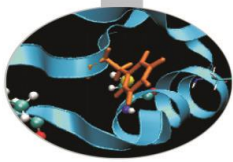
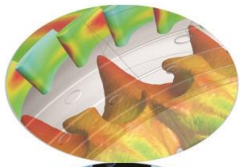
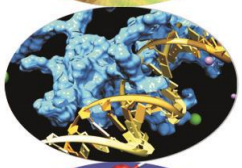
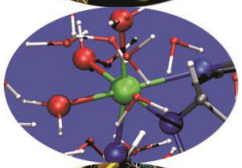
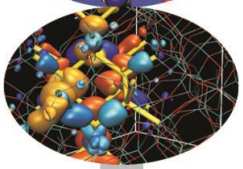
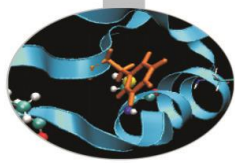


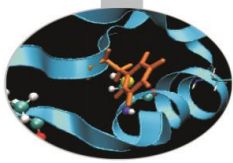
# Eccezioni

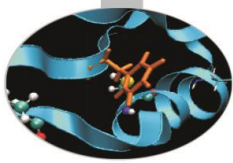
# Indice



# Introduzione



- La *gestione delle eccezioni* rappresenta uno strumento attraverso il quale il C++ riesce a far fronte ad una vasta tipologia di errori che si presentano, tipicamente, in fase di esecuzione (*runtime error*).
- Il controllo degli errori viene trasferito a funzioni implementate ad hoc (*exception handler*).
- La robustezza del codice risulta notevolmente accresciuta perché il programma è in grado di superare situazioni che, altrimenti, ne causerebbero la fine immediata.
- La gestione delle eccezioni è stata progettata per eludere gli errori sincroni, ovvero quelli che si verificano nel momento stesso in cui il programma incontra l'istruzione che li causa.



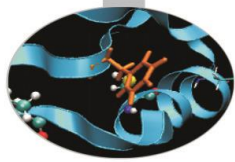
# Esempio

- Esempio: gestione naïf dell'errore dovuto alla divisione per zero

// file div.cpp

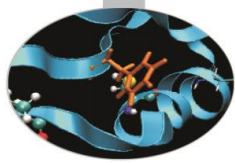
```
#include<iostream>
using namespace std;
int main(){
    double a,b;
    cout << "Program Divisions" << endl;
    cout << "Insert two numbers: " << endl;
    cin >> a;
    cin >> b;
    if(b==0){
        cout << "Error: division by zero!" << endl;
        return 1;
    }
    double d;
    d=a/b;
    cout << "The result is: " << d << endl;
    return 0; }
```

# Esempio



**Output:**

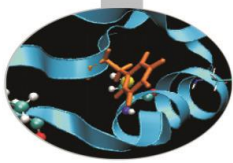
```
> ./div.x
Program Divisions
Insert two numbers:
2
3
The result is: 0.666667
> ./div.x
Program Divisions
Insert two numbers:
2
0
Error: division by zero!
```



# Eccezioni

Utilizzando i comandi propri della gestione delle eccezioni, il programma esegue il controllo del valore del divisore (blocco **try**) e lancia (comando **throw**) un'eccezione se esso è uguale a zero. L'eccezione viene intercettata (blocco **catch**) da un *exception handler* che fa fronte all'errore e consente di proseguire l'esecuzione del programma.

```
#include<iostream>
using namespace std;
int main(){
    double a,b;
    cout << "Program Divisions" << endl;
    cout << "Insert two doubles: " << endl;
    cin >> a;
    cin >> b;
    try{
     if(b==0) throw 10;
     double d=a/b;
     cout << "The result is: " << d << endl;
    }catch(int e){
        cout << "Error: division by zero!" << endl; }
    cout << "End of program. " << endl;
    return 0; }
```



