UNIVERSITÀ PARTHENOPE
DiST
Dipartimento di Scienze e Tecnologie

# Programmazione II e Lab di PII

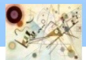
## Classi derivate

Angelo Ciaramella

# Programmazione OO

- La programmazione ad oggetti si basa su principi fondamentali
  - incapsulamento
  - late binding
    - chiamata di un un metodo
      - il codice non determina la funzione finchè non è a runtime
      - il compilatore assicura che il metodo esiste e definisce il tipo dell'argomento di ritorno, ma non sa quale sarà il codice che dovrà eseguire
  - ereditarietà
  - polimorfismo
    - per metodi
      - Overloading
      - Overriding
    - per dati
      - Chiamata virtuale di metodi

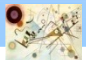




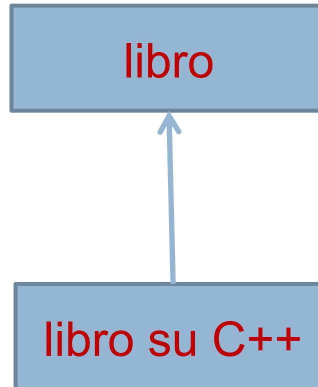
# Ereditarietà

- Ereditarietà
  - definire una classe in termini di una classe definita in precedenza
  - la nuova classe eredita la rappresentazione in memoria e l'interfaccia della sua classe base
  - il programmatore non deve riscrivere tutto il codice che è in comune tra la classe base e la nuova classe
  - il programmatore può modificare l'implementazione di alcune funzioni membro (override)
  - l'ereditarietà consente di implementare relazioni di specializzazione tra tipi di dato



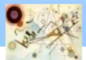


# Derivazione



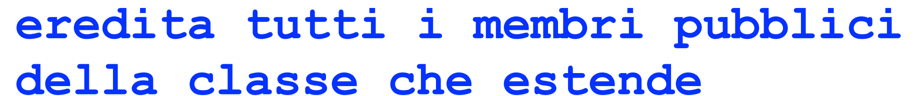

Esempi di derivazione





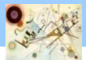
# Parola chiave extends

```
class Libro {
public:
        string titolo;
        string autore;
        string editore;
        string numeroPagine;
        string prezzo;
. . .
}
```




```
class LibroSuCpp : public Libro {
public:
        const string ARGOMENTO_TRATTATO = "C++";
. . .
}
```





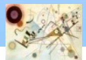
# Classe non derivata

```
struct Employee {
  string first_name , family_name;
  char middle_initial;
  Date hiring_date;
  short department;
// ...
};
```

```
struct Manager {
  Employee emp; //record Employee del manager
  list<Employee*> group; // persone gestite

// ...
};
```

Esempio di classe non derivata (Manager non è anche un Employee)



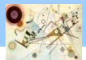


# Classe derivata

```
struct Manager: public Employee{
    list<Employee*> group; // persone gestite
    short level;
// ...
};
```

Esempio di classe derivata (Manager è anche un Employee)



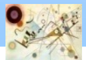


# Esempio

```
class Albero {
public:
void stampa_Albero()
{
        cout << "Albero" << endl;
}
};

class Abete : public Albero {
public:
void stampa_Abete()
{
        cout << "Abete" << endl;
}
};
```

Classi derivate



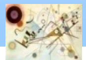


# Esercizio

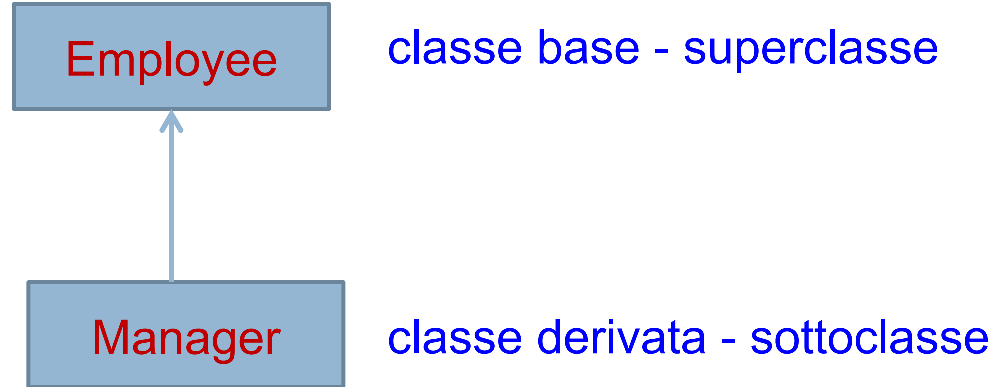
```
Abete* al = new Abete();
al->stampa_Albero();
al->stampa_Abete();
```

