Lezione 3

Introduzione alla Programmazione Orientata agli Oggetti

http://www.mi.infn.it/~palombo/didattica/Lab-TNDS/CorsoLab/LezioniFrontali/

Palombo Fernando

Programmazione Orientata agli Oggetti (OOP)

- OOP si è sviluppata a partire dagli anni ‘60 ed è diventata la tecnica preferita a partire dagli anni ’90
- Si è resa indispensabile per poter gestire programmi sempre più complessi.
- Il progetto viene spezzettato in tanti sottogruppi di parti collegate tra di loro, che hanno dati e codice relativi ai sottogruppi.
- Questi sottogruppi sono organizzati in una struttura gerarchica
- Elemento base di questo tipo di programmazione è l’oggetto : questo interagisce con altri oggetti, esplica determinate funzioni, ecc
- In C++ si utilizzano oggetti di nuovo tipo utilizzando la classe. Questa forma in C++ la base della OOP

Classe

A differenza della struttura (in C) la classe in C++ permette di introdurre nuovi tipi di variabili che racchiudono sia dati che funzioni.

Un oggetto di un certo tipo è una istanza di una certa classe. In C++ la classe è alla base della OOP.

Dichiarazione di una classe:

```
class Nome-classe{
        private:
        membri_accesso_privato
        …….
        protected:
        membri_accesso_protetto
         …….
        public:
         membri_ad_accesso_pubblico
};   //notare il ;  alla fine della dichiarazione;  è essenziale!!

//} oggetto1, oggetto2;   oggetto1, ecc   lista OPZIONALE di nomi di oggetti
```

Specificatori di Accesso

Tre specificatori di accesso:

private: membri (dati e funzioni) con questo accesso solo accessibili solo da altri membri della stessa classe (o di classi friend)

È lo specificatore di default. In una classe ogni membro se non viene dato lo specificatore di accesso è considerato private

protected: membri protected sono accessibili sia dai membri della stessa classe sia dai membri di classi friend o di altre classe derivate da questa classe

public: membri public sono accessibili dovunque sono visibili

Lo specificatore protected è rilevante per le classi derivate e le classi friend che vedremo in seguito. Per ora soffermiamoci su private e public.

Esempio di una classe

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

class Rettangolo {    //Rettangolo è il nome di un nuovo tipo

 private:    // non serve perché nelle classi è tutto privato di default
  double x, y; //sono dati privati e quindi accessibili solo ad altri metodi della stessa classe
  double semiperimetro () {return (x+y);}

 public:
  void set_values (double, double); // prototipo  di questo metodo pubblico.
  double area () {return (x*y);} // Qui definisco già la funzione area !!
  double perimetro() {return 2*(x+y);} //definizione  di questo metodo
  double perimetrobis () { return ( 2*semiperimetro() ); }

};
void Rettangolo::set_values (double a, double b) {  // implementazione fuori della classe
  x=a;   //tramite un metodo pubblico inizializzo i valori delle variabili (private) x e y
  y=b;
}
```

Esempio di una classe

```
int main () {
  Rettangolo ret1, ret2;//creo due oggetti (due istanze) di tipo Rettangolo
  ret1.set_values (2.1,3.6); // metodo pubblico; inizializzo i valori delle due variabili
  cout << "area = " << ret1.area() << " perimetro = " <<  ret1.perimetro() << endl;
  ret2.set_values (5.0, 6.0);
  cout << "area = " << ret2.area() << " perimetro = " <<  ret2.perimetro() << endl;
  //  cout << "accedi e stampa un dato privato " << ret1.x << endl;
  //Se scommento questa istruzione ho un errore perché il programma non può accedere
  //a ret1.x perché x è  privato.
  //A queste quantità private si può accedere solo con una funzione membro che sia pubblica
  //(tipo  set_values o area!)

