



UNIVERSITÀ  
degli STUDI  
di CATANIA

DIPARTIMENTO DI  
MATEMATICA E INFORMATICA

# Classi Derivate in C++

Alessandro Ortis

[alessandro.ortis@unict.it](mailto:alessandro.ortis@unict.it)

[www.dmi.unict.it/ortis/](http://www.dmi.unict.it/ortis/)

# Ereditarietà

Non sempre occorre partire dal nulla nel costruire una classe, soprattutto se si dispone già di una classe che è simile a quella che si vuole costruire. In questo caso si può pensare di **estendere** la classe già esistente per adattarla alle nostre necessità.

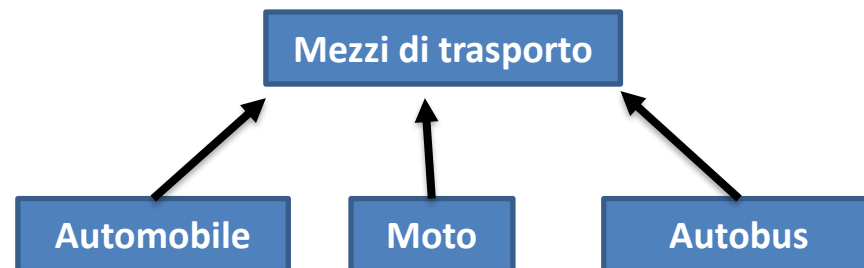
L'**ereditarietà** è lo strumento che permette di costruire nuove classi utilizzando quelle già sviluppate.

Quando una classe viene creata in questo modo, riceve tutti gli attributi ed i metodi della classe generatrice (li eredita). La classe generata sarà quindi costituita da tutti gli attributi e i metodi della classe generatrice più tutti quelli nuovi che saranno definiti.

# Ereditarietà

La classe che è stata derivata prende il nome di **sottoclasse**, mentre la classe generatrice si chiama **sopraclasse**.

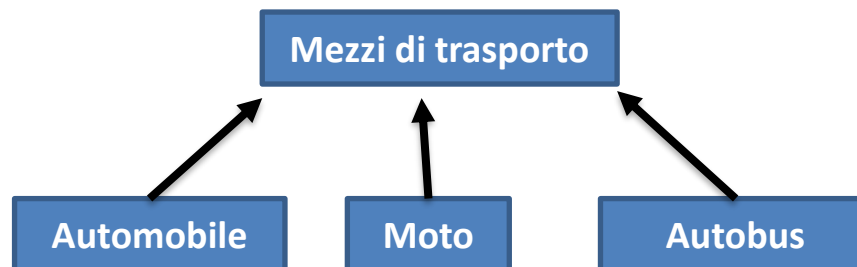
Queste relazioni individuano una gerarchia che si può descrivere usando un **grafo di gerarchia**.



