Programmazione a oggetti

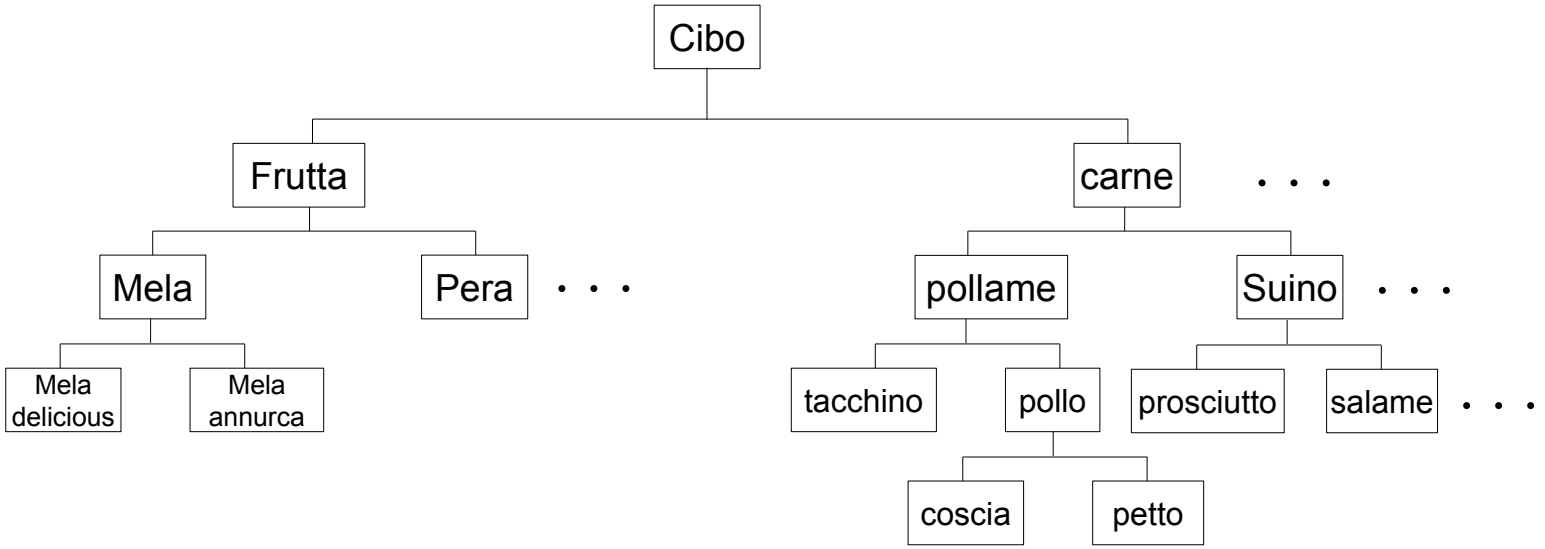
Ereditarietà

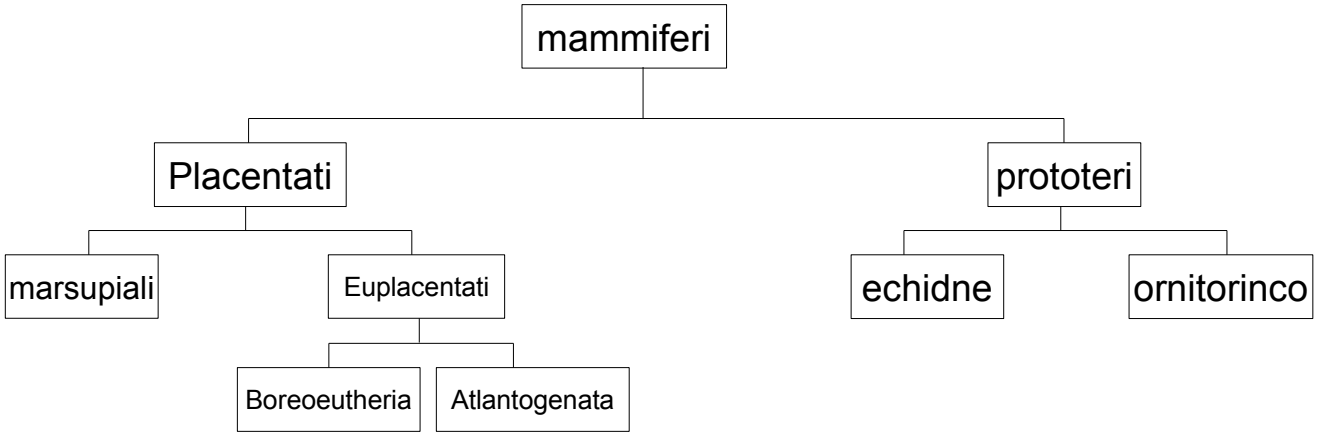
A.A. 2020/2021
Francesco Fontanella

Ereditarietà

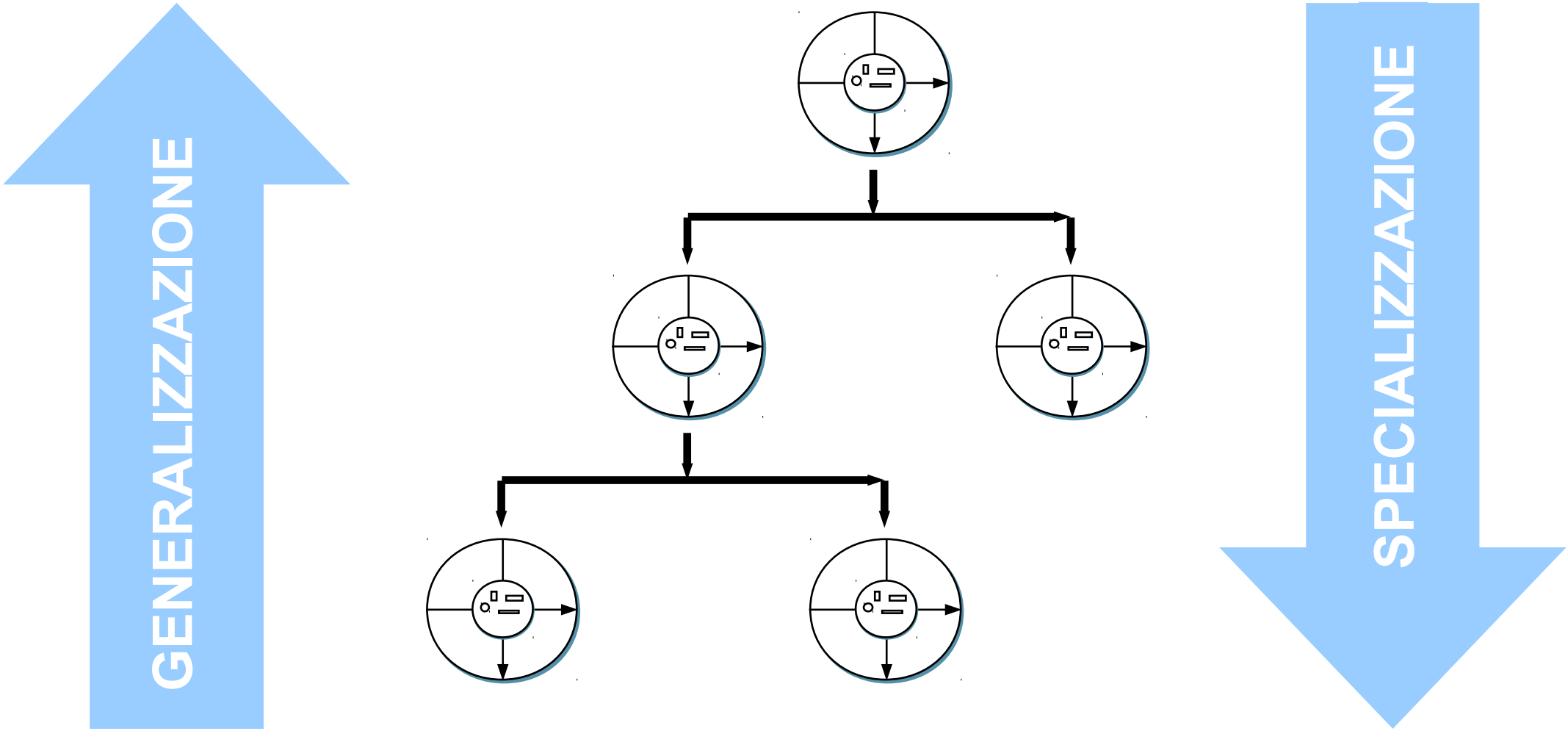
- Meccanismo grazie al quale un oggetto acquisisce le proprietà di un altro oggetto.
- Consente di realizzare relazioni tra classi di tipo generalizzazione-specializzazione
- Una classe, detta **base**, realizza un comportamento generale, comune ad un insieme di entità, mentre le classi cosiddette **derivate** (sottoclassi) realizzano comportamenti specializzati rispetto a quelli della classe base
- In pratica, il meccanismo dell'ereditarietà è utilizzato per modellare le eventuali tassonomie presenti nel sistema in esame
- Il grande vantaggio dell'uso dell'ereditarietà è legato al **riuso del software**.