  //  cout << "Tento di accedere ad una funzione membro (metodo) privato " <<
   //     ret1.semiperimetro()<< endl; //errore !!!!
  cout << "Posso accedere ad una funzione membro (metodo) privato tramite una funzione
              membro pubblica  = " << ret1.perimetrobis() << endl; //corretto !!!!

  return (0);
}
```

Esempio di una classe

Se il codice precedente lo scrivete in un unico file (EsempioClasse.cc) ed eseguite il programma avete questo risultato:

g++ -o EsempioClasse EsempioClasse.cc
./EsempioClasse

area = 7.56 perimetro = 11.4
area = 30 perimetro = 22
Posso accedere ad una funzione membro (metodo) privato tramite una funzione membro pubblica = 11.4

Nello stesso file abbiamo messo tutto il codice della classe e del programma che la usa.

Se in questo file prendo tutto il codice della classe e lo sposto dopo il codice del main
Il compilatore da errore !! È chiaro il perché?

Ma quando si hanno grossi programmi, non possiamo far dipendere l'esecuzione del programma dalla posizione del codice che da il prototipo della classe rispetto al main.

Separazione di Dichiarazione e Definizione

Dovremmo separare la dichiarazione della classe (dare nome delle funzioni, tipo di ritorno, numero e tipo dei parametri) dalla sua definizione dove viene data l'implementazione delle funzioni (cioè il codice!).

Quindi abbiamo un file con la dichiarazione (header file con estensione .hh o anche .h) e un file che contiene la definizione (file con estensione .cc o anche .cpp oppure .cxx, …). Poi c'è il programma che utilizza questa classe.

Vediamo come funziona con la classe appena vista

Header file della classe Rettangolo

Nello header file Rettangolo.hh metto il seguente codice

```
#ifndef RETTANGOLO_HH
#define RETTANGOLO_HH

class Rettangolo {
 private:
  double x, y;
  double semiperimetro ();

 public:
  void set_values (double, double);
  double area ();
  double perimetro();
  double perimetrobis ();
};

#endif
```

File di Implementazione:Rettangolo.cc

```
#include "Rettangolo.hh"

double Rettangolo::semiperimetro () {
return (x+y);
}
double Rettangolo:: area () {
return (x*y);
}
void Rettangolo::set_values (double a, double b) {
  x=a;
  y=b;
}
double Rettangolo::perimetro() {
return 2*(x+y);
}

double Rettangolo:: perimetrobis () {
return ( 2*semiperimetro() );
}
```

Programma che utilizza la classe Rettangolo

```
#include <iostream>
using namespace std;
#include "Rettangolo.hh"


int main () {
  Rettangolo ret1, ret2;
  ret1.set_values (2.1,3.6);
  cout << "area = " << ret1.area() << " perimetro = " <<  ret1.perimetro() << endl;
  ret2.set_values (5.0, 6.0);
  cout << "area = " << ret2.area() << " perimetro = " <<  ret2.perimetro() << endl;
  //  cout << "accedi e stampa un dato privato " << ret1.x << endl;        //errore

//  cout << "Tento di accedere ad una funzione membro (metodo) privato " << ret1.semiperi\
metro(); //errore !!!!
  cout << "Posso accedere ad una funzione membro (metodo) privato tramite una funzione
membro pubblica  = " << ret1.perimetrobis() << endl; //corretto !!!!

  return (0);
}
```

Esecuzione del Programma

Per eseguire il mio programma faccio:
g++ -o programma programma.cc Rettangolo.cc

Ma la classe Rettangolo potrebbe essere stata scritta da un'altra persona, il codice Rettangolo.cc potrebbe essere già stato compilato ed essere per esempio in una libreria.

Per utilizzare la classe Rettangolo devo linkare la libreria che contiene la compilazione di questo codice, inserire l'header file della classe nel mio programma e conoscendo lo header file vedo qual è l'interfaccia per usare la classe (cioè i metodi pubblici, i tipi di ritorno, il numero di parametri ed il loro tipo).

Io utilizzo la classe senza conoscere l'implementazione della classe e senza poter accedere direttamente ai dati della classe. Il funzionamento interno della classe è nascosto.

In questo consiste l'incapsulamento

Puntatori a Classi

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


class Rettangolo {
  double x, y;

 public:
  void set_values (double, double);
  double area () {return (x*y);}
  double perimetro () { return (2 * ( x  + y) );}
};

void Rettangolo::set_values (double a, double b) {
  x=a;
  y=b;
}
```

