CA/NI/4-97

Istituto Nazionale di Fisica Nucleare
Sezione di Cagliari

Programmazione Orientata agli Oggetti

Concetti di base ed introduzione al C++

Dicembre '97

A cura di
Antonio Silvestri

Indice

Capitolo 1

1 Concetti di base

La programmazione orientata agli oggetti (*Object-Oriented Programming*, per brevità faremo riferimento ad essa con la sigla OOP) è una metodologia, non nuova per la verità, che si sta affermando solo in questi ultimi anni nel settore dello sviluppo del software, a causa dei gravissimi problemi del quale soffre il corrente paradigma di progettazione e sviluppo del software.
Questa nuovo metodo di programmazione risale agli anni 80 quando dopo anni di ricerca condotta dal Software Concepts Group presso la Xerox PARC venne presentato il sistema **Smalltalk-80** il quale può senz'altro essere riconosciuto, insieme al linguaggio **Simula**, il capostipite dei linguaggi orientati agli oggetti. La sua influenza su molti prodotti commerciali è stata notevole, basti pensare ai sistemi MacIntosh della Apple, ai sistemi CAD, alle interfacce grafiche per Workstation ed ai sistemi ad icone ed altri ambienti che riguardano in generale *l'Office Automation*.
SmallTalk è un linguaggio di programmazione interpretato ed un ambiente di programmazione e di lavoro integrato dove non esiste soluzione di continuità tra il Sistema Operativo e le applicazioni dell'utente; l'intero sistema cresce e si modifica con le necessità dell'utente. Non esiste tipizzazione dei dati, tutto è dinamico ed ogni cosa è un oggetto, non solo entità di alto livello come finestre o menu ma anche entità di livello più basso come numeri, pixel e caratteri; è quello che si definisce un sistema completamente autoconsistente.
L'ambizione dello SmallTalk è quello di fornire un ambiente integrato dando l'opportunità all'utente di interagire con la macchina, ad un livello molto alto, in un modo più aderente alle problematiche che le persone devono risolvere. Infatti, queste ultime si occupano di concetti nel dominio dei problemi mentre il computer lavora con concetti differenti. Questo ha fatto sì che i progettisti di SmallTalk considerassero di primaria importanza il trasferimento di tutte le funzioni gravose in termini di risorse uomo verso la macchina. SmallTalk incarna tutte queste idee.

1.1 Crisi del software

Negli ultimi 30 anni l'hardware dei computer ha subito numerose evoluzioni e rivoluzioni con un ritmo che non accenna, tuttora, a diminuire. L'ingegneria dell'hardware progredisce ed evolve continuamente. E l'ingegneria del software? Se si confrontano i progressi fatti dall'ingegneria del software rispetto a quelli dell'hardware il risultato è molto deludente, i cambiamenti avvenuti in questo settore sono veramente modesti. Gli ingegneri del software hanno continuato ed in genere continuano ha sviluppare programmi sempre con le stesse regole. Il modello di programmazione corrente è quello del progetto *top-down* che consiste in

Analisi ➔ Scomposizione ➔ Procedure e Funzioni

Con questo metodo di decomposizione algoritmica il progetto di un sistema complesso viene diviso via via in problemi più semplici da affrontare identificando dei moduli che realizzano le operazioni che il sistema deve svolgere; l'intero sistema è quindi concepito come una interazione di procedure e funzioni che operano, in sequenza o in parallelo su strutture dati comuni e predefinite. In definitiva si puó affermare che il sistema complesso del mondo "reale" é stato convertito in un certo numero di problemi del "computer". I linguaggi di programmazione si sono adattati a questa filosofia promuovendo il diffondersi della programmazione strutturata (Pascal, Algol, C, Ada).

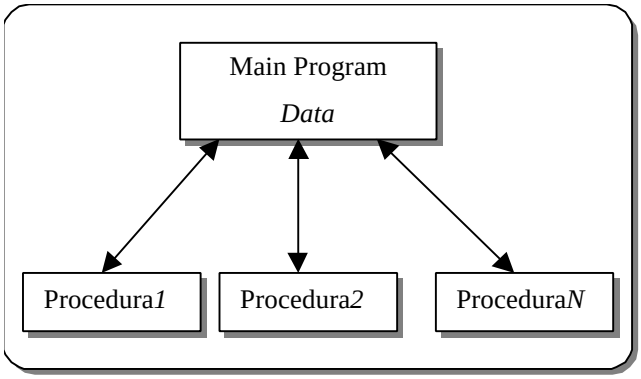


Figura 1.1-1

Possiamo illustrare questo tipo di procedimento come nella Figura 1.1-1 dove esiste un programma principale con una struttura dati ed un insieme di subroutine le quali vengono chiamate per eseguire delle operazioni su questi dati.
Un miglioramento a questo metodo è stato apportato con l'introduzione del concetto di modularità, rappresentato dalla Figura 1.1-2 nel quale un certo numero di procedure con funzionalità simili sono raggruppati in moduli indipendenti. Questo é un primo tentativo di disaccoppiare le procedure, nel quale é stato scomposto l'applicazione, e di limitare l'accesso ai dati globali solo ad alcune specializzate procedure.

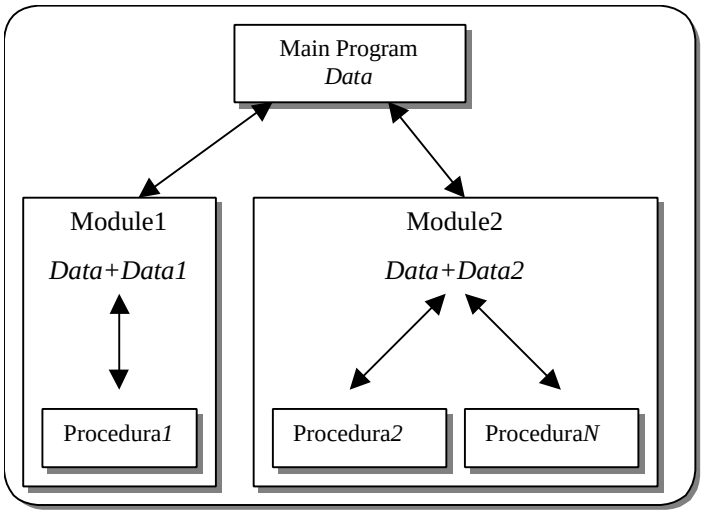


Figura 1.1-2

Questo approccio alla programmazione ha però ancora notevoli punti deboli specialmente a fronte delle modifiche che si rendono necessarie per mutate necessità e di riutilizzo dei programmi. Questo è particolarmente vero per applicazioni di grandi dimensioni che a differenza del singolo programma sono tutt'altro che flessibili anzi di solito sono molto complessi e rigidi, dove un cambiamento in una parte del programma si propaga per tutta la struttura con effetti difficilmente prevedibili.

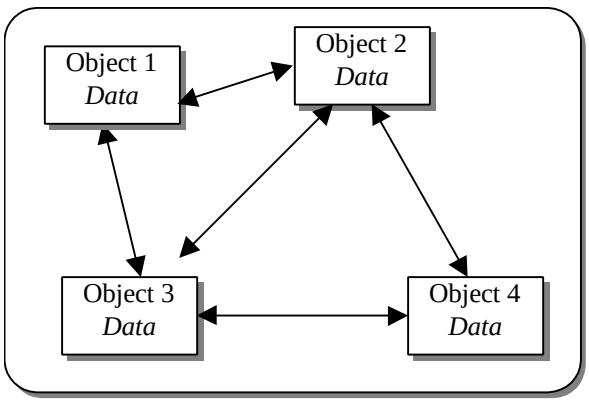
1.2 La programmazione per oggetti

La programmazione per oggetti è un modo alternativo di affrontare e scomporre il progetto software. Il suo paradigma è che l'unità elementare di scomposizione non è più la procedura, la subroutine ma l'*oggetto,* inteso come modello di un entità reale. Quest'ultima operazione chiamata ***abstraction*** (o ***data abstraction***) ha un importanza fondamentale nei linguaggi orientati agli oggetti ed è uno dei metodi fondamentali con il quale gli esseri umani affrontano i sistemi complessi. Esistono varie definizioni del concetto di astrazione, Booch la definisce come segue:

> *Con l'attività di astrazione si individuano le caratteristiche di un oggetto che lo distinguono da tutti gli altri tipi di oggetti fornendo quindi, per l'oggetto in questione, un ben preciso ambito concettuale relativo al punto di vista dell'osservatore.*

L'attività di *astrazione* si concentra sulle caretteristiche esteriori di un oggetto (cioè quali sono le azioni che è in grado di compiere) e quindi serve a separare il comportamento dell'oggetto dai dettagli della sua reale implementazione. Un sistema complesso viene così visto come un insieme di oggetti interagenti (vedi Figura 1.2-1) ciascuna provvisto di una sua struttura dati e dall'insieme delle operazioni che l'oggetto sa effettuare su tale struttura. Un oggetto con inclusa la propria struttura dati non viene influenzato dai cambiamenti che avvengono al di fuori di esso.

