

***Pietro Ducange***

***Complementi di programmazione a oggetti in C++***

***a.a. 2020/2021***

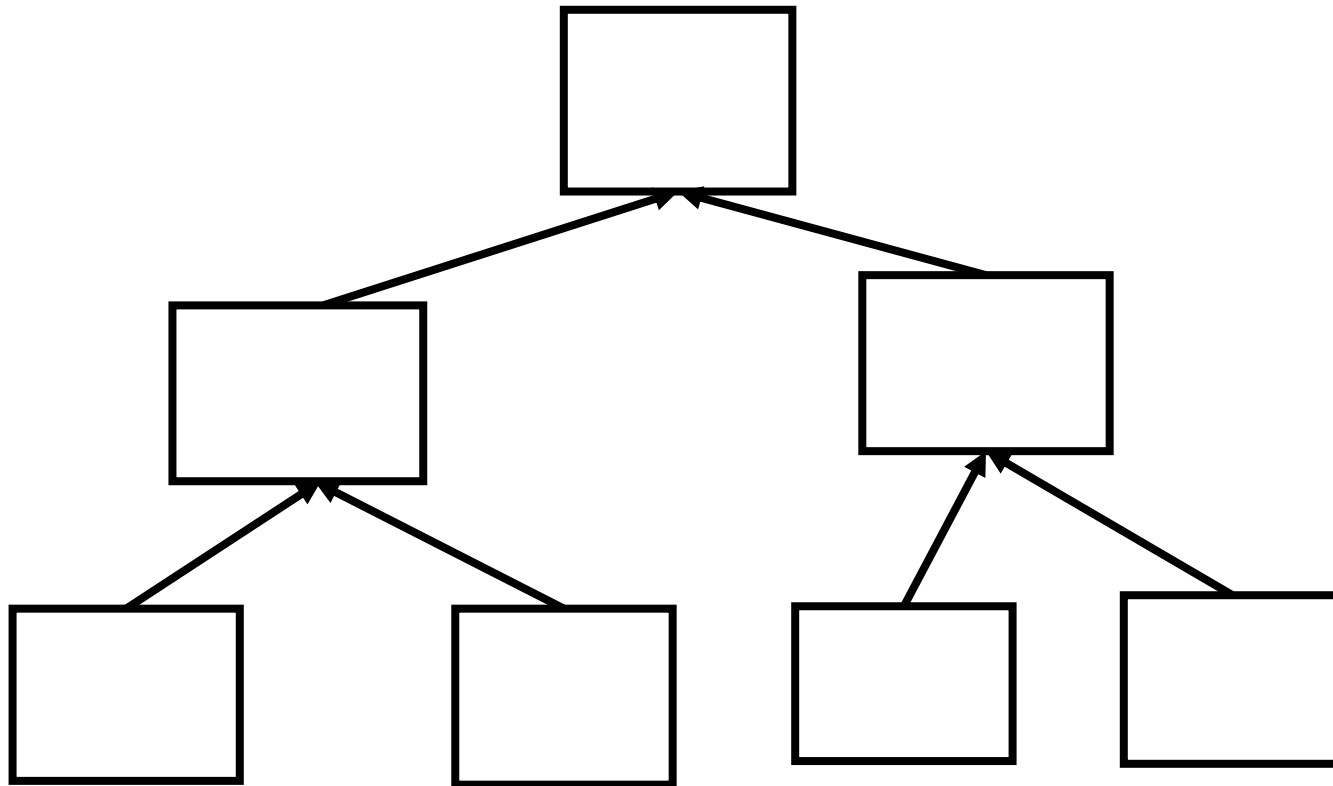
## ***Derivazione semplice***

**Si ringrazia la prof. Nicoletta De Francesco per aver messo a disposizione  
la maggior parte delle slide utilizzate nella presente lezione**

**La derivazione o ereditarietà consente di trasmettere un insieme di caratteristiche comuni da una classe **base** ad una **derivata** senza che ciò comporti una duplicazione del codice, offrendo allo stesso tempo l'opportunità di **adattare** o **estendere** il comportamento a casi d'uso specifici.**

