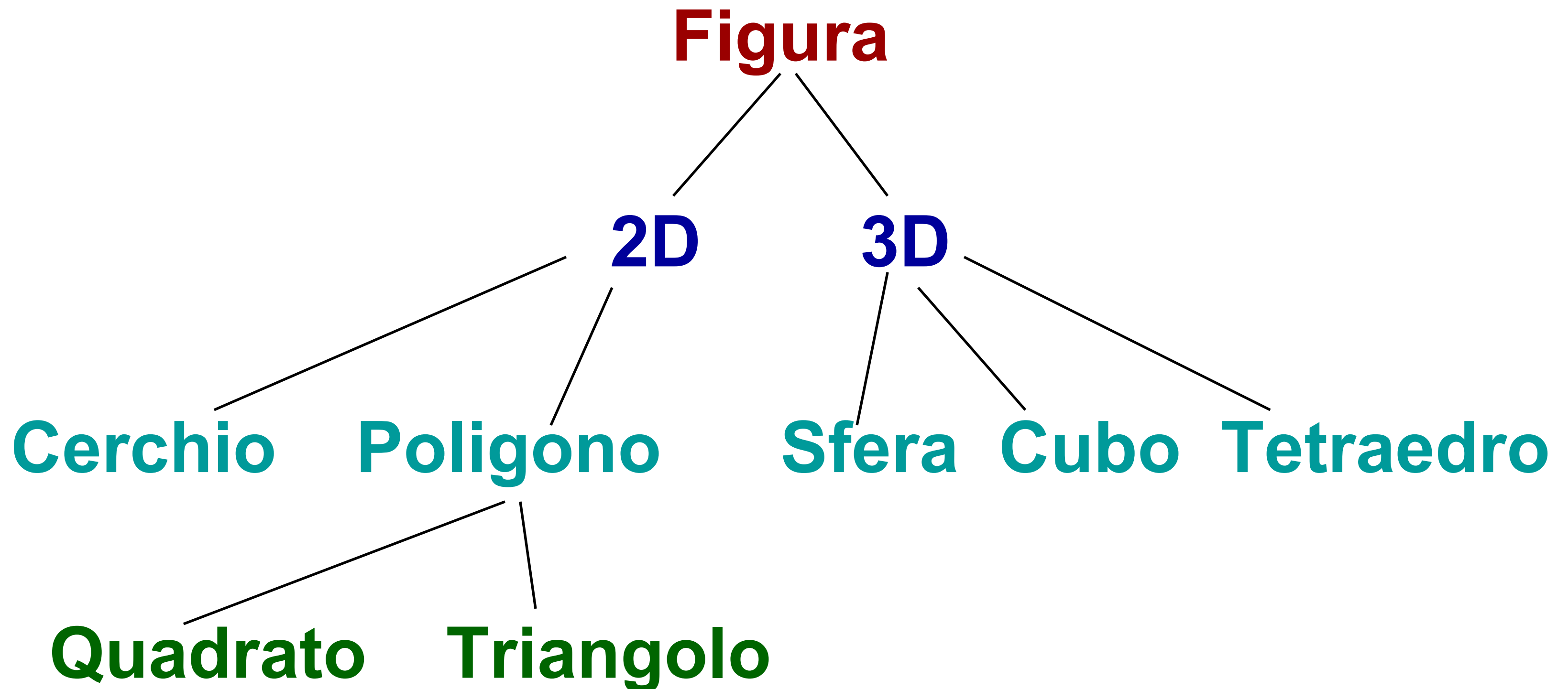
Ereditarietà

Scopo di questa lezione:

- Imparare a creare nuove classi ereditando da classi già esistenti.
- Capire come l'ereditarietà favorisca il riutilizzo del codice esistente.
- Capire le nozioni di classe base e di classe derivata.

Classi base e classi derivate



classi base	classi derivate
Figura	2D
	3D
2D	Cerchio
	Poligono
3D	Sfera
	Cubo
	Tetraedro
Poligono	Quadrato
	Triangolo

Sintassi

class classeDerivata : elencoClassi {

dichiarazione della classe

};

dove l'elencoClassi e' una lista identificatori di classi base separati da virgole, ciascuno preceduto da una parola chiave che specifica il tipo di accesso.

