Sommario

- Programmazione orientata agli oggetti, OOP:
  - Incapsulamento.
  - Ereditarietà (e composizione).
  - Polimorfismo.
- Ereditarietà:
  - Sintassi e metodi.
  - Upcasting.
- Polimorfismo:
  - Parola chiave `virtual`.
  - Overloading e overriding.

Programmazione orientata agli oggetti

- La programmazione orientata agli oggetti (*object-oriented programming*, OOP) si basa su tre principi fondamentali:
  - Incapsulamento (*encapsulation*): come nascondere l'implementazione di una classe.
  - Ereditarietà (*inheritance*): come promuovere un corretto ri-uso del codice.
  - Polimorfismo (*polymorphism*): come manipolare tipi (classi) tra loro collegati in un modo uniforme.

OOP: incapsulamento

- L'incapsulamento permette di nascondere i dettagli di implementazione non necessari all'utente della classe: per esempio, il programmatore utente vuole leggere due immagini, farne la differenza e salvare il risultato:

```
MyImage a("im1.pgm");
MyImage b("im2.pgm");
MyImage res = a-b;
res.Save("diff.pgm");
```


OOP: incapsulamento

- La classe `MyImage` ha incapsulato i dettagli interni di leggere un'immagine dall'hard disk, caricala in memoria, elaborarla e salvare il risultato nuovamente in un file.
- L'OOP permette al programmatore utente della classe di scrivere codice in modo più semplice: non deve preoccuparsi del codice per la gestione delle immagini.
- Si devono solo creare *istanze* (oggetti) e spedire gli appropriati *messaggi* (metodi).

OOP: incapsulamento

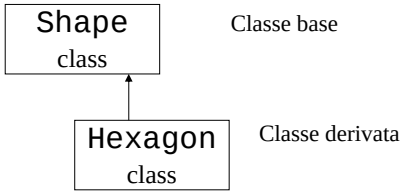
- Un altro aspetto dell'incapsulamento è la protezione dei dati: i campi di una classe (lo stato di un oggetto) devono essere dichiarati `private` e non `public`.
- In tal modo gli utenti della classe devono utilizzare dei metodi per poter leggere o scrivere i dati privati di un oggetto: non si accede mai direttamente all'implementazione interna di una classe.

OOP: ereditarietà

- L'ereditarietà permette di costruire nuove definizioni di classe utilizzando definizioni di classi esistenti.
- L'ereditarietà permette di estendere il comportamento di una *classe base* (base class or parent class), abilitando una *classe derivata* (derived class or child class) ad ereditare i metodi e i campi di tale classe base.

OOP: ereditarietà

- L'ereditarieta classica descrive una relazione del tipo *"è un"* (*"is-a"*).



- Il diagramma si legge come "*un esagono è una forma*": quando le classi sono legate da una relazione del tipo "is-a" allora sono in relazione di ereditarietà.

OOP: ereditarietà

- In questo caso la classe `Shape` descrive campi e metodi comuni a tutte le forme geometriche e quindi è la classe base (quella più generale).
- La classe `Hexagon` è una (is-a) forma, quindi eredita tutti i campi e metodi di `Shape` (*senza doverli riscrivere*) e poi definisce i propri specifici metodi e campi (è una classe derivata, meno generale).

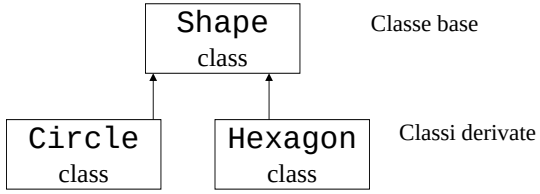
OOP: composizione

- Un'altra forma di riuso del codice nella OOP è la *composizione* (una relazione del tipo "*ha un*", "*has-a*"): alcuni campi della classe sono oggetti di altre classi.
- In questo caso non ci sono relazioni del tipo "is-a", come tra classe base e classi derivate.
- Si ricorda l'esempio della classe `Line` che rappresenta una linea che passa tra due punti (oggetti `Point`) del piano: la linea "has-a" punto e non "is-a" un punto, quindi è composizione.