Puntatori a Classi

```
int main () {
  Rettangolo rettan1, * rettan2, * rettan3;
  //istanzio un oggetto di tipo Rettangolo e due puntatori  ad oggetti   di tipo Rettangolo

  rettan1.set_values (2.0, 4.0); //inizializzo i valori  dell'oggetto rettan1
  cout << "rettan1 area = " << rettan1.area() << "; rettan1  perimetro = ” rettan1.perimetro()
  << endl;

  rettan2 = new Rettangolo; //allocazione dinamica della  memoria necessaria
  rettan2 -> set_values (5.0, 6.0); // inizializzo l'oggetto puntato da rettan2
  cout << "*rettan2 area = " << rettan2->area() << "; *rettan2  perimetro = " <<  rettan2-
>perimetro() << endl;
  cout << "(*rettan2) area = " << (*rettan2).area() << "; (*rettan2) perimetro = " <<
(*rettan2).perimetro() << endl;

  rettan3 = &rettan1; // rettan3 = indirizzo di rettan1
  cout << "*rettan3 area = " << rettan3->area() << "; *rettan3  perimetro = " <<  rettan3-
>perimetro() << endl;
```

Puntatori a Classi

```
Rettangolo  * rettan4 = new Rettangolo [2]; //puntatore ad un array di due oggetti di tipo
Rettangolo

// rettan4[0]  primo oggetto puntato da rettan4; rettan4[1] secondo oggetto puntato
//da rettan4

  rettan4 -> set_values (4.0, 6.0);
  rettan4[1].set_values (10.0, 60.0);

  cout << "rettan4[0] area = " << rettan4[0].area()  << "; rettan4[0] perimetro  = " <<
  rettan4[0].perimetro() << endl;
 cout << "rettan4[1] area = " << rettan4[1].area()  << "; rettan4[1] perimetro  = " <<
rettan4[1].perimetro()  << endl;

 delete [] rettan4; // libera la memoria allocata dinamicamente dall'array di due oggetti
 delete rettan2; // libera la memoria allocata dinamicamente da rettan2
 return (0);
}
```

Puntatori a Classi

Compilo questo programma (il codice si trova nelle Applicazioni-Web)
e lo eseguo

g++ -o EsempioPuntatoreClasse EsempioPuntatoreClasse.cc
./EsempioPuntatoreClasse

==========
rettan1 area = 8; rettan1 perimetro = 12
*rettan2 area = 30; *rettan2 perimetro = 22
(*rettan2) area = 30; (*rettan2) perimetro = 22
*rettan3 area = 8; *rettan3 perimetro = 12
rettan4[0] area = 24; rettan4[0] perimetro = 20
rettan4[1] area = 600; rettan4[1] perimetro = 140

Costruttori e Distruttori

Il processo di creazione di un oggetto richiede inizializzazione dei dati membri e frequentemente anche di allocazione dinamica di memoria mediante new.

Negli esempi appena fatti noi abbiamo inizializzato dati di membri private col metodo public set_values

Questo viene realizzato dal **costruttore**: questo è un metodo con lo stesso nome della classe e che non ha alcun tipo di ritorno.

Il costruttore è un metodo speciale: non può essere chiamato esplicitamente. Viene chiamato automaticamente quando si crea un nuovo oggetto di quella classe

Il **distruttore** ha funzionalità opposta a quella del costruttore. È chiamato automaticamente quando l'oggetto è distrutto. Ha lo stesso nome della classe, non ha alcun tipo di ritorno ed è preceduto da una tilde

Aggiungiamo il costruttore e il distruttore alla classe Rettangolo vista in precedenza

Classe con Costruttore e Distruttore

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

class Rettangolo {

 private:
  double x, y;
  double semiperimetro () {return (x+y);}

 public:
  Rettangolo (double, double); // dichiarazione  del costruttore
  ~Rettangolo();  // dichiarazione del distruttore
  double area () {return (x*y);} // Qui definisco già la funzione area !!
  double perimetro() {return 2*(x+y);} //dichiarazione di questo metodo
  double perimetrobis () { return ( 2*semiperimetro() ); }
};

Rettangolo::Rettangolo (double a, double b) {  //implementazione del costruttore
  x=a;   //il costruttore inizializza le variabili  x e y  (non serve più  il metodo set_values)
  y=b;
}
```

Classe con Costruttore e Distruttore

```
Rettangolo::~Rettangolo(){
        cout << " Distruttore di Rettangolo " << endl;
}

int main () {
  Rettangolo ret1(2.1, 3.6), ret2(5.0, 6.0);//creo due oggetti e inizializzo le  due variabili

  cout << "area = " << ret1.area() << " perimetro = " <<  ret1.perimetro() << endl;
  cout << "area = " << ret2.area() << " perimetro = " <<  ret2.perimetro() << endl;

  return (0);
}

============
g++ -o Distruttore Distruttore.cc
./Distruttore
area = 7.56 perimetro = 11.4
area = 30 perimetro = 22
 Distruttore di Rettangolo
 Distruttore di Rettangolo
```

Costruttore di Default

Il costruttore come ogni altra funzione può avere overloading. Si possono avere più costruttori che hanno lo stesso nome ma che differiscono per i diversi tipi o il diverso numero di parametri. Si dice di default il costruttore con lista di parametri vuota.

Se non date alcun costruttore, il compilatore assume (implicitamente) il costruttore di default. Vediamo il solito esempio della classe Rettangolo:

```
================================================
#include <iostream>
using namespace std;
#include <iomanip>
class Rettangolo {
double x, y;
double semiperimetro () {return (x+y);}

public:
Rettangolo ();   // costruttore di default;
Rettangolo (double, double);   //overloading del costruttore;
double area () {return (x*y);} //
double perimetro() {return 2*(x+y);}
double perimetrobis () { return ( 2*semiperimetro() ); }
};
```

Costruttore di Default

```
Rettangolo:: Rettangolo (double a, double b) {
  x=a;  y=b;
}

Rettangolo:: Rettangolo () {
x=5;  y=5;
}

int main () {
Rettangolo ret1; //uso del costruttore di default. Notate che  non si  mettono le parentesi

Rettangolo ret2(8.0, 10.0); //qui viene usato il costruttore con due parametri

cout << "area (costruttore di default) = " << ret1.area() << ";  perimetro (costruttore di
default) = " <<  ret1.perimetro() << endl;

cout << "area (costruttore con due parametri)  = " << ret2.area() << "; perimetro
(costruttore con due parametri) = " <<  ret2.perimetro() << endl;

return (0);
}
```

Costruttore di Copia e Operatore Assegnazione di default

Vogliamo ora vedere come creare copie di oggetti esistenti. Problema questo abbastanza complesso! Nell'esempio di codice appena visto avevamo:

```
….
Rettangolo ret1(2.0, 3.0),ret2(3.0, 5.0); //creo ed inizializzo due oggetti (della stessa classe)
Rettangolo ret3 = ret1; //CREO un oggetto (ret3) come una copia di un oggetto esistente
                        //(ret1)
ret2 = ret1;  //qui ho già due oggetti inizializzati; all'oggetto ret2 ASSEGNO gli attributi di
              //un altro oggetto anch'esso già esistente.
……
……
```

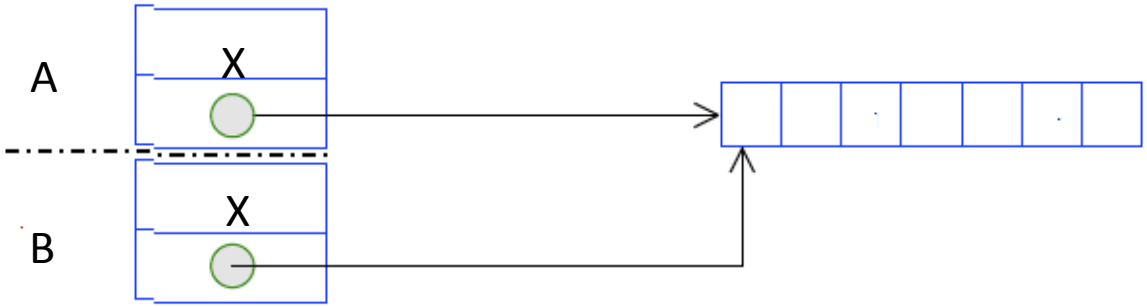
L'oggetto ret3 viene creato dal costruttore di copia (copy constructor) mentre la copia ret2 (di ret1) è dovuta all'operatore assegnazione. Il costruttore di copia e l'operatore assegnazione sono diversi e vengono usati in contesti differenti.

Quando non vengono forniti in modo esplicito , C++ usa automaticamente un costruttore di copia e un operatore di assegnazione di default : un oggetto viene copiato nell'altro bit a bit (*shallow copy*). Vedere esempio **DefaultConstructor.cc** in Applicazioni-Web

Costruttore di Copia e Operatore Assegnazione di default

In casi di classi semplici (come la classe Rettangolo che abbiamo già visto), non abbiamo bisogno di dichiarare un costruttore di copia e un operatore di assegnazione perché C++ fa tutto da solo utilizzando i suoi operatori di default.

Questi operatori di default non fanno bene il loro lavoro quando in una classe appaiono puntatori (per allocare memoria dinamica). In questo caso il costruttore di copia di default copia solo il puntatore e la quantità di memoria necessaria. Si hanno cosi due oggetti (A e B in figura} con puntatori uguali che puntano alla stessa area di memoria

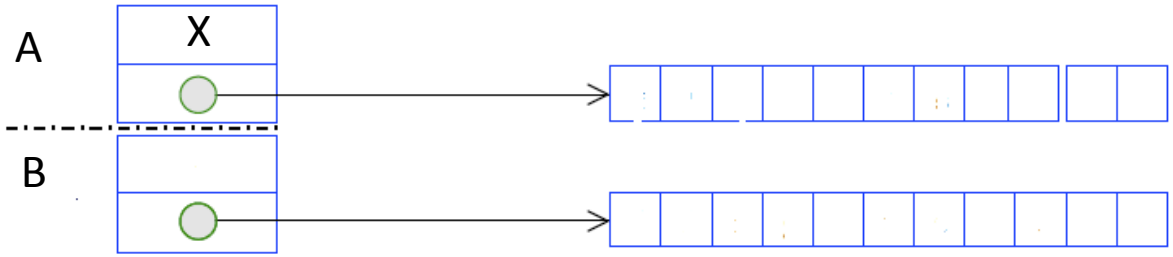


Quando cambio qualcosa nella memoria allocata per A, questo cambiamento si riflette in B.

Problema molto più grave quando nella classe c'è un distruttore che libera la memoria allocata. In questo caso si avrebbe una doppia cancellazione della stessa memoria e questo darebbe errore.

Costruttore di Copia e Operatore Assegnazione: Deep Copy

Per evitare questi errori, abbiamo bisogno non di copiare solo il puntatore ma anche la memoria allocata (*deep copy*):



In questo modo non si hanno i problemi presentati nella slide precedente. Per fare una deep copy dobbiamo scrivere il nostro costruttore di copia.

Il costruttore di copia è chiamato quando:

1) si inizializza un oggetto da un altro esistente (Rettangolo ret3 = ret1;)
2) si passa un oggetto ad una funzione func : func (ret1)
3) si crea un oggetto temporaneo (normalmente quando una funzione ritorna un oggetto) : ret1=func(); //qui ret1 riceve da func() un oggetto temporaneo

Regola generale:Se si ha bisogno o di un costruttore di copia, o di un operatore di assegnazione o di un distruttore, allora quasi sempre c'è bisogno di tutti e tre.

(The Rule of Three)

Sintassi del Costruttore di copia

La forma più generale di un costruttore di copia è:

```
nomeclasse (const  nomeclasse & other) {
    // corpo del costruttore di copia
 }
```

Questo costruttore accetta come parametro un riferimento const ad un altro oggetto della stessa classe. Const è necessario per garantirsi che il costruttore di copia non cambi il parametro e other è l'oggetto che si trova a destra della inizializzazione ed è passato come riferimento. A titolo illustrativo nella classe Rettangolo (dove in realtà non c'è alcun bisogno di scrivere un costruttore di copia) io avrei:

```
Class Rettangolo{
    public:
    ……..
   Rettangolo(const  Rettangolo & other ); //costruttore di copia
     …….
};

Rettangolo::Rettangolo(const Rettangolo& other){
    x= other.x;
    y= other.y;
}
```

Sintassi dell'Operatore Assegnazione

L'operatore assegnazione ha la forma :

```
nomeclasse & operatore= (const nomeclasse & other) {
   // corpo dell'operatore assegnazione
   ……..
 }
```

I corpi del costruttore di copia e dell'operatore assegnazione (e anche del distruttore) sono specifici della classe.

Vedere esempio CostruttoreDiCopie.cc in Applicazioni-Web

Funzioni (e Classi) Friend

Una funzione riesce ad accedere ai membri privati e protetti di
di una classe se è dichiarata friend di questa classe

Il prototipo di questa funzione (preceduto dalla parola chiave friend) va dichiarato (e con accesso public) all'interno della classe di cui è friend

Una funzione friend di una classe può essere anche una funzione membro di un'altra classe.

Non vale la proprietà transitiva. Se non viene esplicitamente dichiarato,
Il friend di un friend non è un friend !!

In modo analogo si hanno classi friend di altre classi.

Esempio di Funzione Friend

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


 class Rettangolo {
  double x, y;

 public:
   Rettangolo();
   Rettangolo (double, double);
   double area () {return (x*y);}
   friend Rettangolo  duplica (Rettangolo);
 };

Rettangolo::Rettangolo(){
  x = 0;
  y = 0;
}
```

