CLASSI DERIVATE

Ereditarietà

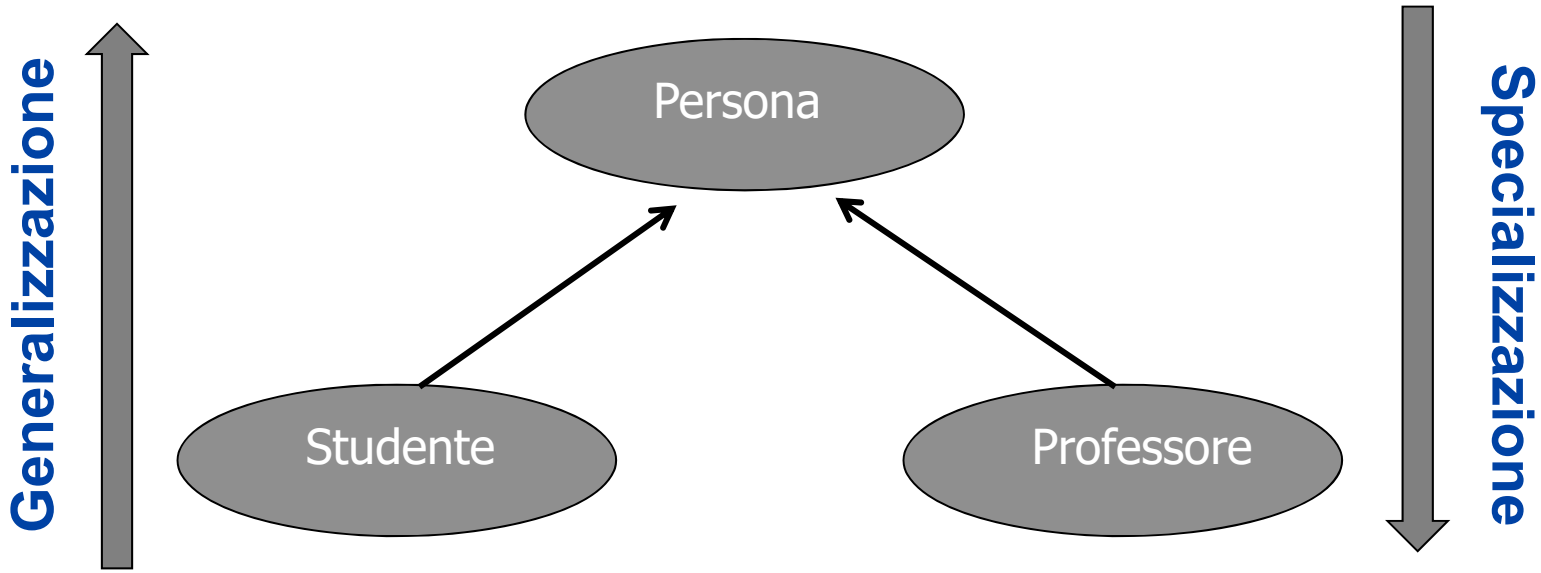
Roberto Nardone, Luigi Romano

ROADMAP

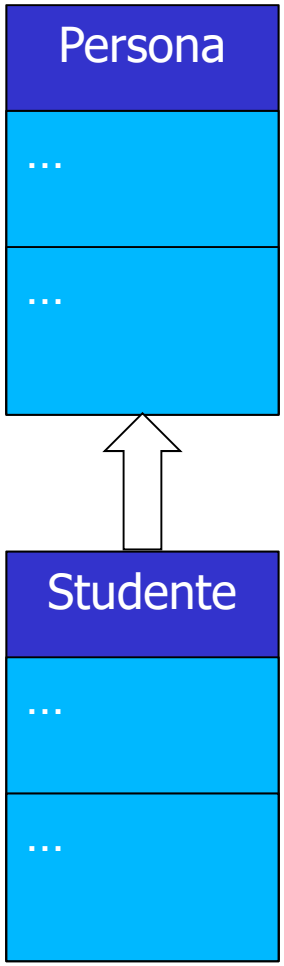
- Concetto di ereditarietà
- Tipi di ereditarietà
  - Public
  - Private
  - Protected
- Costruttori e Distruttori classi derivate
- Ereditarietà multipla
- Binding
- Funzioni virtual
- Polimorfismo

EREDITARIETÀ

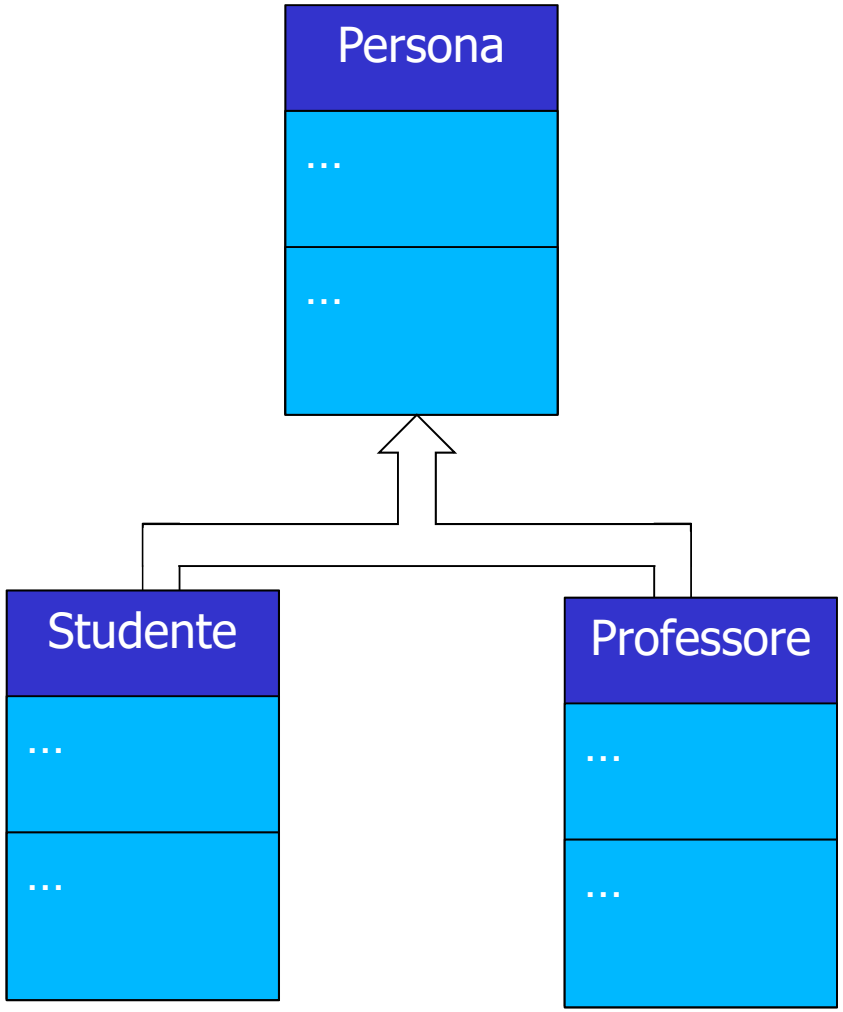
- Nella programmazione orientata agli oggetti l'**ereditarietà** è un concetto che nasce dall'esigenza di costruire classi più specifiche a partire da altre più generali.



