Pietro Ducange

Complementi di programmazione a oggetti in C++

a.a. 2021/2022

Funzioni virtuali, classi astratte e Polimorfismo

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

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";
    }

};
```

oggetto di
tipo studente

Funzioni virtuali

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

public:
     borsista(int e, int m, int b) : studente(e,m) {
        borsa=b;
    };

     void chisei() {
        cout << "sono un borsista";
    }
};
```

ridefinizione della funzione chisei()

oggetto di
tipo borsista

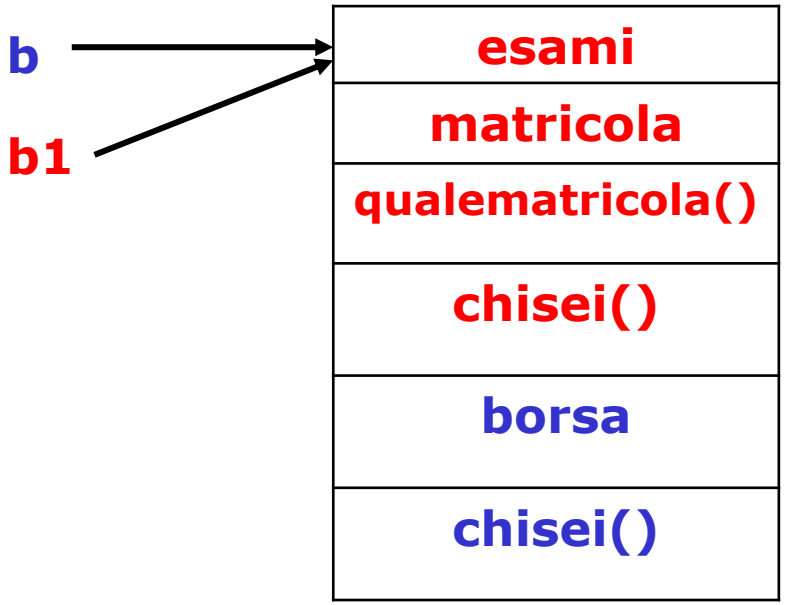
Funzioni virtuali

```
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

}
```

Funzioni virtuali

borsista* b

studente* b1



b1->chisei(); chiama chisei()

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

Funzioni virtuali

In una gerarchia di classi, il metodo (la funzione) da chiamare viene scelto dinamicamente a tempo di esecuzione

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

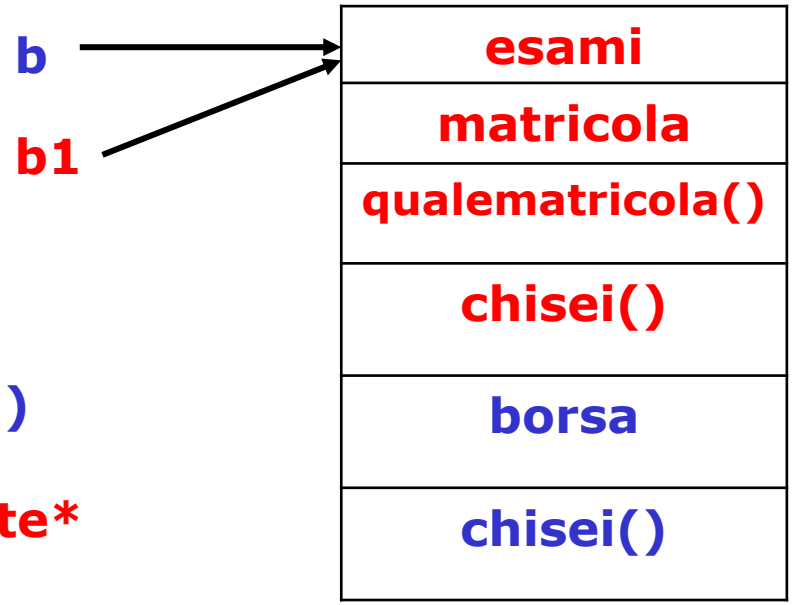
Funzioni virtuali

```
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

}
```

Funzioni virtuali

borsista* b

studente* b1



b1->chisei(); chiama chisei()

anche se b1 è di tipo studente*

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"

Funzioni virtuali

borsista b

Studente b1

b1

b

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 la funzione stampa per avere il seguente output?