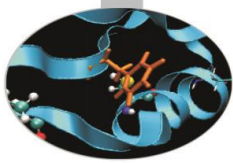
# L'istruzione try

- L'istruzione **try** racchiude un blocco di istruzioni del programma, alcune delle quali potenzialmente in grado di dare luogo ad un'eccezione.
- A seconda del tipo di eccezione generata, il *blocco try* lancerà l'eccezione stessa attraverso il comando throw (o chiamando una funzione che lo contenga) oppure in maniera automatica.
- Sintatticamente il blocco try appare come

```
try{
lista di istruzioni
}
```

Esempio

```
try{
  if(b==0) throw 10;
  double d;
  d=a/b;
  cout << "The result is: " << d << endl;
}
```



# L'istruzione throw

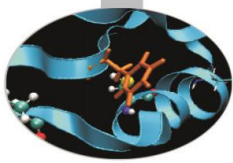
- Il lancio effettivo di un'eccezione è effettuato dall'istruzione **throw** seguita da un operando, necessario per identificare univocamente l'eccezione lanciata:

```
throw operando;
```

- In generale *operando* può essere: una variabile di qualsiasi tipo predefinito; una costante numerica o alfanumerica; un oggetto (chiamato oggetto eccezione) o il suo costruttore; un'espressione condizionale. In quest'ultimo caso la presenza di variabili di tipo diverso può dare luogo ad errori nella gestione delle eccezioni perché il compilatore tende a eseguire automaticamente una promozione di valore (per es. da int a double) per valutare l'espressione, ma così facendo l'eccezione lanciata sarà sempre intercettata dallo stesso blocco catch (in questo caso quello che ha come argomento un double).

Esempio:

```
if(b==0) throw 10;
```



# Il blocco catch

- Il *blocco* **catch** contiene un *gestore di eccezioni*, ovvero una sequenza di istruzioni che il processore deve eseguire in presenza di un particolare errore, allo scopo di evitare la fine prematura del programma.
- E' introdotto dall'istruzione omonima seguita da una coppia di parentesi tonde che contengono un *tipo* (ed, eventualmente, anche un *parametro*) necessario ad identificare l'eccezione:

```
catch (nome_tipo nome_parametro){
    lista di istruzioni
}
```

- Quando l'argomento di catch coincide con il tipo dell'operando del comando throw che ha lanciato l'eccezione, quest'ultima viene intercettata e sono eseguite le istruzioni presenti all'interno del blocco.

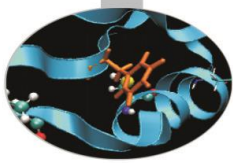
- Se catch racchiude tra parentesi i tre punti di sospensione:

```
catch( … )
```

- allora il blocco corrispondente potrà intercettare qualsiasi tipo di eccezione.

**Esempio:**

```
}catch(int e){
        cout << "Error: division by zero!" << endl;}
```

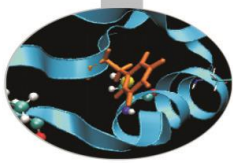


# Try, throw e catch

- Un blocco try può contenere varie istruzioni `throw` e, di conseguenza, può essere associato a *più* blocchi catch.

- Quando è terminata l'esecuzione del gestore delle eccezioni, il programma riprende dalla prima istruzione dopo l'ultimo blocco `catch`.

- Se un'eccezione non viene intercettata da nessuna istruzione `catch`, allora viene invocata automaticamente la funzione terminate che, a sua volta, chiama la funzione `abort`, la quale provoca l'immediato stop dell'esecuzione del programma.



