

***Complementi di programmazione***

***a oggetti in C++***

***a.a. 2019/2020***

# *Funzioni e classi modello*

## 4.2 Funzioni modello

```
int i_max(int x, int y) {  
    return (x>y) ? x : y;  
};  
  
double d_max(double x, double y) {  
    return (x>y) ? x : y;  
};  
  
void main() {  
    int b; double c;  
    // ...  
    a= i_max(3,b);  
    d=d_max(3.6,c);  
}
```

**Le due funzioni hanno la stessa definizione con tipi diversi**

## 4.2 Funzioni modello: costruito template

```
#include<iostream.h>
template<class tipo>
tipo max(tipo x, tipo y) {
    return (x>y) ? x : y;
}

void main() {
    int b; double c;
    // ...
    b= max(3,b);

    // tipo=int max<int>(int,int)

    c=max(3.6,c);

    // tipo = double max<double>(double,double)
}
```

## 4.2 Funzioni modello: compilazione

### Risultato della compilazione

```
a = max(3,b);
```



**tipo=int**

```
max<int>(int x, int y) { return (x>y) ? x : y; }
```

```
d = max(3.6,c);
```



**tipo=double**

```
max<double>(double x, double y) { return (x>y) ? x : y; }
```

## 4.2 Funzioni modello: argomenti impliciti

```
template<class tipo>
```

```
tipo max(tipo x, tipo y) .....
```

```
void main() {
```

```
    int b=2; double c=6.0, d; int array[2]={3,4};
```

```
    cout << max(array[0],b); // OK: int max<int >(int,int)
```

```
    d = max(3.6,c);
```

```
        // OK: double max<double>(double, double)
```

```
    b = max(3.6,c);
```

```
    // OK: double max<double>(double, double) e conversione
```

```
    // d = max(3,c); errore: non si deduce il tipo:
```

```
        // 3 e' intero, c e' double
```

```
}
```

**I tipi devono essere deducibili dalla chiamata**

## 4.2 Esempio di funzione modello

```
template<class tipo>
void primo ( tipo *x ) {
    tipo y= x[0] ;
    cout << y << endl;
};

void main() {
    int array1[2]={3,4};
    double array2[2]={3.5,4.8};

    primo(array1);

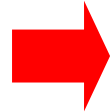
    // 3 tipo=int void primo<int>(int*)

    primo(array2);

    // 3.5 tipo=double void primo<double>(double*)
}
```

## 4.2 Esempi di funzioni modello (cont.)

`primo(array1);`



```
void primo<int> ( int *x ) {  
    int y= x[0] ;  
    cout << y << endl;  
};
```

`primo(array2);`



```
void primo< double > (double *x ) {  
    double y= x[0] ;  
    cout << y << endl;  
};
```



## 4.2 Esempi di funzioni modello

```
template<class tipo>  
void primo ( tipo x ) {  
    cout << x[0] << endl;  
};
```

```
void main() {  
    int array1[2]={3,4};  
    double array2[2]={3.5,4.8};  
  
    primo(array1);
```

```
        // 3   tipo=int*   void primo<int*>(int*)
```

```
    primo(array2);
```

```
        // 3.5  tipo=double* void primo<double*>(double*)
```

```
    }
```

## 4.2 funzioni modello con più parametri

```
template<class tipo1, class tipo2>  
tipo1 max(tipo1 x, tipo2 y) {  
    return (x>y) ? x : y;  
}
```

```
void main() {  
  
    int b=2; double c=6;  
  
    cout << max(3,b); // int max<int,int>(int,int)  
  
        // tipo1=int, tipo2= int  
  
    b = max(3,c); // int max<int,double>(int,double)  
  
        // tipo1=int, tipo2= double  
  
}
```

## 4.2 funzioni modello con più parametri

```
template<class tipo1, class tipo2, class tipo3>  
tipo1 nuovomax(tipo2 x, tipo3 y) {  
    return (x>y) ? x : y;  
}
```

```
void main() {  
    int b; double c=6;  
  
    b = nuovomax(3,c);
```

```
        // NO: tipo1=? , tipo2=int, tipo3=double
```

```
}
```

## 4.4 Funzioni modello: parametri espliciti

```
template<class tipo>  
tipo max(tipo x, tipo y) {  
    return (x>y) ? x : y;  
}
```

```
void main() {  
    double d;  
    cout << max<int>(3,5.5);  
        // 5 max<int>(int,int); conversione del parametro  
  
    cout << max<double>(3,5.5);  
        // 5.5 max<double>(double,double) conversione  
        //del parametro  
  
    d= max<int>(3,5.5);  
        // max<int>(int,int); conversione del valore  
        // assegnato: 5.0  
  
}
```

## 4.4 Funzioni modello: parametri espliciti e impliciti

```
template<class tipo1, class tipo2, class tipo3>  
tipo1 fun(tipo2 x, tipo3 y) {  
    .....
```

Gli argomenti espliciti sono indicati nell'ordine del template

```
fun<int>(9,8.8); // tipo1= int : int fun<int,int,double>
```

```
fun<int,double>(9,8.8); // tipo1=int, tipo2=double :  
int fun<int,double,double>
```

```
fun<int,int,double>(.,..); // int fun<int,int,double>
```

```
fun(9,8); // errore tipo1=tipo2=int, tipo1?
```

