[“Neri”] << endl;
```
VIENE CREATO l’elemento Neri-0 poiché non c’era alcun elemento con chiave Neri[11]

```
map<string,int>::iterator i;
```
iteratore

```
map<string,int,less<string> >::iterator i;
```
iteratore (no arg. default template)

```
i = m.find(“Verdi”);
```
cerca l’elemento con chiave “Verdi” e ne restituisce l’iteratore. Restituisce `end()` se non trovato

```
cout << (*i).first << endl;
```
stampa “Verdi” (la **chiave**)

```
cout << (*i).second << endl;
```
stampa 18 (il **valore**)

```
for (i = m.begin(); i != m.end(); i++)
  cout << (*i).second << endl;
```
stampa **ordinata** dei valori

```
m.erase(i);
```
cancellazione (cancellato Verdi-18)

```
m.erase(“Bianchi”);
```
cancellazione (cancellato Bianchi-27)

```
m.erase(m.begin(),m.end());
```
cancellazione di tutti gli elementi rimasti

```
if (m.empty())
  cout << “ok” << endl;
```
`true` **se** `m` **è vuoto**, (deve effettuare la stampa)

5.5.6 Multimap

È una map che può contenere due o più coppie aventi la stessa chiave.

Caratteristiche:

- permette di trovare l’oggetto `T` associato a una chiave `K`
- con ripetizioni: possono esservi due chiavi uguali
- accesso in tempo logaritmico (*O(log n)*)

In questo caso usiamo come parametro `less<K>`, un *function object* (vedi paragrafo 5.9) che effettua il confronto tra due chiavi `K` utilizzando l’`operator<()` del tipo `K`.

[11] Se si tenta di accedere ad un elemento che non esiste, l’operatore ‘[]’ crea un nuovo elemento con la chiave data e con valore inizializzato dal costruttore senza argomenti della classe `T`. L’operatore ‘[]’ restituisce un riferimento all’elemento creato.

Esempio di uso delle funzioni principali:

```
#include <map>                          file da includere
#include <string>
using namespace std;
...
multimap<string,int> mm;                dichiarazione
multimap<string,int,less<string> > mm;  dichiarazione (no arg. default
                                            template)
                                        inserimento mediante insert() e
                                        costruttore della classe pair che forma la
                                        coppia da inserire nella multimap
mm.insert(pair<string,int>("Rossi",30));
mm.insert(pair<string,int>("Bianchi",27));
mm.insert(pair<string,int>("Verdi",0));
mm.insert(pair<string,int>("Verdi",18));
~~mm[“Neri”] = 24;~~                    ERRORE: operator[]() non definito
cout << mm.size() << endl;              numero di elementi di mm (4)
cout << mm.count(“Verdi”) << endl;      numero di elementi con chiave Verdi
multimap<string,int>::iterator i;       iteratore
                                        iteratore (no arg. default template)
multimap<string,int,less<string> >::iterator i;
i = mm.find(“Verdi”);                   cerca il primo elemento con chiave “Verdi”
                                        e ne restituisce l’iteratore; restituisce
                                        end() se non trovato
cout << (*i).first << endl;             stampa “Verdi” (la chiave)
cout << (*i).second << endl;            stampa 0 (il valore)
                                        stampa ordinata dei valori
for (i = mm.begin(); i != mm.end(); i++)
  cout << (*i).second << endl;
i = mm.find(“Verdi”);
mm.erase(i);                            cancellazione (cancellato Verdi-0)
mm.erase(“Bianchi”);                    cancellazione (cancellato Bianchi-27)
mm.erase(mm.begin(),mm.end());          cancellazione di tutti gli elementi rimasti
if (mm.empty())                         true se m è vuoto,
  cout << “ok” << endl;                 (deve effettuare la stampa)
```

5.5.7 Set

Sono insiemi ordinati di elementi. Tutti gli elementi devono essere differenti tra loro.

Caratteristiche:

- insiemi senza ripetizioni di elementi di un tipo generico, detto chiave
- accesso in tempo logaritmico (O(log n)) a qualsiasi elemento dell’insieme

Esempio di uso delle funzioni principali:

```
#include <set>                          file da includere
using namespace std;
...
set<int> s;                             dichiarazione
set<int, less<int> > s;                 dichiarazione (no arg. default template)
s.insert(10);                           inserimento
int temp[4] = {11,13,14,12};
s.insert(&temp[0],&temp[4]);            inserimento di una sequenza
cout << s.size() << endl;               stampa il numero di elementi di s  (5)
set<int>::iterator i;                   iteratore
set<int, less<int> >::iterator i;       iteratore (no arg. default template)
i = s.find(11);                         i punta a 11 (vale end( ) se non trovato)
s.erase(i);                             cancellazione in mezzo (cancellato 11)
s.erase(13);                            cancellazione dell'elemento di valore 13
for (i = s.begin(); i != s.end(); i++)
  cout << *i << endl;                   stampa ordinata di tutti gli elementi
s.erase(s.begin(),s.end());             cancellazione di tutti gli elementi rimasti
if (s.empty())                          true se ms è vuoto,
  cout << "ok" << endl;                 (deve effettuare la stampa)
```

5.5.8 Multiset

Multiinsiemi, cioè insiemi in cui possono essere presenti elementi uguali.

Caratteristiche:

- insiemi con ripetizioni
- accesso in tempo logaritmico ($O(\log n)$) a qualsiasi elemento dell'insieme

Esempio di uso delle funzioni principali:

```
#include <set>                          file da includere
using namespace std;
...
multiset<int> ms;                       dichiarazione
multiset<int, less<int> > ms;           dichiarazione (no arg. default template)
ms.insert(16);                          inserimento
int temp[6] = {15,12,12,12,11,10};
ms.insert(&temp[0],&temp[6]);           inserimento di una sequenza
cout << ms.size() << endl;              numero di elementi di ms  (7)
cout << ms.count(12) << endl;           numero di elementi uguali a 12
multiset<int>::iterator i;              iteratore
multiset<int, less<int> >::iterator i;
```

```
                                    iteratore (no arg. default template)
i = ms.find(15);                    i  punta all'elemento di valore 15
ms.erase(i);                        cancellazione in mezzo (cancellato 15)
ms.erase(12);                       cancellazione di tutti gli elementi = a 12
for (i = ms.begin(); i != ms.end(); i++)
  cout << *i << endl;               stampa ordinata di tutti gli elementi
if (ms.empty())                     true se ms è vuoto,
  cout << "nok" << endl;            (non deve effettuare la stampa)
```

5.5.9 Esempi di uso del contenitore Map

Come primo esempio di uso della map, vogliamo creare una struttura dati per contare il numero di occorrenze delle lettere minuscole all'interno di una parola. Costruiamo allora una `map<char,int>` in cui ogni coppia contiene una lettera e il numero di occorrenze di quella lettera nella parola.

```
// occorrenze.cpp
#include <iostream.h>
#include <map>
#include <string>

using namespace std;

main()
{ string parola;
  cin >> parola;
  map<char,int,less<char> > occorrenze;
  for (int i = 0; i < parola.length()-1; i++)
  { char c = parola[i];
    if (c >= 'a' && c <= 'z')
      occorrenze[c]++;
  }
  for (char x = 'a'; x <= 'z'; x++)
    if (occorrenze[x] != 0)
      cout << "La lettera " << x << " appare " << occorrenze[x]
           << " volte." << endl;
}
```

Notare il modo di accedere un elemento della map: sia in lettura che in scrittura si può usare una notazione con parentesi quadre, simile a quella degli array, in cui l'indice è del tipo K (chiave), nel nostro caso `char`.

Un accesso in lettura restituisce l'intero associato alla chiave o, se non esiste l'elemento nella map, crea un elemento con la chiave data e il valore 0, e restituisce quindi 0.

Un accesso in scrittura aggiorna l'elemento – se esiste, oppure lo crea automaticamente e lo inizializza con il valore intero passatogli.

Come ulteriore esempio di uso di una map, realizziamo una rubrica telefonica molto semplice. Gli elementi nella map saranno delle coppie nome/numero telefonico, e rappresenteremo i nomi mediante stringhe e i numeri telefonici mediante interi (per semplicità; notare che non si possono rappresentare così i numeri che iniziano per 0).
Usiamo la classe `string` della standard library, dichiarata nel file `string.h` che dobbiamo includere.
La chiave della map è di tipo `string`, e quindi i dati inseriti nella rubrica risulteranno ordinati e accessibili mediante il nome.