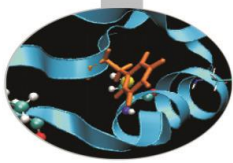
# Try, throw e catch

- **Esempio1: lancio di semplici eccezioni**

```
#include<iostream>
using namespace std;
class Eccezione{};
char ch='a';
int main(){
        try{
          throw Eccezione();
          cout << "Never written!" << endl;
        }catch(Eccezione oggetto){
                cout << "Type: Eccezione" << endl;
        }catch(char){
                cout << "Type: char" << endl;
        }
        cout << "Here!" << endl;

        try{
          throw ch;
        }catch(char){
          cout << "Type: char" << endl;
        }  return 0;}
```

- output:

```
Type: Eccezione
Here!
Type: char
```



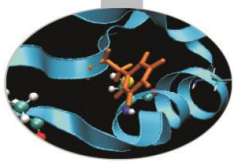
# Try, throw e catch

- Esempio2: throw + espressione condizionale

```
#include<iostream>
using namespace std;

int a;
double b;
int main(){
      try{
         throw( 2 < 5 ? a : b );
      }catch(int){
         cout << "Type: int" << endl;
      }catch(double){
         cout << "Type: double" << endl;
      }
      return 0;
}
```

Output:
Type: double

**a viene convertito a double**

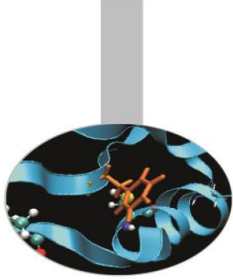


# Rilancio di un'eccezione

- E' possibile che si verifichi il caso in cui un gestore delle eccezioni (blocco catch) "decida" di non poter far fronte all'eccezione intercettata e di doverla, di conseguenza, *rilanciare*. Ciò si realizza attraverso l'istruzione throw, omettendone l'operando:

```
throw;
```

- Dopo essere stata rilanciata, l'eccezione passa sotto il controllo del successivo blocco `try` e può essere intercettata e gestita da un blocco catch che segua il nuovo blocco `try`.

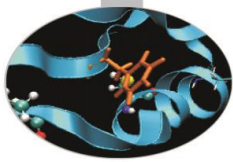


# Esempio

```
#include<iostream>
using namespace std;
void fun_zero(double, double);
int main(){
  double dividendo, divisore;
  while(1){
    cout << endl << "Insert the first number: ";
    cin >> dividendo;
    cout << "Insert the second number: ";
    cin >> divisore;
    try{
   fun_zero( dividendo, divisore);
    }catch( int ){
   cout << "0/0: undetermined" << endl;}
  }
  return 0;
}
void fun_zero(double dividendo, double divisore) {
  try{
    if(divisore==0)  throw 5;
    else  cout << "The result is: " << dividendo/divisore <<
   endl;
  }catch( int ){
    if(dividendo==0) throw;
    else  cout << "Division by zero" << endl;
  }}
```

- output:

```
Insert the first number: 4
Insert the second number: 0
Division by zero

Insert the first number: 0
Insert the second number: 0
0/0: undetermined

Insert the first number:
(CONTRL+C)
```

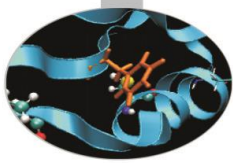


# Le specifiche di eccezione

- Con *specifica di eccezione* si intende l'elenco delle eccezioni che possono essere lanciate da una funzione. Tale elenco va scritto nel prototipo della funzione, all'interno di due parentesi tonde precedute dall'istruzione thrown:
- Tipo_restituito nome_funzione( lista_argomenti )

```
throw( lista_eccezioni ){
    corpo della funzione
    }
```

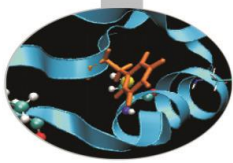
- Se la specifica di eccezione di una funzione consiste nella sola istruzione **`throw()`**, allora la funzione non è in grado di lanciare alcuna eccezione. Viceversa, se l'istruzione `throw(lista_eccezioni)` è assente dal prototipo di una funzione, allora la funzione può lanciare qualsiasi eccezione.
- Qualora le eccezioni specificate da `throw` siano legate ad una *classe*, allora la funzione potrà lanciare eccezioni di tutte le classi da essa derivate con modalità public.