## 4.2 Funzioni modello: parametri costanti

```
template<int n, double m >  
void funzione(int x=n){  
double y=m;  
int array[n];  
.....  
}
```

```
void main () {  
funzione<1+2,2>(8); // n=3, m=2      funzione<3,2>(int)  
  
funzione<2,2>(9); // n=2, m=2      funzione<2,2>(int)  
  
}
```

**I parametri costanti sono necessariamente espliciti:**

**Le istanziazioni di n e m devono essere ESPRESSIONI COSTANTI**

## 4.2 Funzioni modello: parametri costanti (cont.)

**funzione<1+2,2>(8);**



```
void funzione<3,2>(int x=3){  
double y=2;  
int array[3];  
.....  
}
```

**funzione<2,2>(9);**



```
void funzione<2,2>(int x=2){  
double y=2;  
int array[2];  
.....  
}
```

## 4.2 Funzioni modello: parametri costanti e no

```
template< int n, class T>  
int gt(T x){  
  return x>n;  
}
```

```
void main(){  
  cout << gt<50+6>(101);  
  
  // 1  n=56, T=int  int gt<56,int>(int)  
  
  // risoluzione implicita di T  
  
  cout << gt<8, double>(7);  
  
  // 0  n=8, T=double  int gt<8, double>(double)  
  
  // risoluzione esplicita di T
```



## 4.2 Funzioni modello: parametri costanti e no (cont.)

`gt<50+6>(101);`



```
int gt<56,int>(int x){  
    return x>56;  
}
```

`gt<8, double>(7);`



```
int gt<8,double>(double x){  
    return x>8;  
}
```

## Funzioni modello con variabili statiche

```
template<class tipo>
tipo max(tipo x, tipo y) {
    static int a; a++; cout << a << endl;
    return (x>y) ? x : y;
}
```

```
void main(){
```

```
cout << max<int>(101,102) << endl;           // 1 102
cout << max<int>(101,102)<< endl;           // 2 102
cout << max<double>(101,102) << endl;       // 1 102
}
```

Ogni istanza ha la sua variabile statica

## 4.2 Dichiarazione e definizione di template

```
// file templ.h
```

```
template<class tipo>  
void boh(tipo x){  
    // ... definizione  
}
```

```
// file main
```

```
#include"templ.h"
```

```
void main() {  
    //..  
    boh(6);  
  
    // ..  
}
```

**Una funzione modello non può essere compilata senza conoscere le chiamate**

**Anche le classi possono essere definite come classi modello:**

```
template<class tipo1, class tipo2, int n .....>  
class obj { ....
```

**I parametri in questo caso sono sempre  
espliciti**

## 4.1 Classi modello: stack

```
class stack{
    int size;
    int * p;
    int top;
public:
    stack(int n){
        size = n;
        p = new int [n];
        top = -1;
    };

    ~stack() { delete [] p; };

    int empty(){
        return (top==-1); };

    int full(){
        return (top==size-1); }; };
```

```
int push(int s){
    if (top==size-1) return 0;
    p[++top] = s;
    return 1;
};

int pop(int& s){
    if (top==-1) return 0;
    s = p[top--];
    return 1;
}
```

## 4.2 stack modello parametrico rispetto al tipo degli elementi

```
//file stack.h
```

```
template<class tipo>
```

```
class stack {
```

```
    int size;
```

```
    tipo* p;
```

```
    int top;
```

```
public:
```

```
    stack(int n){
```

```
        size = n;
```

```
        p = new tipo [n];
```

```
        top = -1; };
```

```
    ~stack() { delete [] p; };
```

```
    int empty() {
```

```
        return (top == -1);};
```

```
    int full() {
```

```
        return (top == size-1);};
```

```
int push(tipo s) {
```

```
    if (top == size-1) return 0;
```

```
    p[++top] = s;
```

```
    return 1; };
```

```
int pop(tipo& s){
```

```
    if (top == -1) return 0;
```

```
    s = p[top--];
```

```
    return 1; }
```

```
};
```

## 4.2 stack modello parametrico rispetto al tipo degli elementi

```
#include"stack.h"  
  
void main(){  
  
    stack<int> s1 (20), s2 (30);  
    stack<char> s3 (10);  
    stack<float> s4 (20);  
  
    s1.push(3);  
  
    s3.push('a');  
  
    s4.push(4.5);  
  
}
```

## 4.2 stack modello parametrico rispetto al tipo degli elementi

```
class persona {
    char nome [20];
    int eta;
public:
    persona() {}
    persona (char* n, int e){
        strcpy(nome,n);
        eta=e;}
};

void main() {
    persona p ("anna",22);
    stack<persona> pila(10);
    pila.push(p);
}
```



## 4.1 Classi modello

```
class stack{
    int size;
    int * p;
    int top;
public:
    stack(int);
    ~ stack();
    int empty();
    int full();
    int push(int);
    int pop(int&);
};

stack::stack(int n){
    size = n;
    p = new int [n];
    top = -1; }
```

```
stack::~~stack(){ delete [] p; }
int stack::empty(){
    return (top== -1); }
int stack::full(){
    return (top==size-1); }
int stack::push(int s){
    if (top==size-1) return 0;
    p[++ top] = s;
    return 1; }
int stack::pop(int& s){
    if (top== -1) return 0;
    s =p[top--];
    return 1; }
```