Esempi


Cibo
Frutta
carne
. . .
Mela
Pera
. . .
pollame
Suino
. . .
Mela delicious
Mela annurca
tacchino
pollo
prosciutto
salame
. . .
coscia
petto

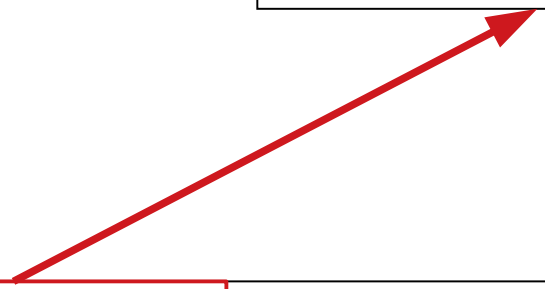

mammiferi
Placentati
prototeri
marsupiali
Euplacentati
echidne
ornitorinco
Boreoeutheria
Atlantogenata

Generalizzazione vs Specializzazione

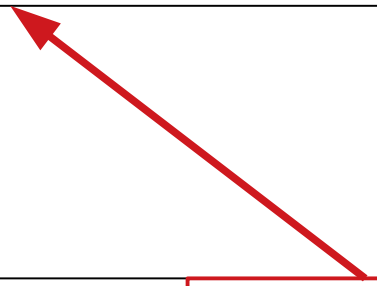


Esempio

```
class Building {
    int rooms;
    int floors;
    float area;
  public:
    void setRooms(int r) {rooms =r;}
    void setFloors(int f) {floors =f;}
    void seArea(float a) {area =a;}
    int  getRooms(){return rooms;}
    int  getFloors(){return floors;}
    int  getArea(){return area;}
};
```



```
class House : public Building{
    int bedrooms;
    int baths;
  public:
    void set_bedrooms(int r) {bedrooms =r;};
    void set_baths(int b) {baths =b;};
    int  get_bedrooms(){return bedrooms;};
    int  get_baths(){return baths;};
};
```



```
class School : public Building{
    int classrooms;
    int offices;
  public:
    void set_classrooms(int c) {classrooms =c;};
    void set_offices(int o) {offices =o;};
    int  get_classrooms(){return classrooms;};
    int  get_offices(){return offices;};
};
```


```
#include house.h
#include school.h


main()
{
  House h;
  School s;

  h.set_rooms(12);
  h.set_floors(3);
  h.set_area(300);
  h.set_bedrooms(5);
  h.set_baths(3);

  s.set_rooms(46);
  s.set_floors(2);
  s.set_area(900);
  s.set_offices(7);
  s.set_baths(3);
     .
     .
}
```

La classe Persona

```
class Persona{
    string nome, cognome;
  public:
    Persona(string n="", string c="");
    void set_nome (string str){nome = str;}
    void set_cognome (string str){cognome = str;}
    string get_nome() {return nome;}
    string get_cognome() {return cognome;}

};
```

La classe studente

```
class Studente{
   string nome, cognome;
   int matr;
  public:
   Studente(string n="", string c="", int m=0);
   void set_nome (string str){nome = str;}
   void set_cognome (string str){cognome = str;}
   void set_matr (int m){matr = m;}
   string get_nome() {return nome;}
   string get_cognome() {return cognome;}
   int get_matr() {return matr;}
   void show();
```

Appartengono anche a persona

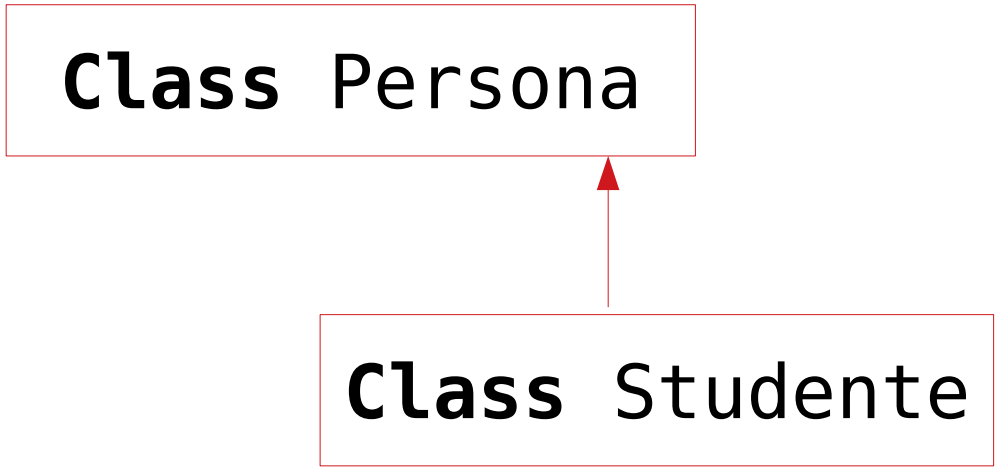
La nuova classe studente

```
class Studente: public Persona{
    int matr;
  public:
    Studente(string n="", string c="", int m=0);
    void set_matr (int m){matr = m;}
    int get_matr() {return matr;}
    void show();
};
```

Osservazioni

- L'ereditarieta classica descrive una relazione del tipo:

è un (*is-a*)



- Il diagramma può essere letto così:

"uno studente è una persona"