# Gerarchia di classi

**generalizzazione**



**specializzazione**

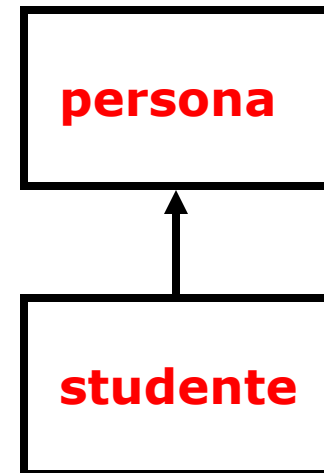
**Attraversare i livelli della gerarchia dall'alto verso il basso significa spostarsi da un livello di astrazione **generico** ad altri sempre più **specifici**.**

# Classi derivate

```
class persona {  
public:  
    char nome [20];  
    int eta;  
};
```

**// classe derivata studente, classe base persona**

```
class studente : public persona {  
public:  
    int esami;  
    int matricola;  
};
```



## Classi derivate

**Un oggetto di una classe derivata ha tutti i campi della classe base più quelli della classe derivata**

|          |               |      |         |
|----------|---------------|------|---------|
| BASE     | char nome[20] | Anna | persona |
|          | int eta       | 22   |         |
| DERIVATA | int esami     | 3    |         |
|          | int matricola | 7777 |         |

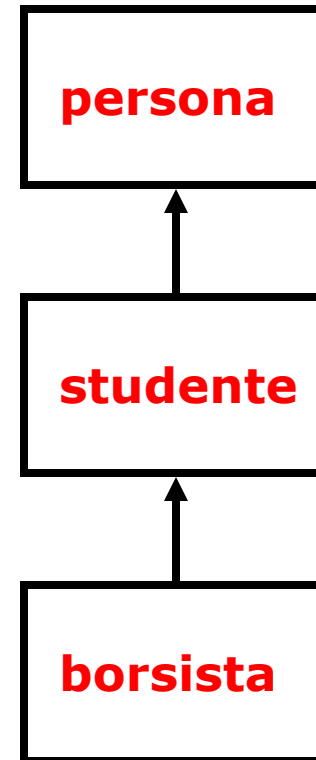
oggetto di tipo studente



# Classi derivate

// classe derivata borsista, classe base studente

```
class borsista : public studente{  
public:  
    int borsa;  
    int durata;  
};
```



## Classi derivate

|          |                |      |
|----------|----------------|------|
| BASE     | char nome [20] | Anna |
|          | int eta        | 22   |
|          | int esami      | 3    |
|          | int matricola  | 7777 |
| DERIVATA | int borsa      | 500  |
|          | int durata     | 3    |

borsista

### Istruzioni possibili

...