OOP: polimorfismo

- Il polimorfismo permette di trattare nello stesso modo oggetti di classi diverse ma che sono collegate tra loro attraverso l'ereditarietà.
- La classe base definisce un insieme di metodi (interfaccia polimorfica) che sono comuni a tutte le classi derivate.
- L'interfaccia polimorfica è basata su *metodi virtuali* che ogni classe derivata può modificare per renderli specifici.
- Attraverso l'interfaccia polimorfica ogni oggetto risponde in modo specifico alla stessa richiesta.

OOP: polimorfismo



- Per esempio, la classe `Shape` definisce un *metodo virtuale* `Draw()` e tutte le sue classi derivate lo ridefiniscono (*overriding*), perché ogni forma si disegna in un modo diverso.

OOP: polimorfismo

- Allora attraverso un puntatore della classe base posso invocare il metodo corretto per ogni oggetto creato:

```
void elabora(Shape *p){
   //...
   p->Draw();
}
int main(){
   Circle c;
   Hexagon h;

   elabora(&c);
   elabora(&h);
   return 0;
}
```

Disegna un cerchio o un esagono, anche se è invocato da un puntatore a `Shape`

Composizione

- Una possibilità per *riusare il codice* già scritto da altri è implementare una classe che ha come *dati membro* degli *oggetti* di altre classi, cioè la composizione (*composition*).
- Il *vantaggio* è che *non* si accede direttamente al *codice*, ma se ne usano le *funzionalità*: in tal modo non si rischia di introdurre nuovi errori in codice che è corretto (è stato verificato da chi ha scritto le classi che si usano per implementare la propria classe).
- Un esempio di composizione (già presentato in una precedete lezione) può essere la classe che rappresenta una retta che usa due `Point` come *attributi privati*.

Composizione

Line.h

```
#ifndef LINE_H
#define LINE_H

#include "point.h“

class Line{
   Point p1,p2;
public:
   Line(float x1, float y1,
        float x2, float y2);
   ~Line();

   void Stampa();
};
#endif
```

Oggetti come attributi

Line.cpp

```
#include "Point.h"
#include "Line.h"
#include <iostream>
using namespace std;
Line::Line(float x1, float y1,
           float x2, float y2) :
        p1(x1,y1), p2(x2,y2)
{
   cout<<"Line()"<<endl;
   //p1(x1,y1);//errore:
                //compile-time
}
Line::~Line(){
   cout<<"~Line()"<<endl;
}
void Line::Stampa(){
   cout<<"\tL: ";
   p1.Stampa(); p2.Stampa();
   cout<<endl;
}
```


Composizione

Test.cpp

```
#include "Point.h"
#include "Line.h"
#include <iostream>
using namespace std;

void test1();

int main(){

   test1();

   return 0;
}

void test1(){
   Line l1(1,1,3,3);
   l1.Stampa();
}
```

Una possibile uscita

```
Point(1,1)
Point(3,3)
Line()
      L: (1,1) (3,3)
~Line()
~Point
~Point
```

Sono chiamati i costruttori dei dati membro e i relativi distruttori

Composizione

- I sotto-oggetti (*subobject*) `p1` e `p2` rappresentano l'implementazione della retta e quindi sono *privati*: in tal modo si può *cambiare l'implementazione* e lasciare la stessa interfaccia pubblica.

- In altri casi un sotto-oggetto potrebbe essere lasciato pubblico in modo da poter utilizzare la sua interfaccia (oppure l'oggetto privato e un insieme di metodi pubblici che richiamano l'interfaccia dell'oggetto privato).
- Vediamo un esempio di una classe che rappresenta un computer con un lettore di DVD pubblico.

Composizione

DVD.h

```
#ifndef DVD_H
#define DVD_H
class DVD{
   //campi
public:
   //costruttori,
   //distruttore, altri
   //metodi
   DVD();
   DVD& open();
   DVD& close();
   void play();
};
#endif
```

DVD.cpp

```
#include "DVD.h"
#include <iostream>
using namespace std;
DVD::DVD(){
   cout<<"\tCostruttore DVD()"<<endl;
}
DVD& DVD::open(){
   cout<<"DVD aperto"<<endl;
   return *this;
}
DVD& DVD::close(){
   cout<<"DVD chiuso"<<endl;
   return *this;
}
void DVD::play(){
   cout<<"DVD..."<<endl;
}
```

Il ritorno di `*this` consente la concatenazione delle chiamate che si riferiscono allo stesso oggetto.