```
// rubrica.cpp

#include <iostream.h>
#include <map>
#include <string>

using namespace std;

main()
{ map<string,int,less<string> > rubrica;
  rubrica["Emanuele"] = 12345;
  rubrica["Giovanni"] = 23456;
  rubrica["Anna"] = 12987;

  string nome;
  while (cin >> nome)
    if (rubrica.find(nome) != rubrica.end())
      cout << "Il numero di " << nome << " e' "
           << rubrica[nome] << endl;
    else
      cout << "nome non presente." << endl;
}
```

In questo semplice esempio memorizziamo nella rubrica tre nomi; poi effettuiamo un ciclo while che continua finché non inseriamo un nome vuoto, digitando il tasto invio senza digitare alcun carattere.
All'interno del while usiamo la funzione `find()` per accertarci che il nome sia presente tra le chiavi contenute nella rubrica. La funzione ci restituisce un iteratore che punta alla coppia nome-valore oppure l'iteratore che punta all'elemento successivo all'ultimo elemento contenuto nella map se non c'è un elemento con la chiave cercata. Possiamo quindi verificare se l'elemento con chiave `nome` sia contenuto nella rubrica confrontando il risultato della `find()` con l'iteratore `end()`.
Se l'elemento è presente stampiamo il numero telefonico, accedendovi mediante l'uso delle parentesi quadre e specificando `nome` come indice. In caso contrario stampiamo la frase "nome non presente".

5.6 Algoritmi generici

Analizziamo l'algoritmo generico `find()` che ricerca un determinato valore all'interno di una sequenza. La sequenza può essere contenuta in qualsiasi contenitore della standard library, e mostreremo nel seguito quattro programmi d'esempio (tratti da [STL]) che applicano l'algoritmo `find()` a un array, a un vector, a una list e a una deque.

5.6.1 Algoritmo `find()` su un array

In questo programma inizializziamo un array di char e cerchiamo un carattere al suo interno. La sequenza su cui effettuiamo la ricerca comprende tutti i caratteri dell'array, quindi utilizziamo i due puntatori `&s[0]` e `&s[len]` per identificarla.

```
// find-arr.cpp

#include <iostream.h>
#include <string>
#include <assert.h>
#include <algorithm>

using namespace std;

main()
{ char* s = "Algoritmo find() su array";
  int len;
  for (len = 0; s[len] != '\0'; len++); // calcolo lunghezza di s

  // ricerca della lettera ‘d’ nella stringa s

  char* pd = find(&s[0],&s[len],'d');

  assert (pd != &s[len] && *pd == 'd');
}
```

5.6.2 Algoritmo `find()` su un vector

Notare l'uso del costruttore a due argomenti della classe vector per inizializzare il vector con i dati contenuti in un array.

```
//find-vector.cpp
```

```
#include <iostream.h>
#include <string>
#include <assert.h>
#include <algorithm>
#include <vector>

using namespace std;

main()
{ char* s = "Algoritmo find() su vector";
  int len;
  for (len = 0; s[len] != '\0'; len++); // calcolo lunghezza di s

  // inizializzazione di un vector con i caratteri di s
  vector<char> v(&s[0],&s[len]);

  // ricerca della lettera ‘d’ nel vector v
  vector<char>::iterator iv = find(v.begin(),v.end(),'d');

  assert (iv != v.end() && *iv == 'd');
}
```

5.6.3 Algoritmo `find()` su una list