# Esempio

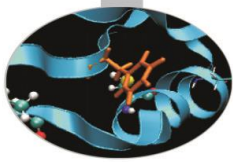
Esempio:

```
int ff (int v_int) throw(double, MiaClasse){
   corpo della funzione
}  // ff può lanciare eccezioni di tipo double e MiaClasse

 int ff (int v_int) throw(){
   corpo della funzione
}  // ff non può lanciare alcuna eccezione

 int ff (int v_int) {
   corpo della funzione
}  // ff può lanciare qualsiasi tipo di eccezione
```

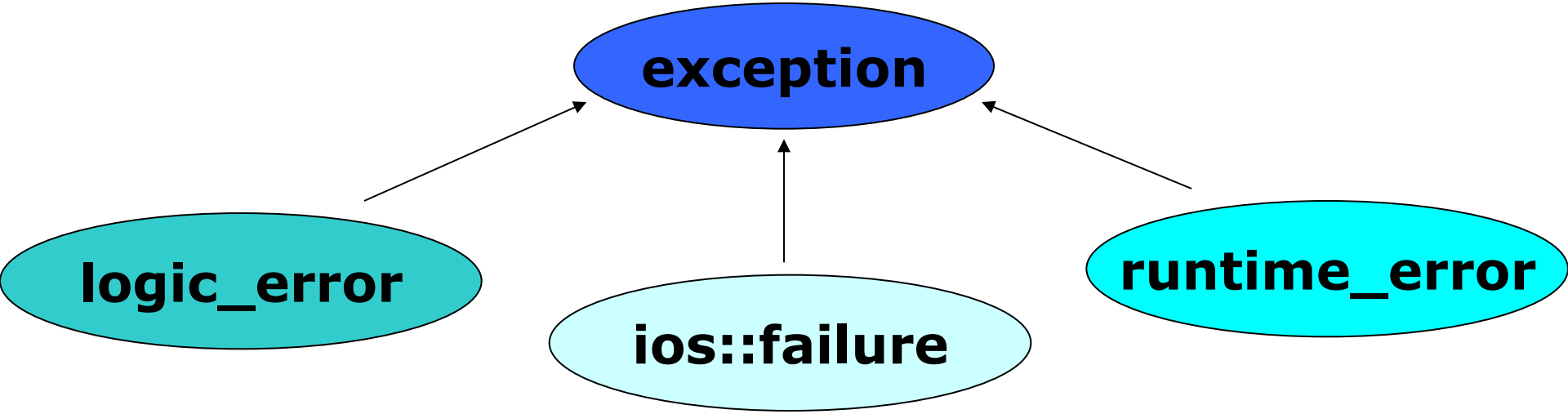


# Le eccezioni standard

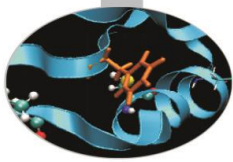
- L'utilizzo più comune dei comandi `try e catch` è legato alla gestione delle eccezioni standard, quelle cioè generate da istruzioni e comandi di uso corrente nella programmazione.
- Il compilatore C++ fornisce la classe base `exception`, insieme con tutte le sue classi derivate, per trattare questo tipo di eccezioni.
- La gestione corretta di un'eccezione standard richiede di: includere nel codice l'header file `<exception>` (o quello legato ad una sua classe derivata); porre all'interno di un blocco `try` le istruzioni che potrebbero generare l'eccezione e scrivere un un blocco `catch` che intercetti un reference a `exception (exception&)`.
- **Non** è necessario utilizzare `throw`, dal momento che il lancio delle eccezioni standard è implementato nei corrispondenti header file.



