

Stream di file

Gli stream associati a file sono oggetti delle classi **ifstream**, **ofstream** e **fstream**. Sono disponibili diversi costruttori (vedere documentazione), i più comuni dei quali sono:



```
ifstream(const char* nomefile, int modalita=ios::in);  
ofstream(const char* nomefile, int modalita=ios::out);  
fstream(const char* nomefile, int modalita=ios::in | ios::out);
```

La stringa **nomefile** è il nome del file associato allo stream, mentre le modalità di apertura dello stream sono specificate da un tipo enum nella classe base **ios**



```

class ios {
public:
    enum openmode {
        in,           // apertura in lettura
        out,           // apertura in scrittura
        ate,           // spostamento a EOF dopo l'apertura
        app,           // spostamento a EOF prima di ogni write
        trunc,         // erase file all'apertura
        binary,        // apertura in binary mode, default text mode
    };
    ...
}

```

member constant	opening mode
app	(append) Set the stream's position indicator to the end of the stream before each output operation.
ate	(at end) Set the stream's position indicator to the end of the stream on opening.
binary	(binary) Consider stream as binary rather than text.
in	(input) Allow input operations on the stream.
out	(output) Allow output operations on the stream.
trunc	(truncate) Any current content is discarded, assuming a length of zero on opening.

Le modalità di apertura di uno stream su file possono essere combinate tramite l'OR bitwise |

Per default, gli oggetti di ifstream sono aperti in lettura mentre quelli di ofstream sono aperti in scrittura. Un fstream può essere aperto sia in lettura che in scrittura.

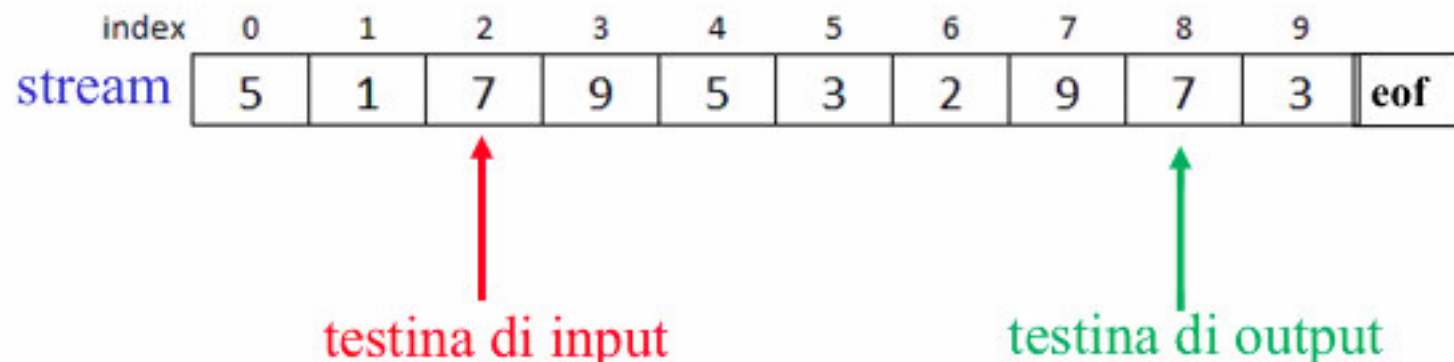
Esempi

```
fstream file("dati.txt", ios::in|ios::out);  
if (file.fail()) cout << "Errore in apertura\n";
```

Apri il file "dati.txt" in i/o.

Il posizionamento in una cella di uno stream associato ad un file si effettua tramite i seguenti metodi (indici nello stream partono da 0)

```
class istream: public virtual ios {
public:
    long tellg(); // ritorna la posizione di input nello stream
    istream& seekg(long p); // setta la posizione di input nello stream
    ...
};
class ostream: public virtual ios {
public:
    long tellp(); // ritorna la posizione di output nello stream
    ostream& seekp(long p); // setta la posizione di output nello stream
    ...
};
```



Il posizionamento in una cella di uno stream associato ad un file si effettua tramite i seguenti metodi (indici nello stream partono da 0).

```
class istream: public virtual ios {
public:
    long tellg(); // ritorna la posizione di input nello stream
    istream& seekg(long p); // setta la posizione di input nello stream
    ...
};
class ostream: public virtual ios {
public:
    long tellp(); // ritorna la posizione di output nello stream
    ostream& seekp(long p); // setta la posizione di output nello stream
    ...
};
```

Le costanti `ios::beg`, `ios::cur` e `ios::end` sono delle posizioni definite in `ios`:

`ios::beg` = posizione iniziale dello stream, cioè vale 0.