**borsista b; borsista \*pb;**

**b.borsa= 500;**

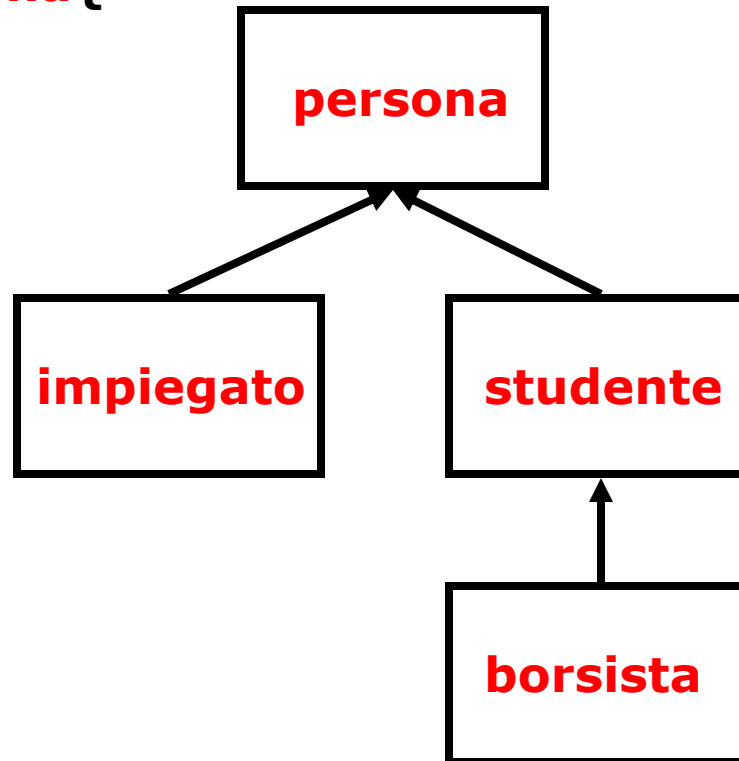
**pb->esami=33;**

**b.eta=22;**

## Classi derivate: gerarchia di classi

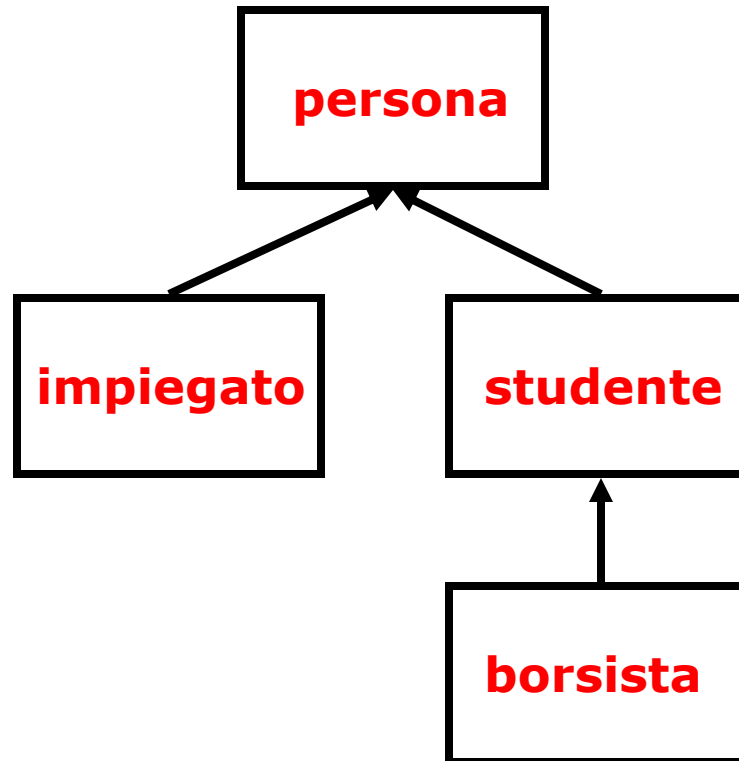
**// classe derivata impiegato, classe base persona**

```
class impiegato : public persona{  
public:  
    int livello;  
    int stipendio;  
};
```



## Classi derivate: esempio di dichiarazioni

```
void main(){  
    persona p;  
  
    studente s;  
  
    impiegato i;  
  
    borsista b;  
}
```

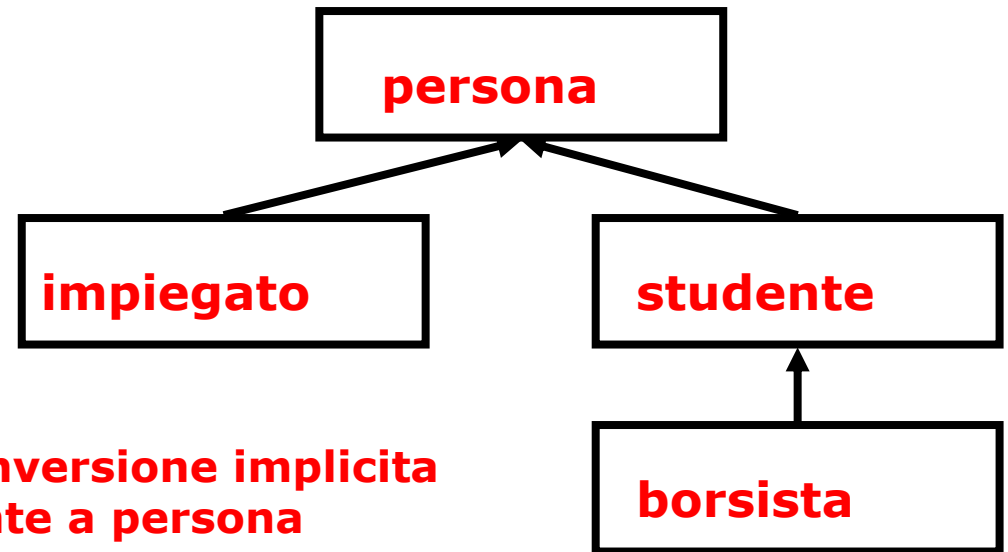


## Classi derivate : compatibilità fra tipi (puntatori)

**Un oggetto (**puntatore ad oggetto**) di un tipo può essere convertito in un supertipo (**puntatore ad un supertipo**), ma non vale il viceversa**

## Classi derivate (cont.): compatibilità fra tipi

```
void main(){  
    persona p;  
  
    studente s;  
  
    impiegato i;  
  
    borsista b;
```



```
p=s;
```

