# Ereditarietà

# Ereditarietà

- L'ereditarietà è importante per la creazione di software riutilizzabile e per controllare la complessita' del codice
- Le classi nuove sono progettate sulla base di classi pre-esistenti
- Le nuove classi acquisiscono gli attributi e i comportamenti (metodi) delle classi vecchie ed aggiungono caratteristiche nuove o raffinano caratteristiche pre-esistenti

# Ereditarietà

- Quando si crea una nuova classe si può fare in modo che questa erediti (acquisisca) i dati membro e le funzioni membro da una classe già definita precedentemente
- La classe precedente prende il nome di *classe base*
- La classe che eredita prende il nome di *classe derivata*

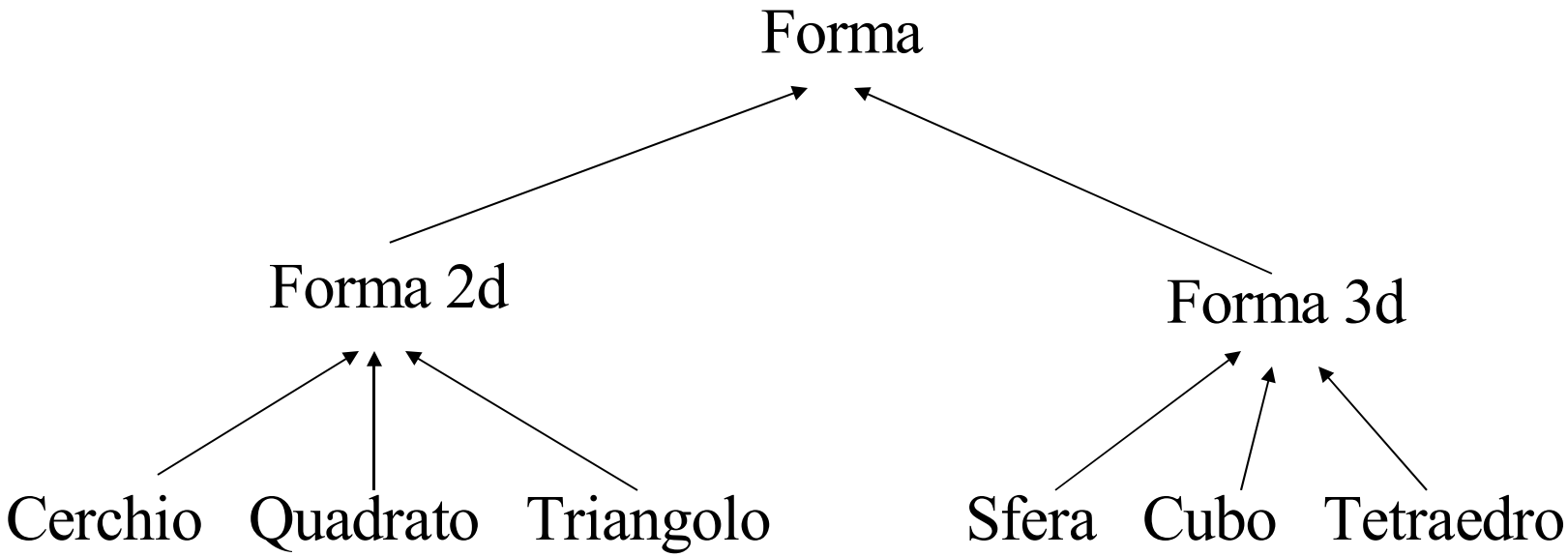
# Ereditarietà gerarchica

- E’ possibile continuare il procedimento di ereditarietà creando una classe che eredita a sua volta da una classe derivata
- Questo procedimento crea una gerarchia
- Una classe base può essere *diretta* o *indiretta* se si trova a livelli più alti della gerarchia di derivazione
- Ovvero se C eredita da B che eredita da A allora:
  - B è una classe base diretta per C
  - A è una classe base indiretta per C