Verificare la correttezza del codice e l'eventuale output





# Ereditarietà

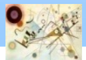



classe base - superclasse

classe derivata - sottoclasse

Esempi di derivazione





# Ereditarietà

tromba

musica

jazz

rock

punk

bi pop

fusion

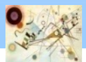
Dizzy Gillespie

Gerarchia di ereditarietà

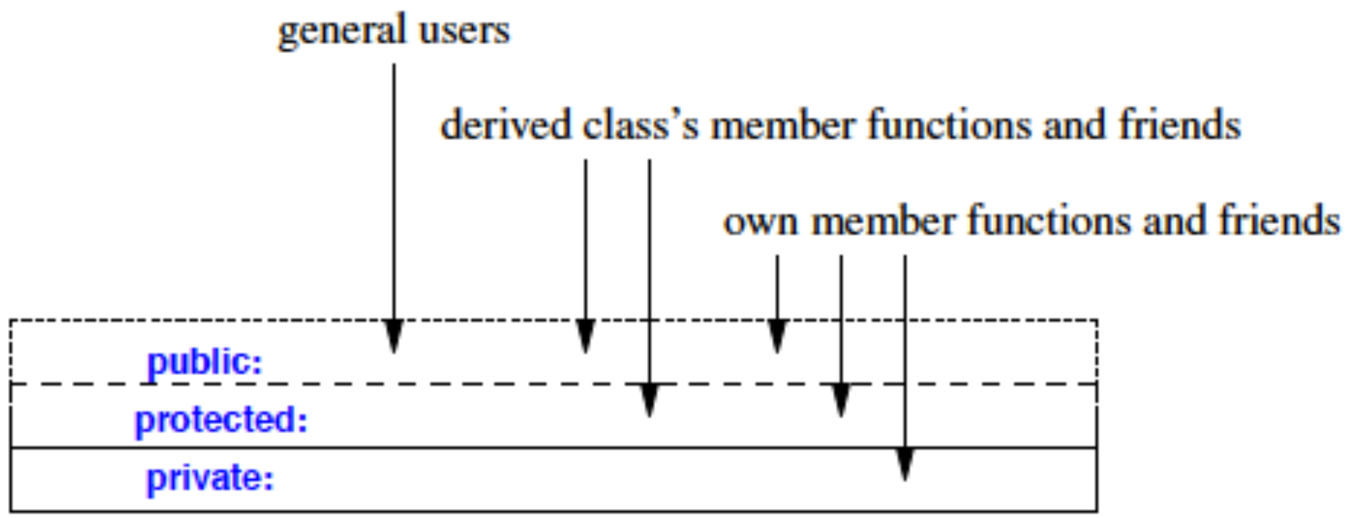
# Controllo di accesso

- Un membro di una classe può essere
  - private
    - il suo nome può essere utilizzato solo da funzioni membro e friend della classe nella quale viene dichiarato
  - protected
    - il suo nome può essere utilizzato solo da funzioni membro e friend della classe nella quale viene dichiarato e da funzioni membro e friend delle classi derivate dalla classe
  - public
    - Il suo nome può essere utilizzato da qualsiasi funzione

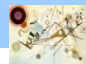




# Controllo di accesso



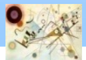


# Class diagram

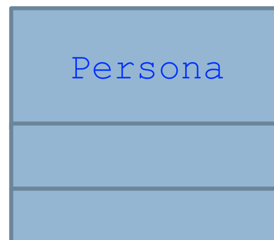
- Il class diagram
  - illustra l'ambito di descrizione da un punto di vista statico
  - evidenzia caratteristiche e mutue relazioni
    - classi
    - relazioni