Lo specificatore protected

- È un ulteriore specificatore di accesso
- I membri dichiarati **protected** sono accessibili SOLO alle classi derivate
- **Esempio**

```
class Base {
  protected:
    int i,j; // privato di base, ma accessibile da derived
  public:
    void set(int a, int b) {i=a; j=b;}
    void show() {cout<<i<<" "<<j;};
};

class Derived : public Base {
    int k;
  public:

    void setk() {k = i*j;}
    void showk() {cout<<k;}

};
```

può accedere a i e j perché sono dichiarate come **protected** nella classe Base

Specificatori di accesso

- L'ereditarietà viene dichiarata usando la sintassi:

```
class classe_derivata : accesso  classe_base
```

- "accesso" specifica il tipo di accesso delle funzioni della classe derivata ai membri della classe base. Esistono tre tipi di accesso:
  - **Public**;
  - **Private**;
  - **Protected**

Specificatore public

- I membri pubblici e protetti della classe base sono tali anche nella classe derivata.
- I membri privati della classe base restano tali, e non sono accessibili nemmeno dalle funzioni membro delle classi derivate.
- **Esempio**

```
class Base {
    int i,j;
  public:
    void set(int a, int b) {i=a; j=b};
    void show() {cout<<i<<" "<<j;};
};

class Derived : public Base {
    int k;
  public:
    Derived (int a) {k=a;}
    void showk() {cout<<k;}
    void setk() {k = i;} // ERRORE!: i è privato!

};
```

```
#include derived.h

int main()
{
  Derived d(3);

  // Accesso a membri della classe base
  d.set(1,2);
  d.show()

  // uso di un membro della classe derivata
  d.showk();

  return 0;
}
```

Specificatore private

- I membri pubblici e protetti della classe base sono tali anche nella classe derivata.
- I membri privati continuano a essere privati, per tutti
- **Esempio**

```
class Base {
    int i,j;
  public:
    void set(int a, int b) {i=a; j=b;}
    void show() {cout<<i<<" "<<j;};
};

class Derived : private Base {
    int k;
  public:
    Derived (int a) {k=a;}
    void showk() {cout<<k;}

};
```

```
#include derived.h

main()
{
  Derived d(3);

  d.set(1,2); // ERRORE!
  d.show();   // ERRORE!


};
```

Specificatore protected: ereditarietà multipla

- Una classe derivata può a sua volta essere usata come classe base per un ulteriore derivazione.
- In questo caso, i membri protected della classe base, se ereditata come public dalla prima classe derivata, potranno essere ereditati nuovamente come protected dalla seconda classe derivata.
- È anche possibile, ereditare una classe base come protected:
  - i membri public e protected della classe base diventano membri protected della classe derivata:
    - Sono accessbili solo dalle classe derivate ma non dall'esterno

```
class Base {
  protected:
    int i,j;
  public:
    void set(int a, int b) {i=a; j=b;}
    void show() {cout<<i<<" "<<j;}
};
class Derived1 : public Base {
    int k;
  public:
    void setk() {k = i*j;}
    void showk() {cout<<k;}
};
class Derived2 : public Derived1 {
    int m;
  public:
    void setm() {m = i-j;}
    void showm() {cout<<m;}
};
```

```
#include derived.h

int main()
{
  Derived2 d1, d2;

  d1.set(2,3);    // membro di Base
  d1.setk();      // membro di Derived1

  d2.set(3, 4);   // membro di Base
  d2.setk();      // membro di Derived1
  d1.show();      // membro di Base

  d1.showk();     // membro di Derived1
  d2.setm();      // membro di Derived2
  d1.showm();     // membro di Derived2
};
```


```
class Base {
  protected:
    int i,j;
  public:
    void set(int a, int b) {i=a; j=b;}
    void show() {cout<<i<<""<<j;}
};

class Derived1 : private Base {
    int k;
  public:
    void setk() {k = i*j;} // OK!
    void showk() {cout<<k;}
};

class Derived2 : public Derived1 {
    int m;
  public:
    void setm() {m = i-j;}// ERRORE!
    void showm() {cout<<m;}
};
```

```
#include derived.h

main()
{
  Derived1 d1;
  Derived2 d2;

  d1.set(2,3);   // ERRORE!
  d1.setk();
  d1.show();     // ERRORE!

  d2.set(3, 4); // ERRORE!
  d2.setk();
  d2.show();    // ERRORE!
  d2.showk();
  d2.showm();
};
```


```
class Base {
  protected:
    int i,j;
  public:
    void set(int a, int b) {i=a; j=b};
    void show() {cout<<i<<" "<<j;};
};

class Derived1 : protected Base {
    int k;
  public:
    void setk() {k = i*j;};
    void showk() {cout<<k;};
};

class Derived2 : public Derived1 {
    int m;
  public:
    void setm() {m = i-j;};   // OK!
    void showm() {cout<<m;};
};
```

```
#include derived.h

int main()
{
  Derived1 d1;
  Derived2 d2;

  d1.set(2,3);   // ERRORE!
  d1.setk();
  d1.show();     // ERRORE!

  d2.set(3, 4); // ERRORE!
  d2.setk();
  d2.show();    // ERRORE!
  d2.showk();
  d2.showm();
};
```