Esempi

```
class Figura {
};     // questa e' una classe base

class 2D : public Figura {
};     // 2D eredita da Figura

class Cerchio : public 2D {
};    // Cerchio eredita da 2D
```

L'ereditarieta' in C++:

- Estensione delle caratteristiche di una classe: la classe derivata è un caso particolare della classe base con alcuni dettagli in più
  - la figura 2D e' un caso particolare di Figura
  - il Cerchio e' una figura 2D
- Adesione ad un modello: la classe base definisce le caratteristiche minimali che devono avere tutte le classi derivate
  - Se Figura ha un metodo che si chiama disegna tutte le classi che ereditano da Figura avranno un metodo disegna

Membri della classe derivata

- Dati membri della classe base
- Metodi membri della classe base
- Nuovi dati membri della classe derivata
- Nuovi metodi membri della classe derivata

Figura

string nome

void disegna( )

2D

string nome
string colore
double superficie

void disegna( )
void trasla( )
void ruota( )

Protezioni

L'accesso ai membri di una classe può essere di tre tipi

- private *// dati e metodi inacessibili dall'esterno*
- public *// dati e metodi accessibili a tutti*
- protected *// public per le classi derivate, private per gli altri*

Analogamente la classe derivata può ereditare dalle classe base in tre modi

- private
- public
- protected

Esempio

```
class B {
  public:
      int x;
  protected:
      int w;
  private:
      int z;
  };
```

Derivazione pubblica

- Ogni membro ***public*** della classe base è ***public*** nella classe derivata
- Ogni membro ***protected*** della classe base è ***protected*** nella classe derivata
- Ogni membro ***private*** della classe base è ***private*** per la classe derivata ed ***è visibile solo dalla classe base***

Ovvero le protezioni dei membri della classe base rimangono inalterate nella classe derivata

```
class D : public B {   // i membri di D hanno
                       // accesso sia ai membri pubblici
                       // che ai membri protetti di B
  // gli utenti della classe D hanno accesso solo ai
  // membri pubblici di B e di D, le classi derivate da D
  // hanno accesso anche ai membri protetti sia di B che di D
  public:
        int a;
  protected:
        int b;
  private:
        int c;
  };
```

Derivazione protetta

- Ogni membro ***public*** della classe base è ***protected*** nella classe derivata
- Ogni membro ***protected*** della classe base è ***protected*** nella classe derivata
- Ogni membro ***private*** della classe base è ***private*** nella classe derivata ed ***è visibile solo dalla classe base***

```
class D : protected B {// i membri di D hanno
                       // accesso sia ai membri pubblici
                       // che ai membri protetti di B
  // gli utenti della classe D hanno accesso solo ai
  // membri pubblici di D, e a nessun membro di B,
  // le classi derivate da D hanno accesso anche ai membri
  // protetti di D e ai membri pubblici e protetti di B
  public:
      int a;
  protected:
      int b;
  private:
      int c;
  };
```

Derivazione privata

- Ogni membro ***public*** della classe base è ***private*** nella classe derivata
- Ogni membro ***protected*** della classe base è ***private*** nella classe derivata
- Ogni membro ***private*** della classe base è ***private*** nella classe derivata ed ***è visibile solo dalla classe base***