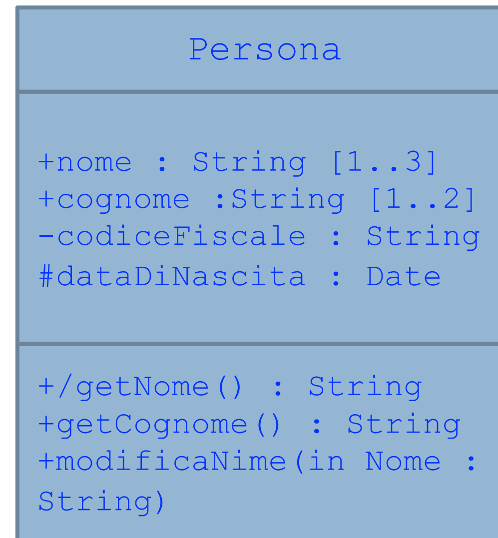




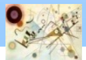
# Classe

- Classe
  - descrive un insieme di entità dotate delle stesse caratteristiche e proprietà
    - oggetti





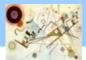



# Classe derivata

```
class canzone: public suono {
 public:
       void forward(unisgned short, int, int);
       // ridefinzione

       double dammi_la_codifica();
      // nuovo metodo

 //...
 protected:
      char *titolo;
      unsigned int codice;
      double costo;
      unsigned short tipo_codifica;

};
```

Esempio di classe derivata

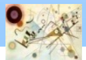




# Esercizio

- Implementare una classe `strumenti_a_corde`
- Implementare le seguenti classi derivate
  - `chitarra`
  - `basso`



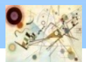


# Funzioni membro

```
class Employee {
public:
        void print() const;
        string full_name() const { return first_name + ' ' + 
middle_initial + ' ' + family_name; }
// ...
private:
        string first_name , family_name;
        char middle_initial;
// ...
};

class Manager : public Employee {
public:
        void print() const;
// ...
};
```

Derivazione di membri





# Funzioni membro

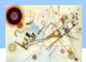
```
void Manager::print() const
{
      cout << "name is " << full_name() << '\n';
// ...
}
```

Ereditarietà dei membri pubblici

```
void Manager::print() const
{
      cout << " name is " << family_name << '\n';
     // errore!
// ...
}
```

Problemi di ereditarietà dei membri privati

Classi derivate

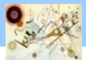


# Funzioni membro

```
void Manager::print() const
{
        Employee::print();
       // stampa informazioni di Employee

       cout << level;
       // stampa informazioni specifiche per Manager

// ...
}
```

Utilizzo dei membri della superclasse





# Gerarchie di classi

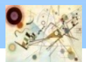
```
class Employee { /* ... */ };
class Manager : public Employee { /* ... */ };
class Director : public Manager { /* ... */ };
```

Gerarchia ad albero

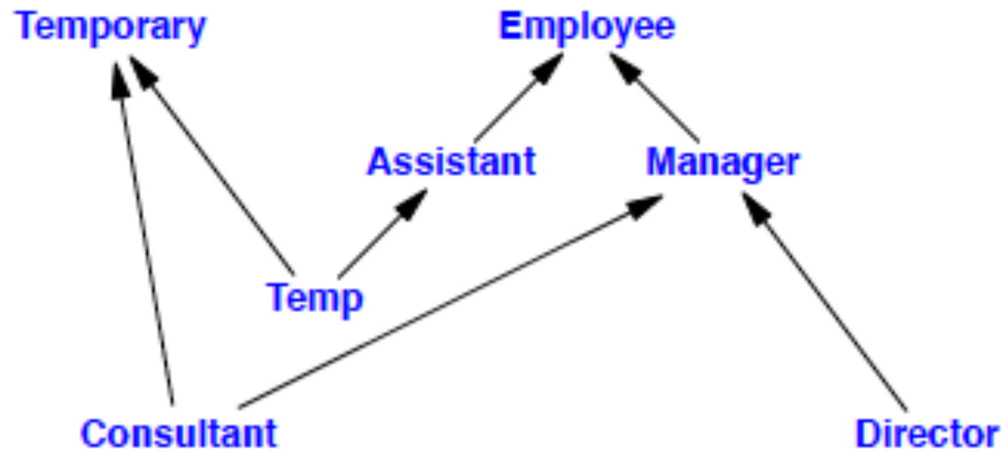
```
class Temporary { /* ... */ };
class Assistant : public Employee { /* ... */ };
class Temp : public Temporary, public Assistant { /* ... */
};
class Consultant : public Temporary, public Manager { /*
... */ };
```