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

Risoluzione a tempo di esecuzione con funzione virtuale

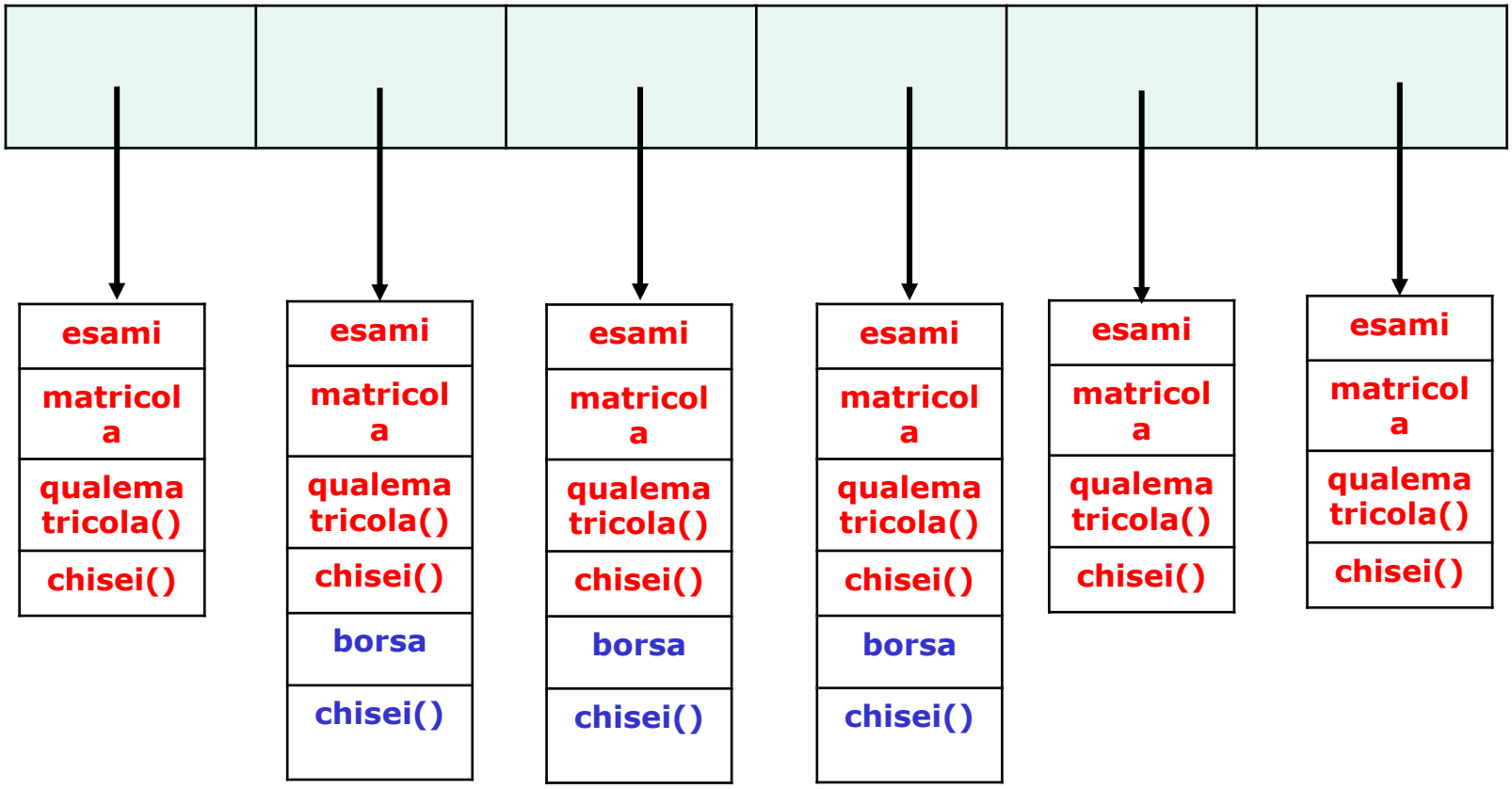
```
class studente {
    ....
public:
    ...
    int qualematricola(){
        return matricola;
    }
    void virtual chisei() { cout << "sono uno studente";}
};

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


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

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
    }
```