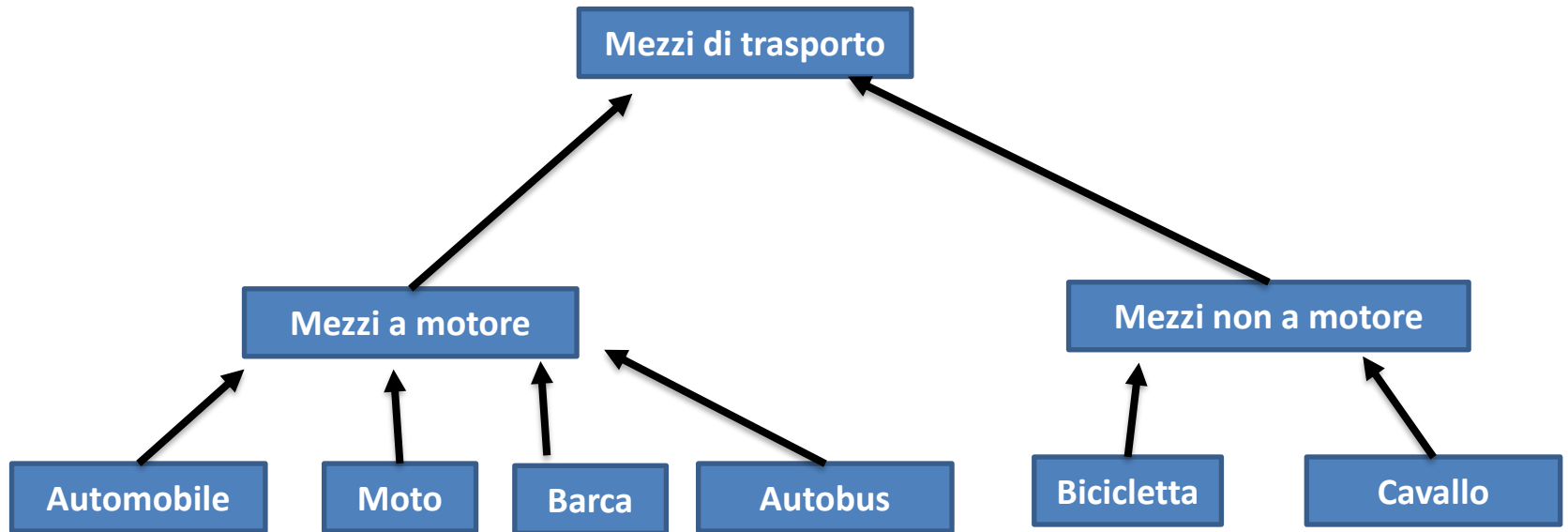
# Ereditarietà

La nuova classe si differenzia dalla sopraclasse in due modi:

- Per estensione: aggiungendo nuovi attributi e metodi
- Per ridefinizione: modificando i metodi ereditati, specificando una implementazione diversa di un metodo (override, overload)

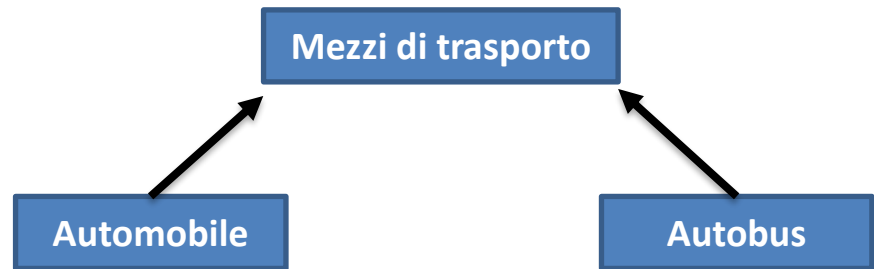


# Ereditarietà

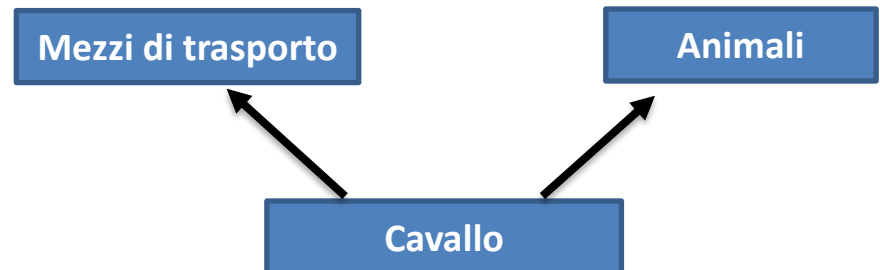


# Ereditarietà

## *Ereditarietà singola*



## *Ereditarietà multipla*



# Ereditarietà

Considerando una relazione di ereditarietà, le sottoclassi hanno la possibilità di ridefinire i metodi ereditati (mantenendo lo stesso nome) oppure lasciarli inalterati perché già soddisfacenti.

Il **polimorfismo** indica la possibilità per i metodi di assumere forme, cioè implementazioni, diverse all'interno della gerarchia delle classi.

Esempio: tutti i veicoli a motore possiedono il metodo "accelera". Le sottoclassi "automobile" e "moto" è probabile che lo ridefiniscano per adeguarlo alle particolari esigenze (es. pedale vs. manopola).

# Ereditarietà

Durante l'esecuzione del programma, un'istanza della classe "veicoli a motore" può rappresentare sia una "automobile" che una "moto".

Quando viene richiesta l'attivazione del metodo "accelera" è importante garantire che, tra tutte le implementazioni, venga scelta quella corretta.