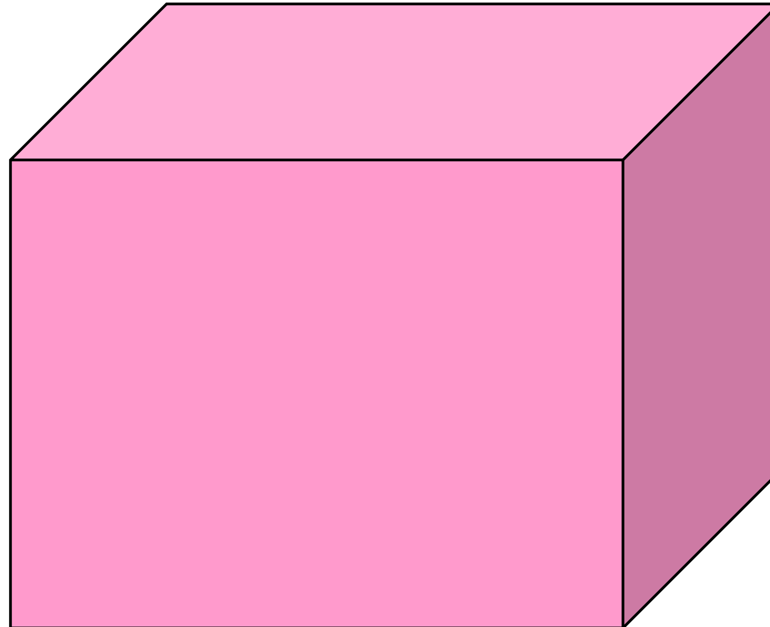
```
class D : private B {    // i membri di D hanno
                         // non hanno accesso a nessun membro
   di B
        // gli utenti della classe D hanno accesso solo ai
        // membri pubblici di D, e a nessun membro di B,
        // le classi derivate da D hanno accesso anche ai membri
        // protetti di D ma non hanno accesso ai membri di B
   public:
        int a;
   protected:
        int b;
   private:
        int c;
   };
```

Costruttori e Distruttori

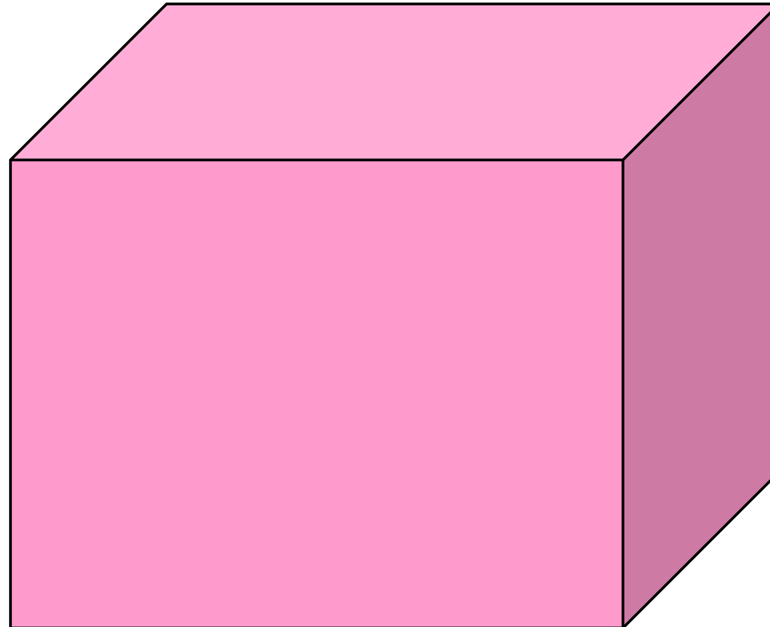
- Quando si crea un oggetto di una classe derivata viene prima chiamato il costruttore della classe base e poi quello della classe derivata
- Quando si distrugge un oggetto di una classe derivata viene prima chiamato il distruttore della classe derivata e poi quello della classe base
- È come se la classe base fosse un contenitore per la classe derivata: prima si crea il contenitore, poi il contenuto, prima si cancella il contenuto e poi il contenitore.