Risoluzione a tempo di esecuzione di stampa

s è un array con elementi di tipo studente *



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

Funzioni virtuali nella gerarchia

```
class uno {
    //..
public:
    uno() {}
    virtual 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();
        uno* p1= new tre;
        p1->f();
}
```

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()
```

Classi astratte e polimorfismo

Classe astratta

Serve come classe base nelle derivazioni.

Viene specializzata nelle classi derivate.

Definisce una interfaccia unica verso le applicazioni.

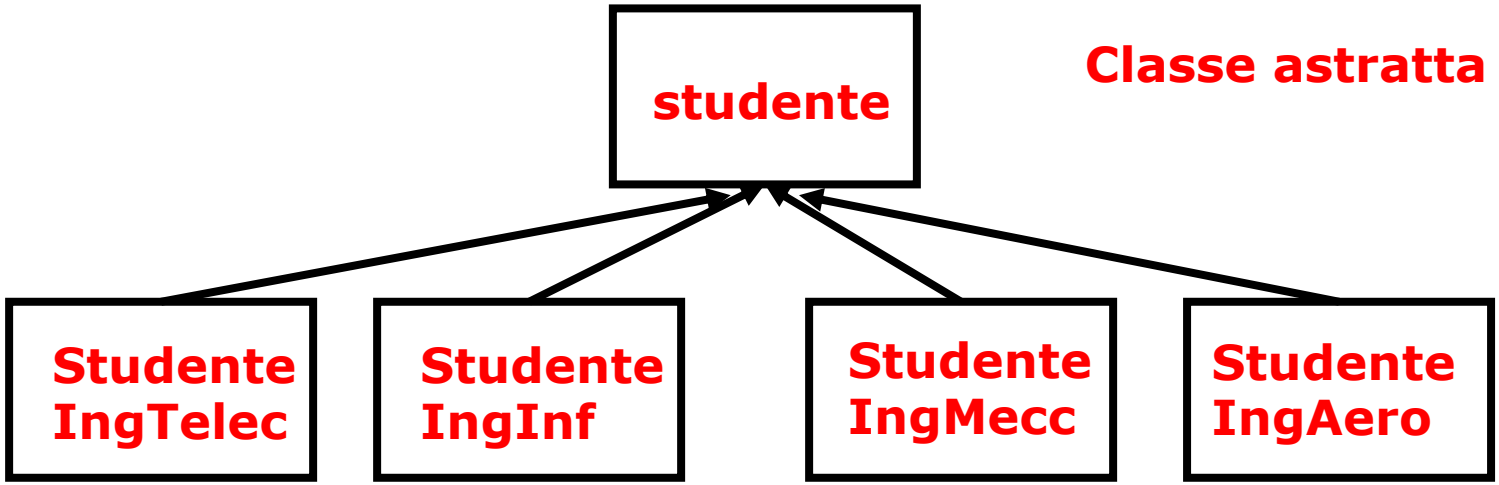
Non viene definita completamente: ha almeno una funzione virtuale pura.

Una funzione virtuale pura è una funzione (ereditata o no) senza definizione: F(...)=0.