Figura 1.2-1



1.3 Oggetti, Metodi e Messaggi

Da quanto detto in precedenza possiamo affermare che qualcosa è definibile come un *oggetto* se possiede le seguenti caratteristiche:

- Una propria struttura dati, chiamati anche *attributi* o *proprietà*, i quali specificano lo ***stato*** nel quale si trova un oggetto
- Un insieme di operazioni chiamati ***metodi***, che sono l'interfaccia dell'oggetto verso l'esterno e che ne definiscono il suo ***comportamento***.

Solo attraverso i metodi, dei quali è nota la funzione ma non la loro implementazione, la struttura dati dell'oggetto è visibile. Questo principio è conosciuto in OOP come ***encapsulation*** il quale permette un alto grado di modularitá: il codice sorgente di un oggetto puó essere scritto e modificato in modo indipendente dai sorgenti degli altri oggetti.
La Figura 1.3-1 illustra un esempio di un oggetto, astrazione di una automobile: Si possono identificare delle variabili che indicano il suo stato corrente (luci, marcia, velocitá) e dei metodi per cambiare la marcia, accendere le luci, frenare o accellerare. Nella figura i metodi avvolgono i dati interni in modo da visualizzare il concetto di *information hiding* e *encapsulation*.

Figura 1.3-1 :rappresentazione dell'oggetto auto

Encapsulation e *abstraction* sono concetti complementari: il primo riguarda l'implementazione dell'oggetto che rende conto del suo comportamento, mentre il secondo attiene al suo comportamento esteriore.
L'*encapsulation* viene ottenuto attraverso l'***information hiding*** il quale è il processo di mascherare tutti i dettagli di un oggetto che non contribuiscono alle sue caratteristiche essenziali; di solito la struttura interna di un oggetto è nascosta come pure l'implementazione dei suoi metodi.

Un oggetto comunica con un'altro oggetto tramite dei ***messaggi***. Un messaggio (con eventuali parametri) è una richiesta inviata ad un oggetto allo scopo di attivare delle operazioni in esso definite. Questo viene realizzato con una chiamata ad un metodo appartenente all'oggetto ricevente. In pratica a differenza di quanto avviene nei comuni linguaggi di programmazione il messaggio comunica all'oggetto cosa deve fare non di come lo deve eseguire (è solo l'oggetto ricevente che conosce il modo nel quale deve reagire alla ricezione di un dato messaggio).
Nel succitato esempio dell'automobile l'oggetto "autista", ai comandi dell'auto, invia messaggi all'oggetto "auto" utilizzando i suoi metodi di interfaccia (interruttore delle

luci, leva di innesto marcia e pedale dell'accelleratore). L'oggetto "autista" ignora completamente come queste azioni verrano eseguite dall'oggetto "auto".

Naturalmente la programmazione orientata agli oggetti è qualcosa di più vasto che comprende altri concetti che aumentano la flessibilità e la riutilizzabilità del software sviluppato con questo criterio.

1.4 Classe

Per trarre vantaggio dalla definizione di oggetto abbiamo bisogno di un sistema di classificazione degli oggetti. A questo scopo si definisce ***classe*** un insieme di oggetti che condividono le stesse proprietà e lo stesso comportamento. Questo corrisponde grosso modo al concetto di tipo di dato negli altri linguaggi di programmazione.

Un oggetto è quindi un esempio, un membro, una *instance* di una data *classe*. Una classe è l'implementazione e la rappresentazione reale di un tipo di dato astratto.

Un abete è un albero in molte caratteristiche è come tutti gli altri alberi tuttavia possiede delle caratteristiche diverse dagli altri alberi, che lo distinguono dai suoi simili. La *instance* abete è un vegetale tangibile, concreto mentre la classe albero è una astrazione.

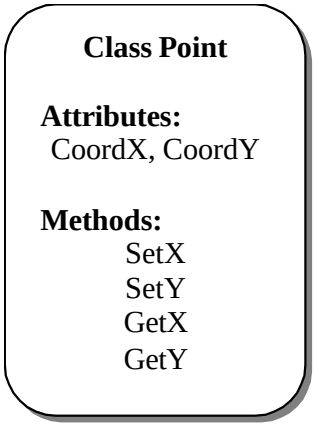


Figura 1.4-1

Come esempio pratico si veda la classe Point descritta nella Figura 1.4-1 essa è costituita da un area dati chiamati attributi (chiamati anche *instance variables*) denominati CoordX e CoordY che indicano le coordinate del punto e da una serie di metodi (SetX, SetY, GetX, GetY) che specificano le operazioni che la classe Point è in grado di fare su queste coordinate cioè leggere e modificare le coordinate del punto.

Un programmatore *object-oriented* modella i problemi reali (fase di *abstraction*) scrivendo classi le cui *instance* (gli oggetti) gli permetteranno di risolvere il suo problema.

1.5 Ereditarietà

L'***ereditarietà*** (*inheritance*) insieme al concetto di *encapsulation* è la caratteristica fondamentale della OOP. In pratica si può definire come quella proprietà che permette di costruire *classi* a partire da altre *classi*. Il vantaggio di questa tecnica è molto importante perché ci dà la chiave per riutilizzare il software già sviluppato o che si ha già a disposizione e permette all'utente di pensare e programmare per differenze.

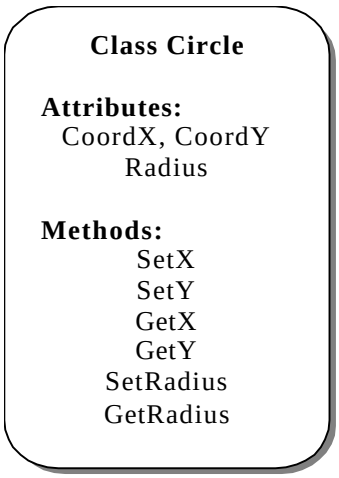


Figura 1.5-1

Come esempio si guardi la Figura 1.5-1 che descrive la classe Circle che a differenza della classe Point della figura precedente ha come attributo aggiuntivo il raggio (Radius) e i metodi (SetRadius, GetRadius) per leggere e variare questo valore; tutti gli altri parametri rimangono gli stessi della classe Point. Ebbene, il principio di ereditarietà afferma che non è necessario scrivere ex novo la classe Circle, ma è sufficiente definire la classe Circle come sottoclasse (*subclass*) della classe Point. La sottoclasse Circle eredita tutte le caratteristiche della classe Point (detta anche *Superclass* di Circle) cioè tutti i *metodi* e tutti gli *attributi*. Quindi la definizione della classe Circle sarà come quella illustrata nella Figura 1.5-2. Nella quale gli attributi e metodi della classe Point sono stati integrati con altri in modo da formare una classe diversa: Circle.