Esempio di Funzione Friend

```
Rettangolo::Rettangolo(double a, double b){
  x = a;
  y = b;
}

Rettangolo duplica (Rettangolo rettparam) {
  Rettangolo rettan;
  rettan.x = rettparam.x*2;
  rettan.y = rettparam.y*2;
  return (rettan); // ritorna un ogggetto !!
}

int main () {
  Rettangolo ret1 (1.0, 1.0), ret2;
  ret2 = duplica (ret1); //duplica NON è un metodo della classe  Rettangolo ma friend
  cout << "area = " << ret1.area() << "; area duplicando i lati  = " <<  ret2.area() << endl;
   return (0);
}
```

Ereditarietà

- In natura si osservano gerarchie e classificazioni di oggetti; ad esempio una mela delizia fa parte della classificazione di mela; questa a sua volta è un particolare frutto; la frutta è a sua volta un particolare cibo. Potrei descrivere tutti questi oggetti dando tutte le loro caratteristiche ma questo non sarebbe efficiente.

- Io posso dare le caratteristiche dell'oggetto cibo (in questo caso la classe che lo crea è detta classe base). Da questa classe base derivo un'altra classe che della classe base eredita la parte pubblica e aggiungo le caratteristiche di questo nuovo oggetto (cioè specifico che si tratta di frutta). Ho la classe frutta derivata dalla classe cibo.

- Dalla classe frutta posso derivare poi una classe mela che eredità la parte pubblica della classe frutta e nella quale aggiungo le caratteristiche della mela.

- L'ereditarietà permette di riutilizzare codice già esistente, definendo nuove classi a partire da classi già definite.

L'ereditarietà è un aspetto importante della OOP

Classe Derivata

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

class Rettangolo {//classe base
  double x, y;
  double semiperimetro () {return (x+y);}

 public:
  Rettangolo (double, double);
  double area () {return (x*y);}
  double perimetro() {return 2*(x+y);}
  double perimetrobis () { return ( 2*semiperimetro() ); }
};

Rettangolo:: Rettangolo (double a, double b){
  x=a;
  y=b;
}

class Quadrato : public Rettangolo {   //classe derivata
  public:
  Quadrato (double a); //dichiarazione del costruttore
};
```

Classe Derivata

```
Quadrato::Quadrato( double x) : Rettangolo( x, x) { //implementazione del costruttore
}

int main () {
  Rettangolo ret1 (2.0, 3.0);
  Quadrato quadr1 (10);
  cout << "area Rettangolo = " << ret1.area() << "; area quadrato  = " <<  quadr1.area() <<
endl;

  Rettangolo * rett;
  Quadrato * quadr;
   rett = new Rettangolo ( 5.0, 8.0 );
   quadr = new Quadrato ( 15 );
   cout << "Usando i puntatori: area Rettangolo = " << rett->area() << "; area quadrato  = " <<
   quadr -> area() << endl;
     delete rett;
     delete quadr;
     return (0);
}
```

Ereditarietà Multipla

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

class Poligono {
  protected:  // specificatore di accesso per le classi derivate. Se  la commentassi?
  double base, altezza;

 public:
  void set_values (double a, double b){base=a; altezza=b; }
  };

class Output {
public :
  void stampa (double x) };

void Output :: stampa (double x){
  cout << x << endl;
}
class Rettangolo: public Poligono, public Output {
public:
  double area () {return (base * altezza);} //
};
```

Ereditarietà Multipla

```
class Triangolo : public Poligono, public Output{
public :
  double area() {return ( base * altezza  / 2); }
};

int main () {
  Rettangolo ret1;
  Triangolo  trian1;
  ret1.set_values(5.0, 6.0);   //Metodo ereditato dalla classe Poligono
  trian1.set_values(10.0, 28.0);
  ret1.stampa (ret1.area());     // Metodo ereditato dalla classe Output
  trian1.stampa (trian1.area());
  return (0);
}
```

==========

```
g++ -o MultipleInheritance MultipleInheritance.cc
./MultipleInheritance
30
140
```

Puntatore a Classe Base

Un puntatore ad una classe derivata è compatibile col puntatore alla sua classe base. Questa particolarità ha conseguenze molto importanti.