La keyword using

- Se una classe è ereditata come **`private`**, tutti i suoi membri **`public`** e **`protected`** divengono private nella classe derivata
- È possibile però, modificare in maniera selettiva, le specifiche originali di uno o più membri ereditati dalla classe base.
- Questa modifica può essere fatta per mezzo della keyword **`using`**

```
class Base {
  protected:
    int i,j;
  public:
    void set(int a, int b) {i=a; j=b;}
    void show() {cout<<i<<" "<<j;}
};

class Derived : private Base {
    int k;
  public:
    using Base::show;
    void setk() {k = i*j;}
    void showk() {cout<<k;}
};
```

```
#include derived.h

int main()
{
  Derived d1;
        .
        .

  d1.set(2,3);   // ERRORE!
  d1.setk();
  d1.show();     // OK!

        .
        .
};
```

Overriding di metodi

- In una gerarchia di classi, è possibile definire metodi con lo stesso nome
- È poi il compilatore ad individuare le funzioni giuste, tramite i tipi
- **Esempio**

```
class Base {
  protected:
    int i;
  public:
    void show() {cout<<"show di Base: "<<endl;}
};

class Derived1 : public Base {
    int j;
  public:
    void show() {cout<<"show di Derived1: "<<endl;}
};

class Derived2 : public Derived1 {
    int k;
  public:
    void show() {cout<<"show di Derived2: "<<endl;}
};
```

```
int main() {
  Base b;
  Derived1 d1;
  Derived2 d2;

  b.show();
  d1.show();
  d2.show();

  return 0;
}
```

OUTPUT

```
show di Base
show di Derived1
show di Derived2
```

Ereditarietà e puntatori

- In generale, un puntatore ad una variabile di un determinato tipo non può puntare ad una variabile di un altro tipo
- Questa regola non vale per le classi derivate: un puntatore della classe base può puntare ad un oggetto di una qualsiasi classe derivata da quella base
- Con i puntatori della classe base però, è possibile accedere SOLO ai membri della relativa classe

Esempio

```
int main() {
  Base b, *bp;
  Derived1 d1, *dp1;
  Derived2 d2;


  // Punto agli oggetti con bp...
  bp = &b;
  bp->show();

  bp = &d1;
  bp->show();

  bp = &d2;
  bp->show();
  cout<<endl;
  // Punto agli oggetti con dp1...
  dp1 = &d1;
  dp1->show();

  dp1 = &d2;
  dp1->show();

  return 0
}
```

OUTPUT

```
show di Base
show di Base
show di Base

show di Derived1
show di Derived1
```

Puntatori a tipi derivati

```
class Base {
  protected:
    int i;
  public:
    void set_i(int a) {i = a;}
};

class Derived1 : public Base {
  protected:
    int j;
  public:
    void set_j(int a) {j = a;}
};

class Derived2 : public Derived1 {
    int k;
  public:
    void set_k(int a) {k = a;}
};
```

```
int main() {
  Base b, *bp;
  Derived1 d1;
  Derived2 d2;

  bp = &b;
  bp->set_i(0);


  bp = &d1;
  bp->set_i(0); // OK!
  bp->set_j(1); // ERRORE!


  bp = &d2;
  bp->set_i(0); // OK!
  bp->set_j(1); // ERRORE!
  bp->set_k(1); // ERRORE!

  return 0;
}
```

- L'assegnazione di un indirizzo di un oggetto di una classe derivata ad un puntatore della classe base può essere fatto SOLO se la classe Base è stata ereditata come **public**, in caso contrario l'assegnazione è vietata

- **Esempio**

```
class Base {
  protected:
    int i;
  public:
    void set_i(int a) {i=a;}
    int get_i() {return i;};
};

class Derived : private Base {
    int j;
  public:
    void set_j(int a) {i=a;}
    int get_j() {return i;};
};
```

```
#include derived.h

int main()
{

  Base *bp;
  Derived d;

  bp = &d;     // ERRORE!

};
```

Object slicing

- È anche possibile inizializzare un oggetto della classe base con uno di una classe derivata:
  - solo la parte base dell'oggetto della della classe derivata viene assegnato. Si parla di **object slicing**.