Composizione

Desktop.h

```
#ifndef DESKTOP_H
#define DESKTOP_H
#include "DVD.h"
class Desktop{
   //campi
public:
   //costruttori,
   //distruttore, altri
   //metodi
   Desktop();
   DVD dvd;
};
#endif
```

Desktop.cpp

```
#include "Desktop.h"
#include <iostream>
using namespace std;
Desktop::Desktop():dvd(){
   cout<<"\tCostruttore
        Desktop()"<<endl;
}
```

test.cpp

```
void test1(){
   Desktop pc;
   pc.dvd.open().close().play();
}
```

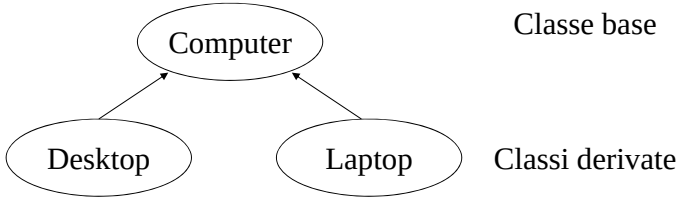
Una possibile uscita

```
      Costruttore DVD()
      Costruttore Desktop()
DVD aperto
DVD chiuso
DVD...
```


Composizione ed ereditarietà

- Un altro modo di *riusare il codice* è attraverso il meccanismo dell'ereditarietà: in tal caso la nuova classe *eredita* attributi e metodi della classe base.
- In generale, la composizione implica una relazione del tipo "*has-a*", mentre l'ereditarietà implica una relazione del tipo "*is-a*": un Desktop *ha un* DVD e non *è un* DVD. Mentre un Desktop *è un* Computer.
- La *classe base* rappresenta una *astrazione* più *generale*, mentre le *classi derivate* ne sono una *specializzazione*.

Ereditarietà



- La definizione della gerarchia di classi mostrata è la seguente:

```
class Computer{…};
class Desktop: public Computer{…};
class Laptop: public Computer{…};
```

Ereditarietà : sintassi

- La parola chiave `public` indica che i membri pubblici della classe base rimangono pubblici per la classe derivata, altrimenti sarebbero privati.
- La classe derivata contiene un *sotto-oggetto* della classe base.
- Per inizializzare un sotto-oggetto si deve chiamare l'opportuno costruttore nella *lista di inizializzazione dei costruttori*.
- I distruttori sono automaticamente chiamati per l'intera gerarchia.

Ereditarietà : sintassi

Base.h

```
#ifndef BASE_H
#define BASE_H
class Base{
   int b;
public:
   Base(int i=0);
   ~Base();
};
#endif
```

Base.cpp

```
#include "Base.h"
#include <iostream>
using namespace std;
Base::Base(int i):b(i){
   cout<<"Base("<<b<<")"<<endl;
}
Base::~Base(){
   cout<<"~Base()"<<endl;
}
```

Derivata.h

```
#ifndef DERIVATA_H
#define DERIVATA_H
#include "Base.h"
class Derivata : public
   Base{
   int d;
public:
   Derivata(int i=0, int
                        j=0);
   ~Derivata();
};
#endif
```

Derivata.cpp

```
#include "Derivata.h"
#include <iostream>
using namespace std;
Derivata::Derivata(int i, int j):
   Base(i) , d(j){
   cout<<"Derivata("<<d<<")"<<endl;
}
Derivata::~Derivata(){
   cout<<"~Derivata()"<<endl;
}
```

Costruttore del sotto-oggetto della classe base

Ereditarietà : sintassi

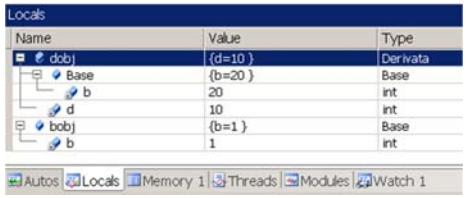
test.cpp

```
#include "Base.h"
#include "Derivata.h"
#include <iostream>
using namespace std;

void test1(){
   Base bobj(1);
   cout<<"-----------"<<endl;
   Derivata dobj(20,10);
   cout<<"-----------"<<endl;
   cout<<"Base "<<sizeof(bobj)<<"
   byte"<<endl;
   cout<<"Derivata "<<sizeof(dobj)<<"
   byte"<<endl;
   cout<<"-----------"<<endl;
}

int main(){
   test1();
   return 0;
}
```