Il **collegamento dinamico** è lo strumento utilizzato per la realizzazione del polimorfismo. È dinamico perché l'associazione tra l'oggetto e il metodo corretto da eseguire è effettuata a ***run-time***, cioè durante l'esecuzione del programma.



# Ereditarietà

Se una classe *A* eredita da una classe *B*, possiamo anche dire che una classe derivata *B* estende le funzionalità della classe base *A*.

Se una classe derivata non fornisce nessuna estensione allora probabilmente abbiamo un errore nel design della classe.

La dichiarazione deve includere il nome della classe base da cui deriva eventualmente, uno specificatore indicante il tipo di ereditarietà (`public`, `private` o `protected`) secondo la seguente sintassi:

```
class ClasseDerivata : specificatore ClasseBase {  
    membri;  
};
```

# Livelli di accesso

Si può dire che una classe ha due interfacce diverse per due categorie di classi:

- Ha una interfaccia `public` per fornire servizi a classi non collegate mediante ereditarietà.
- Ha una interfaccia `protected` per fornire servizi alle classi derivate.

Inoltre, una classe marcata come `final` (posto dopo il nome della classe) non può essere derivata.




# Livelli di accesso

- Una classe derivata in modo `public` eredita i membri pubblici e protetti della classe e ne mantiene tale livello di accesso;
- Una classe derivata in modo `protected` eredita i membri pubblici e protetti della classe e li espone con un livello di accesso `protected`;
- Una classe derivata in modo `private` eredita i membri pubblici e protetti della classe e li espone con un livello di accesso `private`, per cui non li rende fruibili al di fuori della classe stessa.

```
class Derivata :    [virtual][tipo_accesso] Base1,  
                    [virtual][tipo_accesso] Base2,  
                    [virtual][tipo_accesso] ...,  
                    [virtual][tipo_accesso] BaseN {  
  
    ...  
};
```

# Livelli di accesso

Se l'ereditarietà è `protected/private` i membri pubblici e protetti diventano `protected/private`. Se invece l'ereditarietà è pubblica, la classe derivata non può accedere solo ai membri privati di quella base. Una classe derivata non può mai accedere a membri privati della classe base, sebbene questi vengano comunque ereditati.

Tipo di ereditarietà	Accesso a membro classe base		Accesso a membro classe derivata
public	public protected private		public protected inaccessibile
protected	public protected private		protected protected inaccessibile
private	public protected private		private private inaccessibile

# Livelli di accesso

In generale, è bene fare sì che i dati membro della classe base siano privati ma con metodi pubblici per accedervi.

Se vogliamo che i membri di una classe siano visibili ai metodi di una classe derivata e basta dobbiamo dichiararli protetti.

In questo modo questi membri protetti rimarranno accessibili nella gerarchia, a meno che l'ereditarietà non sia privata.

# Livelli di accesso

```
class Base{  
public:  
    void meth1();  
protected:  
    void meth2();  
};  
  
class Derived: public Base{  
public:  
    void meth3() {  
        meth2();  
    }  
};  
  
class DerivedDerived: public Derived{  
public:  
    void meth4() {  
        meth2();  
    }  
};
```

```
int main() {  
  
    Derived d1;  
  
    d1.meth1();  
  
    d1.meth3();  
  
    d1.meth2();  
  
    DerivedDerived d2;  
  
    d2.meth4();  
  
}
```

***Trova l'errore***

# Livelli di accesso

```
class Base{
public:
    void meth1();
protected:
    void meth2();
};

class Derived: public Base{
public:
    void meth3(){
        meth2();
    }
};

class DerivedDerived: public Derived{
public:
    void meth4(){
        meth2();
    }
};
```

```
int main(){

    Derived d1;

    d1.meth1();


    d1.meth3();

    d1.meth2();

    DerivedDerived d2;

    d2.meth4();

}
```



*Base::meth2() is **protected**  
within this context*

# Costruttori e distruttori

Quando si crea un oggetto da una classe derivata i costruttori di ogni classe da cui si deriva sono invocati in sequenza, fino ad arrivare al costruttore della classe più derivata.