- L'operazione inversa è invece illegale: non è possibile assegnare ad un oggetto derivato un oggetto della classe Base

Esempio

```
int main() {
  Base b, *bp;
  Derived1 d1, *dp1;
  Derived2 d2, *dp2;

  // b e pb
  b = d1;      // OK
  b = d2;      // OK

  bp = dp1;    // OK
  bp = dp2;    // OK
  bp = &b;     // OK
  bp = &d1;    // OK
  bp = &d2;    // OK

  // d1 e dp1
  d1  = d2;  // OK
  dp1 = dp2; // OK
  dp1 = &d1;  // OK
  dp1 = &d2; // OK
  return 0;
}
```

```
int main() {
  Base b, *bp;
  Derived1 d1, *dp1;
  Derived2 d2, *dp2;


  d1  = b;      // ERRRORE!
  dp1 = bp;      // ERRRORE!
  dp1 = &b;     // ERRRORE!

  // d2 e dp2
  dp2 = &d2;    // OK
  d2  = b;      // ERRRORE!
  d2  = d1;     // ERRRORE!
  dp2 = bp;     // ERRRORE!
  dp2 = dp1;    // ERRRORE!
  dp2 = &b;     // ERRRORE!
  dp2 = d1;    // ERRRORE!

  return 0;
}
```

Accesso mediante casting

- Con un puntatore alla classe Base è possibile accedere alle funzioni della classe derivata mediante casting:
- **Esempio**

```
int main()
{
  Base *bp;
  Derived d;

  bp = &d;
       .
       .
  ((derived *)bp)->set_j(1); // OK
  cout<<endl<<((derived *)bp)->get_j(); // OK

  return 0;
}
```

Aritmetica dei puntatori

- Con l'ereditarietà bisogna fare attenzione all'incremento dei puntatori.
- **Esempio**

```
#include derived.h

int main()
{
  Base *bp;
  Derived d[2];

  bp = d;
  d[0].set_i(1);
  d[1].set_i(2);

  cout<<endl<<bp->get_i();
  bp++;

  cout<<endl<<bp->get_i();

  return 0;
}
```

```
class Base {
  protected:
    int i;
  public:
    void set_i(int a) {i = a;}
    int  get_i() {return i;}
};

class Derived : public Base {
  protected:
    int j;
  public:
    void set_j(int a) {j = a;}
};
```

Equivale a: `bp = bp + sizeof(Base)`

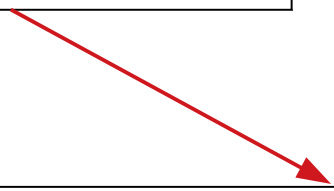
Visualizza un valore casuale!
Perché j **di** d[0] **non è stato inizializzato**

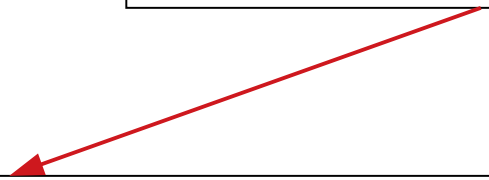
Ereditarietà multipla

- Una classe derivata può ricevere in eredità anche due o più classi
- È necessario un elenco di classi separate da virgole con uno specificatore per ogni classe ereditata
- **Esempio**

```
class Base1 {
  protected:
    int x;
  public:
    void showx() {cout<<x;}
};
```

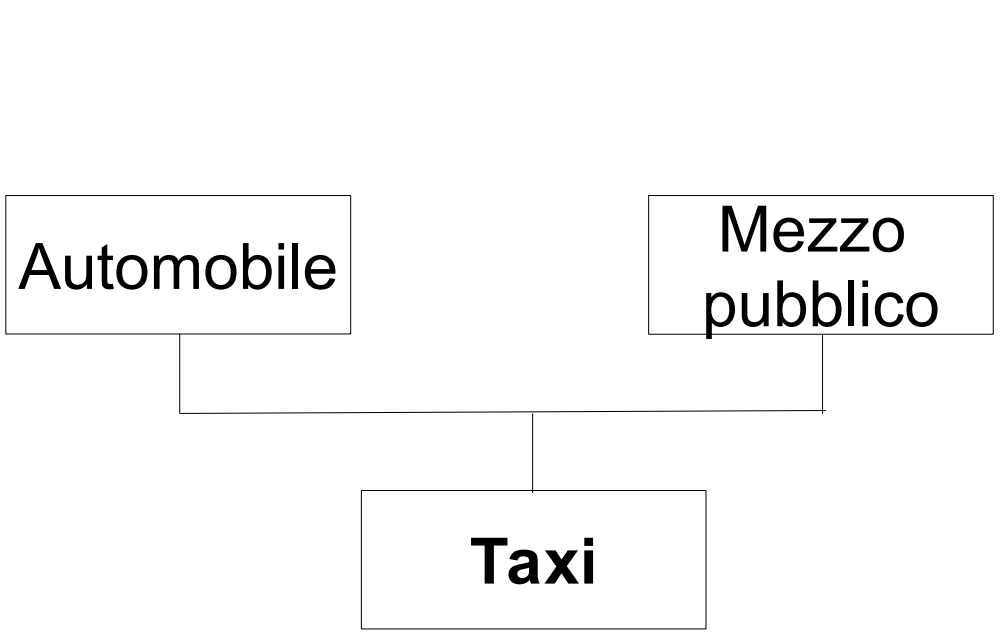
```
class Base2 {
  protected:
    int y;
  public:
    void showy() {cout<<y;}
};
```