Una possibile uscita

```
Base(1)
-----------
Base(20)
Derivata(10)
-----------
Base 4 byte
Derivata 8 byte
-----------
~Derivata()
~Base()
~Base()
```

Debugging



Ereditarietà : metodi

- Se nella classe derivata si ridefinisce un metodo della classe base vi sono due possibilità:
  - Se il metodo ha la stessa firma e tipo di ritorno, si dice che si è eseguito un *redefining*;
  - Se cambia la firma o il tipo di ritorno allora tutti i corrispondenti metodi sovraccaricati della classe base sono nascosti.
- N.B. Le classi sono semplici per poter focalizzare l'attenzione solo sull'ereditarietà.

Ereditarietà : metodi

Base.h

```
#ifndef BASE_H
#define BASE_H
class Base{
   int b;
public:
   Base(int i=0);
   ~Base();
   void Set(int i);
   void Set(int i, int j);
   void Print();
};
#endif
```

Base.cpp

```
#include "Base.h"
#include <iostream>
using namespace std;
Base::Base(int i):b(i){
   cout<<"Base("<<b<<")"<<endl;
}
Base::~Base(){
   cout<<"~Base()"<<endl;
}
void Base::Set(int i){
   b=i;
   cout<<"Base::Set(int )"<<endl;
}
void Base::Set(int i, int j){
   b=i*j;
   cout<<"Base::Set(int, int )"<<endl;
}
void Base::Print(){
   cout<<"Base::Print(): "<<b<<endl;
}
```


Ereditarietà : metodi

Derivata.h

```
#ifndef DERIVATA_H
#define DERIVATA_H
#include "Base.h“’

class Derivata : public Base{
   int d;
public:
   Derivata(int i=0, int j=0);
   ~Derivata();
   void Set(int i);
};
#endif
```

Derivata.cpp

```
#include "Derivata.h"
#include <iostream>
using namespace std;
Derivata::Derivata(int i, int j):
   Base(i), d(j){
   cout<<"Derivata("<<d<<")"<<endl;
}

Derivata::~Derivata(){
   cout<<"~Derivata()"<<endl;
}

void Derivata::Set(int i){
   d=i;
   Base::Set(2*i);
   cout<<"Derivata::Set(int )"<<endl;
}
```

Viene chiamato direttamente un metodo del sotto-oggetto della classe base

Ereditarietà : metodi

test.cpp

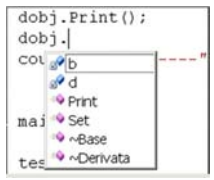
```
void test2(){
   Derivata dobj(20,10);
   cout<<"-----------"<<endl;
   dobj.Set(3);
   //dobj.Set(3,5);//errore
   cout<<"-----------"<<endl;
   dobj.Print();
   cout<<"-----------"<<endl;
}
```

Una possibile uscita

```
Base(20)
Derivata(10)
-----------
Base::Set(int )
Derivata::Set(int )
-----------
Base::Print(): 6
-----------
~Derivata()
~Base()
```

Metodo presente solo nella classe base ed ereditato dalla classe derivata

Visual Studio



Ereditarietà: `protected`

- In alcuni casi è utile fornire l'accesso a una parte dell'implementazione della classe base, in modo che lo sviluppatore delle classi derivate la possa sfruttare. Tuttavia per gli altri utenti deve essere privata. Tale tipo di *accesso differenziato* è permesso dalla parola chiave `protected`.
- Una classe derivata può accedere direttamente ad un membro `protected`, mentre il resto del programma deve utilizzare una funzione pubblica di accesso. Tuttavia l'*accesso* diretto per la classe derivata è relativo solo al proprio *sotto-oggetto* e non ad un oggetto indipendente.