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

class Poligono {
protected:
double base, altezza;
public:
void set_values (double a, double b){ base=a; altezza=b; }
};

class Rettangolo: public Poligono {
 public:  double area () {return (base * altezza);} //
};

class Triangolo : public Poligono{
public :
double area() {return ( base * altezza  / 2); }
};
```

Puntatore a Classe Base

```
int main () {
 Rettangolo ret1;
 Triangolo  trian1;
 Poligono * polptr1 = &ret1;  // notare il puntatore di tipo Poligono per un oggetto Rettangolo
 Poligono * polptr2 = &trian1; //     “          “                “          “        Triangolo
 polptr1->set_values(5.0, 6.0);
 polptr2->set_values(10.0, 28.0);
// cout << " area rettan = " << polptr1->area() << "; area trian = "  << polptr2->area() <<endl;

// con questo comando il compilatore  darebbe errore perché  la classe Poligono
//non ha il metodo area  (che è invece presente nelle classi Rettangolo e Triangolo !

 cout << " area rettan = " << ret1.area() << "; area trian = "  << trian1.area() << endl;
 return (0);
}
==========================================
g++ -o PuntatoreClassBase PuntatoreClassBase.cc
./PuntatoreClassBase
 area rettan = 30; area trian = 140
```

Metodi Virtuali in una Classe

```
//Metodi virtuali in una classe
#include <iostream>
using namespace std;

class Poligono {
protected:
double base, altezza;

public:
 void set_values (double a, double b){ base=a; altezza=b; }
 virtual double area() { return (0); } // metodo virtuale; viene ridefinito nelle sue classi derivate
                                       //La ridefinizione sovrascrive  questa definizione nella classe base
};

 class Rettangolo: public Poligono {
 public:
 double area () {return (base * altezza);}
 };

class Triangolo : public Poligono{
public :
double area() {return ( base * altezza  / 2); }
};
```

Metodi Virtuali in una Classe

```
int main () {
 Rettangolo ret1;
 Triangolo  trian1;
 Poligono  polig1;
 Poligono * polptr1 = &ret1;
 Poligono * polptr2 = &trian1;
 Poligono * polptr3 = &polig1;
 polptr1->set_values(5.0, 6.0);
 polptr2->set_values(10.0, 28.0);
 polptr3->set_values(100.0, 10.0);

 cout << " area rettan = " << polptr1->area() << "; area trian = " << polptr2->area() << ";
area polig = "<< polptr3->area() << endl;
return (0);
}
==========================================
g++ -o MetodiVirtuali MetodiVirtuali.cc
./MetodiVirtuali
 area rettan = 30; area trian = 140; area polig = 0
```

Metodi Virtuali in una Classe

```
//Metodi virtuali in una classe
#include <iostream>
using namespace std;

class FiguraGeometrica {  //classe base
 public:
virtual void IoSono ()  ;
};

void FiguraGeometrica::IoSono () {
cout << “Non so che figura sono ! "  << endl;
};

class Rettangolo: public FiguraGeometrica {  //classe derivata polimorfica: eredita e
 //  dichiara una funzione membro
public:  void IoSono ()  {
cout << " Io sono un rettangolo! "  << endl;};
};

class Triangolo: public FiguraGeometrica {
 public:
void  IoSono ()  ;
};
```

Metodi Virtuali in una Classe

```
void Triangolo:: IoSono() {
 cout << " Io sono un triangolo! "  << endl;
 };

int main () {
FiguraGeometrica * Fig1, * Fig2,  * Fig3;// tutti puntatori a oggetti tipo FiguraGeometrica
Fig1 = new Rettangolo; //oggetti assegnati sono di tipo diverso !!
Fig2 = new Triangolo;
Fig3 = new FiguraGeometrica;
Fig1->IoSono ();  // Qui il compilatore chiama la funzione IoSono() appropriata !!
Fig2->IoSono ();
Fig3->IoSono ();
delete Fig1;
delete Fig2;
delete Fig3;
return (0);
}
=====================================
g++ -o VirtualFunction VirtualFunction.cc
./VirtualFunction
 Io sono un rettangolo!
 Io sono un triangolo!
 Non so che figura sono !
```

Classe base astratta

```
//Metodi virtuali in una classe
#include <iostream>
using namespace std;

class Poligono {
protected:
double base, altezza;
public:
void set_values (double a, double b){ base=a; altezza=b; }
virtual double area() = 0;  // Metodo (o funzione ) virtuale puro. È dichiarato ma non definito

// Qui manca del tutto la definizione della funzione. Il compilatore
//viene informato che c'è una  funzione area che verrà definita nelle classi derivate.
// È  detta funzione virtuale pura; la classe che ha almeno una funzione  virtuale pura  è detta
// classe astratta !!
// Questa classe non può essere istanziata perché ha un metodo non implementato!! Però ...
};

class Rettangolo: public Poligono {
public:  double area () {return (base * altezza);}
 };
```

Classe base astratta (cont)

```
class Triangolo : public Poligono {
public :
double area() {return ( base * altezza  / 2); }
};

int main () {
Rettangolo ret1;
Triangolo  trian1;

//Poligono  polig1; //sbagliato non posso creare l'oggetto polig1 da una classe astratta !!