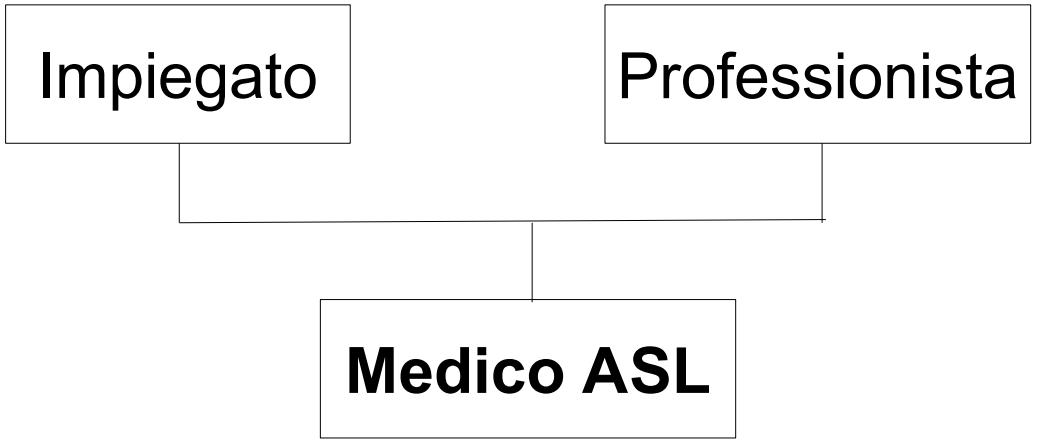




```
class Derived : public Base1, public Base2{
      .
      .
  public:
    void set(int i, int j) {x=i; y=j;}
};
```

Esempi


Automobile
Mezzo pubblico
Taxi


Impiegato
Professionista
Medico ASL

Costruttori e distruttori

- Quando si crea un oggetto di una classe derivata si chiama prima il costruttore della classe base e poi quello della classe derivata.
- Per i distruttori è l'inverso: viene prima chiamato il distruttore della classe derivata e poi quello della classe base.
- **Esempio**

```
class Base {
    int i;
  public:
    Base(){cout<<"costruzione di base"<<endl;}
    ~Base(){cout<<"distruzione di base"<<endl;}
};

class Derived : public Base{
  public:
    Derived(){cout<<"costruzione di derived"<<endl;}
    ~Derived(){cout<<"distruzione di derived"<<endl;}
};
```

```
#include derived.h

int main()
{
  Derived d;

};
```

NOTA
Questa tempficazione è praticamente obbligata: PERCHÈ ?

OUTPUT

```
costruzione di Base
costruzione di Derived
distruzione di Derived
distruzione di Base
```

Passaggio di argomenti

- Per passare argomenti ai costruttori della classe derivata è necessario usare la seguente forma:

```
costruttore_derivata(elenco-argomenti): base1(elenco-
argomenti),
base2(elenco-argomenti), ... baseN(elenco-argomenti)
{
  // Corpo del costruttore derivato
        .
        .
        .

}
```

NOTA
Anche se il costruttore della classe derivata non utilizza argomenti, ne dovrà comunque dichiarare uno se il costruttore della classe base lo richiede.

Esempio

```
class Base {
    int i;
  public:
    Base(int x){i=x; cout<<"costruzione di base, i:"<<i<<endl;}
    ~Base(){cout<<"distruzione di base"<<endl;}
};

class Derived : public Base {
    int j;
  public:
    Derived(int x, int y): Base(y)
      {j=x; cout<<"costruzione di derived, j:"<<j<<endl;}
    ~Derived(){cout<<"distruzione di derived"<<endl;}
};
```

```
#include derived.h

int main()
{
  Derived d(3, 4);

    .
    .

  return 0;
};
```

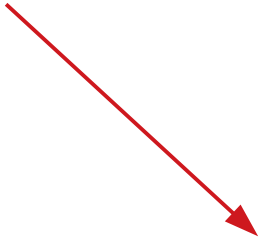
OUTPUT

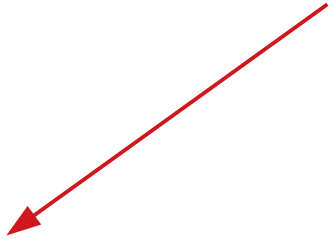
```
costruzione di base, i:4
costruzione di derived, j: 3
distruzione di derived
distruzione di base
```