CLASS DIAGRAM: EREDITARIETÀ



CLASS DIAGRAM: EREDITARIETÀ



GERARCHIE DI CLASSI

- Classi specifiche, note come **classi derivate**, ereditano comportamenti e attributi da classi generali, dette **classi base**, generando una **gerarchia di classi**.

- Ereditare attributi e metodi da una classe base riduce la ridondanza del codice, consentendo di creare programmi adattabili e riutilizzabili.

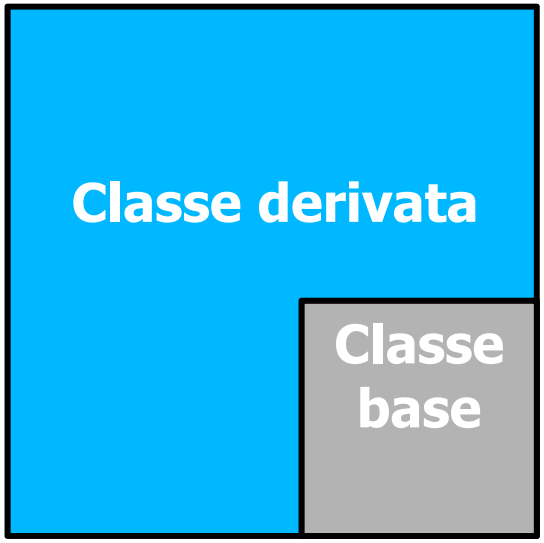
- Trasmettere un insieme di caratteristiche comuni da una classe base ad una derivata minimizza i tempi necessari per l'espansione delle funzionalità del software nel medio-lungo periodo.

DICHIARAZIONE DI UNA CLASSE DERIVATA

- La dichiarazione di una classe derivata deve includere il nome della classe base ed uno specificatore che indica il tipo di ereditarietà fra **public**, **private** e **protected**.

```
class Studente :  [tipo_ereditarietà] Persona
{
    public:
        //nuovi metodi della classe Studente
    private:
        //nuovi membri della classe Studente
};
```

ACCESSO AI MEMBRI IN UNA GERARCHIA DI CLASSI



- Nel contesto della classe derivata, l'accesso ai membri della classe base è definito dalle regole di visibilità stabilite da quest'ultima.
- Al di fuori della classe derivata, la visibilità dei membri della classe base è regolata dal tipo di ereditarietà specificato nella dichiarazione della derivata.
- Un membro ereditato non può mai avere visibilità superiore a quanto specificato dalla classe base.

EREDITARIETÀ PUBBLICA

- Con l'ereditarietà **pubblica**, i membri ereditati mantengono lo stesso livello di visibilità che avevano nella classe base.

Tipo di ereditarietà	Accesso membro classe base	Accesso membro classe derivata
public	public protected private	public protected non accessibili

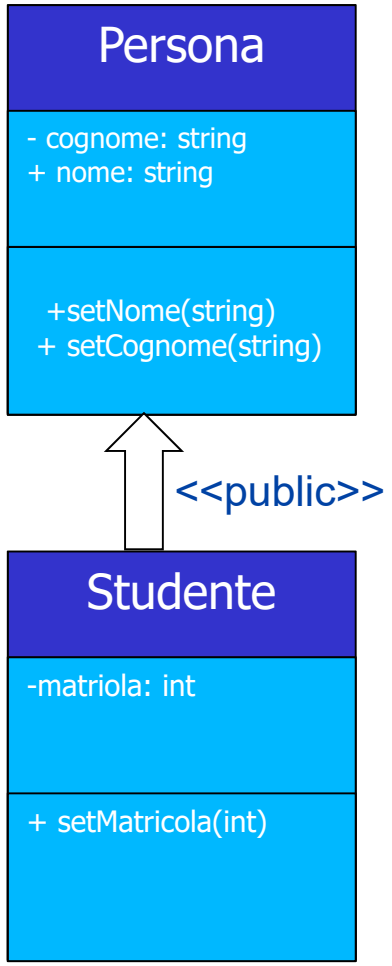
- Ad eccezione di un membro privato della classe base che risulterà **inaccessibile** alla classe derivata.

EREDITARIETÀ PUBBLICA: CLASSE BASE


```
#include<string>
using namespace std;
class Persona
{
        private:
                string cognome;
        public:
                string nome;
                void setNome(string n)
                {
                        nome=n;
                }

                void setCognome(string c)
                {
                        cognome=c;
                }
};
```

EREDITARIETÀ PUBBLICA: CLASSE DERIVATA


```
class Studente : public Persona
{
                    private:
                                int matricola;
                    public:
                                void setMatricola(int m)
                                {
                                            matricola = m;
                                }
};
```



EREDITARIETÀ PUBBLICA: FUNZIONE MAIN()


```
int main()
{
        Studente s;
        s.setCognome("Rossi");  // OK
        s.nome;                 // OK
        s.cognome;              // Errore !!!

}
```

- Il parametro **nome** ed il metodo **setCognome()** della classe base sono **pubblici**, l'ereditarietà è **pubblica** e quindi nel contesto del main() le chiamate **s.nome** e **s.getCognome('Rossi')** sono lecite.