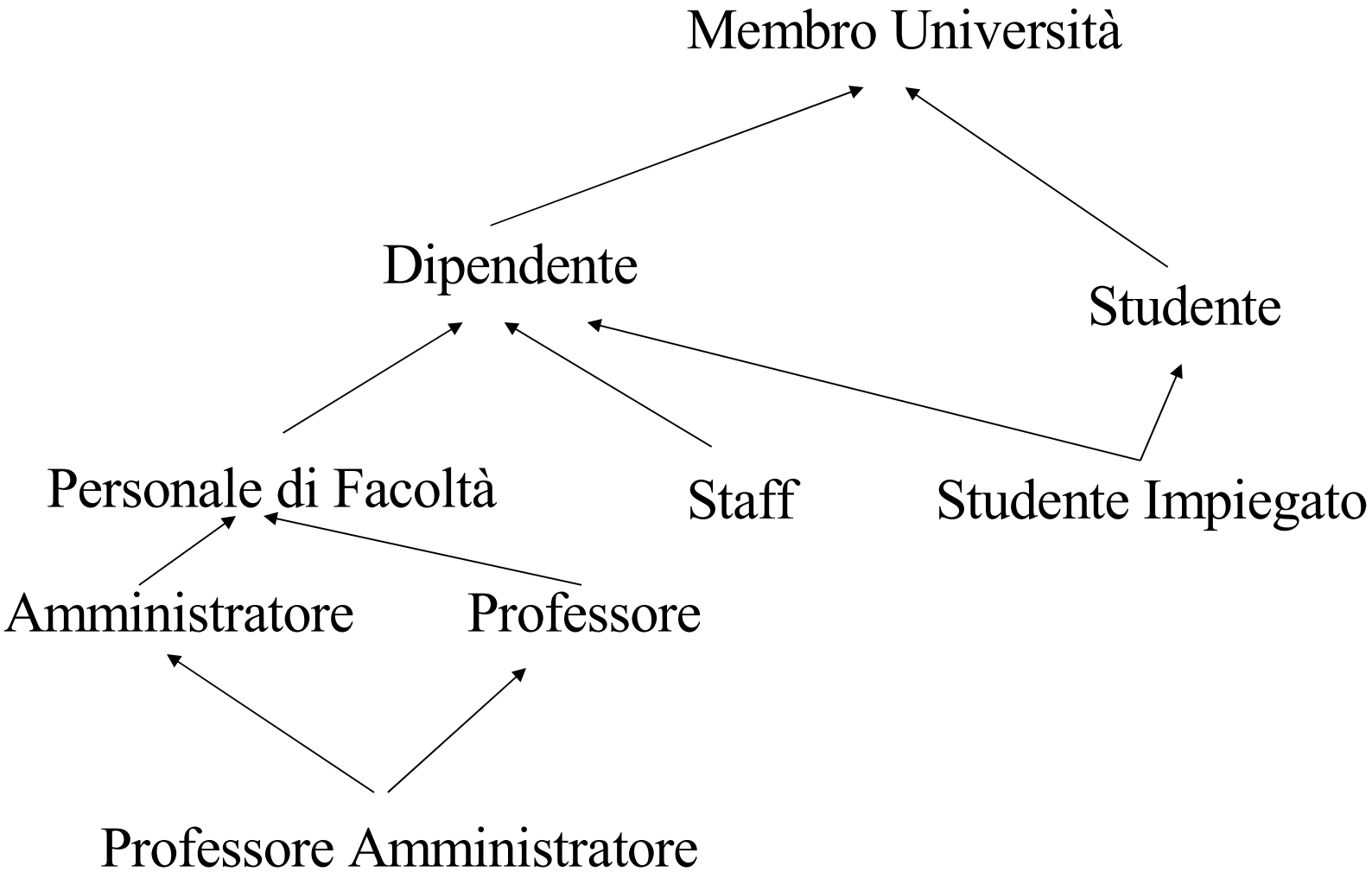
# Ereditarietà

- Esistono due tipi di ereditarietà:
  - *singola*: quando una classe derivata eredita da una sola classe base
  - *multipla*: quando una classe derivata eredita da più classi base che tra loro possono non essere correlate