I costruttori di default sono invocati automaticamente, se presenti, in alternativa è il programmatore ad indicare quale costruttore deve essere invocato.

```
class B {  
    int x,y;  
    B();  
    B(int,int);  
};  
  
class D : public B{  
    int z;  
    D();  
    D(int,int,int);  
};
```

```
// Invocazione automatica di B()  
D::D() { z=0;}  
  
//Invocazione esplicita di B(int,int)  
D::D(int a, int b, int c) :  
    B(a,b), z(c) {}
```



# Costruttori e distruttori

Per quanto riguarda i distruttori avviene l'invocazione di tutti i distruttori delle classi all'interno della gerarchia ma in ordine inverso.

Quindi il distruttore della classe derivata è il primo, a seguire tutti i distruttori delle classi da cui si deriva.

I distruttori non possono essere sovraccaricati, quindi non c'è nessun problema nell'identificare quale metodo deve essere chiamato.

# Polimorfismo: override vs. overload

Ci sono diverse tipologie di polimorfismo in C++:

1. Overloading.
2. Subtyping.
3. Programmazione generica.

In genere quando si parla di polimorfismo nella OOP ci si riferisce al tipo (2).

Il concetto fondamentale è che ci si può riferire ad un'istanza di una classe derivata come se fosse l'istanza della sua super-classe (base), ma ***ogni oggetto risponde alle chiamate ai metodi come specificato dal suo vero tipo.***

# Principio di sostituzione di Liskov

Un'idea intuitiva di sottotipo è quella per cui i suoi oggetti (istanze) forniscono tutti i comportamenti degli oggetti di un altro tipo (il suo super-tipo) e qualche comportamento aggiuntivo.

La sostituibilità (LSP) è un principio della OOP che dice che:

*se  $S$  è un sotto-tipo di  $T$  allora gli oggetti di tipo  $T$  possono essere rimpiazzati da oggetti di tipo  $S$ , senza alterare nessuna proprietà desiderabile del programma (es. correttezza, compiti effettuati, etc.).*

Da questo deriva che è possibile usare un oggetto istanziato da una classe derivata ogni volta che è possibile usare un oggetto istanziato da una classe base.

# Principio di sostituzione di Liskov

```
class Persona {  
    ...  
};  
class Studente : public Persona{  
    ...  
};  
  
void cammina(Persona& p);  
void studia(Studente& s);
```

```
int main() {  
    Persona p;  
    Studente s;  
  
    cammina(p); // OK  
    cammina(s); // OK: uno studente è una persona  
    studia(s); //OK  
    studia(p); // NO!  
};
```

# Binding

L'associazione tra identificatori ed entità è chiamata "binding".

- ***Statico***: il collegamento avviene in fase di compilazione.
- ***Dinamico***: la connessione avviene durante l'esecuzione, fa sì che il codice da eseguire verrà determinato solo all'atto della chiamata; solo durante l'esecuzione del programma si determinerà il binding effettivo (tipicamente tramite il valore di un puntatore ad una classe base) tra le diverse possibilità (una per ogni classe derivata)

# Binding

Vantaggio principale del binding dinamico: offre un alto grado di flessibilità e praticità nella gestione delle gerarchie di classi

Svantaggio principale: è meno efficiente di quello statico

I linguaggi più strettamente OO offrono solo binding dinamico

In C++ il binding per default è quello statico

Per specificare il binding dinamico si fa precedere la dichiarazione della funzione dalla parola riservata `virtual`.

# Funzioni virtuali

`virtual` anteposto alla dichiarazione di una funzione indica al compilatore che essa può essere definita in una classe derivata.

```
class Figura {  
public:  
    virtual double calcolare_area() {};  
    virtual void disegnare() {};  
  
    // ...  
};
```

# Funzioni virtuali

Ogni classe derivata deve definire le sue proprie versioni delle funzioni dichiarate virtuali nella classe base: se le classi Cerchio e Rettangolo derivano dalla classe Figura, debbono entrambe definire le funzioni membro `calcolare_area()` e `disegnare()`.

```
class Cerchio : public Figura
{
    public:
        double calcolare_area();
        void disegnare();
    private:
        double xc, yc;           // coordinata del centro
        double raggio;           // raggio del cerchio
};

double Cerchio::calcolare_area(){
    return 3,1415 * raggio * raggio;
}

void Cerchio::disegnare() const{
    // ...
}
```