- Il tentativo di accesso all'attributo **cognome** non è consentito nel contesto del main() in quanto la sua visibilità è stata specificata come **privata** nella classe base.

EREDITARIETÀ PRIVATA

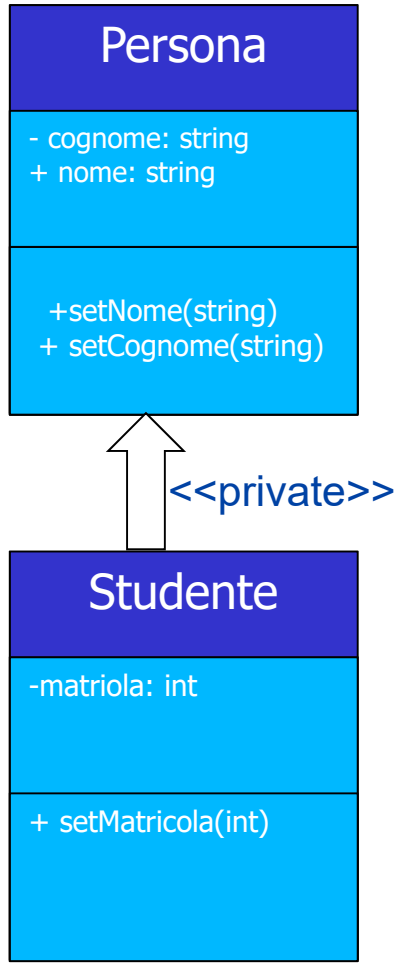
- Con l'ereditarietà **privata** i membri specificati come **pubblici** e **protetti** dalla classe base, una volta ereditati, assumono visibilità **privata** all'esterno della classe derivata.

Tipo di ereditarietà	Accesso membro classe base	Accesso membro classe derivata
private	public protected private	private private non accessibili

- Un membro privato della classe base risulterà **inaccessibile** alla classe derivata.

EREDITARIETÀ PRIVATA: CLASSE DERIVATA


```
class Studente : private Persona
{
                    private:
                                int matricola;
                    public:
                                void setMatricola(int m)
                                {
                                            matricola = m;
                                }
};
```



EREDITARIETÀ PRIVATA: FUNZIONE MAIN()


```
int main()
{
        Studente s;
        s.setCognome("Rossi");  // Errore !!!
        s.nome;                 // Errore !!!
        s.cognome;              //  Errore !!!

}
```

- Il parametro **nome** ed il metodo **setCognome()** della classe base sono **pubblici**, l'ereditarietà è **privata** e quindi nel contesto del main() le chiamata **s.nome** e **s.getCognome('Rossi')** non sono lecite.

- Il tentativo di accesso all'attributo **cognome** non è consentito nel contesto del main() in quanto la sua visibilità è stata specificata come **privata** nella classe base.

EREDITARIETÀ PROTETTA

- Con l'ereditarietà **protetta** i membri **pubblici** e **protected** ereditati assumono visibilità **protetta** all'esterno della classe derivata.

Tipo di ereditarietà	Accesso membro classe base	Accesso membro classe derivata
protected	public protected private	protected protected non accessibili

- Un membro privato della classe base sarà **inaccessibile** alla classe derivata.

EREDITARIETÀ PROTETTA: CLASSE DERIVATA


```
class Studente : protected Persona
{
                    private:
                                int matricola;
                    public:
                                void setMatricola(int m)
                                {
                                            matricola = m;
                                }
};
```

EREDITARIETÀ PROTETTA: FUNZIONE MAIN()


```
int main()
{
        Studente s;
        s.setCognome("Rossi");      // Errore !!!
        s.nome;                     // Errore !!!
        s.cognome;                  // Errore !!!

}
```

- Dal punto di vista dell'utente, ereditarietà privata e protetta hanno un comportamento analogo, rendendo **inaccessibili** i membri della classe base.

MODIFICATORI DI ACCESSO

- Per quanto visto, l'ereditarietà **privata** e **protetta** occulta all'esterno della classe derivata anche i membri con visibilità **pubblica** della classe base.

- Il linguaggio C++ consente di ripristinare il livello di visibilità originario di un membro della classe base.

- Per farlo, all'interno della sezione pubblica della classe derivata, vanno inserite una delle seguenti due possibili dichiarazioni:

  - nome_classe_base :: nome_membro; (**deprecated !!!**)
  - using nome_classe_base :: nome_membro.

ACCESSO MEMBRO PUBBLICO CON DERIVAZIONE PRIVATE/PROTECTED

Con riferimento alla classe base **Persona** definita all'inizio della lezione:

- Per poter usare/accedere ai due membri pubblici **setNome()** e **nome** di Persona, il codice della classe derivata **Studente** va modificato come segue:

```
class Studente : private Persona
{
        private:
                int matricola;
        public:
                using Persona::nome;
                using Persona::setNome;
                void setMatricola(int m)
                {
                        matricola = m;
                }
};
```

Accesso Membro Pubblico con Derivazione Private/Protected: Main() e Compilazione


```
int main()
{
        Studente s;
        s.setNome("Mario");
        cout<<"Il nome inserito è: "<<s.nome<<endl;
}
```

```
Gaetano@DESKTOP-ADCS5PF:/mnt/c/Users/Gaetano Papale/Desktop$ g++ Access.cpp -o ac
Gaetano@DESKTOP-ADCS5PF:/mnt/c/Users/Gaetano Papale/Desktop$ ./ac
Il nome inserito è: Mario
```

SUGGERIMENTI

- Salvo casi particolate, l'ereditarietà **pubblica** è quella più comunemente usata.
- L'ereditarietà **privata** è utile quando si vuole limitare l'accesso ai membri **pubblici** e **protetti** della classe base, alle sole funzioni membro della classe derivata.
- L'ereditarietà **protetta** è utile quando si vuole consentire l'accesso ai membri **pubblici** e **protetti** della classe base, non solo alle funzioni membro della classe derivata, ma anche a ulteriori classi derivate da quest'ultima.