```
//find-list.cpp

#include <iostream.h>
#include <string>
#include <assert.h>
#include <algorithm>
#include <list>

using namespace std;

main()
{ char* s = "Algoritmo find() su list";
  int len;
  for (len = 0; s[len] != '\0'; len++); // calcolo lunghezza di s

  // inizializzazione di una list con i caratteri di s
  list<char> lis(&s[0],&s[len]);

  // ricerca della lettera 'd' nella list lis
  list<char>::iterator il = find(lis.begin(),lis.end(),'d');

  assert (il != lis.end() && *il == 'd');
}
```

5.6.4 Algoritmo `find()` su una deque

```
//find-deque.cpp

#include <iostream.h>
#include <string>
#include <assert.h>
#include <algorithm>
#include <deque>

using namespace std;

main()
{ char* s = "Algoritmo find() su deque";
  int len;
  for (len = 0; s[len] != '\0'; len++); // calcolo lunghezza di s

  // inizializzazione di una deque con i caratteri di s
  deque<char> deq(&s[0],&s[len]);

  // ricerca della lettera 'd' nella deque deq
  deque<char>::iterator id = find(deq.begin(),deq.end(),'d');

  assert (id != deq.end() && *id == 'd');
}
```

5.6.5 Algoritmo `accumulate()`

Come ulteriore esempio mostriamo l'algoritmo `accumulate()` usato per sommare fra loro tutti gli elementi di un contenitore. `accumulate()` richiede tre argomenti: i primi due sono gli iteratori all'inizio e alla fine+1 della sequenza, mentre il terzo è un valore iniziale che viene sommato al totale calcolato e che può essere 0 se non necessario. La accumulate è dichiarata in `numeric.h`.

```
// accu.cpp

#include <iostream.h>
#include <numeric>
#include <set>

using namespace std;

main()
{ // dichiariamo e inizializziamo un insieme di interi
```

```
  set<int,less<int> > s;
  for (int i = 1; i < 11; i++)
    s.insert(i);    // si usa la insert() per inserire
                    // un elemento in un set

  // sommiamo gli interi mediante la accumulate()

  int somma = accumulate(s.begin(),s.end(),0);

  cout << "somma = " << somma << endl;
}
```

5.6.6 Elenco degli algoritmi generici

Elenchiamo nel seguito i nomi degli algoritmi generici della standard library. Il nome della maggior parte di essi spiega ciò che l'algoritmo fa. Per tutti gli algoritmi è comunque possibile trovare informazioni su ciò che fanno e sui parametri da usare per l'invocazione sia in letteratura (ad es. [Strou] e [STL]) che negli help online dei principali compilatori.

Algoritmi dichiarati in `algorithm.h`:

adjacent_find	partition
binary_search	pop_heap
copy	prev_permutation
copy_backward	push_heap
count	random_shuffle
count_if	remove
equal	remove_copy
equal_range	remove_copy_if
fill	remove_if
fill_n	replace
find	replace_copy
find_first_of	replace_copy_if
find_if	replace_if
for_each	reverse
generate	reverse_copy
generate_n	rotate
includes	rotate_copy
inplace_merge	search
iter_swap	set_difference
lexicographical_compare	set_intersection
lower_bound	set_symmetric_difference
make_heap	set_union
max	sort
max_element	sort_heap
merge	stable_partition

min	stable_sort
min_element	swap
mismatch	swap_ranges
next_permutation	transform
nth_element	unique
partial_sort	unique_copy
partial_sort_copy	upper_bound

Algoritmi dichiarati in `numeric.h`:

accumulate	adjacent_difference
inner_product	partial_sum

5.7 Adattatori

Gli adattatori modificano l'interfaccia di un altro componente della standard library, sia esso un contenitore, un iteratore o una funzione.

Un classico esempio è l'uso di un adattatore per modificare l'interfaccia di un contenitore vector al fine di utilizzarlo come una pila, cioè con logica LIFO.

Il programma seguente usa l'adattatore `stack`, dichiarato in `stack.h`. Nella dichiarazione di uno stack è necessario specificare il tipo degli elementi che dovrà contere e il tipo di contenitore che deve utilizzare.