## 4.1 Classi modello

```
// file stack.h
```

```
template<class tipo>
```

```
class stack{
```

```
    int size;
```

```
    tipo * p;
```

```
    int top;
```

```
public:
```

```
    stack(int);
```

```
    ~ stack();
```

```
    int empty();
```

```
    int full();
```

```
    int push(tipo);
```

```
    int pop(tipo&);
```

```
};
```

```
template<class tipo>
```

```
stack<tipo>::stack(int n){
```

```
    size = n;
```

```
    p = new tipo [n];
```

```
    top = -1; }
```

## 4.1 Classi modello (cont.)

```
template<class tipo>  
stack<tipo>::~~stack(){ delete [] p; }
```

```
template<class tipo>  
int stack<tipo>::empty(){ return (top==-1); }
```

```
template<class tipo>  
int stack<tipo>::full(){ return (top==size-1); }
```

```
template<class tipo>  
int stack<tipo>::push( tipo s ){  
    if (top==size-1) return 0;  
    p[++top] = s;  
    return 1; }
```

```
template<class tipo>  
int stack<tipo>::pop( tipo& s ){  
    if (top==-1) return 0;  
    s = p[top--];  
    return 1; }
```

## 4.1 Classi modello

```
// file stack.h  
// contiene dichiarazioni e definizioni
```

```
template<class tipo>  
class stack{  
    // ..  
public:  
    ...  
};  
  
// definizioni
```

```
// file principale
```

```
# include"stack.h"
```

```
...
```

**bisogna includere  
anche la definizione**

## 4.1 stack parametrico anche rispetto alla dimensione

```
template<class tipo, int size>
class stack {
    tipo* p;
    int top;
public:
    stack() {
        p = new tipo [size];
        top = -1; };

    ~stack() { delete [] p; };

    int empty() {
        return (top == -1);};
    int full() {
        return (top == size-1);};

    int push(tipo s) {
        if (top == size-1) return 0;
        p[++top] = s;
        return 1; };
```

```
int pop(tipo& s){
    if (top == -1) return 0;
    s = p[top--];
    return 1; }
};

...

stack<int,10> pila1;
stack<double,20> pila2;

stack<char,20> pila3;
stack<int,20> pila4;
```

## 4.1 esempio

```
template<class tipo, int size>  
class stack {  
// ..  
  
};
```

...

```
stack<int, 100> pila1;  
stack<int, 300> pila2;
```

```
stack<int,100>* ptr = &pila1;
```

```
// ptr = &pila2; errore
```

**stack<int, 300> e stack<int, 100> sono tipi diversi**

## Classi modello con membri statici

```
template<int n>
class cmod{
    static int istanze;
    int m;
public:
    cmod();
    void stampa();
};
template<int n>
int cmod<n>::istanze=0;

template<int n>
cmod <n>::cmod(){
    m=n;
    istanze ++;
};

template<int n>
void cmod <n>::stampa(){
    cout << istanze << '\\t' << m << endl;
}
```

```
void main(){
    cmod<9> nove_a;
    nove_a.stampa();

    // 1 9

    cmod<7> sette;
    sette.stampa();

    // 1 7

    cmod<9> nove_b;
    nove_b.stampa();

    // 2 9
}
```

## *Derivazione semplice*

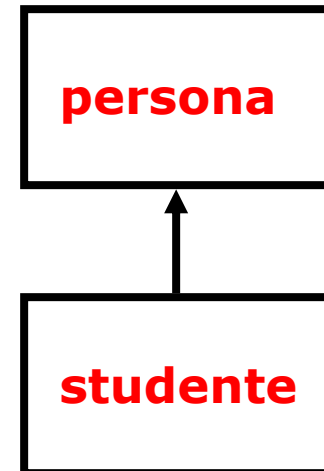


## 5.1 Classi derivate

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

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

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



## 5.1 Classi derivate

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

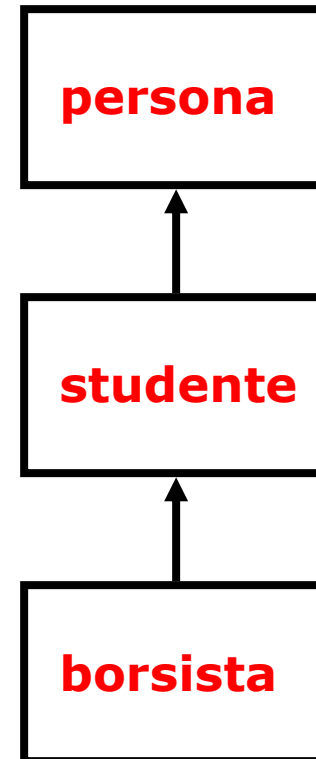
|          |               |      |         |
|----------|---------------|------|---------|
| BASE     | char nome[20] | Anna | persona |
|          | int eta       | 22   |         |
| DERIVATA | int esami     | 3    |         |
|          | int matricola | 7777 |         |