ESERCIZIO

Definire una classe principale **Padre** la cui sezione pubblica sia caratterizzata da:

- 2 attributi interi senza segno **a** e **b**
- 2 funzioni membro senza parametri **somma()** e **sottrazione().**

Somma() ritorna l'intero ottenuto dalla somma tra a e b, sottrazione() la loro differenza in valore assoluto.

Derivare da **Padre** 2 sottoclassi:

- **FiglioPubblico** (derivata come public)
- **FiglioPrivato (**derivata come private)

L'implementazione delle 2 classi dovrà consentire, **a partire da oggetti delle sottoclassi**, la modifica dei parametri della classe principale, l'uso delle funzioni somma() e sottrazione() e la stampa a video dei loro risultati.

*Non è richiesta l'implementazione delle funzioni di set e get per gli attributi a e b.

DEFINIZIONE CLASSI


```
#include<iostream>

using namespace std;

class Padre
{

        public:
                unsigned int a, b;
                int addizione(){return a+b;}
                int sottrazione()
                {
                        if(a>b)
                        return a-b;
                        else
                        return b-a;
                }
};
class FiglioPubblico : public Padre
{

};
class FiglioPrivato : private Padre
{
        public:
                using Padre :: a;
                using Padre :: b;
                using Padre :: addizione;
                using Padre :: sottrazione;
};
```

MAIN()


```
int main()
{
        Padre p;
        cout<<"Inserire primo addendo: ";
        cin>>p.a;
        cout<<"Inserire secondo addendo:";
        cin>>p.b;
        cout<<"La somma è:"<<endl;
        cout<<p.addizione()<<endl;
        cout<<"-----------------------"<<endl;

        FiglioPrivato fpr;
        cout<<"Inserire minuendo: ";
        cin>>fpr.a;
        cout<<"Inserire sottraendo: ";
        cin>>fpr.b;
        cout<<"La differenza è:"<<endl;
        cout<<fpr.sottrazione()<<endl;
        cout<<"-----------------------"<<endl;

        FiglioPubblico fpp;
        cout<<"Inserire minuendo:";
        cin>>fpp.a;
        cout<<"Inserire sottraendo";
        cin>>fpp.b;
        cout<<"La differenza è:"<<endl;
        cout<<fpp.sottrazione()<<endl;
}
```

MODIFICA ESERCIZIO PADRE-FIGLI

Modificare la tipologia di accesso di **a** e **b** in **private** e apportare le dovute correzioni alle 3 classi dell'esercizio precedente al fine di garantire il corretto funzionamento dell'applicazione.

COSTRUTTORI ED EREDITARIETÀ

- In una gerarchia di classi è possibile che classe base e classe derivata abbiano ciascuna uno o più costruttori.

In una situazione del genere, chi sarà responsabile della costruzione di un oggetto della classe derivata?

Il costruttore della classe base, quello della derivata o entrambi?

- Proviamo a capirlo con un semplice esempio.

ESEMPIO:
INVOCAZIONE COSTRUTTORI (DI DEFAULT) ED EREDITARIETÀ


```
#include <iostream>

using namespace std;


class Base{

        public:
                Base()
                {
                        cout<<"Costruttore della classe base"<<endl;
                }
};

class Derivata: public Base{

        public:
                Derivata()
                {
                        cout<<"Costruttore della classe derivata"<<endl;
                }
};

int main()
{
        Derivata d;
}
```

COMPILAZIONE ESEMPIO


```
osboxes@osboxes:~/Desktop$ g++ Costr_Derivata.cpp -o costr
osboxes@osboxes:~/Desktop$ ./costr
Costruttore della classe base
Costruttore della classe derivata
osboxes@osboxes:~/Desktop$
```

- Dalla compilazione dell'esempio si evince che la risposta al quesito proposto è: **entrambi.**
- Prima viene invocato il costruttore della classe base e poi quello della classe derivata.
- Il costruttore della classe base genera la porzione di codice comune, quello della classe derivata specializza l'oggetto creato.

COSTRUTTORI CON PARAMETRI ED EREDITARIETÀ

- Quando una classe base ha un costruttore, la classe derivata deve richiamarlo esplicitamente per inizializzare la porzione di oggetto relativa alla classe base.

- Per richiamare il costruttore della classe base in quello della classe derivata, la dichiarazione di quest'ultimo si espande come segue:

costruttore_derivata(argomenti) : costruttore_base(argomenti)

tipo e **nome** di tutti gli argomenti necessari alla costruzione dell'oggetto **derivato**

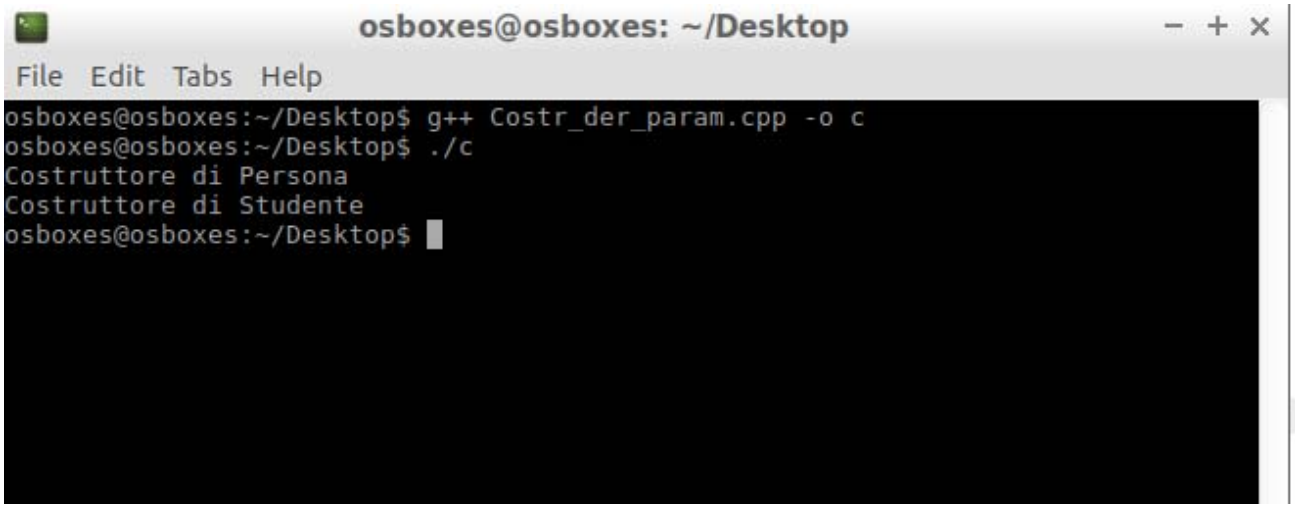
nome dei soli argomenti necessari alla costruzione dell'oggetto **base**