// corretto : conversione implicita  
// da studente a persona

```
// s=p;
```

errato : supertipo assegnato a sottotipo

```
// s=i;
```

errato : tipi diversi

```
p=b;
```

// corretto : conversione implicita  
// da borsista a persona

```
s=b;
```

// corretto : conversione implicita  
// da borsista a studente

```
}
```

## Classi derivate (cont.): compatibilità fra tipi

|           |      |
|-----------|------|
| nome      | Anna |
| eta       | 22   |
| esami     | 3    |
| matricola | 7777 |

**s**

**p=s;**

|      |      |
|------|------|
| nome | Anna |
| eta  | 22   |

**p**

**Nella conversione i campi della classe derivata scompaiono (p ha solo due campi)**

## 5.1 Classi derivate : compatibilità fra tipi (puntatori)

```
void main(){  
    studente s; persona p; borsista b;  
    studente* ps; persona * pp;
```

```
    pp=&p;
```

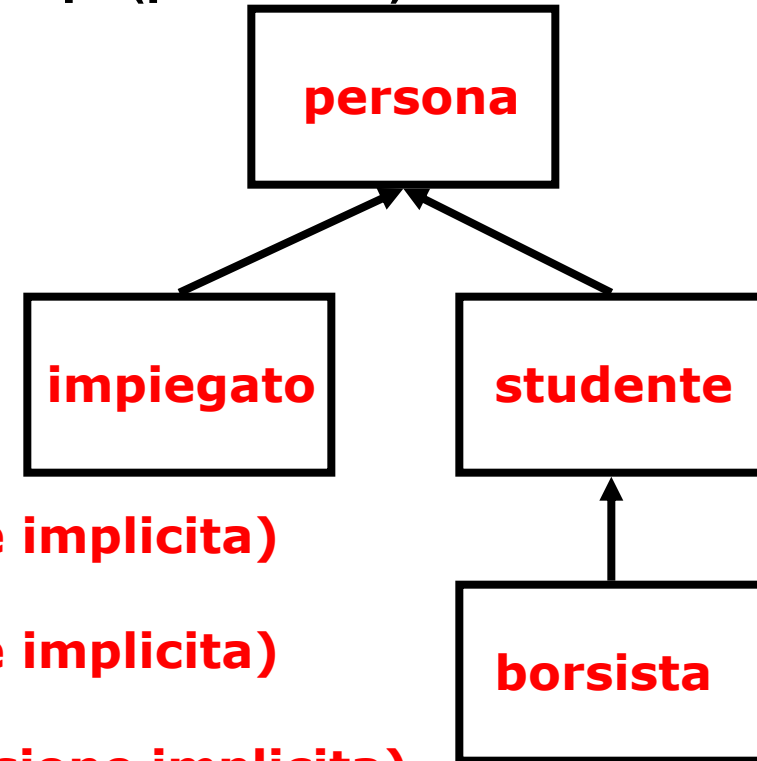
```
    ps =&s           // corretto
```

```
    pp=ps;           // corretto (conversione implicita)
```

```
    pp=&b;           // corretto (conversione implicita)
```

```
    pp=new studente; // corretto (conversione implicita)
```

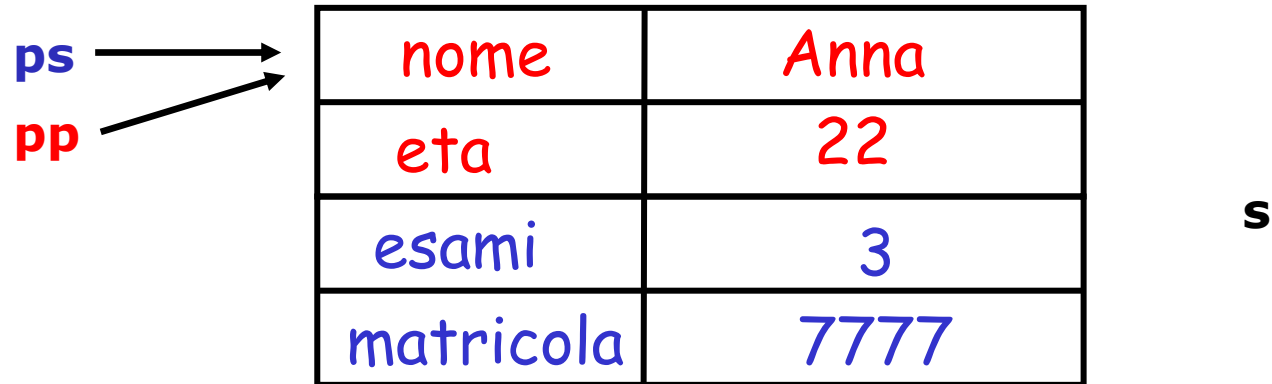
```
    // ps =&p;      errato  
}
```



**Nella conversione i campi non scompaiono ma non sono più accessibili**



## Classi derivate (cont.): compatibilità fra tipi



**pp (tipo \*persona)** e **ps (tipo \* studente)** hanno lo stesso valore, ma possono accedere soltanto ai campi relativi al loro tipo:

Per pp: **pp->nome**, **pp->eta**

Per ps: **ps->nome**, **ps->eta**, **ps->esami**, **ps->matricola**

**pp->esami** **ERRORE**

**La scelta del campo a cui si accede avviene a tempo di compilazione in base al tipo del puntatore**

## Classi derivate con funzioni membro

```
class persona {  
public:  
    char nome [20];  
    int eta;  
    void chisei(){  
        cout << nome << '\t'<< eta << endl;  
    }  
};
```

|                      |
|----------------------|
| <b>char nome[20]</b> |
| <b>int eta</b>       |
| <b>void chisei()</b> |

## Classi derivate con funzioni membro

```
class studente : public persona{  
public:  
    int esami;  
    int matricola;  
    void quantiesami(){  
        cout << esami << endl;  
    }  
};
```

|               |
|---------------|
| nome          |
| eta           |
| chisei()      |
| esami         |
| matricola     |
| quantiesami() |

## Classi derivate con funzioni membro

**// classe derivata borsista**

```
class borsista : public studente{  
public:  
    int borsa;  
    int durata;  
};
```

|                      |
|----------------------|
| <b>nome</b>          |
| <b>eta</b>           |
| <b>chisei()</b>      |
| <b>esami</b>         |
| <b>matricola</b>     |
| <b>quantiesami()</b> |
| <b>borsa</b>         |
| <b>durata</b>        |

## Classi derivate con funzioni membro

```
void main(){
    persona *p;
    studente *s;
    borsista * b;
    // ....
    p->chisei();

    s->chisei();

    b->chisei();

    s->quantiesami();

    b->quantiesami();

    // p->quantiesami()
}
```

|                 |
|-----------------|
| <b>nome</b>     |
| <b>eta</b>      |
| <b>chisei()</b> |

**p**

|                      |
|----------------------|
| <b>nome</b>          |
| <b>eta</b>           |
| <b>chisei()</b>      |
| <b>esami</b>         |
| <b>matricola</b>     |
| <b>quantiesami()</b> |

**s**

|                      |
|----------------------|
| <b>nome</b>          |
| <b>eta</b>           |
| <b>chisei()</b>      |
| <b>esami</b>         |
| <b>matricola</b>     |
| <b>quantiesami()</b> |
| <b>borsa</b>         |
| <b>durata</b>        |

**b**

**errato**

## Regole di visibilità

```
class studente {  
public:  
    int matricola;  
    int esami; // esami sostenuti  
};
```

```
class borsista : public studente{  
public:  
    int borsa;  
    int durata;  
    int esami; // esami dall'inizio della borsa  
};
```

**borsista**

|                  |
|------------------|
| <b>matricola</b> |
| <b>esami</b>     |
| <b>borsa</b>     |
| <b>durata</b>    |
| <b>esami</b>     |

**Ma un borsista può accedere a esami di studente  
con risolutore di visibilità**

## Regole di visibilità

```
void main(){
    studente * s=new studente;
    borsista * b=new borsista;

    b->esami=4;                // = b.borsista::esami

    b->studente::esami=5;      // risolutore di visibilità

    cout << b->esami;          // 4

    s=b;                       // conversione

    cout << s->esami;          // 5
}
```

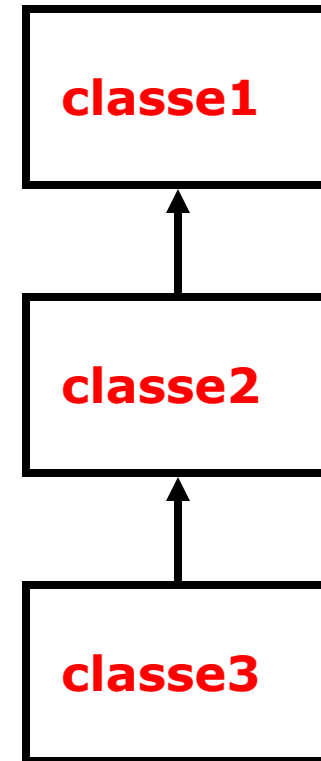
|           |   |
|-----------|---|
| matricola |   |
| esami     | 5 |
| borsa     |   |
| durata    |   |
| esami     | 4 |