# Funzioni virtuali

**Binding Dinamico:** il compilatore C++ non può sapere l'implementazione specifica della funzione `calcolare_area()` che sarà chiamata a tempo d'esecuzione

```
Figura* figs[3]; // array di 3 puntatori a figure

Cerchio c;
Cerchio c2(c);
c2.setRaggio(8);
Rettangolo r;

figs[0] = &c;
figs[1] = &r;
figs[2] = &c2;

for(int i=0;i<3;i++)
    cout << figs[i]->calcolare_area() << endl;
```

# Funzioni virtuali

Linee guida da seguire quando si definisce una classe polimorfica (cioè che ha almeno un metodo virtuale):

- Almeno un metodo virtuale deve essere indicato come tale nella classe base.
- `virtual` si usa solo nella dichiarazione del metodo, non nella definizione.
- Una funzione top-level (una funzione non associata ad una classe) non può essere virtuale.
- Un metodo statico non può essere virtuale.
- `final` indica che un metodo non può essere riscritto nelle classi derivate.
- Il distruttore di una classe base dovrebbe sempre essere dichiarato virtuale.

# Funzioni virtuali

```
class B {  
  
public:  
    virtual void f1();  
};  
  
class D1: public B{  
void f1() final;  
};  
  
class D2: public D1{  
    //D2 non puo' fare  
    //override di f1()  
};
```

```
class B {  
public:  
    ~B(){};  
};  
  
class D1: public B{  
    // ...  
};  
  
int main(){  
  
    D1 d;  
  
    B* b = new B();  
    b = &d;  
  
    delete b;  
    // il distruttore di D1 non  
    // viene chiamato dal delete  
}
```

# Funzioni virtuali

```
class B {  
  
public:  
    virtual void f1();  
};  
  
class D1: public B{  
void f1() final;  
};  
  
class D2: public D1{  
    //D2 non puo' fare  
    //override di f1()  
};
```

*Se dichiaro ~B() virtuale verrà chiamato anche il distruttore di D quando elimino b con delete.*