Gerarchia a grafo



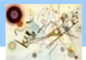


# Gerarchie di classi

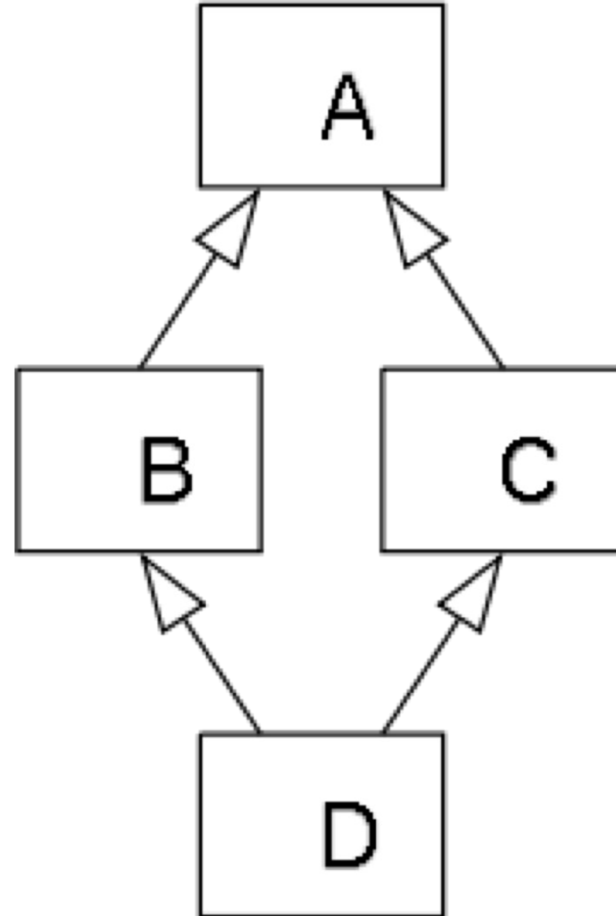



Gerarchia a grafo





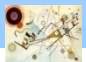
# Diamond problem



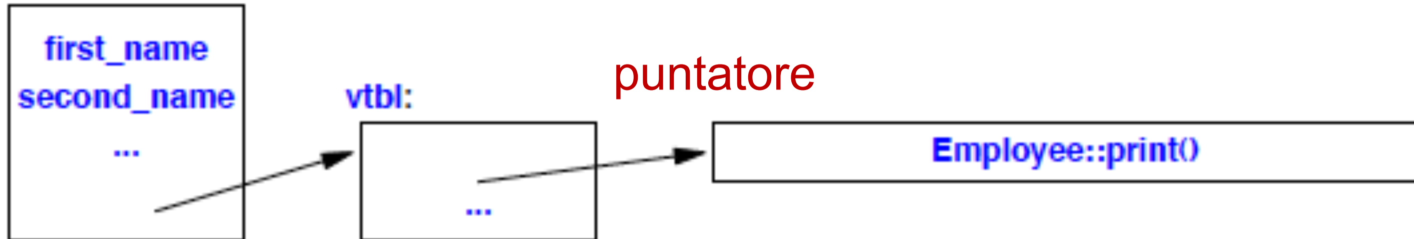
# Funzioni virtuali

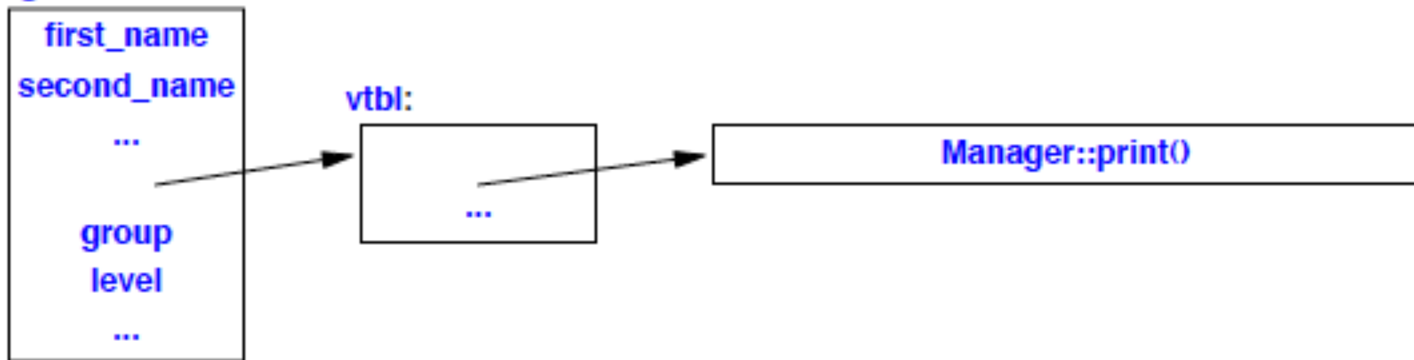
- Funzioni virtuali
  - consentono di dichiarare funzioni in una classe base che possono essere ridefinite in ciascuna classe derivata
  - il compilatore e il linker garantiscono la corretta corrispondenza tra gli oggetti e le funzioni
  - fungono da interfaccia
    - metodo
    - nella classe derivata deve avere la stessa firma (nome e passaggio di parametri)
    - una classe derivata non necessariamente deve fornire un'implementazione
  - Insieme a override caratterizzano il polimorfismo
    - capacità di assumere





# Funzioni virtuali



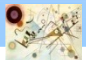

Ogni classe ha una tabella delle funzioni virtuali (vtbl)

# Funzioni virtuali

```
class Employee {
public:
        Employee(const string& name, int dept);
        virtual void print() const;
// ...
private:
        string first_name , family_name;
        short department;
// ...
};
```

Dichiarazione di una funzione virtuale (o metodo)



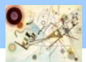


# Funzioni virtuali

```
class Manager : public Employee {
public:
        Manager(const string& name, int dept, int lvl);
        void print() const;
// ...
private:
        list<Employee*> group;
        short level;
// ...
};

void Manager::print() const
