



UNIVERSITÀ
degli STUDI
di CATANIA

Ereditarietà

Corso di programmazione I AA 2021/22

Corso di Laurea Triennale in Informatica

Prof. Giovanni Maria Farinella

Web: <http://www.dmi.unict.it/farinella>

Email: gfarinella@dm.unict.it

Dipartimento di Matematica e Informatica

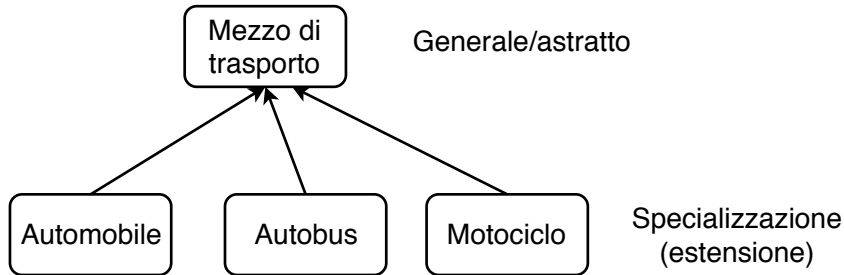
Introduzione all'ereditarietà

Ereditarietà è il meccanismo mediante il quale una classe (che si dice **sottoclasse**) acquisisce tutte le caratteristiche di un'altra classe (**superclasse** o classe "base").

Ereditarietà serve per modellare il software:

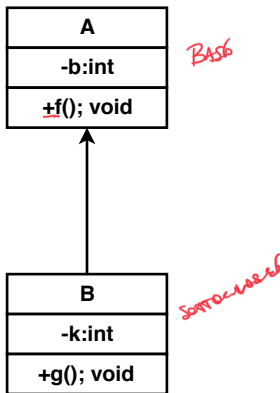
- definizione di entità di carattere generale o astratto, con alcune caratteristiche di base.
- dalle entità di carattere generale vengono definite altre entità **meno generali**, maggiormente definite, con maggiori particolari che le caratterizzano in una categoria meno ampia.

Introduzione all'ereditarietà



Introduzione all'ereditarietà

Diagramma UML.



Ereditarietà in C++

```
BASS
class A{
    private:
        int b; ←
    public:
        void f(); ←
}
ESERCIZIO
class B : <modificatore_accesso> A{
    private:
        int k;
    public:
        void g();
}
```

modificatore_accesso = {public, protected, private}.

Quale sarà la struttura della sottoclasse o classe derivata?

- Essa **eredita** TUTTE le caratteristiche della classe madre (o superclasse):
 - attributi;
 - metodi;
- può inoltre contenere nuovi metodi;
- può inoltre modificare i metodi ereditati
 - In tal caso il metodo ereditato viene **ridefinito** nella classe derivata;
 - tale processo si chiama **OVERRIDING**

Esempi svolti

29_01.cpp – Primo esempio di derivazione

29_02.cpp – Overriding

Overloading vs ereditarietà

Overriding

Overriding avviene con la **ridefinizione** di un **metodo** precedentemente definito in una **classe base**, in una classe derivata. Si ha overriding solo se i due metodi hanno la **stessa intestazione** (o prototipo).

Overloading

Overloading avviene con la **ridefinizione** di un **metodo** presente in uno scope (classe o namespace). con un metodo con **nome uguale ma segnatura differente** (numero di parametri formali e/o tipo)

Overloading vs ereditarietà

Overloading su scope differenti. Possibile ??

```
1 | class A {  
2 |     public:  
3 |         void f();  
4 |         void f(int);  
5 | };  
6 | class B: public A{  
7 |     public:  
8 |         void f(int, int);  
9 |         void f(double);  
10 | }  
11 | // ...  
12 | A a; B b;  
13 | b.f(); // Compile-time error!  
14 | b.f(10); // f(double), anche se argomento e' int...
```