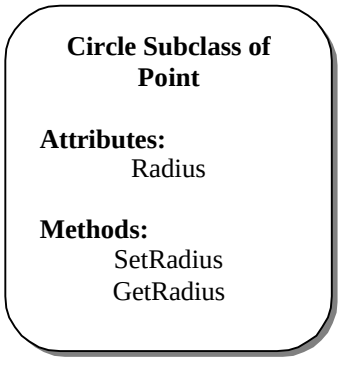


Figura 1.5-2

Una conseguenza importante di questa gerarchia di classi è il fatto che qualsiasi variazione in una data classe (variazioni di un metodo o cambiamento delle proprietà) si ripercuote in modo automatico su tutte le classi derivate senza dover operare nessun aggiustamento manuale che potrebbe essere scomodo e difficile da eseguire.

1.6 Binding statico e dinamico

Questo concetto è molto importante nei linguaggi orientati agli oggetti e si riferisce al tempo nel quale i nomi (identificatori delle variabili e funzioni) che si usano nel

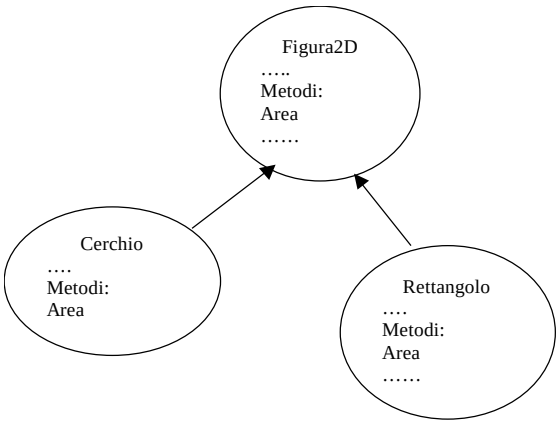
contesto di un dato linguaggio vengono associati (*bound*) ad un certo tipo (o classe). Un *binding statico* (chiamato anche *early binding*) significa che il tipo di tutte le variabili (o la classe di appartenenza di tutti gli oggetti) ed espressioni sono fissati al momento della compilazione.
Mentre un *binding dinamico* (detto anche *late binding*) vuol dire che i tipi di tutte le variabili (o la classe di appartenenza di tutti gli oggetti) ed espressioni non sono noti fino al momento dell'esecuzione del programma.

1.7 Polimorfismo

Il *polimorfismo* nell'OOP ha molte definizioni, in genere con questo termine si intende la caratteristica che possiedono gli *oggetti* di mostrare un **comportamento diverso** nonostante ricevano *messaggi* identici (cioè esistono oggetti i cui metodi hanno nomi uguali). L'uso di metodi che presentano lo stesso nome, pur non essendo un vincolo, ha una grande importanza concettuale: assegnare il medesimo nome a metodi diversi in classi diverse significa aver riconosciuto un comportamento simile ed ha quindi senso usare lo stesso nome.

Una forma di polimorfismo riguarda le funzioni (o i metodi). Nei linguaggi di programmazione tradizionali due subroutine o funzioni non possono avere lo stesso nome. Accade spesso, però che delle procedure o funzioni debbano essere *polimorfiche* cioè trattare argomenti di tipo diverso. La soluzione convenzionale è quella di fornire un argomento supplementare di controllo per gestire questa eventualità. Nei linguaggi orientati agli oggetti è possibile avere funzioni con lo stesso nome ma che differiscono dal **tipo** e dal **numero** degli argomenti. Questo tipo di polimorfismo è noto anche come **overloading**.

L'altro tipo di polimorfismo (detto anche *polimorfismo puro*) è quello che si ottiene in presenza dell'ereditarietà e del binding dinamico.
Si consideri la seguente gerarchia di classi:



Le classi Cerchio e Rettangolo sono sottoclassi di Figura2D; esiste un metodo Area comune a tutti che calcola e stampa la superfice della figura corrispondente (il metodo Area nella classe Figura2D è solo dichiarato ma non implemetato). Si consideri, inoltre, le seguenti righe di pseudo codice:

```
…
Cerchio cerchio        // definizione oggetto cerchio
Rettangolo rettangolo // definizione oggetto rettangolo
…
```

```
if (z==1) A:=cerchio // z è definito altrove nel programma
if (z==0) A:=rettangolo
A<-Area              // invio messagio Area all'oggetto A
…
```

Questo ci dice che al momento della compilazione non è noto il tipo di oggetto a cui punterà la variabile A (cerchio o rettangolo), ciò viene determinato al run-time e di conseguenza è in quel momento che viene fatta l'associazione (A è del tipo cerchio o rettangolo) e di conseguenza selezionato il metodo corrispondente.

Capitolo 2

2 Il C++

Lo Smalltalk al quale abbiamo più volte fatto riferimento è quello che viene chiamato un linguaggio orientato agli oggetti puro. Volendo significare che tutte le entità del linguaggio sono oggetti che appartengono a qualche classe.
Il C++ da questo punto di vista è un **ibrido** poichè esistono oggetti come ad esempio: i numeri interi, i caratteri e i numeri reali (**int**, **char**, **float**) che non sono *instance* di nessuna classe.
Questo capitolo offre una carrelata molto veloce del linguaggio su quegli aspetti non prettamente "*object oriented*" mentre, i successivi, si propongono di descrivere i modi nei quali sono stati implementati nel C++ i concetti della programmazione orientata agli oggetti. Naturalmente questa descrizione non ha la pretesa di essere esaustiva e dettagliata, è solo un tentativo di fornire una visione generale della struttura del linguaggio.

2.1 Nota storica

Il C++ è stato inventato da Bjarne Stroustrup verso la fine degli anni 80 presso i Bells Laboratories della AT&T partendo dal linguaggio C sviluppato anni prima presso la stessa organizzazione. Infatti, il C può essere visto come un sottoinsieme del C++. Gli altri linguaggi che hanno influenzato la realizzazione dello C++ sono Simula67, per quanta riguarda il concetto di classe, e Algol68 per l'overloading. La standardizzazione del linguaggio è iniziata nel 1990 con un sforzo congiunto tra l'ANSI e l'ISO. Una prima bozza dello standard è stato pubblicato nell'aprile del 1995 e si è in attesa per il 1998 della definitiva approvazione da parte dell'ISO[1].

2.2 Struttura del linguaggio

Un programma C++ consiste in una serie di dichiarazioni di tipi di dato, funzioni, variabili e costanti. Almeno una delle funzioni definite deve avere il nome **main() {…}** da questa, infatti, inizia l'esecuzione del programma. Il linguaggio usa un certo numero di direttive che iniziano con il carattere "#", tale direttive vengono eseguite prima della reale compilazione da un modulo chiamato pre-compilatore o *pre-processor*.

```
#include "header.h"
...
int f1(void)
{
int z;
...
return (z);
```

[1] International Standard Organization

```
}
void f2(float x,int y)
{
/* fai
    qualcosa */
}
//
// funzione principale
//
main() {
      int a;
      float c;
      a=f1();
      f2(c,a);
}
```

2.2.1 Identificatori

Un identificatore (nomi di funzioni, variabili e costanti) è costituito da uno o più caratteri. Il primo deve essere una lettera o un underscore. Le successive possono essere lettere, numeri o underscore. In teoria un identificatore non ha limitazioni di lunghezza ma di solito è il sistema operativo nel contesto nel quale il compilatore agisce a porre dei limiti.

2.2.2 Commenti

Spesso è necessario inserire delle spiegazioni, significative solo per il lettore, all'interno dei programmi. In C++ questo si può fare in due modi:
I caratteri **/*** contrassegnano l'inizio di un commento che si conclude con i caratteri ***/**. Questo metodo è molto utile se i commenti occupano più righe.
I caratteri **//** iniziano i commenti che terminano con la fine della riga. Questo sistema è utile per commenti brevi.

2.2.3 Header file

La direttiva **#include "header.h"** impartisce al pre-compilatore del linguaggio C++ di inserire in quel punto il file **"header.h"**. Quest'ultimo è quello che si chiama *header file*, il suo scopo è quello di garantire, per quanto possibile, un certo livello di coerenza con il corpo del programma specie se quest'ultimo è composto da diversi file. Esso contiene in genere le dichiarazioni delle classi, dei prototipi delle funzioni e delle costanti.
Lo scopo di questo tipo di file è anche quello di fornire un sistema per separare l'interfaccia del programma dalla sua implementazione.