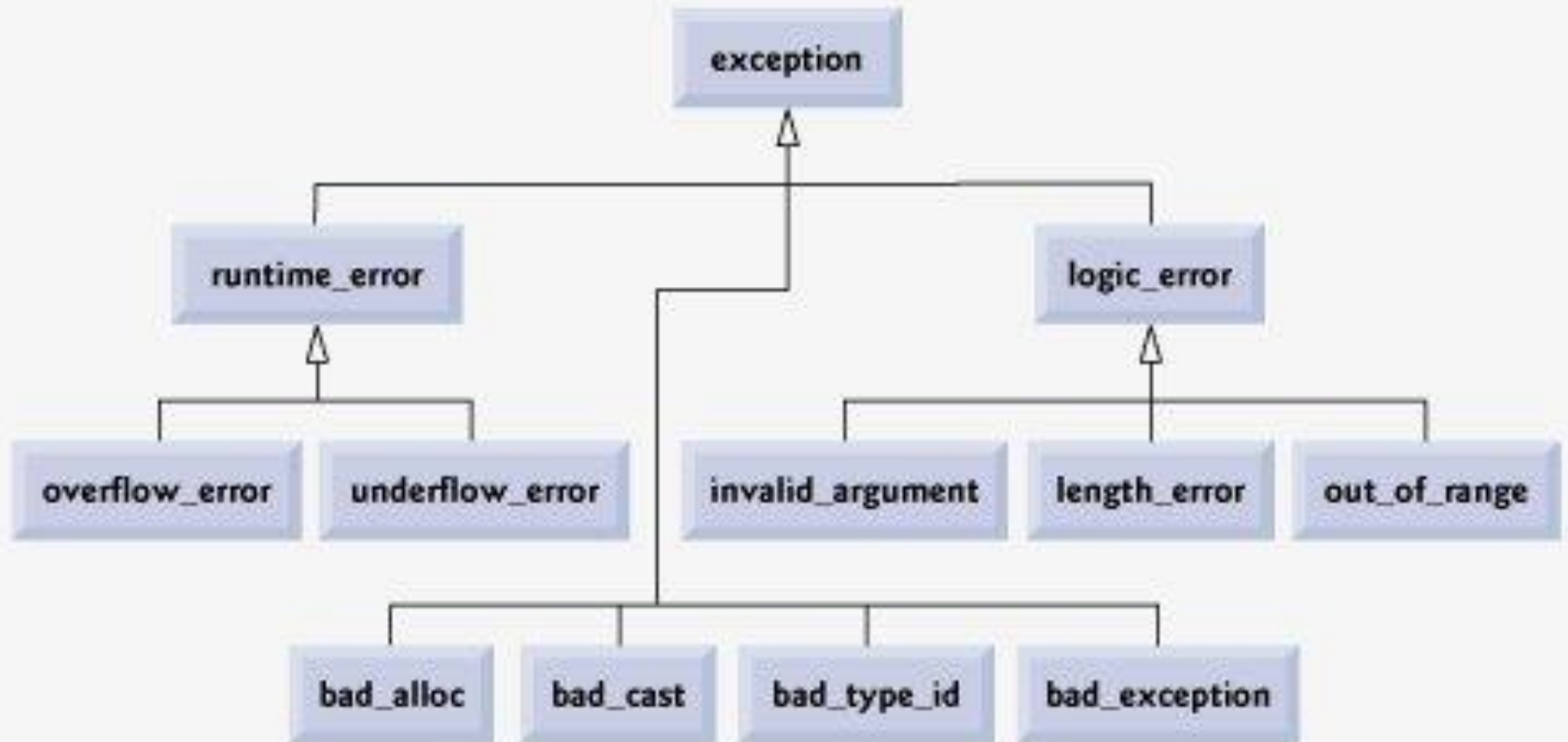
```
class B {  
public:  
    ~B(){};  
};  
  
class D1: public B{  
    // ...  
};  
  
int main(){  
  
    D1 d;  
  
    B* b = new B();  
    b = &d;  
  
    delete b;  
    // il distruttore di D1 non  
    // viene chiamato dal delete  
}
```

# Gestione delle eccezioni

Se mettiamo in cascata più clausole catch che tengono conto di una gerarchia di classi bisogna specificare prima quelle derivate, altrimenti verrà sempre catturata la “versione” base dell’eccezione/classe.

[illegible]

# Gestione delle eccezioni



# Gestione delle eccezioni

Ogni classe di eccezioni che deriva da `exception` contiene la funzione virtuale `what` che restituisce un oggetto "messaggio dell'eccezione".

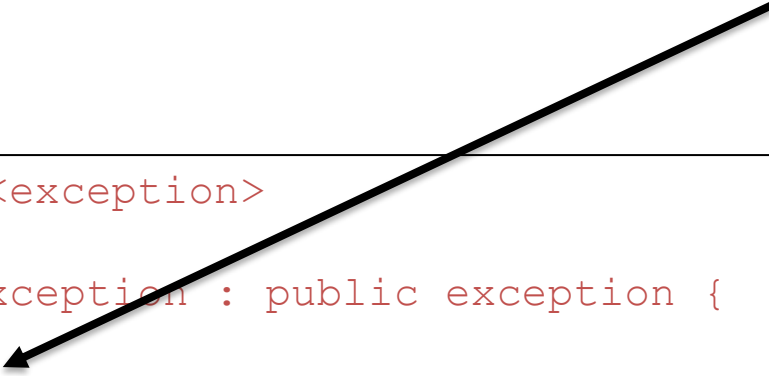
```
#include <exception>

class MyException : public exception {
public:
    virtual const char* what() const throw() {
        return "My exception";    }
};

int main() {
    try {
        throw MyException();
    } catch (exception& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0; }
```

# Gestione delle eccezioni

*Il metodo overridden è già virtuale, si potrebbe omettere*



```
#include <exception>

class MyException : public exception {
public:
    virtual const char* what() const throw() {
        return "My exception";    }
};

int main() {
    try {
        throw MyException();
    } catch (exception& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0; }
```