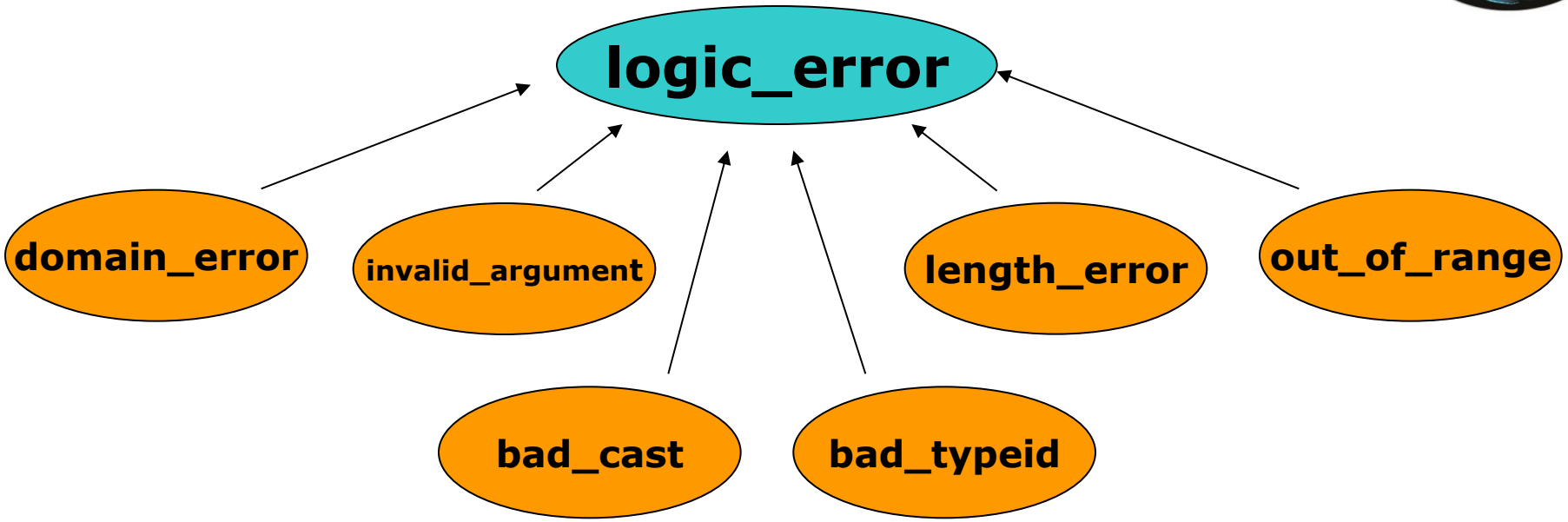
# Le eccezioni standard



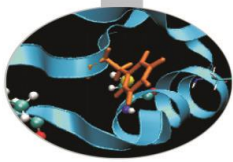

- **exception:** è la classe base per tutte le eccezioni standard. Contiene il metodo what() che restituisce un messaggio sul tipo di eccezione lanciata.
- **logic_error:** deriva direttamente da exception. Gestisce errori logici che, in alcuni casi, sono rilevati prima di eseguire le istruzioni che li contengono.
- **runtime_error:** deriva direttamente da exception. Gestisce errori rilevati in fase di esecuzione.
- **ios::failure:** deriva direttamente da exception. Gestisce errori presenti nelle operazioni di I/O. Non ha sottoclassi.