Non si possono istanziare oggetti di una classe astratta

POLIMORFISMO

Classi astratte (cont.)



Classi astratte

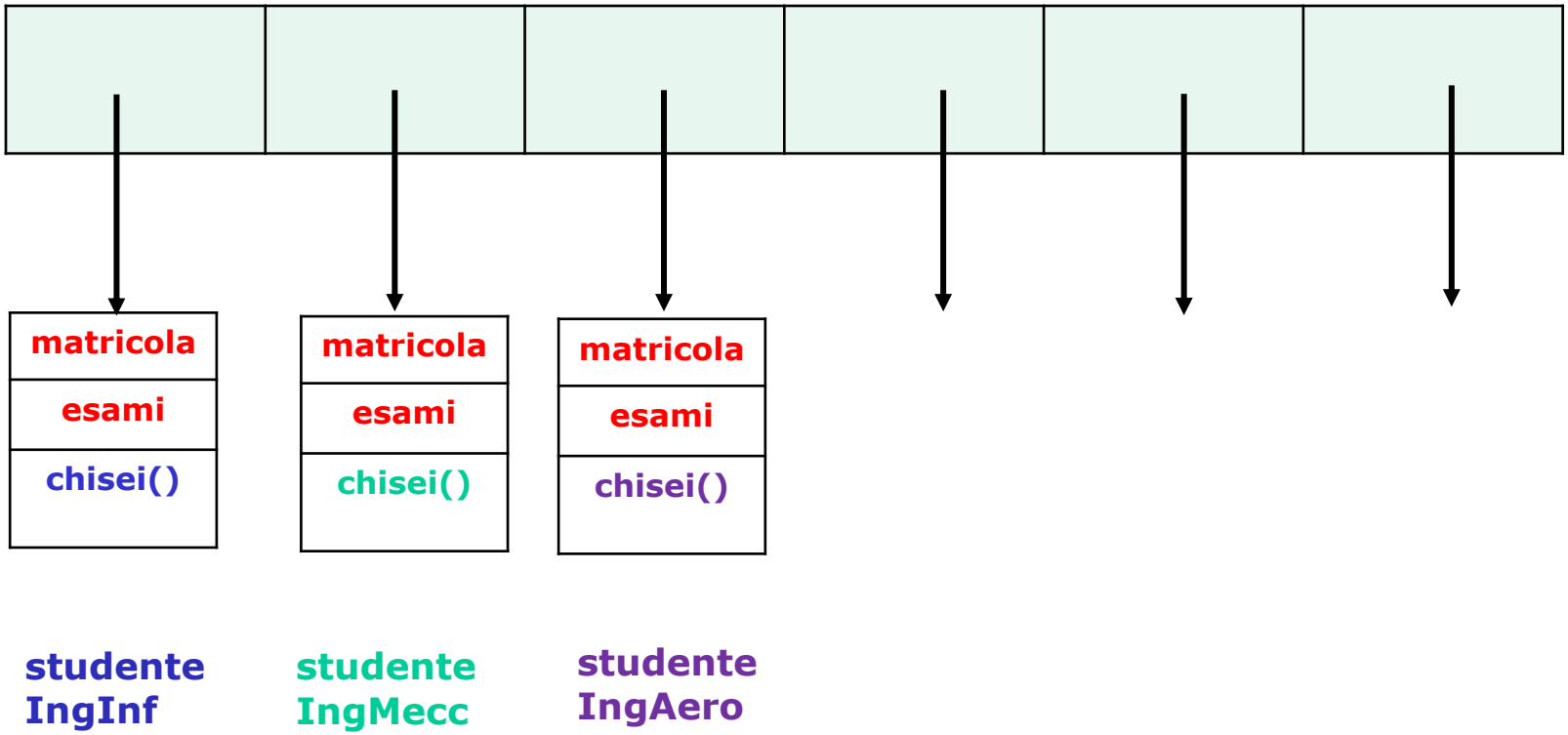
```
class studente {
        int matricola; int esami;
public:
        studente (int m){ esami=0; matricola=m; }
        // ...
        void virtual chisei() =0;

                        // funzione virtuale pura
};
```

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;
        }
};
```

Classi astratte (cont.)

array con elementi di tipo studente*



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();
}
```

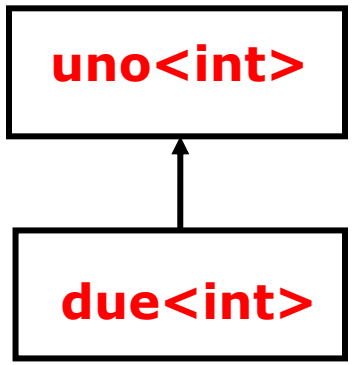
studente di ingegneria informatica
studente di ingegneria meccanica
studente di ingegneria informatica