# Ereditarietà singola



# Ereditarietà multipla



# Cosa può ereditare una classe?

- La classe derivata acquisisce i dati membro e le funzioni membro della classe base

# La sintassi

- Nella definizione della classe derivata si aggiunge la specifica della classe base da cui si eredita ed il tipo di eredità

```
class BaseClass{
  //dichiarazione
};

class DerivedClass: public BaseClass{
  //dichiarazione
};
```

- Vedremo in seguito che esistono tre tipi di ereditarietà:

  *public, private, protected*

# Costruttore di classe derivata

- Il costruttore della classe derivata si deve occupare di inizializzare i dati membri aggiuntivi, quelli cioè che sono introdotti in più rispetto ai dati membro della classe base
- Si può utilizzare il costruttore della classe base per inizializzare i dai membri *condivisi* con la classe base
- La sintassi è:

```
NomeClassDeriv(T prm_bas,T prm_drv):NomeClassBase(prm_bas){
        //init con prm_drv
   }
```

# Dichiarazione di una classe base

```
#ifndef POINT_H
#define POINT_H

class Point{
   friend ostream operator<<(ostream &, const Point &);

public:
   Point(int=0, int=0);
   void setPoint(int, int);
   int getX() const {return x;}
   int getY() const {return y;}

protected:
   int x,y;
};

#endif
```

# Definizione delle funzioni

```
#include<iostream>
#include "point.h"

Point::Point(int a, int b){set(a,b);}

void Point::set(int a, int b){x=a;y=b;}

ostream& operator<<(ostream &out, const Point &p){
   out<<"["<<p.x<<","<<p.y<<"]"<<endl;
   return out;
}
```

# Dichiarazione di una classe derivata

```
#ifndef CIRCLE_H
#define CIRCLE_H

class Circle: public Point{
   friend ostream& operator<<(ostream &, Circle &);
public:
   Circle(double r=0.0, int x=0, int y=0);
   void setRadius(double);
   double getRadius() const {return radius;}
   double area() const;
protected:
   double radius;
}

#endif
```

# Definizione delle funzioni

```
#include<iostream>
#include "circle.h"
 Circle::Circle(double r, int a, int b):
   Point(a,b){//Costruttore per la classe base
   setRadius(r);
}
void Circle::setRadius(double r){
   radius=(r>=0 ? r: 0);
}
double Circle::area()const{
   return 3.14159 * radius *radius;
}
ostream & opertor<<(ostream &out, Circle &c){
   out<<"Center:"<< static_cast<Point>( c ) //cast esplicito
   <<"Radius:"<<c.radius<<endl;
   return out;
}
```

# Esempio

```
#include<iostream>
#include "point.h"
#include "circle.h"
int main(){
   Point p(10,20);
   Circle c(2.1,30,40);
   cout<<c; //Stampa: Center:[30,40] Radius:2.1
   Point *pPtr=&c;
   cout<<(*pPtr); // Stampa: [30,40] il punt vede solo
                  //i dati membri della classe base
   Circle *cPtr=static_cast<Circle *>(pPtr);
   cout<<(*cPtr); // Stampa: Center: [30,40] Radius:2.1.
//I dati ci sono sempre. Erano solo non visibili prima
   cPtr=static_cast<Circle *>(&p);
   cout<<(*cPtr); // Stampa: Center:[30,40] Radius:???
//Adesso invece non sono mai esistiti e quindi si va ad
//accedere in memoria casualmente. ERRORE
}
```

# Note

- Il cast esplicito ha sintassi:

```
static_cast<tipo>(oggetto)
```

- il suo compito è di eseguire (al tempo di compilazione) una conversione forzata al tipo indicato dell'oggetto passato come parametro