```
// stack.cpp

#include <iostream.h>
#include <vector>
#include <stack>

using namespace std;

main()
{ stack<int,vector<int> > pila;
  pila.push(10);
  pila.push(20);
  cout << "dimensione della pila: " << pila.size() << endl;
  while (!pila.empty())
  { cout << pila.top() << endl;
    pila.pop();
  }
}
```

È possibile dichiarare uno stack utilizzando altri contenitori (es. `stack<int,list<int> >`, `stack<int,deque<int> >`).

In maniera simile è possibile dichiarare una coda usando l'adattatore `queue`.

```
// queue.cpp

#include <iostream.h>
#include <list>
#include <queue>

using namespace std;

main()
{ queue<int,list<int> > coda;
  coda.push(10);
  coda.push(20);
  cout << "dimensione della coda: " << coda.size() << endl;
  while (!coda.empty())
  { cout << coda.front() << endl;
    coda.pop();
  }
  cout << "dimensione della coda: " << coda.size() << endl;
}
```

Mostriamo adesso un adattatore per un iteratore.
Supponiamo di voler scorrere una sequenza partendo dall'ultimo elemento. Per fare ciò possiamo utilizzare un `reverse_iterator`, cioè un adattatore che modifica un iteratore permettendo di utilizzare gli operatori '++' e '—' per puntare rispettivamente all'elemento precedente e seguente.

L'iteratore `ri` utilizzato nel programma è di tipo `vector<int>::reverse_iterator`. Utilizziamo le funzioni `rbegin()` e `rend()` per inizializzarlo e per controllare la condizione di uscita dal loop. Queste due funzioni restituiscono rispettivamente gli iteratori all'ultimo e al primo elemento del vector. L'operatore '++', applicato a un `reverse_iterator`, permette di modificare il `reverse_iterator` in modo che punti all'elemento precedente a quello corrente.

```
// reverse_iterator.cpp

#include <iostream.h>
#include <vector>

using namespace std;

main()
{ vector<int> v;
  v.push_back(10);
  v.push_back(20);
  v.push_back(30);
  v.push_back(40);
```

```
  v.push_back(50);

  vector<int>::reverse_iterator ri;

  // stampa degli elementi in ordine inverso
  for (ri = v.rbegin(); ri != v.rend(); ri++)
    cout << *ri << endl;
}
```

La classe `vector<int>::reverse_iterator` usata nel programma è definita nella standard library come un adattatore per iteratori. Se avessimo voluto dichiarare l'adattatore esplicitamente nel programma avremmo dovuto dichiarare `ri` come segue:

```
reverse_iterator<vector<int>::iterator,int,int&,ptrdiff_t> ri;
```

che mostra più chiaramente il fatto che si sta dichiarando un adattatore per un `vector<int>::iterator`.

5.8 Allocatori

Non trattati in questo corso.

5.9 Function objects

Non trattati in questo corso.

6 Ereditarietà

L'ereditarietà permette di attribuire delle proprietà a una classe non definendole esplicitamente ma derivandole da altre classi più generali.

La definizione della classe consiste quindi nella specifica delle differenze che la classe ha rispetto alla classe da cui eredita.
Le due classi sono dette *classe base* e *classe derivata*.

Data una classe base `B` possiamo derivare da essa una nuova classe `D` specificando ciò che `D` ha in più rispetto a `B` (campi e/o funzioni).