Creazione di un oggetto base

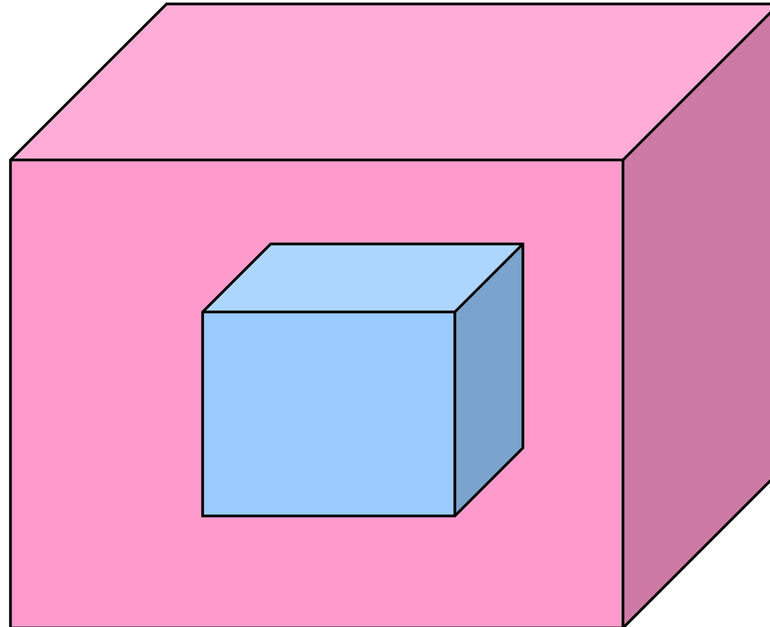
Creazione di un oggetto base



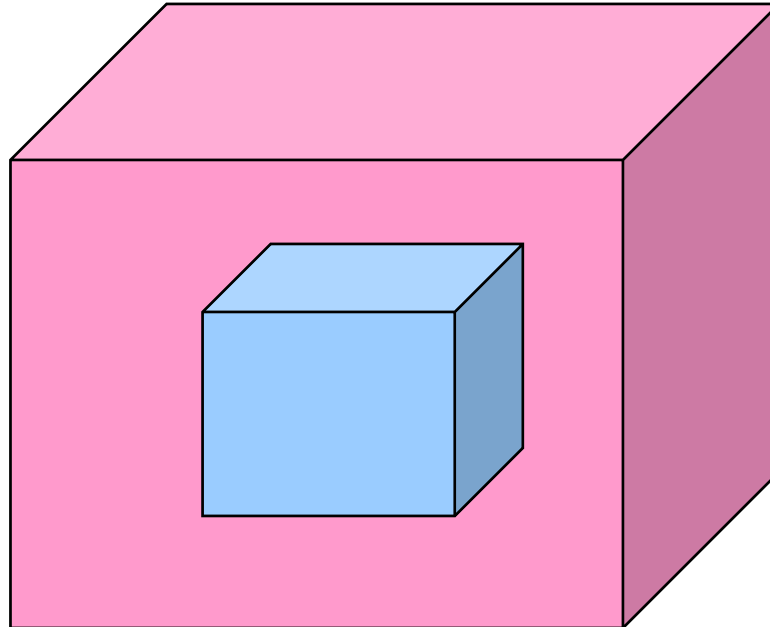
Creazione di un oggetto derivato



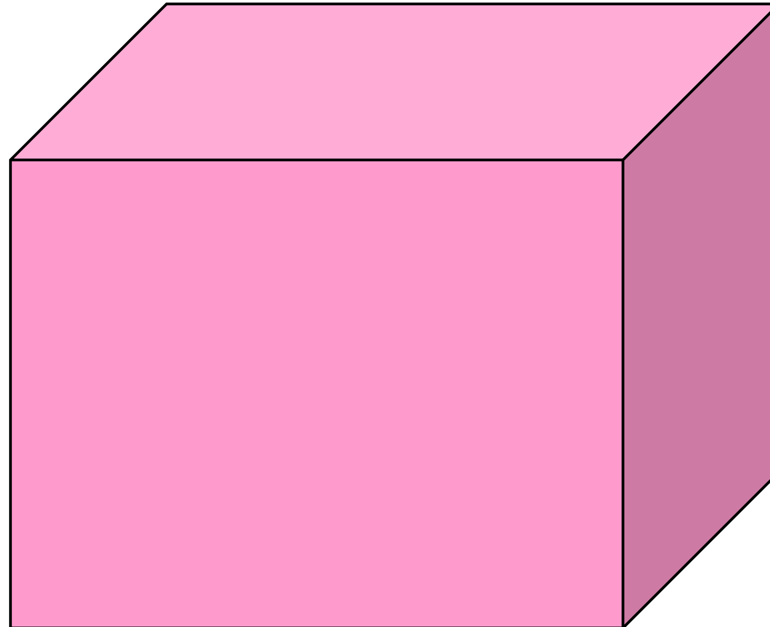
Creazione di un oggetto derivato



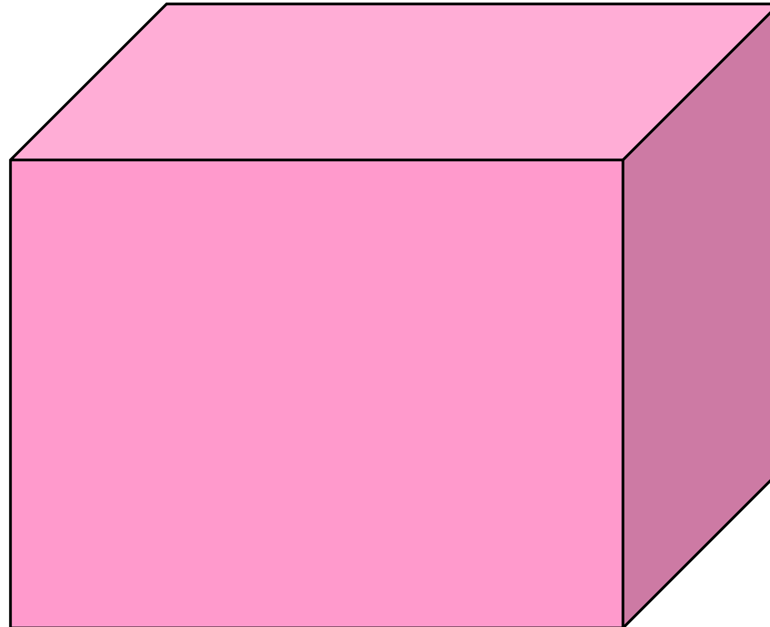
Cancellazione di un oggetto derivato



Cancellazione di un oggetto derivato



Cancellazione di un oggetto base



Cancellazione di un oggetto base

Conversione di tipo tra classi derivate e classi base

- Se la classe D eredita da B e abbiamo le seguenti dichiarazioni:

  D d;

  B b;

- Possiamo convertire un oggetto di tipo D in un oggetto di tipo B con l'istruzione

  b=d;

- Ma non possiamo fare il contrario

  d=b;   // NO!!!!!

Polimorfismo

- Abbiamo già visto che in C e C++ è possibile dare lo stesso nome a più funzioni (o metodi) che vengono distinte dal compilatore grazie alle differenze nella lista dei parametri:

  void stampa( );

  *void stampa(char * commento);*

- Ma quando una classe eredita da un'altra ha la possibilità di ridefinire uno o più metodi della classe base (con la stessa lista di parametri!): ad esempio il metodo *double funzione(double x)* nella classe eqdiff sostituisce il metodo *double funzione(double x)* della classe integrale (cf. cap. 16). Questa proprietà si chiama polimorfismo.

Polimorfismo e compilazione

Una funzione polimorfa può essere applicata ad oggetti appartenenti a classi diverse:

Cerchio.disegna( );

Sfera.disegna( );

3D.disegna( );

Talvolta non è possibile per il linker decidere quale metodo selezionare tra I vari metodi disegna a lui noti, perchè non gli è possibile capire la classe di appartenenza di ogni oggetto…

binding

- *Binding*: collegamento tra la chiamata ad una funzione de la sua definizione
- *Compile-time, static o early binding*: il collegamento viene effettuato dal linker
- *Run-time, dynamic o late binding*: il collegamento viene effettuato durante l'esecuzione del codice, al momento della chiamata
- Una *virtual function* è una particolare dichiarazione C++ che consente il *late binding*