oggetto di tipo studente

## 5.1 Classi derivate

// classe derivata borsista, classe base studente

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



## 5.1 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;
```

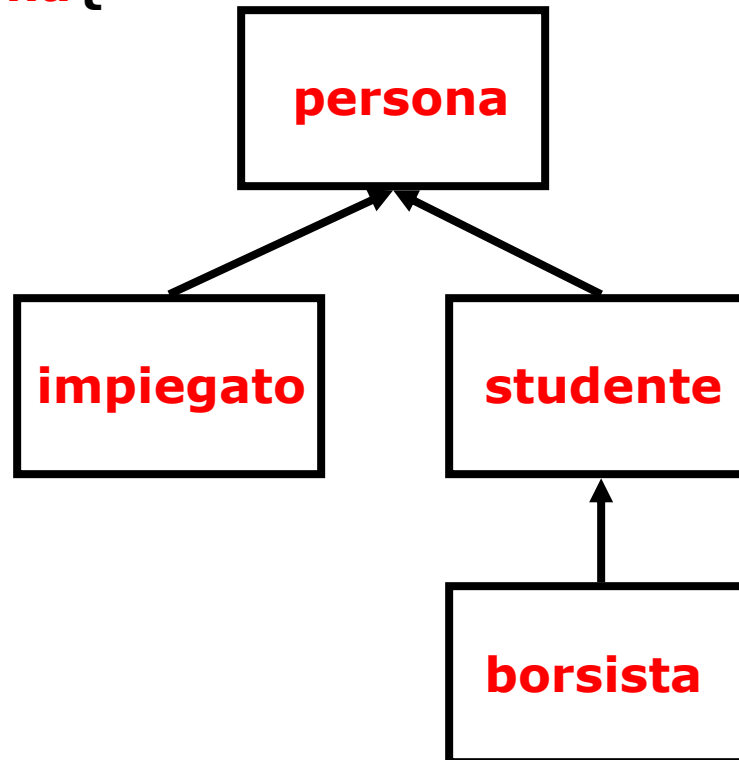
```
pb->esami=33;
```

```
b.eta=22;
```

## 5.1 Classi derivate: gerarchia di classi

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

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



## 5.1 Classi derivate

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

## 5.1 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**

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

```
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  
  
}
```



## 5.1 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;           // corretto

    pp=&s;           // corretto (conversione implicita)

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

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

    // ps =&p;           errato
    // ps -> esami;     errato
}
```

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

## 5.1 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> |

## 5.1 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() |

## 5.1 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>        |

## 5.1 Classi derivate con funzioni membro

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

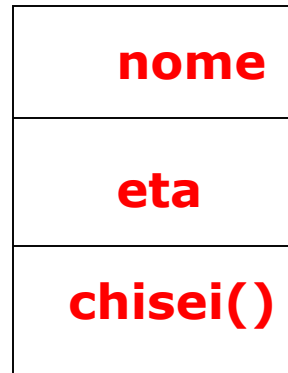
  s->chisei();

  b->chisei();

  s->quantiesami();

  b->quantiesami();

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



p



s

errato



b

## 5.1.1 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**

## 5.1.1 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
}
```

|                  |          |
|------------------|----------|
| <b>matricola</b> |          |
| <b>esami</b>     | <b>5</b> |
| <b>borsa</b>     |          |
| <b>durata</b>    |          |
| <b>esami</b>     | <b>4</b> |

**valgono le stesse regole dei blocchi**

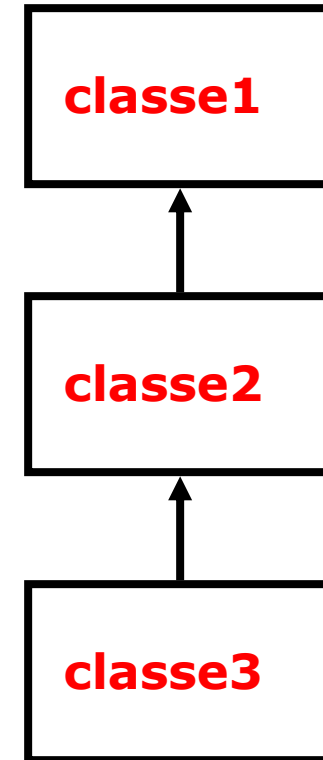


## 5.1.1 Regole di visibilità

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

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

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



## 5.1.1 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 |

obj

```
cout << obj.a;           // 2
```

```
cout << obj.classe1::a; // 7
```

```
cout << obj.classe2::a; // 8
```

...

## 5.1.1 Regole di visibilità (puntatori)

```
classe1* p1=&obj;           // conversione
classe2* p2=&obj;           // conversione
classe3* p3=&obj;
cout << p1->a;              // 7
cout << p2->a;              // 8
cout << p3->a;              // 2
}
```

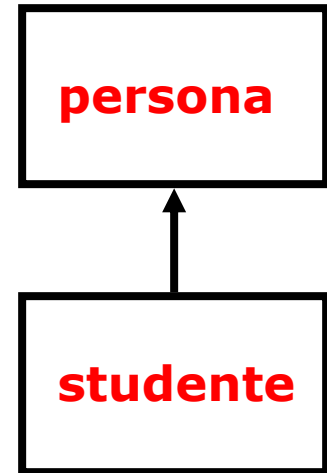
p1->a : i  
p2->a : i+1  
p3->a : i+2

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

## 5.1.1 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;  
    }  
};
```



## 5.1.1 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

## 5.1.1 Regole di visibilità (funzioni membro)

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

## 5.1.1 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
}
```

**no overloading per  
funzioni  
appartenenti a classi  
diverse**

## 5.2 Specificatori di accesso

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

```
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
```



## 5.2 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  
};
```

```
due * s = new due;  
s->x=2 ; // no perchè x è privato di uno  
s->y; // no perchè y è privato di due
```