2.2.4 Funzioni

Una funzione è un insieme di istruzioni avente lo scopo di risolvere un problema specifico. La scambio di informazioni tra la funzione ed il mondo esterno avviene tramite i suoi argomenti. La funzione può ritornare un un valore appartenente ad un certo tipo di dato. Tipicamente una funzione ha la seguente struttura:

tipo_di_ritorno **Nome_Funzione(*arg1,arg2,...argN*)**

{*corpo della funzione*}

Dove con ***tipo_di_ritorno*** si intende il tipo di dato che viene restituito alla fine dell'esecuzione della funzione, con ***arg1,arg2, … argN*** le dichiarazioni del tipo di

argomenti che si passano alla funzione e con ***corpo della funzione*** tutte le dichiarazioni e le istruzioni che formano la funzione. Nel caso venga ritornato un valore la funzione termina con l'istruzione **return (espressione)** Il tipo di dato di **espressione** deve coincidere con quello dichiarato. In C++ è obbligatorio dichiarare il prototipo di una funzione che si vuole definire. Il prototipo di una funzione è una dichiarazione preliminare (fatta in un punto diverso e prima della sua effettiva implementazione) che informa il compilatore del numero e del tipo di argomenti della funzione. Esempi:

```
void fun_Prova(double z);           // prototipo
float funProvaFun(void);                  // prototipo
//
int Media(int x, int y);            // prototipo
//
int Media(int x, int y) {           // definizione
                  return ((x+y)/2);
                  }
```

Il primo esempio dichiara una funzione che non ritorna nessun dato e che ha per argomento un numero o variabile doppia precisione. Il secondo dichiara una funzione che ritorna un valore in reale in semplice precisione, non è previsto nessun argomento. Il terzo dichiara che la funzione Media ritorna un intero e che ha due argomenti anche essi interi. Segue le definizione vera e propria della funzione.

2.3 Keywords

Con il termine *keywords* si designano degli identificatori che sono riservati al linguaggio C++ e che quindi non possono essere ridefiniti o usati in contesti diversi. Nella seguente tabella sono elencati quelle comuni a quasi tutte le versioni del linguaggio.

Keywords			
asm	**auto**	**bool**	**break**
case	**catch**	**else**	**enum**
private	**explicit**	**extern**	**false**
true	**float**	**operator**	**protected**
public	**register**	**throw**	**typedef**
double	**friend**	**class**	**delete**
virtual	**this**	**inline**	**template**
new	**const**	**sizeof**	**for**
while	**switch**	**do**	**void**
return	**static**	**union**	**signed**
unsigned	**char**	**struct**	**short**

2.4 Tipi di dato

I tipi fondamentali di dati predefiniti nel C++ sono **bool** (booleano), **char** (carattere), **short** (intero corto), **int** (intero), **long** (intero lungo), **float** (reale, precisione semplice) e **double** (reale, doppia precisione).

2.5 Pointers e array

Un **array** ad una (vettore) o più dimensioni (matrice) è un tipo di dato che viene usato per rappresentare una sequenza di valori omogenei. Agli elementi di una **array** si fa riferimento per mezzo di indici. Il primo elemento di un array è quello con indice 0. Tipici esempi di dichiarazioni di vettori o matrici sono:

```
int a[5];     //dichiarazione di un vettore di 5 interi
int a[3]={4 ,5,6};        //dichiarazione con inizializzazione a[0]=4,a[1]=5,a[2]=6
char list[]={'a','b','c'}; è equivalente a
char list[2]= {'a','b','c'};
float h[3][3];      //dichiarazione di una matrice bidimensionale 3x3
```

Un **pointer** è una variabile che può contenere come valore un indirizzo il quale viene usato per accedere ai dati di un certo tipo. La dichiarazione del tipo pointer ha la forma seguente:

```
int *p;  // p è un pointer ad un intero, cioè p contiene un indirizzo che punta ad un intero
```

Gli array e pointer sono strettamente legati, infatti il nome di un array non è altro che un pointer il cui valore (un indirizzo) è costante e non può essere cambiato. Quindi se abbiamo

```
int a[2], *p;
```

allora

```
p=&a[0] è equivalente a p=a
p= a+1 è equivalente a p=&a[1]
```

L'operatore "**&**" ritorna l'indirizzo della variabile alla sua destra.

2.6 Blocchi

Un blocco è una lista di istruzioni racchiuse tra parentesi graffe le quali vengono trattate come una istruzione unica. Esempio di blocco:

```
{a=b; vvv(b);
      ...
     i++;}
```

2.7 Variabili

Le variabili in C++ possono essere dichiarati in qualsiasi punto del programma. Una variabile inizia ad esistere al momento della dichiarazione, la sua esistenza cessa quando viene raggiunto la fine del blocco corrente. Esempio:

```
{
      int i;
      ...
      int k;
      ...
      int z = Media(k,i);
```

```
      ...
}
```

Tutte e tre le variabile iniziano ad esistere nel momento della dichiarazione e vengono distrutte alla chiusura del blocco.
Il C++ permette inoltre di dichiarare degli *alias*, cioè un altro nome, per una certa variabile:

```
int i = 2; //dichiarazione di i
int &ref_i =i; // dichiarazione di ref_i, un altro nome per i
```

2.8 Costanti

Il C++ dispone di una notazione per i valori dei tipi di dato predefiniti: costanti carattere, costanti intere e floating point. Le stringhe di caratteri sono vettori di tipo **char**. Nella tabella seguente sono riportati alcuni esempi di costanti.

Esempi di costanti	Descrizione
234 0377 0x1aa	interi: in base 10 8 e 16
‘A’ ‘a’ ‘3’	character: A, b e 3
“stringa”	una stringa di caratteri
3.1456f	costante di tipo float
3e-1	costante di tipo double

2.8.1 const

Un modo più sicuro per definire una costante è quella fatta, ad esempio, con la dichiarazione

```
const int Massimo=100;
```

la quale definisce *Massimo* come una costante di tipo intero e quindi il suo valore non può essere cambiato. Il modificatore **const** può essere usato in vari contesti per rafforzare il concetto di non modificabilità di certi valori. Per esempio nella dichiarazione degli argomenti della funzione:

```
f1(const int j) {
… }
```

vuol dire che l’argomento della funzione, la variabile **j,** non può essere alterata dalla funzione **f1**.

2.8.2 Enum

Un altro modo per definire delle costanti intere è quello di utilizzare l’istruzione di *enumeration* **enum**. Esempio:

```
enum colori {rosso, blu, verde};
```

definisce 3 costanti intere, più precisamente definisce un nuovo tipo di dato “colori” il quale può avere uno dei seguenti valori:

```
rosso=0
blu=1
verde=2
```

cioè agli identificatori delle costanti della *enumeration* "colori" vengono assegnati automaticamente valori interi a partire da zero. E' anche possibile assegnare dei valori in modo esplicito:

```
enum direction {nord=0, est=90, sud=180, ovest=270);
```

2.9 Istruzioni

Il C++ come tutti gli altri linguaggi di programmazione possiede tutte le strutture fondamentali di iterazione e di controllo di flusso.

Istruzione	Esempio	Nota
vuota	;	Ogni istruzione deve terminare con un ";"
espressione	i = j +k;	Puó comportare anche delle conversioni
if	if (x == 1) cout << "x is 1";	cout è l'operatore di output

if-else	if (x==y)
	cout << "equal";
	else
	cout << "not equal";	

for	for (i=0; i<n;i++) a[i]=b[i]+c[i];	
while	while (x !=0) { … cout<<"loop"; }	Nessuna o piú iterazioni
do-while	do { x=y+1; … } while (y>7);	Una o piú iterazioni

switch	switch (s) {
	case 1: cout<< "one"; break;
	case 2: cout<<"two"; break;
	…
	default: cout<<"default";
	};
		Usare sempre break per evitare il test sui rimanenti "case" ed evitare il default

break	break;	Viene usato per uscire dal loop più interno delle istruzioni di iterazione. Termina l'istruzione di switch.
return	return (x*x);	Ritorna un valore al chiamante

2.10 Espressioni ed Operatori

Il C++ é un linguaggio molto ricco di operatori e di espressioni di vario tipo; per una trattazione completa si consulti uno dei libri nella bibliografia.