# Gestione delle eccezioni

*Funzione membro what() che restituisce un const char\**

```
#include <exception>

class MyException : public exception {
public:
    virtual const char* what() const throw() {
        return "My exception",    }
};

int main() {
    try {
        throw MyException();
    } catch (exception& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0; }
```

# Gestione delle eccezioni

*what() è const. Quindi può essere invocata da un puntatore const o una reference di un oggetto di questa classe o una derivata da essa.*


```
#include <exception>

class MyException : public exception {
public:
    virtual const char* what() const throw() {
        return "My exception";    }
};

int main() {
    try {
        throw MyException();
    } catch (exception& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0; }
```

# Gestione delle eccezioni

*Indica al compilatore che la funzione non genera eccezioni. Più recentemente si suggerisce di usare la clausola **noexcept***



```
#include <exception>

class MyException : public exception {
public:
    virtual const char* what() const noexcept {
        return "My exception";
    }
};

int main() {
    try {
        throw MyException();
    } catch (exception& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0; }
```

# Classi astratte

Una funzione *puramente virtuale* è definita nel seguente modo.

```
virtual void abstractFunction() = 0;
```

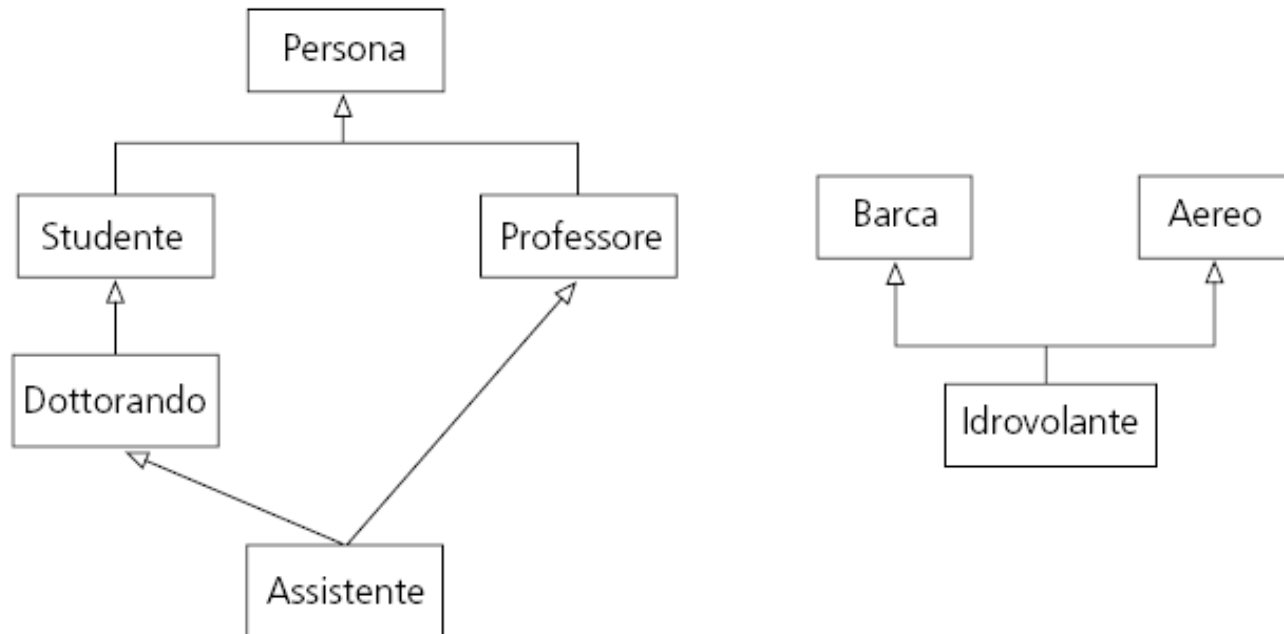
Una classe astratta è una classe che contiene almeno un metodo puramente virtuale. Come conseguenza, non possiamo istanziare oggetti di una classe astratta.

Queste infatti vengono usate come strumenti di specifica delle funzionalità richieste all'interno di una gerarchia di classi (cioè per definire interfacce).

Una classe derivata da una classe astratta deve necessariamente fare l'override di tutti i metodi puramente virtuali ereditati per poter essere istanziata.