## 5.2 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
```

## 5.2 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 privati, protetti e pubblici rimangono tali in tutta la gerarchia**

## 5.3 costruzione degli oggetti

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

**Viene quindi prima chiamato il costruttore della classe base e poi quello della classe derivata.**

**Se la classe base ha dei 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.**

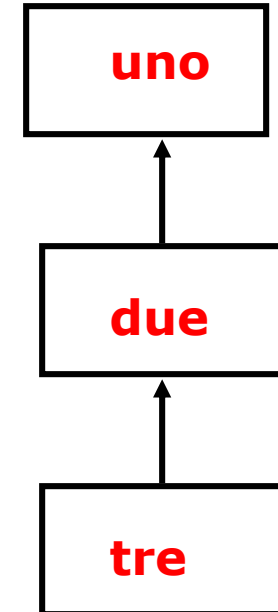
## 5.3 Costruttori

```
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  
}
```

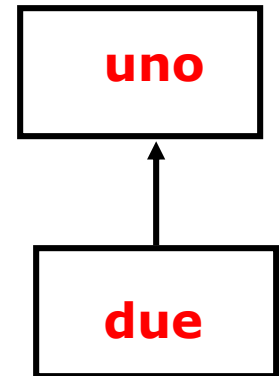


## 5.3 Costruttori

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

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



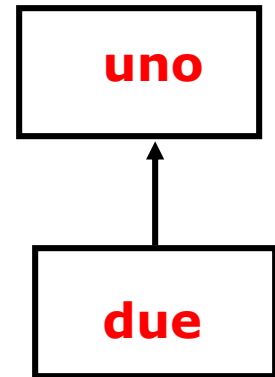
## 5.3 Costruttori

```
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
}
```



## 5.3 costruzione di obj2

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

|   |  |
|---|--|
| a |  |
| b |  |

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

|   |   |
|---|---|
| a | 9 |
| b |   |

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

|   |   |
|---|---|
| a | 9 |
| b | 8 |



## 5.3 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  
};
```

## 5.3 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  
}
```

## Costruzione con membri oggetti

**costruzione di un oggetto della della classe O**

**COSTRUZIONE(O):**

**se O non e' una classe derivata:**

- **si costruiscono le variabili di istanza di O (chiamando gli opportuni costruttori nel caso che siano oggetti);**
- **si chiama il costruttore di O;**

**se O deriva da una classe base B:**

**COSTRUZIONE (B);**

**si chiama il costruttore di O;**

**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 classi derivate

```
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> |  |

# *Funzioni virtuali e Polimorfismo*

## 6.1 Funzioni virtuali

```
class studente {  
    int esami;  
    int matricola;  
  
public:  
    studente (int e, int m){  
        esami=e;  
        matricola=m;  
    };  
  
    int qualematricola(){  
        return matricola;  
    }  
  
    void chisei() {  
        cout << "sono uno studente";  
    }  
  
};
```



## 6.1 Funzioni virtuali

```
class borsista : public studente {  
    int borsa;  
  
    public:  
        borsista(int e, int m, int b) : studente(e,m) {  
            borsa=b;  
        };  
  
        void chisei() {          // ridefinizione della funzione chisei  
            cout << "sono un borsista";  
        }  
};
```

## 6.1 Funzioni virtuali

```
void main () {  
    studente* s= new studente (5,777777);  
    borsista* b= new borsista(10,888888,500000);  
    studente* b1= b;  
  
    s->chisei();  
        // sono uno studente  
  
    b->chisei();  
        // sono un borsista  
  
    b1->chisei();        // studente::chisei();  
        // sono uno studente  
  
}
```

La scelta della funzione avviene a tempo di compilazione  
in base al tipo del **puntatore**

## 6.1 Funzioni virtuali

```
class studente {
    ....
public:
    ...
    void virtual chisei() { cout << "sono uno studente";}
};

class borsista : public studente{
    ....
public:
    ....
    void virtual chisei() { cout << "sono un borsista";}
}; // virtual puo' mancare
```

La scelta della funzione avviene a tempo di esecuzione in base al tipo dell'oggetto **effettivamente puntato**

## 6.1 Funzioni virtuali

```
void main () {  
  
    studente* s= new studente (5,777777);  
    borsista* b= new borsista(10,888888,500000);  
    studente* b1= b;  
  
    s->chisei();  
        // sono uno studente  
  
    b->chisei();  
        // sono un borsista  
  
    b1->chisei();  
        // sono un borsista  
  
}
```

## 6.1 Funzioni virtuali : non hanno effetto se sono chiamate dall'oggetto

```
void main () {  
  
    studente s(5,777777);  
    borsista b(10,888888,500000);  
    studente b1= b;  
  
    s.chisei();  
        // sono uno studente  
  
    b.chisei();  
        // sono un borsista  
  
    b1.chisei();  
        // sono uno studente  
  
}
```

**b1 ha un solo campo "chisei"**

## 6.1 Funzioni virtuali esempio di utilizzo

```
void main(){  
    studente* s [2];  
    s[0] = new studente(7,77777);  
    s[1] = new borsista(10,888888,500000);  
  
    for(int i=0; i< 2; i++) stampa(s[i]);  
}
```

**Come definisco stampa per avere il seguente output?**

**sono uno studente matricola=77777**  
**sono un borsista matricola=888888**

## 6.1 Risoluzione a tempo di esecuzione con funzione virtuale

```
class studente {
    ....
public:
    ...
    void virtual chisei() { cout << "sono uno studente";}
};

class borsista : public studente{
    ....
public:
    ....
    void chisei() { cout << "sono un borsista";}
};

void stampa (studente* s){
    s->chisei();
    cout << " matricola=";
    cout << s->qualematricola() << endl;
}
```