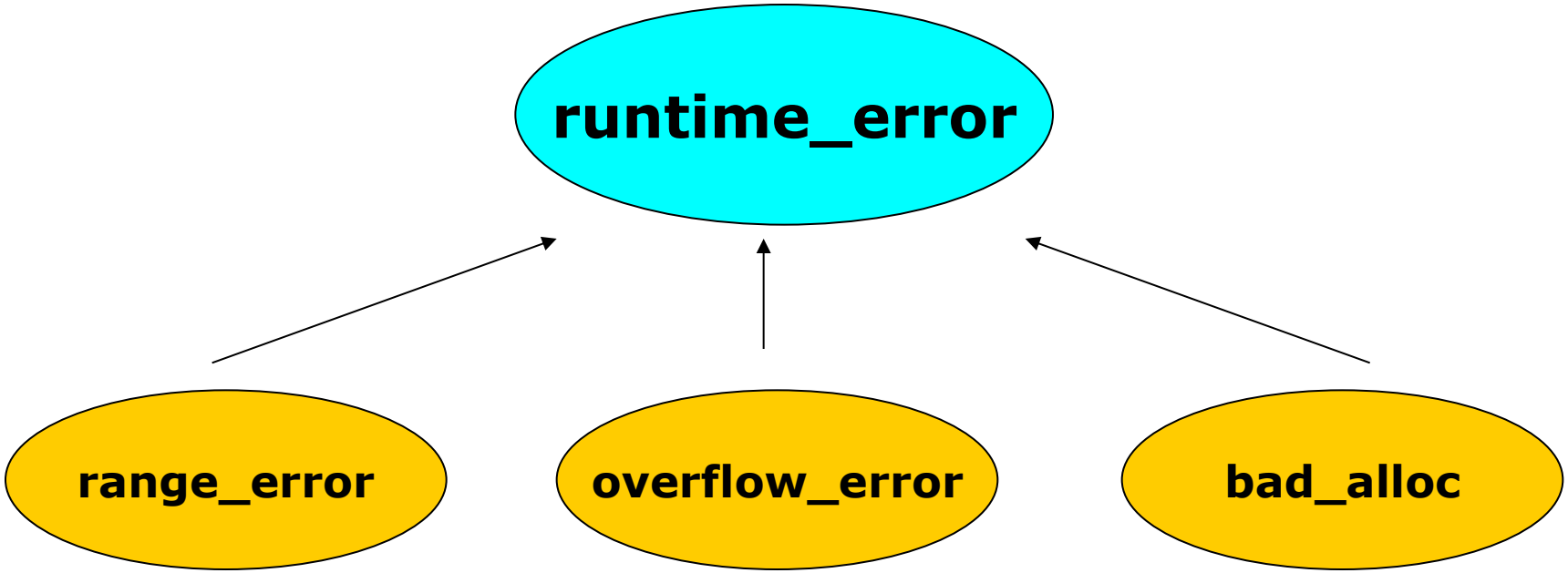
# Le eccezioni standard



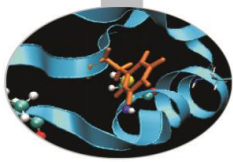

- **domain_error:** tratta violazioni di precondizioni.
- **invalid_argument:** rileva il passaggio di un argomento sbagliato ad una funzione.
- **length_error:** indica il tentativo di creare un oggetto la cui dimensione oltrepassa quella consentita (NPOS).
- **out_of_range:** rileva l'errore di out-of-range, tipico per gli array.
- **bad_cast:** scatta in presenza di un errore con un dynamic_cast.
- **bad_typeid:** segnala un puntatore nullo **p** in un espressione del tipo **typeid(*p)**.



# Le eccezioni standard




- **range_error:** individua violazioni su postcondizioni.
- **overflow_error:** scatta in presenza di un overflow aritmetico.
- **bad_alloc:** segnala che un'operazione di allocazione della memoria è fallita.



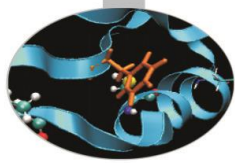
# Le eccezioni standard

- **Esempio1: gestione dell'eccezione di tipo bad_alloc.**

```
#include<iostream>
#include<exception>
using namespace std;
int main(){
        int* a;
        try{
          a = new int[1000000000];
        }catch(exception& e){
          cout << "Err: " << e.what() << endl;
        }
        return 0;
}
```

- **output:**

```
Err: St9bad_alloc
```



# Le eccezioni standard

- Esempio2: riscrittura di what().

```
#include<exception>
#include<iostream>
using namespace std;

class new_exception: public exception{
        const char* what() const throw(){
           return "new_exception caught";}
};
int main(){
        new_exception n;
        try{
          throw n;
        }catch(exception& e){
          cout << e.what() << endl;
        }
        return 0;}
```

- **output:**

```
new_exception caught
```