ESEMPIO COSTRUTTORI CON PARAMETRI ED EREDITARIETÀ


```
#include <iostream>
#include <string>

using namespace std;


class Persona{

        private:
                string nome, cognome;
        public:
                Persona()
                {
                        nome = "";
                        cognome = "";
                }
                Persona(string n, string c)
                {
                        nome = n;
                        cognome = c;
                        cout<<"Costruttore di Persona"<<endl;
                }

};

class Studente: public Persona{

        private:
                int matricola;
        public:
                Studente()
                {
                        matricola = 0;
                }
                Studente(string n, string c, int m):Persona(n,c)
                {
                        matricola = m;
                        cout<<"Costruttore di Studente"<<endl;
                }
};

int main()
{
        Studente s1("Mario", "Rossi", 12345);
}
```

Il costruttore della classe derivata dichiara esplicitamente qual è il costruttore della classe base da invocare e, all'interno del suo corpo, inizializza solo i sui parametri. Gli altri, sono già stati inizializzati dal costruttore della classe base.

COMPILAZIONE ESEMPIO COSTRUTTORI CON PARAMETRI ED EREDITARIETÀ



- Ancora una volta, viene prima invocato il costruttore con parametri della classe base (Persona) che inizializza gli attributi **nome** e **cognome** dello Studente**.**
- Poi, si procede con l'invocazione del costruttore con parametri di Studente, che completa l'oggetto **s1** inizializzando l'unico suo parametro **matricola**.
- Non esplicitare il costruttore della classe base all'interno della dichiarazione del costruttore della classe derivata, comporta l'invocazione automatica del costruttore base di default (avrò uno studente con nome e cognome **vuoti**).

DISTRUTTORI ED EREDITARIETÀ

- In una gerarchia di classi, i distruttori vengono invocati in ordine inverso a quanto visto per i costruttori.
- Quindi, il primo ad essere invocato sarà il distruttore della classe derivata e successivamente, quello della classe base.
- Se non esplicitamente incluso nella dichiarazione della classe, il distruttore viene generato di default dal compilatore.
- Nota bene che il costruttore di default non rilascia la memoria di eventuali membri di tipo **reference**.

ESEMPIO DISTRUTTORE ED EREDITARIETÀ


```
#include <string>

using namespace std;

class Persona{

        private:
                string nome, cognome;
        public:
                Persona()
                {
                        nome = "";
                        cognome = "";
                }
                Persona(string n, string c)
                {
                        nome = n;
                        cognome = c;
                        cout<<"Costruttore di Persona"<<endl;
                }
                ~Persona()
                {
                        cout<<"Distruttore di Persona"<<endl;
                }
};
class Studente: public Persona{

        private:
                int matricola;
                int *ptr;
        public:
                Studente()
                {
                        matricola = 0;
                }
                Studente(string n, string c, int m):Persona(n,c)
                {
                        ptr = new int;
                        matricola = m;
                        cout<<"Costruttore di Studente"<<endl;
                }
                ~Studente()
                {
                        delete ptr;
                        cout<<"Distruttore di Studente"<<endl;
                }
};
int main()
{
        Studente s1("Mario", "Rossi", 12345);
        return 0;
}
```

Nella classe derivata **Studente** è presente un puntatore ad intero. Il distruttore di default avrebbe liberato il solo spazio occupato dall'oggetto e non l'area puntata da ptr. Quest'ultima va deallocata esplicitamente attraverso una **delete** all'interno del distruttore.

COMPILAZIONE ED ESECUZIONE
ESEMPIO DISTRUTTORE ED EREDITARIETÀ


```
osboxes@osboxes:~/Desktop$ g++ Costr_der_param.cpp -o dis
osboxes@osboxes:~/Desktop$ ./dis
Costruttore di Persona
Costruttore di Studente
Distruttore di Studente
Distruttore di Persona
osboxes@osboxes:~/Desktop$
```

- L'invocazione dei distruttori avviene in ordine inverso rispetto a quella dei costruttori.

ESERCIZIO

Definire una classe **Edificio** caratterizzata dalle seguenti variabili membro:

- **superficie** (in m^2) dell'edificio
- **num_piani** dell'edificio.

Implementare i costruttori e metodi di lettura e modifica dei suoi attributi.

Definire una classe **Scolastico** che specializza la classe **Edificio** aggiungendo gli attributi:

- **nome** della scuola
- **num_aule** della scuola
- **num_uffici** della scuola

Oltre ai costruttori e i metodi di lettura e modifica dei suoi attributi, la classe deve essere dotata di una funzione membro **capienza** che, considerato pari a **30** il massimo numero di studenti per aula, restituisca il numero totale di alunni che un edificio scolastico è in grado di ospitare.

Scrivere un programma che inserite 5 scuole, stampa a video il nome delle scuole e la loro capienza complessiva data dalla somma delle singole capienze.

DEFINIZIONE CLASSE EDIFICIO E CLASSE SCOLASTICO


```
#include<iostream>
#include<string.h>

using namespace std;

class Edificio
{
        private:
                float superficie;
                int num_piani;
        public:
                Edificio();
                Edificio(float, int);
                void setSuperficie(float);
                void setNumPiani(int);
                float getSuperficie();
                int getNumPiani();
};

class Scolastico : public Edificio
{
        private:
                string nome;
                int num_aule;
                int num_uffici;
        public:
                Scolastico();
                Scolastico(float, int, string, int, int);
                void setNome(string);
                void setNumAule(int);
                void setNumUffici(int);
                string getNome();
                int getNumAule();
                int getNumUffici();
                int capienza(int);
};
```

COSTRUTTORI CLASSE PADRE VS CLASSE DERIVATA

Costruttore parametrico classe padre:

```
Edificio :: Edificio(float s, int np)
{
        this->superficie = s;
        this->num_piani = np;
}
```