Nella seguente tabella vengono riassunti gli operatori più comuni.

Operatore	Descrizione
[]	Operatore accesso elemento di un array

->	Accesso elemento di una classe o struttura
!	NOT logico
++	Post/pre incremento
--	Post/predecremento
-	Meno unario
*	Operatore indirezione
&	Operatore indirizzo
*	Operatore di moltiplicazione
/	Operatore di divisione
%	Operatore modulo
+	Operatore di addizione
-	Operatore di sottrazione
<<	Operatore di bit shift a sinistra
>>	Operatore di bit shift a destra
<	Operatore minore-di
<=	Operatore minore o uguale
>	Operatore maggiore
=>	Operatore maggiore uguale
==	Operatore uguale
!=	Operatore non uguale
~	NOT binario
&	AND binario
^	XOR binario
|	OR binario
&&	AND logico
||	OR logico
=	Operatore assegnamento

Nella tabella sottoriportata si elencano le espressioni di uso più comune.

Espressioni	**Descrizione**
j=++i	Assegnazione con pre-incremento equivalente a i=i+1; j=i;
a=b=c=10	Assegnamento multiplo
j=i++	Assegnazione con post-incremento equivalente a j=i;i=i+1;
a*=b	Assegnazione breve equivalente a *a=a*b*
a += b	Assegnazione breve equivalente a *a=a+b*
int a=5	Dichiarazione con inizializzazione
int *p =&a	il puntatore p contiene l'inidirizzo di a; questa scrittura è equivalente alle due istruzioni: *int *p; p=&a;*
*p=7	Ad a viene assegnato 7 (equivalente a *a=7*)
int &x=a	Un'altro nome per la variabile a

2.11 Funzioni

La seguente tabella mostra un riassunto dei possibili tipi di funzioni esistenti nel C++.

Tipo di funzione	Esempio	Nota

Funzione	double power2(double x)
	{
	return x*x;
	}
		Il passaggio dei parametri è del tipo call-by-value, il valore di x rimane invariato per il chiamante;
		l'espressione di ritorno deve essere compatibile col tipo dichiarato.

Procedura	void display(int i)
	{
	cout<<i<<endl;
	}
		void denota una procedura; non viene ritornato nessun valore al chiamante

Nessun argomento	void display()
	{
	cout <<"hello world"<<endl;
	}
		Chiamata ad una procedura senza nessun argomento; si suggerisce di usare la forma display(void).

Call by reference	void change (int& i)
	{
	i=10;
	}
	
		Chiamata con passaggio dei parametri del tipo call-by-reference il valore di i viene cambiato dalla funzione

Numero di argomenti variabile	int display(char *,…)	Questa funzione può essere chiamata con un qualsiasi numero di argomenti; naturalemente il primo deve essere di tipo char
inline	inline cube(int x);	Una funzione dichiarata inline vuol dire che non viene generato nessuna chiamata a detta funzione, tutto il codice della funzione viene inserito nel punto di chiamata
Argomenti di default	int power(double x, int n=2)	Questo tipo di dichiarazione di funzione significa che se nella chiamata viene omesso il secondo argomento viene utilizzato quello presente nella dichiarazione; power(3) sarà uguale a 9; power(3,3) sarà uguale a 27;

2.12 Overloading

L'oveloading dà la possibilità al programmatore di usare lo stesso nome per una funzione od uno operatore ed assegnare ad essi significati differenti. Il significato selezionato dipenderà dalla "*signature*" della funzione. Per *signature* di una funzione si intende la lista che comprende il tipo ed il numero degli argomenti della funzione.

Si considerino le seguenti dichiarazioni della funzione **add**:

```
add(int x, int y)
add(double x, double y)
```

La scelta della funzione da eseguire verrà fatta in base agli argomenti presenti nella chiamata; ad esempio con **add(4,5)** viene eseguita la prima, mentre con la chiamata **add(4.0,5.0)** la seconda.
Il C++ permette inoltre con l'istruzione **operator** di usare l'overloading su quasi tutti gli operatori del linguaggio (+,*,=,ecc.). Questo significa che, a questi ultimi, si possono assegnare significati diversi da quello di origine. Per esempio nell'ambito di una classe (per esempio quella delle matrici: matrix) si può definire la moltiplicazione tra matrici facendo l'overloading dell'operatore "*":

```
matrix operator*( matrix a, matrix b) {…}
```

Questa scrittura definisce una funzione la quale verrà eseguita se gli argomenti dell'operatore "*" sono del tipo "matrix". In pratica potremo scrivere il seguente codice senza ambiguità:

```
…
matrix c,a,b;   // dichiarazione degli oggetti (variabili) a,b,c
…
c=a*b; // c conterrà il prodotto matriciale di a e b
…
```

Avere una notazione unica per operazioni simili è molto importante e conveniente. Infatti in molti casi sarà possibile scrivere programmi più brevi e leggibili.

2.13 Input/output

Nello C++ l'input ed output risulta molto facilitato con l'utilizzo di librerie standard già incluse nel linguaggio. Esiste una gerarchia di classi di tipo *stream* che gestiscono tutti i problemi riguardanti l'interazione con l'esterno. L'oggetto che si occupa dall'ouput verso il terminale (video) è **cout** quello di input dalla tastiera è **cin**, entrambi sono *instance* della classe *iostream*. Questi oggetti (creati dal compilatore per default) usano gli operatori inserimento "<<" e di estrazione ">>" che sono stati "*overloaded*" in modo da ottenere un comportamento polimorfico e fare input ed output di qualsiasi tipo di dato pre-definito dal linguaggio. Per usare questi oggetti è sufficiente che vengano inclusi i file "**iostream.h**" e nel caso si vogliano utilizzare i manipolatori di input/output anche il file "**iomanip.h**". Esempio:

```
#include <iostream.h> //direttiva necessaria per attivare la classe
                      //iostream
void main()
{
int i;
char buffer(80];
cout << "Inserire un valore intero:"
cin >> i;
cout << "Scrivi una riga:";
cin.getline(buffer,80); //leggi una riga dalla tastiera
cout <<buffer;
}
```

In alternativa agli operatori "<<" e ">>" possono essere usati alcuni metodi della classe *ostream* e *istream*, i quali permettono un miglior controllo dell'attività di input ed output. I più usati sono **get()**, **getline()**, **put()**.
L'output generato dall'esempio precedente non è formattato, cioè non vi sono spazi

tra i dati stampati. Per migliorare la forma dell'output ci sono due possibili strategie complementari: usare dei codici speciali (*escape codes*) e stringhe di caratteri poste tra doppi apici (Tabella 2-1), e l'uso di particolari "*stream manipulators*" o delle funzioni corrispondenti. (Tabella 2-2).

Tabella 2-1: Codici di escape

Sequenze di escape	Significato
\n	Newline – vai alla riga sequente
\t	Tab orizzontale
\v	Tab verticale
\r	Return
\f	Form-feed cambio pagina
\\	Stampa il carattere "\"
\?	Stampa "?"
\'	Stampa apice
\"	Stampa doppio apice

Tabella 2-2: Formattazione dati

Manipolatore	Funzione	Descrizione	File da includere
endl		Outputs a LF	iostream.h
dec		Usa numeri decimali in I/O	iostream.h
hex		Usa numeri esadecimali in I/O	iostream.h
oct		Usa numeri ottali in I/O	iostream.h
ws		Ignora gli spazi in input	iostream.h
flush	flush()	Flush output	iostream.h
ends		Output null in una stringa	iostream.h
setw(int)	widh(int)	Fissa la larghezza del campo di I/O	iomanip.h
setfill(int)	fill(int)	Fissa il carattere di riempimento	iomanip.h
setbase(int)		Fissa la base per i numeri	iomanip.h
setprecision(int)	precision(int)	Fissa la precisione dei numeri in floating point	iomanip.h
setiosflags(long)	setf(long)	Fissa altri parametri di formattazione	iomanip.h
resetiosflags(long)	setf(long)	Cancella altri parametri di formattazione	iomanip.h

Di seguito si propone un esempio di programma dove vengono usati alcuni istruzioni di formattazione.