{
Employee::print();
cout << "\t level " << level << " \n ";
// ...
}
```

Dichiarazione di una funzione virtuale (o metodo)

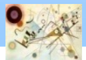




# Override

- Override
  - è usato per definire che una funzione in una classe derivata è concepita per prevalere su una funzione virtuale di una classe base





# Override

```
struct B0 {
        void f(int) const;
        virtual void g(double);
};
struct B1 : B0 { /* ... */ };
struct B2 : B1 { /* ... */ };
struct B3 : B2 { /* ... */ };
struct B4 : B3 { /* ... */ };
struct B5 : B4 { /* ... */ };
```

Gerarchia di classi

```
struct D : B5 {
        void f(int) const override;
        // errore : B0::f() non è virtuale
        void g(double) override;
       // OK
        virtual int h() override;
      // errore : nessuna function h()
};
```

Esempi di utilizzo di override in classe derivate

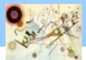
# Final

- Final
  - è usato in una classe derivata per prevenire la possibilità di prevalere ulteriormente

```
class Derived : public Base {
void f() final; // OK se Base ha una f. virtuale f()
void g() final; // OK se Base ha una f. virtuale g()
// ...
};
```

Clausola final

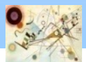




# Polimorfismo

- I metodi pubblici di una classe costituiscono l'interfaccia della classe
  - i messaggi che l'oggetto può interpretare
- La funzione è assegnata al messaggio in fase di codifica
  - early binding
- Può essere necessario assegnare la funzione al messaggio run-time
  - late binding