Ereditarietà multipla


```
class Base1 {
    int i;
  public:
    Base1(int x) {
      i=x;
      cout<<"costruzione di Base1"<<endl;
    }
    ~Base1(){
      cout<<"distruzione di Base2"<<endl;
    }
};
```

```
class Base2 {
    int k;
  public:
    Base2(int x) {
      k=x;
      cout<<"costruzione di Base2"<<endl;
    }
    ~Base2() {
      cout<<"distruzione di Base2"<<endl;
    }
};
```





```
class Derived: public Base1, public Base2 {
    int j;
  public:
    Derived(int x, int y, int z): Base1(y), Base2(z)
      {j=x; cout<<"costruzione di derived"<<endl;}

    ~Derived(){cout<<"distruzione di derived"<<endl;}
};
```

- Gli argomenti passati ai costruttori delle classi base possono essere usati anche dal costruttore della classe derivata
- **Esempio**

```
class Derived: public Base
    int j;
  public:
    Derived(int x, int y): Base(x)
      { j=x*y;
        cout<<"costruzione di derived"<<endl;
      }
};
```

Ereditarietà: esempio

- Supponiamo di avere a disposizione una classe Lista:

```
class Lista{
 public:
    void Lista() {n=0; l =0;}
    int size() const;
    int clear();
    bool empty() const;
    void insert(TipoValue val, int pos);
    void remove(int pos);
    void set(TipoValue val, int pos);
    TipoValue get(int pos) const;
  private:
    nodo *l;
    int n;
};
```

La classe Pila


```
// pila.h

#include "Lista.h"

class Pila : private Lista {
  Public:
    using Lista::size;
    using Lista::clear;
    using Lista::empty;
    Pila() {} // Costruttore
    void push(TipoValue val) {insert(val, 0);}
    TipoValue top() const { return get(0);}
    TipoValue pop() {
      TipoValue tmp = get(0);
      remove(0);

      return tmp;
    }
};
```

La classe Coda


```
// coda.h

#include "Lista.h"

class Coda : private Lista {
  Public:
    using Lista::size;
    using Lista::clear;
    using Lista::empty;
    Pila() {}
    void add(TipoValue val) {insert(val, n);}
    TipoValue head() const { return get(0);}
    TipoValue take() {
      TipoValue tmp = get(0);
      remove(0);

      return tmp;
    }
};
```

Ereditarietà: operatore di assegnazione

- Deve essere realizzato per classi con estensione dinamica.
- A differenza degli altri operatori NON viene ereditato.
- Possono verificarsi 4 diversi casi

SEGUE...

- Nè la classe base nè la derivata implementano **`operator=`**
  - Il compilatore utilizza l'operatore di default (bit a bit) per entrambi le classi.
- Solo la classe base implementa **`operator=`**
  - Il compilatore utilizza l’operatore di assegnazione della classe base e l’operatore di default per la parte derivata

- Solo la classe derivata implementa **`operator=`**
  - Viene richiamato SOLO **`operator=`** della derivata che è responsabile di assegnare TUTTO l'oggetto derivato.
  - Il compilatore NON assegna il sotto-oggetto base (non viene chiamato l'operatore di default della classe base).

- Sia la classe base che la derivata implementano **operator=**:
  - L'operatore della classe derivata deve esplicitamente richiamare l'operatore di assegnazione della classe base

```
Derived& Derived::operator=(const Derived& d)
{
    Base::operator=(d);
  // codice per la parte derivata
 return *this;
}
```

Ereditarietà: costruttore di copia

- Possono verificarsi gli stessi 4 casi visti in precedenza per l'operatore di assegnazione
- il compilatore si comporta allo stesso modo

- Nè la classe base nè la derivata implementano il Costruttore di Copia (CC):
  - Il compilatore usa il CC di default (copia bit a bit) per entrambi
- Solo la classe base implementa il CC:
  - Il compilatore usa il CC della classe base per la parte base ed effettua la copia bit a bit per la parte derivata

SEGUE...

- Solo la classe derivata implementa il CC:
  - Viene richiamato SOLO il CC della derivata che è responsabile di costruire TUTTO l'oggetto derivato. Il compilatore NON chiama il CC di default della classe base.
- Sia la classe base che la derivata implementano il CC
  - Il CC della classe base deve essere chiamato mediante lista di inizializzazione dal CC della classe derivata. Viene costruito prima il sotto-oggetto base e poi quello derivato.

Composizione (composition)

- Riuso di classi come variabili membro di altre classi
- implica una relazione del tipo: has-a (ha-un)
- **Esempio**

```
class Point {
   float x,y;
 public:
   Point(float c1, float c2): x(c1), y(c2){}
   float getX() const {return x;}
   float getY() const {return y;}
   }
};
```

```
#include "point.h"

class Retta {
   Point p1,p2;
 public:
   Retta(Point o1, Point o2): p1(o1),p2(o2){}
   float pendenza() const {
    return (p2.getY()-p1.getY())/(p2.getX()-p1.getX());
   }
};
```

```
#include "retta.h"

int main(){

   Point p1(1,1), p2(2,2);
   Retta r(p1,p2);
   cout<<"Pendenza "<<r.pendenza()<<endl;

   return 0;
}
```

I/O class hierarchy