Nello esempio precedente, **almeno una versione di $f()$ viene definita nella sottoclasse B**, di conseguenza:

- il compilatore, in assenza di direttive esplicite, cerca di risolvere invocazioni di $f()$ mediante **regole di risoluzione overloading** nello scope di B;

Overloading vs ereditarietà

Per invocare versione di `f()` definita in `A` adottare una della segg. soluzioni:

A. **usare operatore risoluzione di scope**: `(b.A::f());` 

- Risoluzione della versione corretta del metodo `f()` a carico del programmatore (`b.A::f(10.5)` vs `b.f(5)`);

B. (oppure) **includere direttiva using** in apposita sezione: `(using A::f);`

- Risoluzione della versione corretta del metodo `f()` automatica (`b.f(10.5)` oppure `b.f(5)`);

Esempi svolti

29_03.cpp – Funzioni overloaded in classe base, invocazione nella sottoclasse

29_04.cpp – Funzioni overloaded in classe base e sottoclasse, problemi di risoluzione (overriding vs overloading)

29_05.cpp – Funzioni overloaded in classe base e sottoclasse, risoluzione esplicita con operatore di risoluzione di scope

29_06.cpp – Funzioni overloaded in classe base e sottoclasse, risoluzione automatica mediante direttiva using

Filosofia alla base della ereditarietà

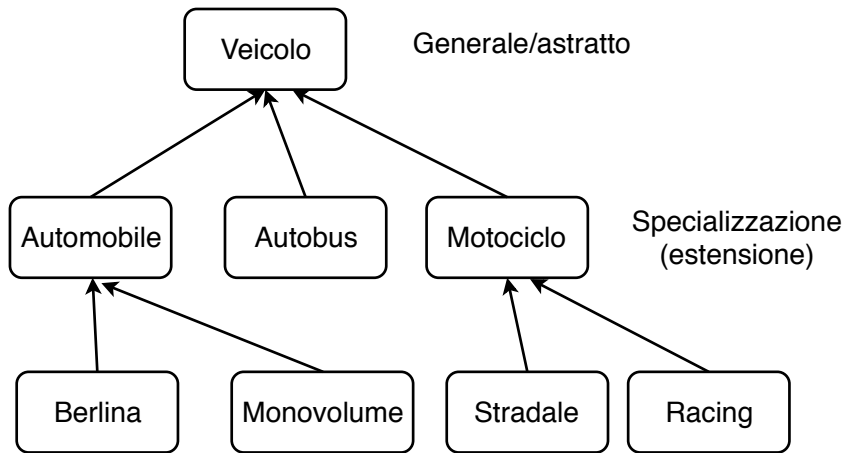
```
class Quadrilatero { //...  
    double perimetro(){return l1+l2+l3+l4;}  
    //... };  
class Trapezio: public Quadrilatero { //...  
    //non ridefinisce perimetro()  
    //... };  
class Parallelogrammo: public Trapezio{ //...  
    double perimetro(){return 2*l1+2*l2;}  
    //... };  
class Rettangolo: public Parallelogrammo { //...  
    //non ridefinisce perimetro()  
    //... };  
class Quadrato: public Rettangolo{ //...  
    double perimetro(){return 4*l1;} //l1==l2  
    //... };
```



Filosofia alla base della ereditarietà

In una gerarchia ereditaria:

- ogni classe può essere **base di una o più sottoclassi**, ma anche **derivata di un'altra classe**;
- ogni classe “offre” le sue **funzionalità** a **tutte le eventuali sottoclassi** (funzionalità di base);
- ogni **sottoclasse estende** le funzionalità ereditate dalla superclasse:
 - per **aggiungere ulteriori funzionalità** che caratterizzano l'entità che la classe stessa rappresenta;
 - per svolgere le stesse funzionalità in modo più **efficiente**;



Ereditarietà rappresenta un terzo caso di relazione tra classi. Le tre relazioni sono:

- **Ereditarietà:** una automobile è un veicolo.
(regola “..is a..” o ISA);
- **Composizione:** una automobile ha un motore
(regola “..has a..”);
- **Aggregazione:** una automobile ha un guidatore
(regola “..has a..”);

Gerarchie ereditarie ben strutturate

In ogni gerarchia ereditaria che sia strutturata in modo corretto, per ogni classe deve valere la regola is-a, ovvero ogni classe derivata e' un tipo particolare di quella rappresentata dalla sua classe base.