```
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)
```

obj

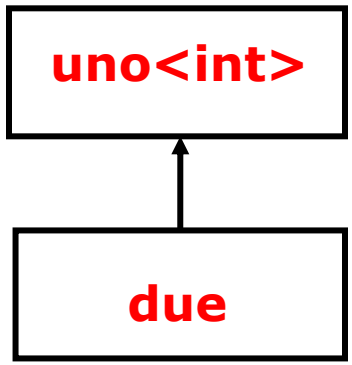


```
template <class T>
class uno {
    T a;
public:
    uno(T x) {
        a=x;
    cout << a << endl;
    }
};

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)
```

obj

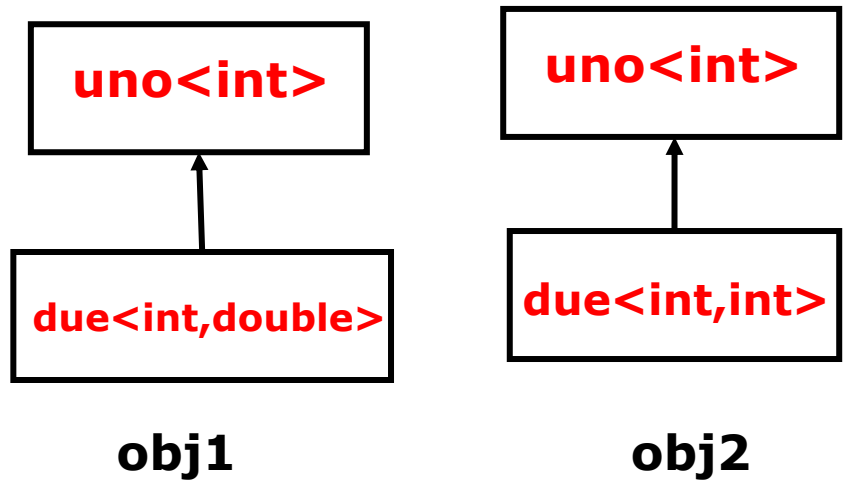


classi modello e derivazione: classe base modello, classe derivata modello

```
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

Eccezioni

Errori a runtime (es. divisione per 0, indice array fuori dall'intervallo)

Situazioni anomale non rilevabili dal compilatore

Possono causare il crash dell'applicazione

Eccezioni: costrutto sintattico

Possibilità di individuare le eccezioni e gestirle da programma a tempo di esecuzione

Metodo formale e ben definito

Netta separazione tra il codice che rileva l'eccezione e il codice che lo gestisce

Eccezioni: costrutto sintattico

```
try   {
..
throw espressione1;          // lancio eccezione
.....
throw espressionem;          // lancio eccezione
.....
}

   catch (tipo1 e) {  .....  }    // gestione eccezione
   ..
   catch (tipon e) { ......  }    // gestione eccezione

comandi_successivi ...
```

Eccezioni: costrutto sintattico

- **Se viene lanciata una eccezione (throw), l'esecuzione del blocco try si interrompe**

- **le eccezioni lanciate durante l'esecuzione del blocco try sono gestite dai gestori (clausole catch): la gestione è scelta in base al tipo dell'eccezione lanciata**