## Regole di visibilità

```
class classe1 {  
public:  
    int a;  
    //..  
};
```

```
class classe2 : public classe1{  
public:  
    int a;  
    //..  
};
```

```
class classe3 : public classe2{  
public:  
    int a;  
    //..  
};
```





## Regole di visibilità

```
void main(){
  classe3 obj;
  obj.a=2;           // obj.classe3::a
  obj.classe1::a=7;
  obj.classe2::a=8;
```

|   |   |            |
|---|---|------------|
| a | 7 | classe1::a |
| a | 8 | classe2::a |
| a | 2 | classe3::a |

```
  cout << obj.a;           // 2           obj
  cout << obj.classe1::a;   // 7 risolutore di visibilità
  cout << obj.classe2::a;   // 8 risolutore di visibilità
```

...

## Regole di visibilità (puntatori)

**classe1\* p1=&obj;**

**// conversione**

**classe2\* p2=&obj;**

**// conversione**

**classe3\* p3=&obj;**

**cout << p1->a;**

**// 7**

**p1->a : i**

**cout << p2->a;**

**// 8**

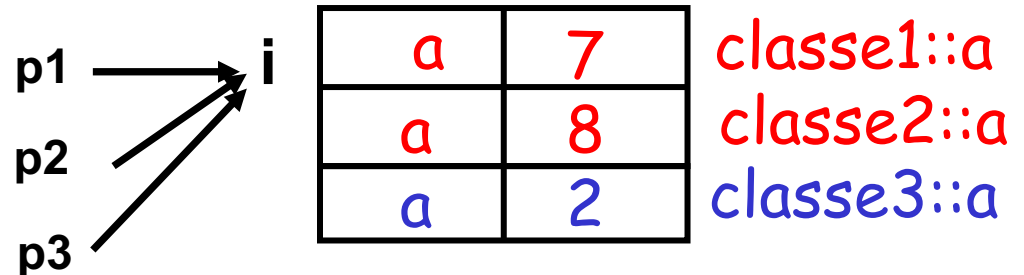
**p2->a : i+1**

**cout << p3->a;**

**// 2**

**p3->a : i+2**

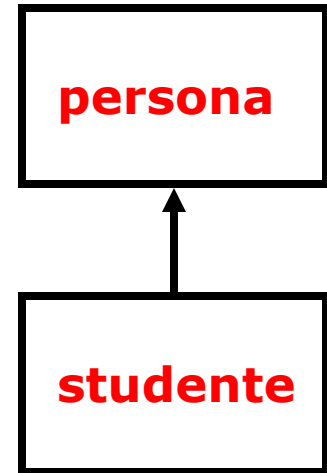
**}**



## Regole di visibilità (funzioni membro)

```
class persona {  
public:  
    char nome [20];  
    int eta;  
    void chisei(){  
        cout << nome << '\\t'<< eta << endl;  
    }  
};
```

```
class studente : public persona{  
public:  
    int esami;  
    int matricola;  
    void chisei(){  
        cout << nome << '\\t'<< eta << '\\t'  
            << matricola <<  
            '\\t'<< esami << endl;  
    }  
};
```



## Regole di visibilità (funzioni membro)

```
void main(){
    studente s;
    strcpy(s.nome, "anna"); s.eta=22;
    s.esami=3; s.matricola=444444;

    s.chisei();          // anna 22 444444 3
                        // chiamata a studente::chisei()

    s.persona::chisei(); // anna 22

    persona *p=&s;

    p->chisei();         // anna 22
}
```

|           |
|-----------|
| nome      |
| eta       |
| chisei()  |
| esami     |
| matricola |
| chisei()  |

s

## Regole di visibilità (funzioni membro)

|          |           |      |   |
|----------|-----------|------|---|
| persona  | nome      | Anna |   |
|          | eta       | 22   |   |
|          | chisei()  |      |  |
| studente | esami     | 3    |   |
|          | matricola | 4444 |   |
|          | chisei()  |      |  |