Poligono * polptr1 = &ret1;
Poligono * polptr2 = &trian1;
polptr1->set_values(5.0, 6.0);
polptr2->set_values(10.0, 28.0);
cout << " area rettan = " << polptr1->area() << "; area trian = " << polptr2->area() << endl;
return (0);
}
=======================================
g++ -o AbstractBaseClass AbstractBaseClass.cc
./AbstractBaseClass
area rettan = 30; area trian = 140
```

Polimorfismo

Abbiamo visto che un puntatore alla classe base può essere usato per puntare ad un oggetto di una classe derivata. Questo è un fatto molto importante perché permette di trattare molti tipi diversi (derivati da una stessa classe base) come se fossero lo stesso tipo. Le stesse linee di codice possono funzionare egualmente bene per tutti questi diversi tipi.

La funzione, se dichiarata virtuale, viene definita (o ridefinita) in modo appropriato per il tipo della classe derivata. La funzione adatta alla classe derivata viene scelta durante il run-time (late binding).

La funzione virtuale permette di differenziare un tipo da tutti gli altri derivati da una stessa classe base.

Polimorfismo: Stessa interfaccia, metodi multipli

Il polimorfismo è il terzo aspetto essenziale (dopo l'incapsulamento e l'ereditarietà) della OOP .

Nei programmi di grandi dimensioni il polimorfismo permette di migliorare la organizzazione e leggibilità del codice come anche la loro estensibilità con l' aggiunta di parti previste o non previste nel progetto iniziale.

Struttura in C++

A differenza della struttura in C, la struttura in C++ ha oltre ai dati ha anche funzioni interne (metodi). È praticamente simile alla classe con una sola differenza però. Nella struttura tutto di default è public a meno che non sia dichiarato private (proprio l'opposto della classe!)

Di fatto non serve perché si usa la classe. È rimasta nel linguaggio C++ per compatibilità col C. Esempio di struttura in C++:

```
// Crea la struttura Time
#include <iostream>
using namespace std;
#include <iomanip>

// structure definition
struct Time {
public:  // non è necessario perché senza specificatore si ha public !!
  void set_values (int, int, int);
  int ore() {return (hour);};
  int minuti() {return (minute);};
  int secondi() {return (second);};
private:
  int hour; // 0-23 (24-hour clock format)
```

Struttura in C++

```
  int minute; // 0-59
  int second; // 0-59
}; // end struct Time

void Time::set_values (int a, int b, int c){
hour = a;
minute = b;
second = c;
}

void printUniversal (const  Time & );
void printStandard  (const   Time & );

int main() {
Time dinnerTime;
dinnerTime.set_values(18,30,0);
cout << "Dinner will be held at ";

printUniversal( dinnerTime );//struttura passata per riferimento ad una funzione
 cout << " universal time, \n which is ";
 printStandard( dinnerTime );
 cout << " standard time." << endl;
```

Struttura in C++

```
 return (0);
 } // end main

// print time in universal-time format
void printUniversal(  Time& t )
 {cout << setfill( '0' ) << setw( 2 ) << t.ore() << ":” << setw( 2 ) << t.minuti() << ":"
       << setw( 2 ) << t.secondi() ;
   }


 // print time in standard-time format
 void printStandard ( Time &t ) {
   cout << ( ( t.ore() == 0 || t.ore() == 12 ) ?  12 : t.ore() % 12 ) << ":" << setfill( '0' ) <<
   setw( 2 ) << t.minuti() << ":”   << setw( 2 ) << t.secondi()  << ( t.ore() < 12 ? " AM" :
“PM" ) ;
}

=====================================
   g++ -o timeC++ TimeC++.cc
   ./timeC++
   Dinner will be held at 18:30:00 universal time,
   which is 6:30:00 PM standard time.
```