Ereditarietà : protected

Base.h

```
#ifndef BASE_H
#define BASE_H
class Base{
protected:
   int b;
public:
   Base(int i=0);
   ~Base();
   void Set(int i);
   void Print()const;
};
#endif
```

Base.cpp

```
#include "Base.h"
#include <iostream>
using namespace std;

Base::Base(int i):b(i){
   cout<<"Base("<<b<<")"<<endl;
}

Base::~Base(){
   cout<<"~Base()"<<endl;
}

void Base::Set(int i){
   b=i;
}

void  Base::Print()const{
   cout<<"b="<<b<<endl;
}
```


Ereditarietà : protected

Derivata.h

```
#ifndef DERIVATA_H
#define DERIVATA_H
#include "Base.h"
class Derivata : public Base{
   int d;
public:
   Derivata(int i=0, int j=0);
   ~Derivata();
   void Set(int i);
   void Elab(Base &bobj);
   void Print()const;
};
#endif
```

Derivata.cpp

```
#include "Derivata.h"
#include <iostream>
using namespace std;
Derivata::Derivata(int i, int j):
   Base(i) , d(j){
   cout<<"Derivata("<<d<<")"<<endl;
}
Derivata::~Derivata(){
   cout<<"~Derivata()"<<endl;
}
void Derivata::Set(int i){
   d=i;
   b=2*i;
}
void  Derivata::Print()const{
   cout<<"b="<<b<<" d="<<d<<endl;
}
void Derivata::Elab(Base &bobj){
   //d=bobj.b;//errore
   bobj.Print();
}
```

Un campo del proprio sotto-oggetto

Un campo di un oggetto esterno

Ereditarietà : protected

test.cpp

```
void test3(){
   Base bobj(10);
   Derivata dobj(2,1);
   //cout<<bobj.b;//errore
   dobj.Set(3);
   dobj.Print();
   cout<<"-----------"<<endl;
   dobj.Elab(bobj);
}
```

Rimane privato per gli utenti della classe

Una possibile uscita

```
Base(10)
Base(2)
Derivata(1)
b=6 d=3
-----------
b=10
~Derivata()
~Base()
~Base()
```


Ereditarietà: upcasting

- Un aspetto importante dell'ereditarietà non è solo il fatto che le classi derivate hanno nuove funzionalità, ma è la *relazione di tipo* che esiste tra classe base e derivate: *la nuova classe è un tipo della classe esistente*.
- In tal modo tutte le *funzionalità* della classe *base* sono *presenti* nella classe *derivata* e quindi qualsiasi messaggio che può essere spedito ad un oggetto della classe base può essere spedito ad un oggetto della classe derivata.

Ereditarietà : upcasting

- La *conversione* di un *puntatore* o *riferimento* ad un oggetto di una classe derivata in un puntatore o riferimento alla classe base è detto *upcasting* ed è supportato dal compilatore, perché risulta sicuro convertire un tipo specifico/derivato a uno più generale/base (fornisce le *funzionalità* minime e quelle *comuni*)

```
void test4(){
   Base bobj(10);
   Derivata dobj(2,1);
   Base *p = &dobj;
   //p->Elab(bobj);//errore
   p->Print();
}
```

Solo l'interfaccia della classe base è disponibile

Una possibile uscita

```
Base(10)
Base(2)
Derivata(1)
b=2
~Derivata()
~Base()
~Base()
```


Ereditarietà : upcasting

- Per default le funzioni membro sono invocate in base al *tipo statico* del riferimento o puntatore attraverso il quale sono invocate: in tal modo l'invocazione è determinata a tempo di compilazione.
- Questa scelta è stata fatta per *efficienza*.
- Il termine *upcasting* si riferisce a come normalmente vengono disegnate le gerarchie di classi derivate: la classe base è in alto.

Ereditarietà: metodi speciali

- Vi sono alcuni metodi speciali che non sono ereditati ma devono essere direttamente *implementati*: i *costruttori* e il *distruttore*, in particolare il *costruttore per copia* e *l'operatore di assegnamento*.

OOP: polimorfismo

- Se la classe base e derivata condividono la *stessa interfaccia pubblica*, la classe derivata è detta *sottotipo* (*subtype*).
- Un *puntatore* o un *riferimento* a una classe base possono *riferirsi* a qualunque *sottotipo derivato* senza l'intervento esplicito del programmatore.
- Manipolare più di un tipo con un puntatore o riferimento a una classe base è detto *polimorfismo*.
- Il *polimorfismo* dei sottotipi permette di scrivere la parte centrale e più importante di un'applicazione *senza curarsi dei singoli tipi* che si vogliono manipolare.