## Regole di visibilità (funzioni membro)

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

```
class uno {  
    // ..  
    public:  
        uno() { }  
        void f(int) {  
            cout << "uno";  
        }  
};
```

```
class due: public uno {  
    //..  
    public:  
        due() {}  
        void f() {  
            cout << "due";  
        }  
};
```

```
void main () {  
    due* p= new due;  
    // p->f(6);    errore  
    p->uno::f(6); // uno  
    p->f();        // due  
}
```

## Specificatori di accesso

I campi **privati** di una classe non sono accessibili dalle sottoclassi nè dall'esterno

I campi **protetti** di una classe sono accessibili dalle sottoclassi, ma non dall'esterno

I campi **pubblici** di una classe sono accessibili anche dall'esterno

I campi mantengono la stessa specifica in tutta la gerarchia.

## Specificatori di accesso

**I campi pubblici di una classe sono accessibili dalle sottoclassi e dall'esterno**

```
class uno {  
public:  
    int x;  
};
```

```
class due : public uno{  
public:  
    int y;  
    void f() {x=5; y=6; } // corretto perchè x è pubblico  
};
```

```
due * s = new due;
```

```
s->x=2 ; // corretto perchè x è pubblico
```



## Specificatori di accesso

**I campi privati di una classe non sono accessibili dalle sottoclassi**

```
class uno {  
    int x;  
};
```

```
class due : public uno{  
    int y;  
    void f() {x=5; y=6; } // no perchè x è privato di uno  
};
```

## Membri protetti

I campi **protetti** di una classe sono accessibili dalle sottoclassi

```
class uno {  
    protected:  
        int x;  
};
```

```
class due : public uno{  
    int y;  
    void f() {x=5; y=6; } // ok perchè x è protetto  
};
```

```
due * s = new due;
```

```
s->x=2 ; // no perchè x è protetto ma non pubblico
```

## Costruzione degli oggetti

**Quando un oggetto di una classe derivata viene costruito si costruisce prima la parte **BASE** e poi la parte **DERIVATA**.**

# Costruzione di un oggetto della classe O

## **COSTRUZIONE(O):**

- **se O deriva da una classe base B: COSTRUZIONE (B);**
- **si costruiscono i campi di O chiamando gli opportuni costruttori nel caso che siano oggetti;**
- **si chiama il costruttore di O;**

## Costruzione con membri oggetti

**Se la classe base ha più costruttori, il costruttore di una classe derivata deve chiamarne uno nella lista di inizializzazione.**

**Può non chiamarlo esplicitamente se la classe base ha un costruttore di default, che in questo caso viene chiamato automaticamente.**

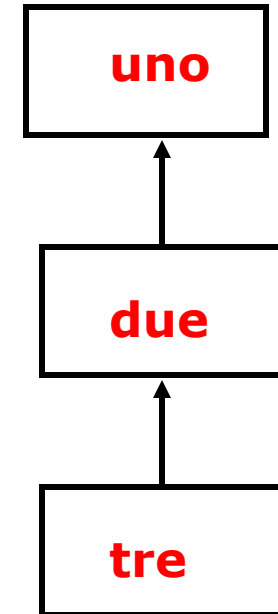
## Costruttori

```
class uno {  
public:  
    uno(){cout << "nuovo uno" << endl;}  
};
```

```
class due: public uno {  
public:  
    due() {cout << "nuovo due"<< endl;}  
};
```

```
class tre: public due {  
public:  
    tre() {cout << "nuovo tre"<< endl;}  
};
```

```
void main (){  
    due obj2;    // nuovo uno  
                // nuovo due  
    tre obj3;    // nuovo uno  
                // nuovo due  
                // nuovo tre  
}
```

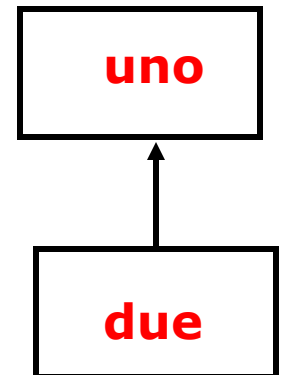


# Costruttori

```
class uno {  
protected:  
    int a;  
public:  
    uno() {a=5; cout << "nuovo uno" << a << endl;}  
    uno(int x) {a=x; cout << "nuovo uno" << a << endl;}  
};
```

```
class due: public uno {  
    int b;  
public:  
    due(int x) {b=x; cout << "nuovo due" << x << endl;}  
};
```

```
void main (){  
    due obj2(8);    // nuovo uno 5  
                   // nuovo due 8  
}
```