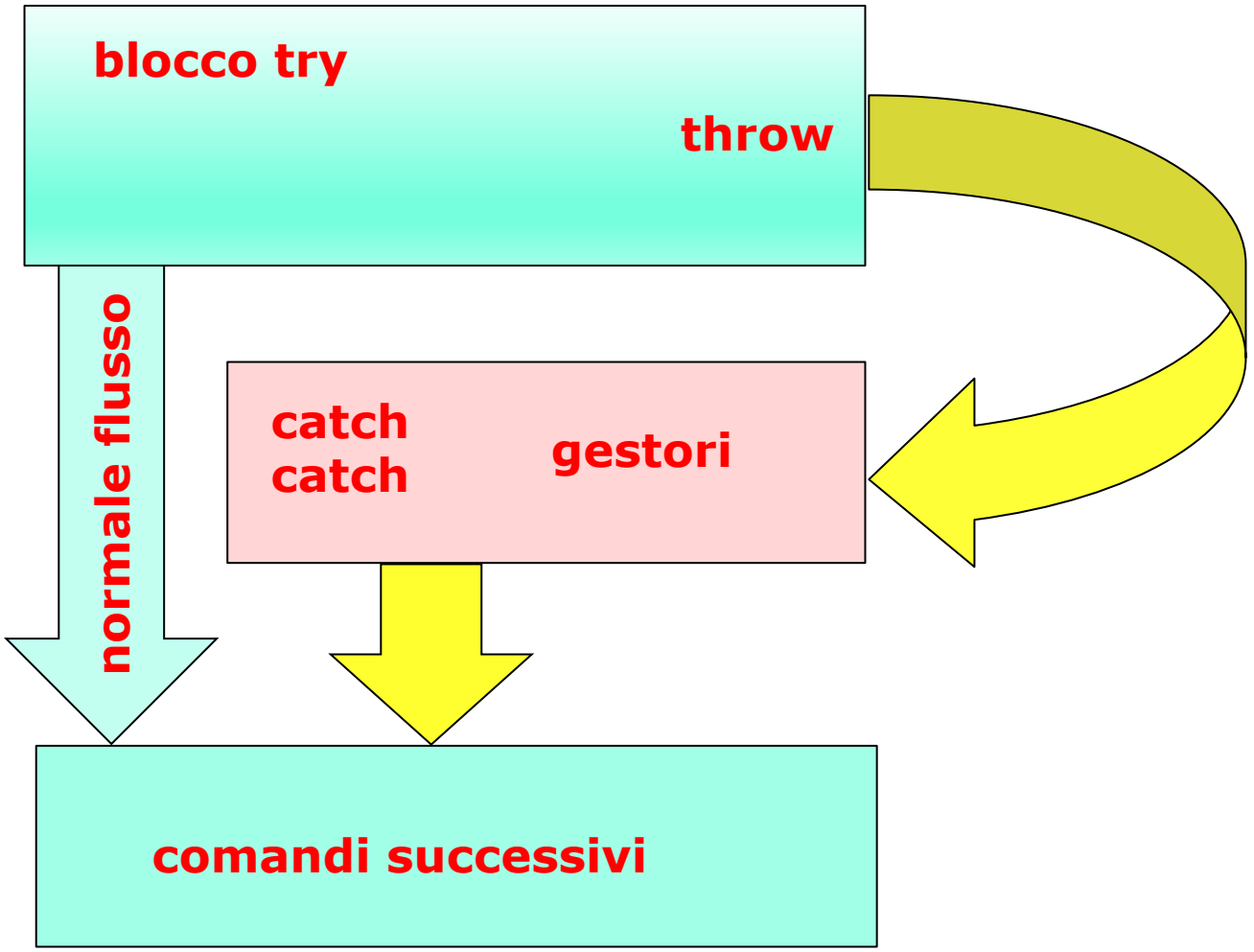
- **dopo la gestione dell'eccezione, l'esecuzione prosegue normalmente con comandi_successivi non considerando le altre clausole catch**

- **se nessuna eccezione viene lanciata, l'esecuzione prosegue con comandi_successivi**

- **se un'eccezione lanciata non viene catturata, il programma termina con errore**

- **Un'eccezione può essere lanciata soltanto durante l'esecuzione di un blocco try**