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

void main() {
     int a=10;
     double b=322.123456;
     char *s="Test i/o:";
```

```
      cout <<s<< a << endl;
      cout <<s<<"\t"<< a << endl;
      cout << s << " "<<a<<endl;
      cout <<s <<hex<<a<< endl;
      cout <<s <<oct<<a<< endl;
      cout <<s<<b<<endl;
      cout <<s<<setw(15)<<b<<endl;
      cout <<s<<setw(20)<<setfill('*')<<b<<endl;
      cout <<s<<setprecision(3)<<b<<endl;
      cout <<s<<setprecision(5)<<b<<endl;
      cout<<setiosflags(ios::uppercase);
      cout<<s<<hex<<a<<endl;
      cout<<resetiosflags(ios::uppercase);
      cout <<s<<hex<<a<<endl;

}
```

L'output generato è il seguente:

```
Test i/o:10
Test i/o:       10
Test i/o: 10
Test i/o:a
Test i/o:12
Test i/o:322.123
Test i/o:        322.123
Test i/o:*************322.123
Test i/o:322
Test i/o:322.12
Test i/o:A
Test i/o:a
```

In ogni momento può essere verificato lo stato di un oggetto di tipo stream con i metodi **good()**, **eof()**, **fail()** e **bad()**. Questi ritornano informazioni sulla correttezza della operazioni di input/output.

2.14 File Input/Output

Per la lettura e scrittura su file su disco vengono usati oggetti appartenenti alle classi *ofstream* per l'output e *ifstream* per l'input. L'accesso a queste classi è reso disponibile dall'inclusione del file **fstream.h**. L'esempio seguente mostra l'utilizzo dell'input/output su file:

```
#include <fstream.h>
void main() {
      char c;
      ifstream fin("ifile.txt"); //apre il file "ifile.txt"
      ofstream fout("ofile.txt"); //apre il file "ofile.txt"
      fout<<"Inizio file di output…\n"; //scrive riga su ofile
 while (fin.eof()) { // loop di copia di ifile su ofile
      c=fin.get();
      fout<<c;
                  }
      fout.close(); // chiusura file
      fin.close();
}
```

Capitolo 3

3 Oggetti e Classi nel C++

Le classi in C++ sono un modo alternativo e molto potente per definire nuovi tipi di dati da parte dell'utente. La dichiarazione di una classe ha la seguente sintassi nella sua forma più generale:

```
class nomeclasse {
      private:
            // private data class members
            // private function members
      protected:
            // protected data class members
            // protected function members
      public:
            // public data class members
            // public function members
                        };
```

Chi conosce lo C avrà subito notato che definizione di classe è simile a quella di una **struct** alla quale sono state aggiunte nuove direttive. Queste direttive (private, protected e public) sono opzionali; per default tutti i membri sono privati se non altrimenti specificato.
La direttiva **private** vuol dire che i suoi membri sono accessibili solo dalla classe nella quale sono definiti. Con **public** si indicano tutte le funzioni e tutti i dati disponibili all'esterno della classe. La parola chiave **protected** è una via di mezzo tra le precedenti due definizioni infatti, i suoi membri sono accessibili non solo dalla classe ove sono definiti ma anche dalle sue sottoclassi (o classi derivate) per il resto del mondo esse sono private. Nell'esempio seguente

```
class Point {
            int Coordx;
            int Coordy;
      public:
            void Getx(int x);
            void Gety(int y);
            void Setx(int x);
            void Sety(int y);
      };
```

viene definita la classe Point in modo analogo a quello che abbiamo fatto introducendo i concetti dell'OOP. Come si può notate gli attributi (CoordX, CoordY) sono diventati i dati (variabili) privati (non è stato indicato nessuna direttiva quindi vale la regola di default) mentre i metodi corrispondono alle funzioni public. Si noti che queste funzioni sono state solo dichiarate la loro effettiva definizione viene, di solito, fatta fuori dalla classe. In C++ le funzioni che appartengono ad una data classe vengono chiamati *member functions*. Ecco un esempio di definizione di una funzione public:

```
void Point::Getx(int x)
```

```
{
        x = Coordx;
}
```

Come si vede il nome della funzione da definire è preceduta da "`::`" (*scoping operator*) il quale a sua volta è preceduto dal nome della classe alla quale appartiene.

3.1 Creazione degli oggetti

Una volta creata una classe essa si può usare come ogni altro tipo di dato già predefinito nel C++ in modo equivalente a quello che si fa con le direttive **int** o **char**. Per Esempio possiamo dichiarare una variabile di tipo point (in termini di programmazione orientata agli oggetti creare una *instance*) come segue:

```
point screen;
```

Sebbene *screen* sia una variabile, in termini di OOP è da considerarsi un *oggetto* poiché ne possiede tutte le caratteristiche, infatti, contiene sia dati che funzioni (attributi e metodi). Quindi in C++ il termine variabile è sinonimo di oggetto.

Siamo adesso in grado di scrivere il nostro primo programma in C++ che calcola l'area di un cerchio:

```
#include <stdio.h>
      class circle {
                        int radius;
                  public:
                        void set_radius(float x);
                        float area(void);
                  };
//
void circle::set_radius(float x)
            {
                  radius = x;
            }
//
float circle::area(void)
            {
                  float pi=4.0*atan(1);
                  return pi*radius*radius;
            }
//
void main() {
      circle c1, c2;
      c1.set_radius(3.0);
      c2.set_radius(4.5);
      printf("Area di c1: %d\n",c1.area());
      printf("Area di c2: %d\n",c2.area());
            }
```

Questo piccolo programma ci permette di introdurre la notazione con la quale vengono chiamate le funzioni public o come si accede ai dati public (nel nostro caso non esistono) della classe a cui appartiene; la notazione usata è

```
<oggetto>.<funzione>(<argomenti>)
<oggetto>.<variabile>
```

come si può verificare dalle parti evidenziate del listato. Quindi le scritture

```
c1.area() e c2.set_radius(4.5)
```

significano rispettivamente l'invio del messaggio *area* all'oggetto *c1* (ovvero chiamata del metodo area dell'oggetto c1) e l'invio del messaggio *set_radius* con argomento *4.5* all'oggetto *c2*.

3.2 Oggetti Dinamici

Talvolta accade che per programmi complessi è necessario creare un grande numero di oggetti, in questo caso risulterebbe conveniente "allocare" questi oggetti in modo dinamico con la possibilità di riservare spazio di memoria quando serve e di liberarla quando non é più utile. Il C++ fornisce due operatori **new** e **delete** per la gestione dinamica della memoria (allocazione e liberazione) con l'ausilio dei puntatori.

```
int *p;
p = new int[10];
...
p[5]=25;
...
delete[] p;
```

```
int *p = new int;
...
*p=40;
...
delete p;
```

```
Point *apoint;
apoint = new Point[1024];
...
apoint[0].SetX(10);
apoint[0].SetY(10);
...
delete[] apoint;
```

Negli esempi sopra riportati si vede come sono usati questi operatori; nell'esempio di sinistra viene riservato, usato ed infine distrutto un vettore di 10 numeri interi, in quello di centro una variabile intera ed a destra un vettore (apoint) di oggetti Point. Si noti la differente sintassi dell'operatore delete nel caso si debba de-allocare un singolo oggetto o un vettore.

3.3 Constructors

Con il termine constructor si intende una o più funzioni di tipo speciale che vengono usati per inizializzare un oggetto al momento della sua definizione. In C++ la loro esecuzione è automatica appena l'oggetto è dichiarato. Come esempio possiamo modificare la classe point che abbiamo già usato nel seguente modo:

```
class Point {
                int Coordx;
                int Coordy;
        public:
                Point() { CoordX=CoordY=0;}
                void Getx(int x);
                void Gety(int y);
                void Setx(int x);
                void Sety(int y);
        };
```

L'effetto dell'aggiunta del costructor ci assicura che nell'istante di creazione di un oggetto di tipo Point le variabili Coordx e CoordY verranno inizializzate al valore (0,0). Le funzioni Constructors hanno lo stesso nome della classe e possono avere degli argomenti. In quest'ultimo caso sfruttando la tecnica dell'overloading possiamo definire un secondo constructor per la classe Point che inizializzi le variabili CoordX e CoordY ad un valore diverso di (0,0).