## costruzione di obj2

```
due obj2(8);
```

chiamata a **uno::uno()**

|          |          |
|----------|----------|
| <b>a</b> | <b>5</b> |
|----------|----------|

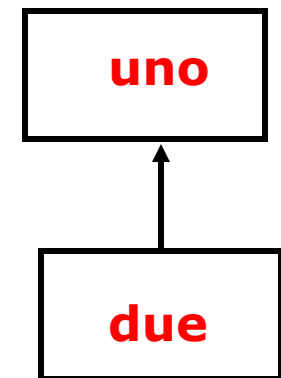
chiamata a **due::due(8)**

|          |          |
|----------|----------|
| <b>a</b> | <b>5</b> |
| <b>b</b> | <b>8</b> |



# Costruttori

```
class uno {  
protected:  
    int a;  
Public:  
    uno() {a=5; cout << "nuovo uno" << a << endl;}  
    uno(int x) {a=x; cout << "nuovo uno" << a << endl;}  
};  
  
class due: public uno {  
    int b;  
public:  
    due(int x): uno(x+1) {b=x; cout << "nuovo due" << x << endl;}  
};  
  
void main (){  
    due obj2(8);    // nuovo uno 9  
                   // nuovo due 8  
}
```



## Costruzione di obj2

```
due obj2(8);
```

chiamata a **uno::uno(9)**

|          |          |
|----------|----------|
| <b>a</b> | <b>9</b> |
|----------|----------|

chiamata a **due::due(8)**

|          |          |
|----------|----------|
| <b>a</b> | <b>9</b> |
| <b>b</b> | <b>8</b> |

# Costruttori

```
class uno {
```

```
public:
```

```
    uno(int x) {cout << "nuovo uno" << endl;}
```

```
};
```

```
class due: public uno {
```

```
public:
```

```
    // due(int x) {...} ERRORE: manca il costruttore di default
```

```
    // nella classe uno
```

```
};
```

## **ORDINE DI CHIAMATA DEI COSTRUTTORI PER UNA GERARCHIA A DUE LIVELLI**

- 1. costruttori degli oggetti membri della classe base**
- 2. costruttore della classe base**
- 3. costruttori degli oggetti membri della classe derivata**
- 4. costruttore della classe derivata**

## Con membri oggetto

```
class uno {  
public:  
  uno() {  
    cout << "nuovo uno "  
          << endl;  
  }  
};  
class due {  
  uno a;  
public:  
  due() {  
    cout << "nuovo due "  
          << endl;  
  }  
};  
class tre: public due {  
  uno b;  
public:  
  tre() { cout << "nuovo tre" << endl; }  
};
```

```
void main () {  
  tre obj;  
}  
  
nuovo uno // uno::uno() per a  
nuovo due // due::due() per obj  
nuovo uno // uno::uno() per b  
nuovo tre // tre::tre() per obj
```

|              |  |
|--------------|--|
| <b>uno a</b> |  |
| <b>uno b</b> |  |

## Distruzione degli oggetti

**Quando un oggetto di una classe derivata viene distrutto viene distrutta prima la parte **DERIVATA** e poi la parte **BASE**.**

# **Distruzione di un oggetto della classe O**

## **DISTRUZIONE(O):**

- **i campi di O vengono distrutti;**
- **viene chiamato il distruttore di O;**
- **se O deriva da una classe base B: DISTRUZIONE (B);**

## Distruttori

```
class uno {
public:
    uno();
    ~uno();
};

uno::uno(){cout << "nuovo uno" << endl;}
uno::~~uno(){cout << "via uno" << endl;}

class due: public uno {
public:
    due();
    ~due();
};

due::due(){cout << "nuovo due" << endl;}
due::~~due(){cout << "via due" << endl;}
```

```
void main (){
    due obj2;
    // nuovo uno
    // nuovo due

    // via due
    // via uno
}
```



## Membri statici

```
class A {  
public:  
    static int quantiA;  
    A(){  
        cout << "A = "  
        << ++quantiA << endl;}  
};
```

```
int A::quantiA=0;
```

```
class B : public A{  
public:  
    static int quantiB;  
    B(){  
        cout << "B = "  
        << ++quantiB << endl;}  
};
```

```
int B::quantiB=0;
```

```
void main(){  
    A p1;  
                                     // A = 1  
    B s1;  
                                     // A = 2  
                                     // B = 1  
    A p2;  
                                     // A = 3  
    B s2;  
                                     // A = 4  
                                     // B = 2  
}
```