# Ereditarietà multipla

Una classe può ereditare attributi e comportamento di più di una classe base.

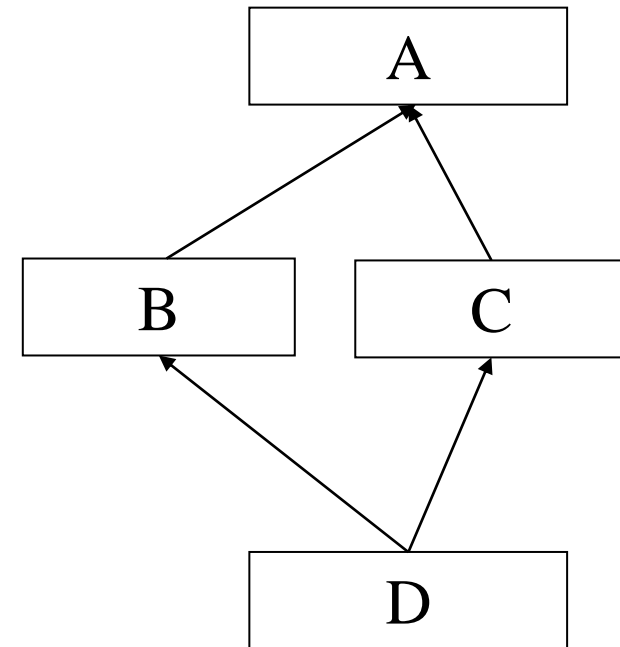


# Uso di virtual: dettagli

## Problema del diamante

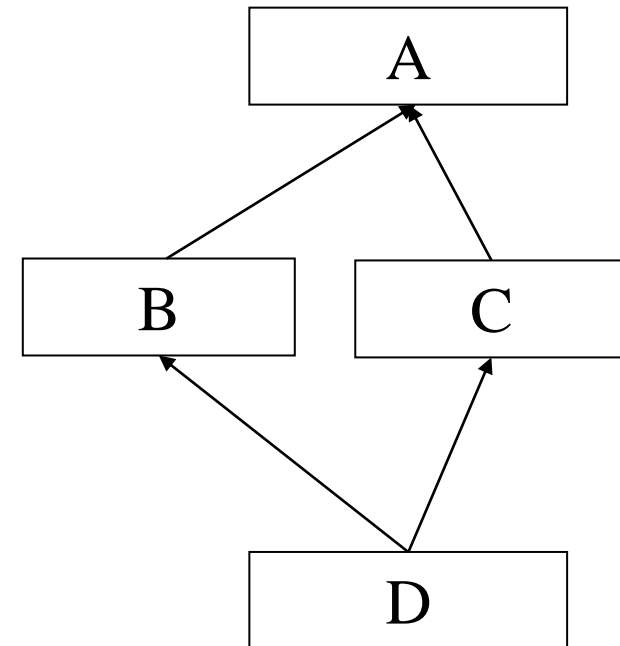
- B e C ereditano dalla classe A e la classe D eredita sia da B che da C
- se un metodo in D chiama un metodo definito in A, da quale classe viene ereditato?

Differenti linguaggi di programmazione hanno risolto quest'inconveniente in modi diversi.



# Uso di virtual: dettagli

- C++, per default, segue ogni percorso (di ereditarietà) separatamente, quindi D conterrà due (separati) oggetti di A
- **Soluzione:** se le ereditarietà da A a B (e da A ad C) sono “virtual”, C++ crea un solo oggetto A
- L'effetto della parola chiave virtual in una clausola di derivazione è quello di forzare il compilatore a includere la base virtuale una sola volta nella definizione degli oggetti derivati, anche se essa appare più volte nella catena di derivazione. In questo modo si ottimizza l'uso delle risorse, e si risolvono a monte eventuali conflitti di nomi.



# Uso di virtual: dettagli

```
class A{
public:
int x = 5;
};
class B: public virtual A{//...};
class C: public virtual A{//...};

class D: public B, public C{
    //...
};

int main(){
    D d;
    // richiesta non (piu') ambigua
    cout << d.x << endl;
}
```