```
class Point {
        int Coordx;
        int Coordy;
    public:
        Point() { CoordX=CoordY=0;}
```

```
Point(int x, int y) {CoordX=x; CoordY=y;}
      void Getx(int x);
      void Gety(int y);
      void Setx(int x);
      void Sety(int y);
};
```

La parte evidenziata mostra il nuovo constructor della classe Point; quindi quando si dichiarano gli oggetti:

```
Point punto1;
Point punto2(10,4);
```

Nel primo caso viene chiamata implicitamente la funzione Point(); nel secondo caso è la funzione Point(int x, int y) ad essere eseguita.

3.4 Destructors

Le funzioni destructors hanno il compito opposto di quelle constructors anch'esse sono chiamate in modo automatico ma al termine della routine che ha dichiarato l'oggetto che contiene il destructors. Esse in genere sono utilizzate quando una funzione constructor ha riservato dello spazio in memoria che deve essere liberato.

I destructors hanno lo stesso nome della classe, ma sono precedute dal carattere "~".

3.5 Funzioni e classi "friend"

Una funzione definita come *friend* (amica) di una certa classe ha la possibilità di accedere alle variabili e alle funzioni non pubbliche (private e protected) di quella classe. In pratica una tale funzione possiede il privilegio di violare il principio di *data encapsulation*. Questo è considerato in alcuni casi un male necessario ed inevitabile. Esempio:

```
class blabla {
      …
      int jfk();  //funzione membro della classe blabla
      friend int alice(); //funzione friend della classe blabla
      …
};
```

Una funzione *friend* deve essere dichiarata all'interno della classe della quale è friend.

Una classe friend è una classe i cui membri (funzioni) sono tutti friend di un'altra classe. In questo caso la dichirazione avviene come nel seguente esempio:

```
class ciccio {
      …
      friend class blabla;
      …
};
```

Qusto significa che tutti i membri della classe **blabla** hanno accesso a tutti quelli della classe **ciccio**.

3.6 Static members: variabili e funzioni

Quando una variabile, membro di una classe, è dichiarata *static* vuol dire che tutti gli oggetti che appartengono a quella classe condividono la stessa copia della variabile.

Questo è utile quando oggetti di una medesima classe hanno necessità di accedere gli stessi dati.
Una variabile di una classe viene definita static facendo precedere alla dichiarazione della variabile la parola *static*. Questo, in pratica, significa che esiste solo una copia di tale variabile a livello globale.

Nel caso di una funzione l'attributo *static* ha lo scopo di far eseguire (si potrebbe dire: alla classe, a prescindere dall'esistenza o meno di oggetti) qualche operazione generale che riguarda tutti gli oggetti di quella classe. In particolare uno degli usi è quello di accedere alle variabili dichiarate static.

Capitolo 4

4 Ereditarietà

L'ereditarietà è una importante caratteristica del C++ perché ci permette di fare, in modo semplice naturale, quello che con i linguaggi convenzionali è quasi impossibile: **l'estensione ed il riutilizzo del codice** esistente senza dover riscrivere programmi.

4.1 Sottoclassi o classi derivate

Data una classe (detta classe base o superclasse) se ne può derivare un'altra (sottoclasse o classe derivata). La classe derivata **eredita** i dati (data members) e i metodi (function members), questa classe può modificare sia i dati che i metodi ereditati dalla classe base oppure aggiungerne di nuovi. Questo processo può continuare cosicché una classe derivata diventa a sua volta una classe base e così via. La dichiarazione di una classe derivata si comprenderà meglio con l'esempio seguente:

```
class Circle : public Point {
                        int radius;
                public:
                        void SetRadius(int x)
                                { radius=x; }
                };
```

La sopraindicata scrittura mostra la dichiarazione della classe Circle derivata dalla classe Point; l'operatore ":" costruisce le classi derivate. Ai dati della classe Point viene aggiunto il dato radius ed alle function members la funzione SetRadius.

Quindi la dichiarazione di una sottoclasse è del tipo

```
Class sottoclasse : public (o private) classe_base {
            private:
                  // private data class members
                  // private function members
            protected:
                  // protected data class members
                  // protected function members
            public:
                  // public data class members
                  // public function members
                     };
```

Una classe B che eredita da un altra classe base A in modo private significa che si vogliono nascondere le funzionalità della classe di base. In altre parole tutti i componenti della classe di base vengono trattati come privati anche se sono (tutti o in parte public). Dall'altra parte la derivazione di tipo public vuol dire che i componenti pubblici della classe di base sono da considerare pubblici anche nella classe derivata.

4.2 Constructors e Destructors nelle classi derivate

Molta attenzione deve essere posta, in una gerarchia di classi, alle funzioni *Constructors* e *Destructors*. Il loro comportamento è, infatti, differente da ogni altra *member function*. Come illustra l'esempio seguente, prima di essere eseguito, ogni constructors e ogni destructors chiama il corrispondente constructor e destructor della classe base:

```
class X {
    public:
        X()    { cout << "Costruttore di X\n"; }
        ~X()  { cout <<"Distruttore di X\n"; }
};


class Y : public X {
    public:
        Y()    { cout << "Costruttore di Y\n"; }
        ~Y()  { cout <<"Distruttore di Y\n"; }
};

class Z : public Y {
    public:
        Z(int n)    { cout << "Costruttore di Z\n"; }
        ~Z()  { cout <<"Distruttore di Z\n"; }
};
int main()
{
  Z a(2);
}
```

L'output di questo esempio è il seguente:

```
Costruttore di X
Costruttore di Y
Costruttore di Z
Distruttore di Z
Distruttore di Y
Distruttore di X
```

Quando viene viene dichiarato l'oggetto **a(2)** appartenente alla classe Z viene chiamato il costruttore di di Z::Z() questo, prima della sua esecuzione chiama il costruttore della sua classe base Y cioè Y::Y() che a sua volta chiama il costruttore di X::X(). Quando il costruttore X::X() termina la sua esecuzione, il controllo ritorna al costruttore Y::Y(); al termine dell'esecuzione di quest'ultimo, il controllo ritorna al costruttore Z::Z(int n), il quale inizia la propria esecuzione. Il risultato è che tutti i costruttori in una gerarchia di classi vengono eseguiti dal'alto verso il basso.

I Distruttori hanno un comportamento simile, con la differenza che essi eseguono il proprio codice **prima** di chiamare il distruttore della classe base, quindi tutti i distruttori in una gerarchia di classi vengono eseguiti dal basso verso l'alto.

4.3 Ereditarietà multipla

Una recente ed importante aggiunta al C++ è quella di poter creare sottoclassi a partire da più classi contemporaneamente. In pratica ereditare metodi e variabili da più di una classe. Questa nuova caratteristica è detta *Ereditarietà multipla*.
Il giudizio, da parte degli addetti ai lavori, sulla sua effettiva utilità è però discorde

poichè possono sorgere gravi problemi quando sono presenti nomi di funzioni o variabili che sono uguali nelle classi base dal quale si eredita.

4.4 Polimorfismo e Funzioni Virtual

Il polimorfismo è implementato nello C++ con le funzioni *virtual*. Questo è reso possibile dal fatto che un puntatore ad una *instance* di una classe A (un oggetto) può puntare ad una qualsiasi *instance* di una classe B *sottoclasse* di A:

```
class A {
// …
};
class B : public A {
// …
};
main () {
A* p ; // p è puntatore  ad oggetti della classe A
B b ;
p = &b; // p può puntare ad oggetti della sottoclasse B di  A
}
```

Supponiamo che sia nella classe base A sia nella sua sottoclasse B esista una medesima funzione **f()**, quale delle due verrà eseguita con la chiamata (***p).f()** (questa scrittura per motivi pratici si può anche scrivere **p->f()**) ? Anche se **p** punta alla sottoclasse **B** rimane un puntatore ad uno oggetto della classe **A** quindi verrà eseguita sempre la funzione **A::f().** Si veda l'esempio seguente:

```
class A {
  public:
     void f() { cout << "Viene eseguita A::f()\n"; }
};
class B : public A {
  public:
     void f() { cout << "Viene eseguita B::f()\n"; }
};
int main () {
      A a ;
      B b ;
      A* p ; // p è puntatore  ad oggetti della classe A
      p = &x;
      p->f();
      p = &b; // p può puntare ad oggetti della sottoclasse B di  A
      p->f();
}
```

L'output è il seguente:

```
Viene eseguita A::f()
Viene eseguita A::f()
```

Le due chiamate alla funzione eseguono la medesima versione, quella definita nella classe base A. Il fatto che il puntatore punti, in certo istante. ad uno oggetto della classe B è irrilevante.

Adesso modifichiamo il programma in modo da dichiarare la funzione f() della classe A *virtual*:

```
class A {
```

```
  public:
     virtual void f() { cout << "Viene eseguita A::f()\n"; }
};
class B : public A {
  public:
     void f() { cout << "Viene eseguita B::f()\n"; }
};
int main () {
      A a ;
      B b ;
      A* p ; // p è puntatore  ad oggetti della classe A
      p = &x;
      p->f();
      p = &b; // p può puntare ad oggetti della sottoclasse B di  A
      p->f();
}
```

Il risultato è diverso:

```
Viene eseguita A::f()
Viene eseguita B::f()
```

Adesso la seconda chiamata p->f() esegue B::f() invece di A::f(). Questo esempio illustra come funziona il polimorfismo nel C++: La medesima chiamata provoca l'esecuzione di funzioni differenti in accordo con la classe dell'oggetto a cui punta p.

I Distruttori possono (ed in alcuni casi devono) essere dichiarati virtual, mentre non lo possono essere i costruttori.

4.5 Classi astratte

Una funzione si dice virtuale pura (*pure virtual function*), se essa, nella classe è dichiarata uguale a zero. Questo ha il significato che la funzione verrà implementata in ciascuno delle classi derivate. La classe contenente almeno una funzione virtuale pura è chiamata *classe astratta*. Una classe astratta non può avere *instances*, serve solo come intelaiatura dalla quale verranno derivati i dettagli nelle sottoclassi "concrete".

Supponete di possedere 3 videoregistratori (VCR) di marche diverse; naturalmente avranno funzionalità differenti, e telecomandi non compatibili. Poi trovate sul mercato un telecomando molto semplice che può operare con qualsiasi modello di VCR. Per esempio, un singolo tasto "PLAY" comanda, a qualsiasi VCR sia puntato, il playback del nastro inserito.
Questo dispositivo rappresenta l'essenza della programmazione orientata agli oggetti: semplificazione concettuale delle diverse implementazioni per mezzo di una singola interfaccia . Nel nostro esempio, l'interfaccia è il telecomando, e le implementazioni sono le operazioni (invisibili all'utente) all'interno del telecomando.
L'interfaccia potrebbe essere la classe base astratta seguente:

```
class VCR {
     public:
      virtual void on() = 0;
      virtual void off() = 0;
      virtual void record() = 0;
      virtual void stop() = 0;
      virtual void play() = 0;
};
```

Le implementazioni sono le classi concrete derivate:

```
class Panasonic : public VCR {
   public:
      void on();
      void off();
      void record();
      void stop();
      void play();
};
class Hitachi : public VCR {
   public:
      void on();
      void off();
      void record();
      void stop();
      void play();
};
class Philips : public VCR {
   public:
      void on();
      void off();
      void record();
      void stop();
      void play();
};
```

Questo tipo di approccio conferma la proprietà di facile estensibilità del sistema. Il telecomando (visto dall'utente) non cambia aggiungendo la possibilità di gestire altri modelli di VCR (cioè aggiungere altre classi derivate concrete).

Capitolo 5

5 Template

Una recente aggiunta al linguaggio C++ è stata quella di fornire un sistema per creare classi e funzioni generiche parametrizzate, cioè dei modelli sulla base dei quali creare delle famiglie di classi e funzioni simili tra loro. In altre parole mentre l'ereditarietà fornisce la possibilità di riutilizzare il codice oggetto, il template ci permette di riutilizzare il codice sorgente.

5.1 Funzioni template

Una funzione template definisce una famiglia di funzioni. Tutte queste famiglie sono tra loro simili. Ogni funzione della famiglia differisce da un altra per il valore di uno o più parametri. Esempio:

```
template <class T>
void swap( T a, T b) {
      T x;
      z=a;
      a=b;
      b=z;
}
void main() {
      int i=1,j=2;
      double z=2.5,v=5.0;
      swap(i,j);
      swap(z,v);
}
```

Questo piccolo programma dà un'idea di come funzionano i template. La funzione template **swap** è definita a meno del parametro T il quale serve a specificare il tipo di dato o la classe degli argomenti di swap. Il template non genera codice eseguibile finchè non viene fatto riferimento alla funzione swap. Solo allora viene fatta la generazione (*instantiation*) del codice. Nell'esempio verranno generate due copie di detta funzione una per gli argomenti di tipo **int** e l'altra per quelli di tipo **double**.

5.2 Classi template

Una classe template definisce una famiglia di classi. La famiglia viene parametrizzata attraverso una o più classi che vengono elencate nella lista degli argomenti del template: **template<class T, class B, …>**. In tale elenco possono anche comparire argomenti appartenenti ai tipi di dato predefiniti nel linguaggio. L'esempio che segue mostra come specificare un template Vector che può contenere dati di qualsiasi tipo.

```
#include <iostream.h>
#include <stdlib.h>
//
template <class T, int sz>
class Vector
```

```
{
    public:
      Vector(){}
      T &operator()(const int index);

    private:
      T vdata[sz];
};
template <class T, int sz>
T &Vector<T,sz>::operator()(const int index) {
     if (index>=sz) {
        cout << "Error: index out of bound"<<endl;
        exit(1);
        }
     return vdata[index];
}
//
//
void main() {
     Vector<int, 10> vi;
     vi(8)=13;
     cout<<vi(8)<<endl<<vi(7)<<endl;
     Vector<double, 4> vd;
     vd(1)=3.12345;
     cout<<vd(1)<<endl;
}
```

Si noti la particolare sintassi nella definizione della funzione “operator()” se viene fatta al di fuori della classe. Ad una classe template si applicano le stesse regole valide per una qualunque altra classe, per esempio è possibile applicare il principio di ereditarietà e creare una classe template da un altra classe template.

Le classi template sono comunemente associati ad oggetti che contengono altri oggetti (containers). Esempi di classi container sono i vettori, matrici, liste, code, stack, ecc. Esistono delle librerie apposite per la gestione di questo tipo di classi chiamate *Standard Template Library* (STL). Esse forniscono un valido aiuto al programmatore fornendogli una varietà di “pezzi di codice” che possono essere integrati ed usati per generare una applicazione.

Appendice

Bibliografia

Il Linguaggio C++ - Stroustrup, B., Addison-Wesley

Simple C++ - Cogswell, J. M., Waite Group Press, 1994

The C++ Programming Language – Stroustrup B., Addison Wesley,1997, 3rd edition

C++ for dummies – Davis, S. R., IDG Books, 1996

More C++ for dummies - Davis, S. R., IDG Books, 1996

Teach Yourself ANSI C++ in 21 days – Liberty, J., Hord, J. M., Sams, 1996

The complete C++ Primer – Weiskamp, K., Flaming, B., Academic Press, 1992

C++ for Fortran Programmers – Pohl, I., Addison Wesley,1997

An introduction to numerical methods in C++ - Flowers, B. H., Claredon Press, 1995

C++ for Pofessional Programmers with PC and UNIX Applications – Blaha, S., ITC Press, 1995

Thinking in C++ - Eckel, B., Prentice Hall, 1995

Scientific and Engineering C++ - Barton, J. J., Nackman, L. R., Addison Wesley, 1994