## 6.1 Risoluzione a tempo di esecuzione con funzione virtuale (cont.)

```
void main(){  
    stampa(s[0] );  
        // sono uno studente matricola=777777  
    stampa(s[1] );  
        // sono un borsista matricola=888888  
}
```



## 6.1 Funzioni virtuali nella gerarchia

```
class uno {
    //..
public:
    uno() {}
    void f() {
        cout << 1 << endl; }
};

class due : public uno{
public:
    due () {}
    void virtual f() {
        cout << 2 << endl; }
};

class tre: public due {
public:
    tre () {}
    void f() {
        cout << 3 << endl; }
};
```

```
void main(){
    due* p2= new tre;
    p2->f(); // 3 tre::f()
    uno* p1= new tre;
    p1->f(); // 1 uno::f()
}
```

**f è virtuale in due e tre ma non in uno**

**Una funzione e' virtuale in tutte le classi che si trovano sotto quella che la definisce come virtuale**

## 6.3 Distruttori virtuali

```
class uno {
public:
    uno() {};
    virtual ~uno() {cout << "via uno" << endl;}
};
class due: public uno {
public:
    due(){};
    ~due() {cout << "via due" << endl
};
```

```
void main (){
    uno* obj=new due;
    //...
    delete obj;}

```

```
// via due           ~due()
// via uno
```

**senza virtual :**

```
// via uno           ~uno()
```

## 6.4 Classi astratte

```
class studente {  
    int matricola; int esami;  
public:  
    studente (int m){ esami=0; matricola=m; }  
    // ...  
    void virtual chisei() =0;  
        // funzione virtuale pura  
};
```

**Una classe e' astratta se ha almeno una funzione virtuale pura (ereditata o no)**

**Non si possono istanziare oggetti di una classe astratta**

**Serve come classe base nelle derivazioni**

## 6.4 Classi astratte (cont.)

```
class studenteIngInf : public studente {  
    //...  
public:  
    studenteIngInf(int m) : studente(m) {}  
    // ...  
    void chisei() {  
        cout << "studente di ingegneria informatica" <<endl;  
    }  
};
```

```
class studenteIngMecc : public studente{  
    // ..  
public:  
    studenteIngMecc(int m) : studente(m) {}  
    // ...  
    void chisei() {  
        cout << "studente di ingegneria meccanica" << endl;  
    }  
};
```

## 6.4 Classi astratte (cont.)

```
void main(){  
    // studente s; errato studente e' una classe astratta  
  
    studente* s; // OK viene dichiarato un puntatore  
  
    studente* studenti [3];  
    studenti[0]= new studenteIngInf(777777) ;  
    studenti[1]= new studenteIngMecc(888888);  
    studenti[2]= new studenteIngInf(888888) ;  
  
    for (int i=0; i<3; i++)  
        studenti[i]->chisei();  
}
```

**studente di ingegneria informatica**  
**studente di ingegneria meccanica**  
**studente di ingegneria informatica**

classi modello e derivazione: classe base modello, classe derivata modello con lo stesso tipo

```
template <class T>
class uno {
    T a;
public:
    uno(T x) {
        a=x;
        cout << a << endl;
    }
};

template <class tipo>
class due: public uno<tipo> {
    tipo b;
public:
    due(tipo x, tipo y):
        uno<tipo>(x) {
        b=y; cout << b << endl;
    }
};
```

```
void main (){
    due<int> obj(7,8);
}

7  uno<int>::uno<int>(7)
8  due<int>::due<int>(7,8)
```

```
template <class T>
class uno {
    T a;
public:
    uno(T x) {
        a=x;
        cout << a << endl;
    }
};

class due: public uno<int> {
    int b;
public:
    due(int x, int y):
        uno<int>(x) {
        b=y; cout << b << endl;
    }
};
```

```
void main (){
    due obj(7,8);
}

7  uno<int>::uno<int>(7)
8  due::due(7,8)
```

```
template <class T>
class uno {
    T a;
public:
    uno(T x) {
        a=x;
        cout << a << endl;
    }
};
```

```
template <class tipo1, class tipo2>
class due: public uno<tipo1> {
    tipo2 b;
public:
    due(tipo1 x, tipo2 y):
        uno<tipo1>(x) {
        b=y; cout << b << endl;
    }
};
```

```
void main (){
    due<int,double> obj1(7,8.5);
    due<int,int> obj2(7,8.5);
}

7    uno<int>::uno(7)
8.5 due<int,double>::due<int,double>(7,8.5)

7    uno<int>::uno(7)
8    due<int,int>::due<int,int>(7,8.5)
```



***Gestione delle Eccezioni***

**a.a. 2001/2002**

## 9.2 Eccezioni: costrutto sintattico

```
try {  
  ..  
  throw espressione1;           // lancio eccezione  
  ....  
  throw espressionem;         // lancio eccezione  
}  
  
catch (tipo1 e) { ..... }    // gestione eccezione  
..  
catch (tipon e) { ..... }    // gestione eccezione  
  
comandi_successivi ...
```

## 9.2 Eccezioni: costruito sintattico

- Se viene lanciata una eccezione (**throw**), l'esecuzione del blocco **try** si interrompe
- le eccezioni lanciate all'interno del blocco **try** tramite **throw** sono gestite dai gestori (clausole **catch**): la gestione è scelta in base al tipo dell'eccezione lanciata
- dopo la gestione dell'eccezione, l'esecuzione prosegue normalmente con comandi successivi non considerando le altre clausole **catch**
- se nessuna eccezione viene lanciata, l'esecuzione prosegue con comandi successivi
- se un'eccezione lanciata non viene catturata, il programma termina con errore