OOP: polimorfismo

- Si *programma l'interfaccia* della classe base della nostra gerarchia mediante puntatori o riferimenti alla classe base. Il *tipo effettivo* cui essi si riferiscono è *risolto durante l'esecuzione* (*run-time*) quando viene chiamata l'istanza appropriata dell'interfaccia pubblica.
- La risoluzione durante l'esecuzione del metodo da invocare è detta *legame dinamico*, *dynamic binding* (o *late binding*). Per default i metodi sono risolti *staticamente* a tempo di compilazione, *static binding* (o *early binding*).
- Il legame dinamico è supportato dal meccanismo delle *funzioni virtuali* (*virtual functions*).

Ereditarietà : upcasting

- Per default le funzioni membro sono invocate in base al *tipo statico* del riferimento o puntatore attraverso il quale sono invocate: in tal modo l'invocazione è determinata a tempo di compilazione.

```
void test4(){
   Base bobj(10);
   Derivata dobj(1);
   Base *p = &dobj;

   p->Print();
}
```

Viene invocato il metodo `Print()` di `Base`

Una possibile uscita

```
Base(10)
Base(2)
Derivata(1)
b=2
~Derivata()
~Base()
~Base()
```


Polimorfismo

Base.h

```
#ifndef BASE_H
#define BASE_H
class Base{
protected:
   int b;
public:
   Base(int i=0);
   ~Base();
   void Set(int i);
   virtual void Print()const;
};
#endif
```

Il metodo `Print()` di `Base` viene dichiarato virtuale

Base.cpp

```
#include "Base.h"
#include <iostream>
using namespace std;

Base::Base(int i):b(i){
   cout<<"Base("<<b<<")"<<endl;
}

Base::~Base(){
   cout<<"~Base()"<<endl;
}

void Base::Set(int i){
   b=i;
}

void  Base::Print()const{
   cout<<"b="<<b<<endl;
}
```


Polimorfismo

Derivata.h

```
#ifndef DERIVATA_H
#define DERIVATA_H
#include "Base.h"
class Derivata : public Base{
   int d;
public:
   Derivata(int i=0, int j=0);
   ~Derivata();
   void Set(int i, int j);

   void Print()const;
};
#endif
```

Overriding del metodo virtuale `Print()` della classe `Base`.

Derivata.cpp

```
#include "Derivata.h"
#include <iostream>
using namespace std;
Derivata::Derivata(int i, int j):
   Base(i) , d(j){
   cout<<"Derivata("<<d<<")"<<endl;
}
Derivata::~Derivata(){
   cout<<"~Derivata()"<<endl;
}
void Derivata::Set(int i , int j){
   d=j;
   b=i;
}

void  Derivata::Print()const{
   Base::Print();
   cout<<"d="<<d<<endl;
}
```

Polimorfismo

- Se le funzioni membro della classe base sono dichiarate `virtual`, allora sono invocate in base al *tipo dinamico* del riferimento o puntatore attraverso il quale sono invocate: in tal modo l'invocazione è determinata a tempo di esecuzione.

```
void test4(){
  Base bobj(10);
  Derivata dobj(2,1);
  Base *p = &dobj;

  p->Print();
}
```

Viene invocato il metodo `Print()` di `Derivata`

Una possibile uscita

```
Base(10)
Base(2)
Derivata(1)
b=2
d=1
~Derivata()
~Base()
~Base()
```


Polimorfismo

- In corrispondenza di ogni *invocazione* durante l'*esecuzione* del programma, viene determinato l'*effettivo tipo* (classe) a cui punta il riferimento o il puntatore e viene chiamata l'*istanza appropriata* del metodo, che deve essere dichiarato *virtuale nella classe base*. Il meccanismo che supporta tale comportamento è il *dynamic binding*.

- Nel *paradigma orientato agli oggetti* il programmatore manipola oggetti principalmente attraverso puntatori e riferimenti alla classe base.
- Nel *paradigma basato sugli oggetti* si manipola un'istanza di un singolo tipo fissato al momento della compilazione.