Eccezioni: costrutto sintattico



Esempio: divisione per 0 (I)

```
void div (int x, int y){
   try {
         if (y==0) throw "divisione per 0";
         cout << x/y  << endl;
        }
        catch (const 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 (const 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 o negativa (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

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 (const char* p) {
                    cout << p << endl;
          }
     cout << endl << "fine main";
}
```

con 10 0

divisione per 0

fine main

corrispondenza fra throw e catch

- **No conversioni implicite (a parte sottotipo->sopratipo)**

- **L'eccezione viene gestita a tempo di esecuzione esaminando i gestori nell'ordine in cui compaiono a partire dal blocco più recente incontrato.**

- **Viene scelto il primo con argomento corrispondente all'eccezione lanciata**

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

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(int) {
    cout << "eccezione da main"
    << endl; }
   cout << "fine main";
}
```

con input 0:

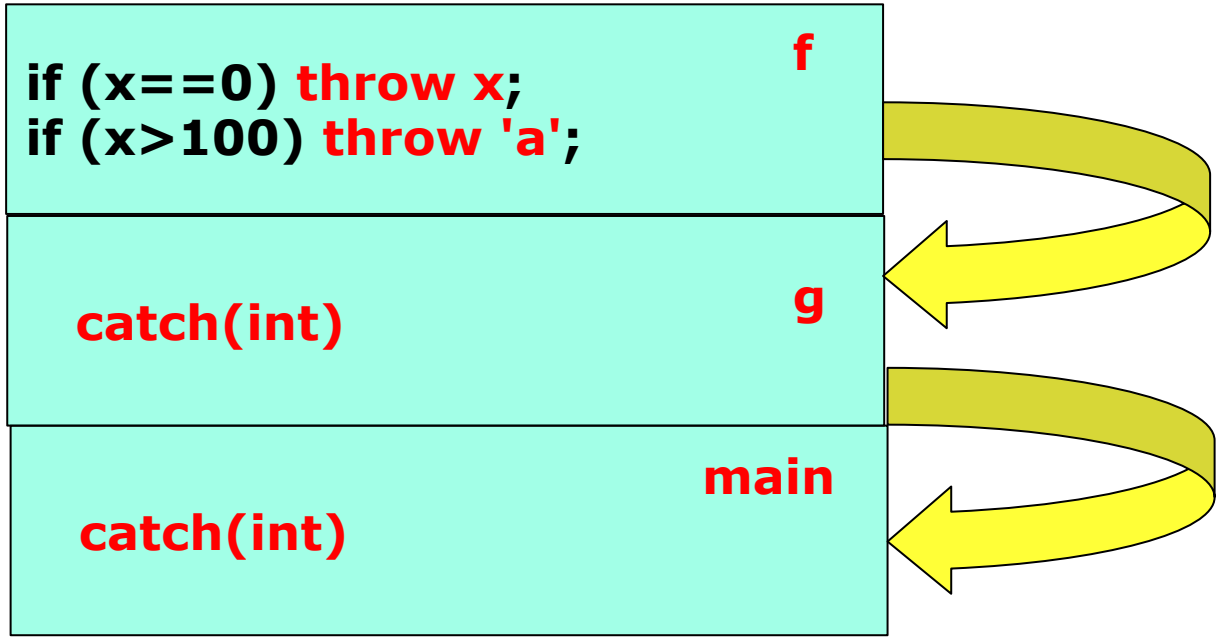
eccezione da g

fine g

fine main

con input 200:

errore



Stack di esecuzione

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

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
```

Stack con 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;
        }
};
```

Stack con 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

Stack 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;
        }
};
```