Costruttore parametrico classe derivata:

```
Scolastico :: Scolastico(float s, int np, string n, int na, int nu) : Edificio(s,np)
{
        this->nome = n;
        this->num_aule = na;
        this->num_uffici = nu;
}
```

MAIN()


```
int main()
{
        Scolastico* s = new Scolastico[5];

        float sup;
        int np, na, nu, capTotale=0;
        string n;

        for(int i = 0; i<5 ; i++)
        {

                cout<<"Inserire superficie della scuola"<<endl;
                cin>>sup;
                s[i].setSuperficie(sup);
                cout<<"Inserire i piani della scuola"<<endl;
                cin>>np;
                s[i].setNumPiani(np);
                cout<<"Inserire il nome della scuola"<<endl;
                cin>>n;
                s[i].setNome(n);
                cout<<"Inserire il numero di aule"<<endl;
                cin>>na;
                s[i].setNumAule(na);
                cout<<"Inserire il numero di uffici"<<endl;
                cin>>nu;
                s[i].setNumUffici(nu);
                cout<<"----------------------------------------"<<endl;
                capTotale = capTotale + s[i].capienza(s[i].getNumAule());
        }

        for(int i = 0; i<5 ; i++)
        {
                cout<<s[i].getNome()<<endl;
        }

        cout<<"La capienza complessiva è:"<<capTotale<<endl;
        delete[] s;
}
```

MODIFICA ESERCIZIO EDIFICIO-SCOLASTICO

Apportare le dovute modifiche alle classi in modo che, anche in presenza di una derivazione di tipo **privato**, l'applicazione risulti correttamente funzionante.

ESERCIZIO

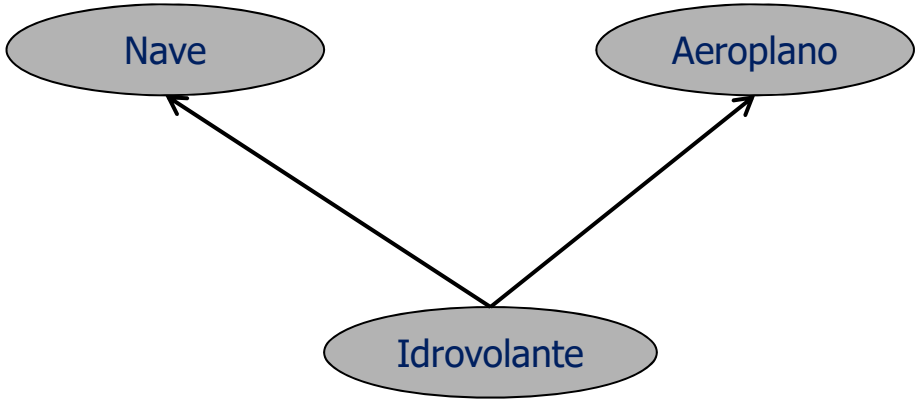
In un negozio ogni **Articolo** in vendita è catalogato in base a **bar_code**, **descrizione** e **prezzo**. Un articolo può appartenere alla categoria **Alimentare** (con parametro anno_di_scadenza) o **Non_Alimentare** (con parametro paese di produzione)**.** Un cliente possessore della tessera fedeltà, ha diritto ad uno sconto del 5% su ogni articolo acquistato.

Realizzare una applicazione che dati 3 articoli acquistati da parte del cliente, stampa a video: descrizione articolo, prezzo articolo (eventualmente scontato) e totale speso.

Implementare l'opportuna gerarchia di classi, i metodi **costruttori** e **1** metodo **sconta()**[funzione membro della classe base]. Risolvere l'esercizio specificando una ereditarietà di tipo **privato.**

EREDITARIETÀ MULTIPLA

- Il linguaggio C++ prevede la possibilità di incorporare all'interno di una classe dati e comportamenti definiti in un numero arbitrario di classi base.



- Idrovolante eredita dati e comportamenti dalle due classi base Nave e Aeroplano.

DICHIARAZIONE DI UNA CLASSE IN PRESENZA DI UNA DERIVAZIONE MULTIPLA


```
class Idrovolante : [tipo_di_accesso] Nave,
                    [tipo_di_accesso]Aeroplano
{
    private:
        //sezione privata di Idrovolante
    public:
        //sezione pubblica di Idrovolante
};
```

- Il tipo_di_accesso può essere **public, private** e **protected**, con le stesse regole viste per l'ereditarietà semplice.

PROBLEMI DELLA DERIVAZIONE MULTIPLA

- La derivazione multipla è uno strumento molto potente che consente al programmatore di stabilire relazioni complesse fra classi, ma ha alcune controindicazioni:
  - **Ambiguità dei nomi**
  - **Poca efficienza** nella ricerca dei metodi definiti nelle classi
  - **problema della precedenza** fra funzioni o dati

- Supponiamo che sia la classe Nave che la classe Aeroplano abbiano un metodo **vira_a_dx()**. Cosa succede se invoco **vira_a_dx()** su di un oggetto della classe **Idrovolante**?

Il compilatore non saprà quale delle due funzioni invocare.

RISOLUZIONE DELL'AMBIGUITÀ DEI NOMI

- Per risolvere l'ambiguità dei nomi possiamo seguire due strade:
  a) utilizzando l'operatore di risoluzione di visibilità :: all'invocazione del metodo
  b) ridefinire il metodo all'interno della classe, specificando il metodo della classe base da richiamare

```
int main(){
        Idrovolante i;
        i.Nave::vira_a_dx();
        i.Aeroplano::vira_a_dx();
}
```