Polimorfismo

- Il polimorfismo esiste soltanto all'interno di singole gerarchie di classi.

- I puntatori `void *` possono essere considerati polimorfici, ma sono privi di un supporto *esplicito* da parte del linguaggio: devono essere gestiti mediante *cast espliciti* e qualche forma di *discriminante* che tenga traccia del tipo effettivo puntato. Quindi non sono usati per il polimorfismo.

- Per default, le funzioni membro di una classe *non* sono *virtuali* (per ragioni di efficienza): in tal caso la funzione invocata è quella definita dal *tipo statico* (quello a tempo di compilazione).

Polimorfismo

- Quando una funzione membro è dichiarata *virtuale*, allora la funzione invocata è quella definita dal *tipo dinamico* (quello a tempo di esecuzione).
- Questo permette di *estendere* la propria applicazione con nuovi tipi ed utilizzare sempre le funzionalità fornite dalla classe base (si programma l'interfaccia della classe di base).
- Il meccanismo delle funzioni virtuali è supportato solo con puntatori o riferimenti alla classe base.
- *Nel caso degli oggetti si utilizza il tipo statico, cioè non vi è ambiguità sul tipo.*

Overloading e overriding

- Quando si esegue un *overriding* di una funzione virtuale, eventuali metodi sovraccaricati della classe base sono nascosti.
- Non si può cambiare il tipo di ritorno di una funzione *overridden*. È possibile quando il tipo ritornato è un riferimento o puntatore ad una classe derivata dal tipo base.
- *Le funzioni virtuali disponibili per un riferimento o puntatore della classe base sono quelle della classe base e non eventuali nuove funzioni definite nelle classi derivate.*

Classi astratte

- Quando si vuole che la *classe base* rappresenti solo un'*interfaccia* (comportamento) per le sue classi derivate, ma che *non* sia *istanziata*, in questo caso si rende la *classe astratta*. Una classe è astratta quando contiene almeno una *funzione virtuale pura*: una dichiarazione di membro `virtual` seguita da `=0`.
- Quando una classe astratta è ereditata, la *classe derivata* deve fornire le *implementazioni* delle funzioni virtuali pure, altrimenti diventa essa stessa astratta.
- È utile quando la classe base rappresenta azioni che si applicano a tutte le classi derivate, ma non ha senso creare un oggetto della classe base.

Classi astratte

- È possibile fornire una definizione di una funzione virtuale pura: per esempio, per contenere codice utile alle classi derivate.

- Quando si invoca una funzione virtuale usando l'operatore di risoluzione di scopo di classe, `::`, si supera il meccanismo delle funzioni virtuali, forzando la risoluzione statica. È utile per invocare funzioni virtuali pure.

Costruttori e distruttori

- I costruttori *non* possono essere virtuali, perché creano oggetti di una *classe specifica* e non puntatori. I puntatori possono essere usati per manipolare oggetti già creati.
- I costruttori della classe base sono chiamati per primi nella lista di inizializzazione dei costruttori.
- Eventuali funzioni virtuali invocate nel costruttore fanno riferimento alle *versioni locali*, cioè al tipo "corrente".
- Lo stesso comportamento si ha per i distruttori: eventuali funzioni virtuali invocate nel distruttore fanno riferimento alle *versioni locali*.
- I *distruttori* sono chiamati in *ordine inverso* rispetto ai *costruttori*.

Costruttori e distruttori

- I *distruttori* possono essere *virtuali*: in generale, se una classe presenta una funzione virtuale *dovrebbe fornire* anche il distruttore virtuale.
- Poiché è frequente *manipolare* un *oggetto* attraverso un *puntatore* della sua classe *base*, quando è necessario utilizzare `delete`, allora dovrebbe essere invocato il distruttore appropriato. Ciò accade solo se il distruttore della classe base è dichiarato virtuale.
- Se il distruttore non è virtuale, vi è il rischio che non vengano rilasciate le risorse della classe derivata.
- I distruttori virtuali puri sono legali, ma deve essere fornita anche una definizione. Tuttavia, in una classe derivata non è richiesto di fornirne esplicitamente una definizione.