## Esempio: divisione per 0 (I)

```
void div (int x, int y){  
    try {  
        if (y==0) throw "divisione per 0";  
        cout << x/y << endl;  
    }  
    catch (char* p) { cout << p << endl;}  
  
    cout << "fine div" << endl ;  
}
```

```
void main(){  
    int x,y;  
    cin >> x >> y;  
    div(x,y);  
    cout << "fine main";  
}
```

con 10 5:

2

fine div

fine main

con 10 0:

divisione per 0

fine div

fine main

## Esempio: divisione per 0 (II): eccezione non catturata

```
void div (int x, int y){  
    try {  
        if (y==0) throw 0;    // nota: eccezione intera  
        cout << x/y << endl;  
    }  
    catch (char* p) { cout << p << endl;}  
  
    cout << "fine div" << endl ;  
}
```

```
void main(){  
    int x,y;  
    cin >> x >> y;  
    div(x,y);  
    cout << endl << "fine main";  
}
```

con **10 0** il programma termina con errore

## Esempio: divisione per 0 (III)

```
void div (int x, int y){
    try {
        if (y==0) throw 0;
        cout << x/y << endl;
    }

    // nota
    catch (int) {
        cout << "divisione per 0" << endl;}

    cout << "fine div" << endl ;
}

void main(){
    int x,y;
    cin >> x >> y;
    div(x,y);
    cout << endl << "fine main";
}
```

**con 10 0:**

**divisione per 0**

**fine div**

**fine main**

## Esempio: divisione per 0 (IV)

```
void positive_div (int x, int y){  
    try {  
        if (y==0) throw 0;  
        if ( (x<0 && y>0) || (x>0 && y<0) ) throw 1;  
        cout << x/y << endl;  
    }  
    catch (int n) {  
        if (n==0) cout << "divisione per 0" << endl;  
        else cout << "risultato negativo" << endl;  
    }  
    cout << "fine div" << endl ;  
}
```

```
void main(){  
    int x,y;  
    cin >> x >> y;  
    positive_div(x,y);  
    cout << "fine main";  
}
```

con input -2 3

risultato negativo

fine div

fine main

## Esempio: divisione per 0 (V)

```
void positive_div (int x, int y){
    try {
        if (y==0) throw 0; // intero
        if ( (x<0 && y>0) || (x>0 && y<0) ) throw '0'; // carattere
        cout << x/y << endl;
    }
    catch (int) {
        cout << "divisione per 0" << endl; }
    catch (char) {
        cout << "risultato negativo" << endl; }

    cout << "fine div" << endl ;
}
void main(){
    int x,y;
    cin >> x >> y;
    positive_div(x,y);
    cout << "fine main";
}
```

con input -2 3

risultato negativo

fine div

fine main



## 9.3 Throw e try-catch in funzioni diverse

```
void div (int x, int y){
    // lancio eccezione
    if (y==0) throw "divisione per 0";

    cout << x/y << endl;
    cout << "fine div" << endl ;
}

void main(){
    try { // gestione eccezione nel main
        int x,y;
        cin >> x;
        cin >> y;
        div(x,y);
    }
    catch (char* p) {
        cout << p << endl;
    }
    cout << endl << "fine main";
}
```

con 10 0

divisione per 0

fine main

## 9.7 corrispondenza fra throw e catch

- **No conversioni implicite**
- **Oggetto lanciato: stesso tipo o sottotipo dell'argomento del catch**
- **I gestori sono esaminati nell'ordine in cui compaiono a partire dal blocco più interno e viene considerato il primo con argomento corrispondente all'eccezione lanciata**

## 9.6 Ordine dei gestori

```
void f(int x) {  
    if (x==0) throw x;  
    if (x>100) throw 'a';  
    cout << "fine f" << endl;  
}  
void g(int x) {  
    try { f(x);}  
    catch(int) {  
        cout << "eccezione da g"  
        << endl; }  
    cout << "fine g" << endl;  
}  
void main(){  
    try { int x; cin >> x; g(x);}  
    catch(char) {  
        cout << "eccezione da main"  
        << endl; }  
    cout << "fine main";  
}
```

**con input 0:**

**eccezione da g**

**fine g**

**fine main**

**con input 200:**

**eccezione da main**

**fine main**

## 9.6 Ordine dei gestori :l'eccezione viene gestita dal gestore più interno incontrato

```
void f(int x) {  
    if (x==0) throw x;  
    if (x>100) throw 'a';  
    cout << "fine f" << endl;  
}  
  
void g(int x) {  
    try {  
        f(x);}  
    catch(char) {  
        cout << "eccezione da g"  
            << endl; }  
    cout << "fine g" << endl;  
}
```

```
void main(){  
    try {  
        int x; cin >> x; g(x);  
    }  
    catch(char) {  
        cout << "eccezione da main"  
            << endl; }  
    cout << "fine main";  
}
```

con input = 200:

eccezione da g  
fine g

fine main

con input 0:

**ECCEZIONE NON GESTITA**

## 9.6 Clausola catch generica: cattura qualsiasi eccezione

```
void main(){
    try {
        int x; cin >> x;
        if (x==0) throw x;
        if (x <0) throw 7.8;
    }
    catch(int) {
        cout << "eccezione da main" << endl;
    }

    catch(...) {
        cout << "eccezione non prevista da main" << endl; }
    cout << "fine main";
}
```

con input=-1:

eccezione non prevista da main  
fine main

## 9.8 Rilancio delle eccezioni

```
void f(int x) {
    if (x==0) throw x;
    cout << "fine f" << endl;
}
void g(int& x) {
    try {
        f(x);
    }
    catch(...) {
        throw; // = catch(int) { throw; }
    }
}
void main(){
    try {
        int x; cin >> x; g(x);
    }
    catch(int) {
        cout << "eccezione da main" << endl; }
    cout << "fine main";
}
```

**con input 0:**  
**eccezione da main**  
**fine main**

## Esempio: stack (I)

```
class stack{
    int size;
    int * p;
    int top;
public:
    stack(int);
    ~ stack();
    stack& push(int);
    int pop();
};

stack::stack(int n){
    size = n;
    p = new int [n];
    top = -1; }

stack::~~stack(){ delete [] p; }
```

```
stack& stack::push(int s){
    if (top==size-1) throw 0;
    p[++top] = s;
    return *this;}

int stack::pop(){
    if (top==-1) throw 1;
    return p[top--]; }
```

## Esempio: stack (II)

```
void main(){
stack pila(2);

    try { // ...
        pila.push(4).push(5).push(6);
    }
    catch (int n) {
        if (n==0) cout << "stack pieno";
        else if (n==1) cout << "stack vuoto";
    }

    cout << endl << "fine";
}
```

stack pieno  
fine



## Esempio: stack (III)

```
try { // ...  
    pila.push(4).push(5);  
    cout << pila.pop() << endl;  
    cout << pila.pop() << endl;  
    cout << pila.pop() << endl;  
}  
catch (int n) { // come sopra }  
cout << endl << "fine";  
}  
  
5  
4  
stack vuoto  
fine
```

## 9.4 con una classe eccezione (I)

```
class eccezione{  
  
    int e;  
  
    public:  
        eccezione(int n) {  
            e=n;  
        }  
  
        void print(){  
  
            if (e==0) cout << "stack pieno" << endl;  
            else cout << "stack vuoto" << endl;  
  
        }  
  
};
```

## 9.4 con una classe eccezione (II)

.....

```
stack& stack::push(int s){  
    if (top==size-1)  
        throw eccezione(0);  
    p[++top] = s;  
    return *this;}  

```

```
int stack::pop(){  
    if (top==-1)  
        throw eccezione(1);  
    return p[top--]; }  

```

```
void main(){  
    stack pila(2);  
  
    try { // ...  
        pila.push(4).push(5).push(6);  
    }  
    catch (eccezione ecc) {  
        ecc.print();  
    }  
}
```

**stack pieno**

## 9.4 con due classi eccezione (I)

```
class StackFull {  
    int e;  
public:  
    StackFull(int n) {  
        e=n;  
    }  
    void print(){  
        cout << e << " non inserito" << endl;  
    }  
};
```

```
class StackEmpty {  
  
public:  
    StackEmpty() {}  
    void print(){  
        cout << "stack vuoto" << endl;  
    }  
};
```

## 9.4 con due classi eccezione (II)

```
stack& stack::push(int s){  
    if (top==size-1) throw StackFull(s);  
    p[++top] = s;  
    return *this;  
}
```

```
int stack::pop(){  
    if (top==-1) throw StackEmpty();  
    return p[top--];  
}
```

## 9.4 con due classi eccezione (III)

```
void main(){  
    stack pila(2);  
  
    try { // ...  
        pila.push(4).push(5).push(6);  
    }  
    catch (StackFull ecc) {  
        ecc.print();  
    }  
  
    catch (StackEmpty ecc) {  
        ecc.print();  
    }  
  
}
```

**6 non inserito**

## 9.4 con riferimenti e puntatori

```
throw StackFull(s);  
...  
throw StackEmpty();  
  
.....  
  
catch (StackFull &ecc) {  
    ecc.print();  
}  
catch (StackEmpty &ecc) {  
    ecc.print();  
}
```

```
throw &StackFull(s);  
...  
throw &StackEmpty();  
  
.....  
  
catch (StackFull *ecc) {  
    ecc->print();  
}  
  
catch (StackEmpty *ecc) {  
    ecc->print();  
}
```

## 9.7 Con una gerarchia di classi (I)

```
class StackEcc { // classe astratta
public:
    void msg() {cout << "attenzione: "; };
    void virtual print()=0;
};

class StackFull: public StackEcc {
    int e;
public:
    StackFull(int n) { e=n; }
    void print(){
        msg();cout << e << " non inserito" << endl;}
};

class StackEmpty : public StackEcc {
public:
    StackEmpty() {}
    void print(){
        msg(); cout << "stack vuoto" << endl;}
};
```



## 9.7 Con una gerarchia di classi (II)

```
stack& stack::push(int s){  
    if (top==size-1) throw &StackFull(s);  
  
    p[++top] = s;  
    return *this;  
}  
  
int stack::pop(){  
    if (top==-1) throw &StackEmpty();  
  
    return p[top--];  
}
```

## 9.7 Con una gerarchia di classi (III)

```
void main(){
    stack pila(2);
    try { // ...
        pila.push(4).push(5).push(6);
    }
    catch (StackEcc* ecc) {
        ecc->print();
    }
}
```

**attenzione: 6 non inserito**

```
void main(){
    stack pila(2);
    try { // ...
        int x=pila.pop();
    }
    catch (StackEcc* ecc) {
        ecc->print();
    }
}
```

**attenzione: stack vuoto**