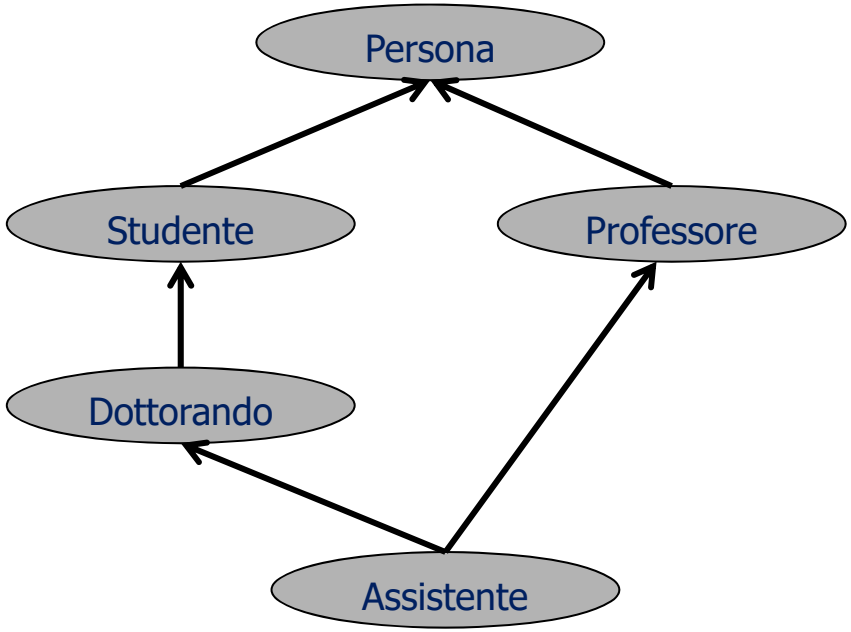
(a)

```
class Idrovolante() : public Nave, public Aeroplano
{
  public:
     void n_vira_a_dx(){  Nave::vira_a_dx()  };
     void a_vira_a_dx(){  Aeroplano::vira_a_dx()  };

};
```

(b)

PROBLEMA DELL'EFFICIENZA

- Consideriamo la seguente gerarchia di classi:



- Nella gerarchica di classi in figura la strada della ricerca lineare per identificare la corrispondenza tra metodi e classi di appartenenza non è più utilizzabile. L'uso di algoritmi di backtracking risolvono il problema, ma con un incremento dei tempi di ricerca rispetto al caso di derivazioni semplici.

PRECEDENZA FUNZIONI IN PRESENZA DERIVAZIONE SEMPLICE

- In presenza di **ereditarietà semplice**, quando una funzione della classe derivata ha lo stesso nome di una funzione della classe base (**anche in presenza di argomenti diversi**), quella della derivata occulta la funzione della classe base.

```
#include<string>
#include<iostream>

using namespace std;

class Base{
        public:
                void f(int);
                void f(string s)
                {
                        cout<<"f della classe base"<<endl;
                }
};

class Derivata : public Base{
        public:
                void f(string s)
                {
                        cout<<"f della classe derivata"<<endl;
                }
};

int main()
{
        Derivata d1;
        d1.f("Hello");
        d1.f(10);
}
```

Nel main() non è possibile invocare su di un oggetto della classe derivata la funzione **f(int)** della classe base, in quanto occultata dalla funzione **f(string)** della derivata.

```
Precedenza.cpp: In function 'int main()':
Precedenza.cpp:28:9: error: no matching function for call to 'Derivata::f(int)'
  d1.f(10);
         ^
```

PRECEDENZA FUNZIONI IN PRESENZA EREDITARIETÀ MULTIPLA

- Quando l'ereditarietà è multipla, alla precedenza si associa anche il problema della ambiguità dei nomi.

- In questi casi è buona norma far compiere chiamate dirette nella classe derivata alle rispettive funzioni della classe base.

- Vediamolo con un esempio.

ESEMPIO PRECEDENZA FUNZIONI IN PRESENZA DI EREDITARIETÀ MULTIPLA


```
#include<string>
#include<iostream>

using namespace std;

class Base1{
        public:
                void f(int);
                void f(string s)
                {
                        cout<<"f della classe base1"<<endl;
                }
};

class Base2{
        public:
                void f(int)
                {
                        cout<<"f della classe base2"<<endl;
                }
                void f(string s);
};
class Derivata : public Base1, public Base2{
        public:
                void f(string s)
                {
                        Base1::f(s);

                }
                void f(int i)
                {
                        Base2::f(i);
                }
};

int main()
{
        Derivata d1;
        d1.f("Hello");
        d1.f(10);
}
```

Una volta specificato nella classe derivata quale funzione delle classi base utilizzare, il compilatore identifica correttamente qual è la funzione che deve essere invocata.

```
osboxes@osboxes:~/Desktop$ g++ Precedenza.cpp -o pre
osboxes@osboxes:~/Desktop$ ./pre
f della classe base1
f della classe base2
osboxes@osboxes:~/Desktop$
```

ESERCIZIO

Definire una classe **Persona** con variabili di stato **nome** e **cognome.**

Ereditare da **Persona** una classe **Studente** con variabili di stato **facoltà** di appartenenza e **crediti** formativi acquisiti.

Ereditare da **Persona** una classe **Lavoratore** con variabili di stato **azienda** di appartenenza e **stipendio** mensile.

Definire una classe **StudenteLavoratore** che eredita le informazioni delle sottoclassi **Studente** e **Lavoratore.**

Dotare tutte le classi degli opportuni **costruttori** e funzioni di **set/get.**

Realizzare un'applicazione che chiede all'utente il numero di studenti-lavoratori da inserire. Effettuare l'inserimento degli studenti-lavoratori da tastiera e stampare a video facoltà e azienda dello studente-lavoratore con il maggior numero di crediti.

MODIFICA ESERCIZIO STUDENTELAVORATORE

1) Stampare a video anche il nome e cognome dello studente-lavoratore con il massimo numero di crediti*

2) Gestire il caso in cui 2 o più studenti abbiano lo stesso numero massimo di crediti.

* **Problema del diamante**

BINDING STATICO

- In un linguaggio di programmazione il termine **binding** indica il tempo in cui avviene il collegamento tra la chiamata ad una funzione e il codice che la implementa.

- Si parla di **binding statico**, noto anche come **early binding**, quando compilatore e linker determinano univocamente la posizione del codice che deve essere eseguito per ogni chiamata di funzione.

- La maggior parte dei linguaggi procedurali adottato questa tipologia di binding.

BINDING DINAMICO

- Il **binding dinamico**, o late binding, stabilisce che l'associazione fra chiamata a funzione e codice da eseguire venga stabilito a tempo di esecuzione.