- E' possibile passare da una classe derivata alla sua classe base
- E' errato passare da una classe base ad una derivata! Infatti non esistono e non sono accessibili i dati e funzioni membro aggiunte dalla classe derivata: se si tenta di accedervi si genera un errore di violazione di memoria

# Specializzazione

- La classe derivata aggiunge dati membro e funzioni membro a quelle della classe base
- La classe derivata specializza, raffina, reimplementa le funzioni membro della classe base
- Una classe derivata è più *grande* di una classe base nel senso che occupa più spazio, ha più dati e funzioni membro
- Una classe derivata rappresenta tuttavia un gruppo più *ristretto* di oggetti, è più specializzata

# Overriding di funzioni membro

- Una classe derivata può ridefinire una funzione membro della classe base
- Attenzione non è overloading: infatti la funzione ha lo stesso nome e gli stessi parametri
- Se fosse stato un caso di overloading la nuova funzione si sarebbe dovuta distinguere per qualche parametro
- La redefinizione si chiama *overriding*
- La funzione nella classe base è mascherata dalla funzione ridefinita nella classe derivata

# Accedere a funzioni overridden

- Una classe derivata può aver bisogno di accedere alle funzioni della classe base
- Se le funzioni sono state ridefinite tramite overriding sorge il problema di indicarle senza ambiguità
- Lo si può fare utilizzando l'operatore di risoluzione :: ed indicando il nome della classe base

# Esempio

```
// Definition of class Employee
#ifndef EMPLOY_H
#define EMPLOY_H

class Employee {
public:
   Employee( const char *, const char * );  // constructor
   void print() const;  // output first and last name
   ~Employee();         // destructor
private:
   char *firstName;     // dynamically allocated string
   char *lastName;      // dynamically allocated string
};
#endif
```

```
// Member function definitions for class Employee
#include <iostream>
#include <cstring>
#include <cassert>
#include "employ.h"
Employee::Employee( const char *first, const char *last ){
   firstName = new char[ strlen( first ) + 1 ];
   assert( firstName != 0 ); // terminate if not allocated
   strcpy( firstName, first );
   lastName = new char[ strlen( last ) + 1 ];
   assert( lastName != 0 );  // terminate if not allocated
   strcpy( lastName, last );
}
void Employee::print() const
   { cout << firstName << ' ' << lastName; }
Employee::~Employee(){
   delete [] firstName;    // reclaim dynamic memory
   delete [] lastName;     // reclaim dynamic memory
}
```

```
// Definition of class HourlyWorker
#ifndef HOURLY_H
#define HOURLY_H

#include "employ.h"

class HourlyWorker : public Employee {
public:
   HourlyWorker( const char*, const char*, double, double );
   double getPay() const;  // calculate and return salary
   void print() const;     // overridden base-class print
private:
   double wage;            // wage per hour
   double hours;           // hours worked per week
};

#endif
```

```
// Member function definitions for class HourlyWorker
#include <iostream>
#include <iomanip>
#include "hourly.h"

HourlyWorker::HourlyWorker( const char *first,
                            const char *last,
                            double initHours, double initWage )
   : Employee( first, last )   // call base-class constructor
{
   hours = initHours;  // should validate
   wage = initWage;    // should validate
}

// Get the HourlyWorker's pay
double HourlyWorker::getPay() const { return wage * hours; }
```

```
// Print the HourlyWorker's name and pay
void HourlyWorker::print() const
{
   cout << "HourlyWorker::print() is executing\n\n";
   Employee::print();   // call base-class print function

   cout << " is an hourly worker with pay of $"
        << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 2 ) << getPay() << endl;
}
```

```
#include "hourly.h"
int main()
{
   HourlyWorker h( "Bob", "Smith", 40.0, 10.00 );
   h.print();
   return 0;
}
```