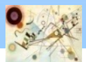




# Polimorfismo

- binding dinamico (o late binding)
  - la versione del metodo da eseguire viene scelta sulla base del tipo di oggetto effettivamente contenuto in una variabile a runtime, invece che al momento della compilazione
  - *Se ho una variabile di tipo A, e il tipo A ha due sottoclassi B e C, che ridefiniscono entrambe il metodo m(), l'oggetto contenuto nella variabile potrà essere di tipo A, B o C, e quando sulla variabile viene invocato il metodo m() viene eseguita la versione appropriata per il tipo di oggetto contenuto nella variabile in quel momento*



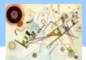


# Binding dinamico

```
#include<iostream>

class B{
public:
virtual void bar(){};
virtual void qux(){};
};

class C : public B{

public:
void bar() override{};
};
```

Esempio di binding dinamico





# Binding dinamico

| Vtable of class B |
| --- |
| bar |
| qux |

| Vtable of class C |
| --- |
| bar |
| qux |

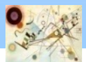
B::bar()

B::qux()

C::bar()

Esempio di binding dinamico



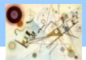


# Overriding vs Hiding

```
#include <iostream>

using namespace std;
class Parent {
public:
void doA() { cout << "doA in Parent" << endl; }
virtual void doB() { cout << "doB in Parent" <<
endl; }
};
class Child : public Parent {
public:
void doA() { cout << "doA in Child" << endl; }
void doB() { cout << "doB in Child" << endl; }
};
```

Esempio di binding dinamico



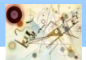


# Overriding vs Hiding

```
main(){
Parent* p1 = new Parent();
Parent* p2 = new Child();
Child* cp = new Child();
p1->doA();
p2->doA();
cp->doA();
p1->doB();
p2->doB();
cp->doB();
}
```

Esempio di binding dinamico

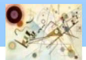




# Classe astratta

- Classe astratta
  - classe utilizzata come interfaccia per altre classi
    - non è possibile istanziare oggetti
  - classe contenente una o più funzioni virtuali pure
  - una classe derivata che non implementa una funzione virtuale pura è una classe astratta



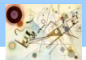


# Classe astratta

```
class Shape { // classe astratta
public:
        virtual void rotate(int) = 0;
         // funzione virtuale pura
        virtual void draw() const = 0;
       // funzione virtuale pura
       virtual bool is_closed() const = 0;
       // funzione virtuale pura
// ...
virtual ~Shape(); //virtuale
};
```

Dichiarazione di una funzione virtuale pura e classe Astratta



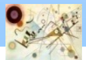


# Classe astratta

```
class Point { /* ... */ };
class Circle : public Shape {
public:
        void rotate(int) override { }
        void draw() const override;
        bool is_closed() const override { return true; }
        Circle(Point p, int r);
private:
        Point center;
        int radius;
};
```

Implementazione di classe Astratta

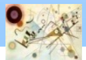




# Polimorfismo

- Polimorfismo
  - dal greco “molte forme”
  - consente di riferirci con un unico termine a “entità” diverse
  - e.g., sia un telefono fisso sia un portatile permettono di telefonare
    - telefonare può essere considerata un’azione polimorfica





# Esercizio



**_Albero_**

+_cresce()_
+stampa()

stampa(): visualizza "Albero"

**Abete**

+cresce()
+stampa()

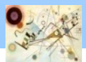
cresce(): visualizza "Cresce Abete"
stampa(): visualizza "Abete"

**Pino**

+cresce()
+stampa()

cresce(): visualizza "Cresce Pino"
stampa(): visualizza "Pino"

Data la classe astratta Albero implementare le classi derivate Abete e Pino



# Esempio

```
Albero* al;                        // esempio di polimorfismo

al = new Albero; // ?
al-> stampa();

al = new Abete();
al->cresce();
al->stampa();

al = new Pino();
al->cresce();
al->stampa();
```

Effettuare i seguenti test