- Il C++ per default adotta il binding statico, quando si vuole forzare l'uso del binding dinamico, alla dichiarazione della funzione va anteposta la parola chiave **virtual.**

- Il binding dinamico è in generale meno efficiente di quello statico, ma garantisce una maggiore flessibilità nella gestione delle gerarchie di classi.

POLIMORFISMO

- Letteralmente il termine **polimorfismo** indica:

« La facoltà di assumere aspetti, comportamenti, forme diverse a seconda delle circostanze.»

- In un linguaggio di programmazione, il termine polimorfismo può essere descritto come segue:

« Un'interfaccia, molti metodi»

- In questa lezione tratteremo il **polimorfismo in esecuzione** che viene attuato nel linguaggio C++ attraverso l'utilizzo di classi **derivate** e funzioni **virtual**.

FUNZIONI VIRTUAL

- Una funzione si definisce virtuale se nella sua dichiarazione all'interno della classe viene preceduta dalla parola chiave **virtual.**

- All'interno di una classe una funzione va dichiarata **virtual** quando esistono classi derivate da essa che necessitano di ridefinire quella stessa funzione.

- Una funzione si dice **virtual pura** quando all'interno della classe base non è implementata ma solo dichiarata e posta uguale a zero.

virtual tipo_ritorno nome_funzione()=0;

- La presenza, anche solo di una funzione virtual pura, rende la classe corrispondente **astratta.** Questo obbliga **tutte** le classi derivate a fornire la propria implementazione della funzione virtual ereditata.

FUNZIONI VIRTUAL: PURA vs NON PURA

- Una funzione si dice **virtual non pura** quando è anche implementata all'interno della classe base.

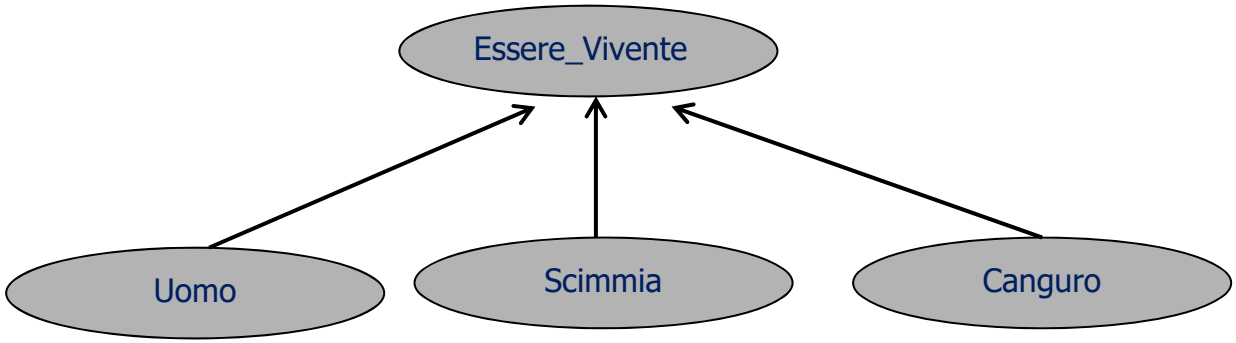
```
virtual tipo_ritorno nome_funzione(){
    //codice della funzione o nessuna istruzione
}
```

- Questo non obbliga tutte le classi derivate a fornirne una loro implementazione.

- Una classe astratta **non può essere istanziata,** cosa che può essere fatta per una classe con sole funzioni virtual non pure.

- Posso in alternativa ricorrere a puntatori del tipo della classe astratta.

ESEMPIO POLIMORFISMO



- Tutti gli esseri viventi camminano, ma ciascuno di essi lo fa in maniera diversa (l'uomo ha una camminata eretta, la scimmia ricurva, un canguro saltella).
- Il polimorfismo a tempo di esecuzione consiste nel determinare a run-time la corretta funzione cammina() da lanciare in relazione all'oggetto che l'ha invocata.
- Vediamo una possibile implementazione della gerarchia in figura.

ESEMPIO POLIMORFISMO: CODICE E COMPILAZIONE


```
#include<iostream>

using namespace std;

class Essere_Vivente{
        public:
                virtual void cammina()=0;
};

class Uomo : public Essere_Vivente{
        public:
                void cammina(){
                        cout<<"L'uomo ha una camminata eretta."<<endl;
                }
};

class Scimmia : public Essere_Vivente{
        public:
                void cammina(){
                        cout<<"La scimmia ha una camminata ricurva."<<endl;
                }
};

class Canguro : public Essere_Vivente{
        public:
                void cammina(){
                        cout<<"Il canguro saltella."<<endl;
                }
};

int main(){
        Essere_Vivente* v[3] = {new Uomo, new Scimmia, new Canguro};
        for(int i=0; i<3; i++)
        {
                v[i]->cammina();
        }
}
```

1. La funzione cammina() della classe base viene dichiarata come **virtual pura;**
2. Ogni classe derivata deve fornire la sua **specifica implementazione** della funzione cammina();
3. Nel main(), dato che Essere_Vivente è una classe **astratta**, non è consentito istanziare un suo oggetto;
4. L'uso di un puntatore a Essere_Vivente consente di identificare a tempo di esecuzione la versione corretta della funzione cammina() da invocare.

```
osboxes@osboxes:~/Desktop$ g++ Polimorfismo.cpp -o pol
osboxes@osboxes:~/Desktop$ ./pol
L'uomo ha una camminata eretta.
La scimmia ha una camminata ricurva.
Il canguro saltella.
osboxes@osboxes:~/Desktop$
```

ESERCIZIO

Definire una classe base **Figura** caratterizzata da:

- **1** parametro intero **n_lati,** gli opportuni **costruttori** e **1** funzione virtual **calcolaArea()** con tipo di ritorno opportuno.

Derivare da **Figura** due classi **Quadrato** e **Rettangolo**:

- **Quadrato è caratterizzata da:**

- **1** parametro double **dim_lato,** gli opportuni **costruttori** e **1** funzione **calcolaArea()** che ritorna l'area del quadrato.

- **Rettangolo**

- **2** parametri double **dim_base** e **dim_altezza,** gli opportuni **costruttori** e **1** funzione **calcolaArea()** che ritorna l'area del rettangolo.

Risolvere l'esercizio verificando il polimorfismo a tempo di esecuzione