Un membro **protected** di una classe sarà **visibile**:

1. alle funzioni membro della **classe stessa**;
2. alle funzioni membro della **classe derivata**;

Quindi i membri dichiarati in sezione **protected** **non saranno accessibili** dalle funzioni appartenenti a scope differenti dalla classe base e della classe derivata.

Accesso ai membri della classe base

```
1  class A{  
2      protected:  
3          void f();  
4          int a;  
5  }  
6  
7  class B: A{  
8      void g(){  
9          a*=2; // accesso ad a, OK  
10         f(); // accesso ad f(), OK  
11     }  
12 }
```

Ogni **membro private** di una classe sarà visibile **SOLO** alle **funzioni membro della classe stessa**.

I membri di una classe derivata:

- **non possono accedere direttamente** ai membri con accesso **private** della classe base;
- **possono accedere indirettamente** ai membri **private** della classe base **mediante invocazione di funzioni membro (public/protected)** **ereditate dalla classe base**;

Accesso ai membri della classe base

```
1  class A{
2      void p(); float x; // accesso private
3      public:
4          void h() { p(); }
5      protected:
6          void f(){ p(); }
7  };
8  class B: A{
9      void g(){
10         x*=2; // Compile-time error!
11         p(); // Compile-time error!
12         f(); // OK, accesso a p() mediante f();
13         h(); // OK, accesso a p() mediante h();
14     } };
```

Specifiche di accesso alla classe base

```
1  class <class_name> : [ ACCESS ] {  
2      // ...  
3  }
```

ACCESS può essere

1. public
2. protected
3. private.

ACCESS modifica la **visibilità dei membri public e protected** della classe base, **ereditati dalla sottoclasse**.

Specifiche di accesso alla classe base

Derivazione con **accesso public**: l'accesso ai membri ereditati rimane **invariato nella classe derivata**.

```
1  class A{
2      int a; void f(); // private
3      protected:
4      float x; void g();
5      public:
6      double y; void h(); }; //end class A
7  class B: public A{ /* ... */ }; //and class B
8  //...
9  B b;
10 b.h(); // OK, h() public in B..
11 b.g(); // Compile-time error! g() protected in B..
```

Specifiche di accesso alla classe base

Derivazione con **accesso public** vs interfaccia della classe base.

```
1  class A{  
2      //...  
3  }  
4  class B: public A{ /* ... */ };
```

Accesso all'interfaccia di A (membri public) **non viene in alcun modo ristretto** con istanze di B:

- membri dichiarati in sezione **public in A** ancora **accessibili tramite istanze di B** (fuori dallo scope di A e B);
- membri dichiarati in sezione **protected in A** accessibili da:
 - **membri e friend di B;**
 - **membri e friend di classi derivate da B;**

Specifiche di accesso alla classe base

Derivazione con **accesso protected**: l'accesso ai membri ereditati ad accesso protected e public in A **diviene protected nella classe derivata** in B.

```
1  class A{
2      int a; void f(); // private
3      protected:
4          float x; void g();
5      public:
6          double y; void h(); }; //end class A
7  class B: protected A{ /* ... */ }; //end class B
8  //...
9  B b;
10 b.h(); // Compile-time error! Adesso h() protected in B..
11 b.g(); // Compile-time error! g() protected in B..
```

Specifiche di accesso alla classe base

Derivazione con **accesso protected** vs interfaccia della classe base.

```
1  class A{  
2      //...  
3  }  
4  class B: protected A{ /* ... */ };
```