```
class B
{
  // ...
};

class D : public B
{
  // ...
};
```

Ogni oggetto `d` della classe `D` è considerato anche un oggetto di `B`, ed è possibile utilizzare `d` in qualsiasi contesto in cui sarebbe possibile utilizzare un oggetto `b` della classe `B`: come parametro di funzione, come oggetto di invocazione, etc.

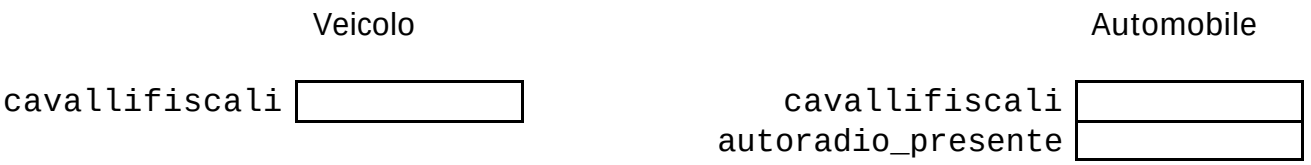
Tutte le funzioni proprie della classe `B` sono considerate come funzioni proprie anche per la classe `D`, quindi gli oggetti di `D` possono essere utilizzati da tutte le funzioni proprie delle due classi.

Esempio:

```
class Veicolo
```

```
{private:
  int cavallifiscali;
 public:
  int PrezzoBollo();
};

int Veicolo::PrezzoBollo()
{ return cavallifiscali*5000; }

class Automobile : public Veicolo
{private:
  bool autoradio_presente;
};
```

Gli oggetti della classe `Automobile` sono formati da due campi: `cavallifiscali` e `autoradio_presente`. Gli oggetti di `Veicolo` hanno invece solo il primo campo `cavallifiscali`.



Una porzione di codice come la seguente:

```
Automobile a;
cout << a.PrezzoBollo();
```

stampa in output il prezzo del bollo dell'automobile senza considerare il fatto che l'autoradio sia presente o meno e quindi senza aggiungere il prezzo dell'abbonamento radiofonico. Se vogliamo considerare l'abbonamento radiofonico, e considerarlo solo per le automobili (gli altri veicoli non hanno in generale l'autoradio, e non abbiamo infatti neanche definito il campo `autoradio_presente` nella classe base) dobbiamo ridefinire all'interno della classe `Automobile` la funzione `PrezzoBollo`. La classe viene modificata come segue:

```
class Automobile : public Veicolo
{private:
  bool autoradio_presente;
 public:
  int PrezzoBollo();
};

int Automobile::PrezzoBollo()
{ int abbonamentoradio = 20000;
  if (autoradio_presente)
    return cavallifiscali * 5000 + abbonamentoradio;
  else
    return cavallifiscali * 5000;
```

```
}
```

A questo punto le seguenti istruzioni chiamano, nell'ordine, `Automobile::PrezzoBollo()` e `Veicolo::PrezzoBollo()`:

```
Automobile a;
Veicolo v;
cout << a.PrezzoBollo();
cout << v.PrezzoBollo();
```

Che succede nel caso in cui a compile time non si conosce il tipo effettivo di un oggetto? Osserviamo questo esempio:

```
Veicolo* pv;
pv = new Automobile; // lecito
cout << pv->PrezzoBollo() << endl;
```

Quale funzione viene chiamata? Quella della classe `Veicolo`, poiché il compilatore sa che `pv` è un puntatore a veicolo e non può tenere memoria dei valori che gli vengono assegnati (nel nostro caso, il compilatore non può "ricordare" il fatto che abbiamo assegnato a `pv` un puntatore a `Automobile`).

Allora in questo caso il calcolo del prezzo del bollo sarebbe errato.

Per risolvere questo problema si può dichiarare la funzione `PrezzoBollo()` come *virtuale*.

Questo significa che il compilatore non prenderà decisioni su quale delle due funzioni debba essere chiamata ma rimanderà questa decisione al momento dell'esecuzione del programma. Infatti a runtime è possibile verificare effettivamente di che tipo sia l'oggetto puntato da `pv`, e scegliere la funzione appropriata.

Per dichiarare una funzione come virtuale è necessario usare la parola chiave `virtual` nella dichiarazione fatta all'interno della classe base. Riportiamo la nuova definizione della classe `Veicolo`:

```
class Veicolo
{private:
  int cavallifiscali;
 public:
  virtual int PrezzoBollo();
};
```

---

14 luglio 2000; 13 novembre 2000; 8 gennaio 2001

Indice