Stack 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--];
}
```

Stack con due Classi eccezione (III)

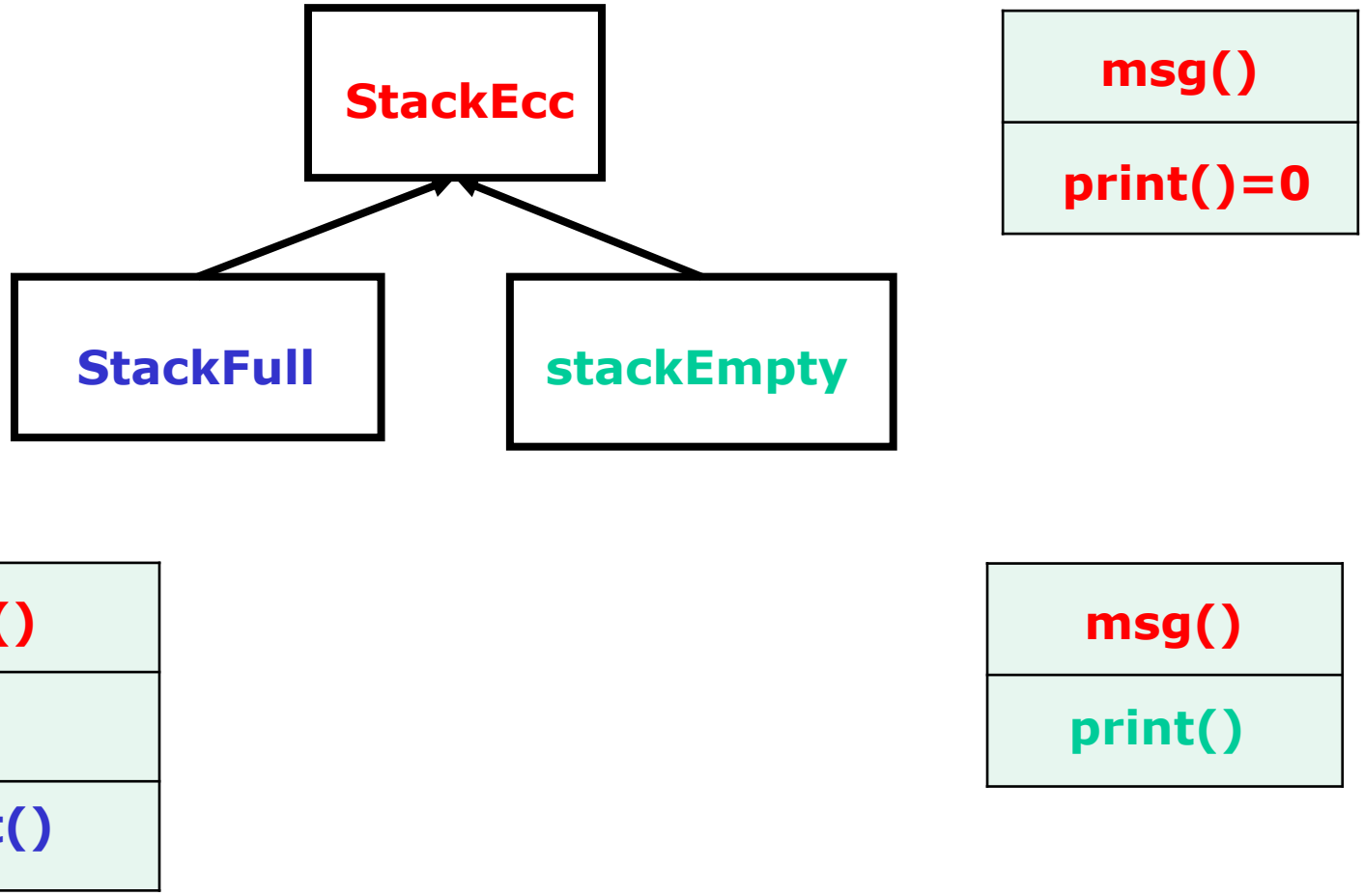
```
void main(){

        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 puntatori

```
throw new StackFull(s);

...

throw  new StackEmpty();

.............


catch (StackFull *ecc) {
    ecc->print();
    delete ecc;
}

 catch (StackEmpty *ecc) {
    ecc->print();
    delete ecc;

}
```

Con una gerarchia di classi (I)



Stack 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;}
};
```

Con una gerarchia di classi (II)

```
stack& stack::push(int s){

        if (top==size-1) throw new StackFull(s);

        p[++top] = s;
        return *this;
}


int stack::pop(){

        if (top==-1)  throw new StackEmpty();

        return p[top--];
}
```

Con una gerarchia di classi (III)

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

attenzione: 6 non inserito

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

attenzione: stack vuoto