Accesso all'interfaccia di A (membri public) soggetto a **restrizioni nello scope di B**:

- membri dichiarati in sezione **public e protected in A** **accessibili** da:
 - **membri e friend di B**;
 - **membri e friend di classi derivate da B**;

Specifiche di accesso alla classe base

Derivazione con **accesso private**: l'accesso ai membri ereditati ad accesso `protected` e `public` in A **diviene private nella classe derivata** in B.

```
1  class A{
2      int a; void f(); // private
3      protected:
4          float x; void g();
5      public:
6          double y; void h(); }; //end class A
7  class B: private A{
8      //...
9      void foo(){
10 }
```

Specifiche di accesso alla classe base

Derivazione con **accesso private** vs interfaccia della classe base.

```
1  class A{  
2      // ...  
3  }  
4  class B: private A{ /* ... */ };
```

Interfaccia di A (membri public) soggetto a **restrizioni nello scope di B e non accessibile nello scope delle classi derivate:**

- membri dichiarati in sezione **public e protected in A accessibili** da:
 - **membri e friend di B;**
 - ~~membri e friend di classi derivate da B;~~

Specifiche di accesso alla classe base

RIASSUMENDO:

```
class B : <SPECIFICATORE_ACCESSO> A { ... }
```

Specificatore di accesso ai Membri nella classe base

SPECIFICATORE_ACCESSO alla classe base	Membro	
	Public in A	Protected in A
→ class B: public A	public in B	protected in B
class B: protected A	protected in B	protected in B
class B: private A	private in B	private in B

Esempi svolti

29_07.cpp – esempio di derivazione public

29_08.cpp – esempio di derivazione protected

29_09.cpp – esempio di derivazione private

Specifiche di accesso alla classe base

Ripristinare lo specificatore di accesso di uno o più membri.

```
1 → class A{  
2     in x;  
3     public:  
4     → getX();  
5     setX(int v);  
6 };  
7 class B: private A{  
8     public:  
9     using A::getX;  
10 }  
11 //...  
12 B b;  
13 → b.getX(); //OK
```

Esempi svolti

29_10.cpp – using per ripristinare visibilità di un membro;

Alla creazione di un **oggetto a partire da una classe derivata** (bottom up), invocazione:

1. (eventuale) costruttore classe base
2. (eventuale) costruttore classe derivata

Alla **distruzione di un oggetto di una classe derivata** (top down), invocazione:

1. (eventuale) distruttore **classe derivata;**
2. (eventuale) distruttore **classe base;**

Costruzione e distruzione di oggetti vs Ereditarietà

Costruttore classe derivata:

- **non può inizializzare** direttamente gli **attributi della classe base**;
- **non può invocare esplicitamente un costruttore** della classe base;

Allora come **passare i parametri al costruttore della classe base**?

Risposta: **Lista di inizializzazione**. Essa permette di:

- selezionare la **versione** desiderata del **costruttore**;
- **specificare** gli argomenti (**parametri attuali**) per il costruttore selezionato;

Esempi svolti

29_11.cpp – Costruzione e distruzione di oggetti vs ereditarietà.

Limitazioni.

Non è possibile ereditare:

- costruttori e distruttori;
- operatore di assegnamento “=”;
- le funzioni `friend`;

Composizione vs ereditarietà.

Un oggetto **composto** o contenitore contiene uno o più oggetti della classe “contenuto”. Controlla la creazione e la distruzione.

Un oggetto creato da una classe derivata, formalmente **non contiene alcun oggetto della classe base**.

Composizione vs ereditarietà (cont.)

Un oggetto composto **può accedere ai soli membri** `public` degli oggetti contenuti. Tuttavia:

- esso può avere **opportuni metodi** `friend` delle classi degli oggetti contenuti..

Un oggetto costruito a partire da una classe derivata **può accedere anche ai membri** `protected`

FINE