`ios::cur` = posizione corrente

`ios::end` = posizione finale dello stream, cioè la cella di EOF successiva all'ultimo byte dello stream

Esempio

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

int main(){ // trunc: crea se non esiste
    fstream f("dati.txt", ios::trunc|ios::in|ios::out);
    if ( f.fail()) cout << "Errore in apertura\n"; ...
    f << "Pippo";
    cout << f.tellp() << endl; // posizione testina di output: 5
```

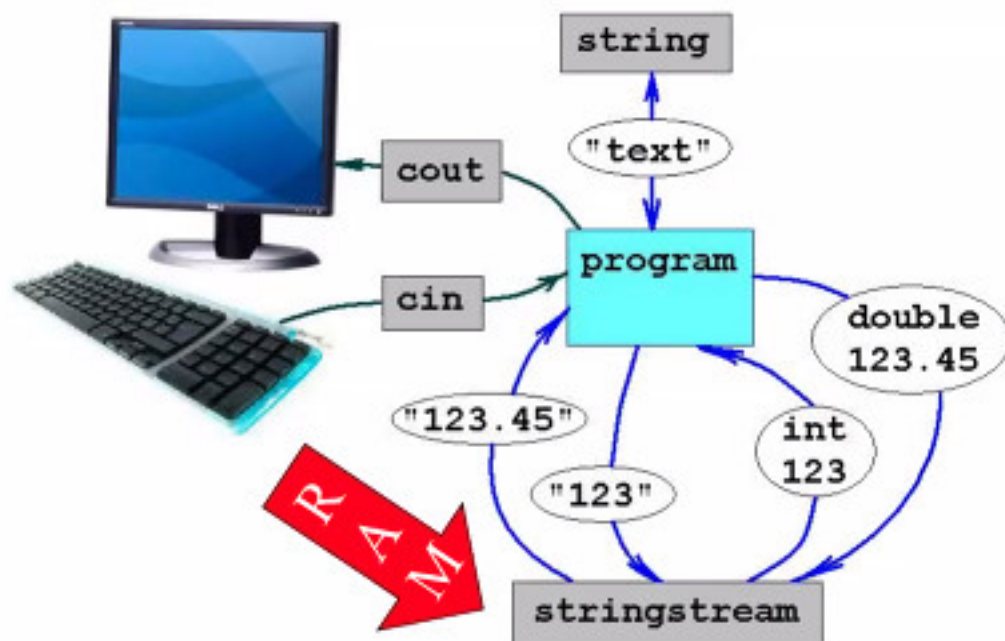
Esempio

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

int main(){
    fstream f("dati.txt", ios::trunc|ios::in|ios::out);
    if ( f.fail()) cout << "Errore in apertura\n"; ...
    f << "Pippo";
    cout << f.tellp() << endl;
    f.seekp(ios::beg);
    f << "Topolino";
    cout << f.tellp() << endl;
    f << " Pluto"; // append
    cout << f.tellp() << endl; // posizione testina di output: 14
    f.seekg(ios::beg); // testina di input all'inizio
    char c; while (f.get(c)) cout << c; // stampa: Topolino Pluto
}
```

Stream di stringhe

Si possono definire **stream associati a stringhe**, ossia sequenze di caratteri memorizzate in RAM (si parla anche di i/o in memoria). Il carattere nullo di terminazione gioca il ruolo di marcatore di fine stream.



Stream di stringhe

Le classi da utilizzare sono: `istringstream`, `ostringstream` e `stringstream`, il file header che le dichiara è `<sstream>`.

I costruttori sono i seguenti.

```
istringstream(const char* initial, int = ios::in);  
ostringstream(int = ios::out);  
ostringstream(const char* initial, int = ios::out);  
stringstream(int = ios::in|ios::out);  
stringstream(const char* initial, int = ios::in|ios::out);
```

I metodi di scrittura/lettura sono quelli ereditati da `istream`, `ostream` e `iostream`. Il metodo `str()` applicato ad uno stream di stringhe ritorna la stringa associata allo stream. Vediamo un esempio.

```

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

int main(){
    stringstream ss;
    ss << 236 << ' ' << 3.14 << "  pippo  "; // output su stringstream
    cout << ss.tellp() << ' ' << ss.tellg() << endl;
    // posizioni di testina di output e input: 17 0
    // la stringa in memoria è: "236 3.14  pippo  "
    // la testina di output è avanzata alla fine ios::end
    // la testina di input è ancora a ios::beg
    int i; ss >> i; // input da stringstream
    cout << i << endl; // stampa: 236
    double d; ss >> d; cout << d << endl; // stampa: 3.14
    string s; ss >> s; cout << "*" << s << "*\n"; // stampa: *pippo*
}

```

A cosa può servire?

Ad implementare input/output mediante **operator>>** ed **operator<<** su stringhe, ad esempio fornite dall'interazione con una GUI.

```

int main() {
    string x("(1a2.23,35)\n(12.23,a35)\n(12.23,35)\n(a14.2,5)\n");
    // ad esempio, stringa x ricevuta in input da GUI
    stringstream ss(x); // stringstream ss inizializzato con x
    Punto p; // posso invocare parsing di un Punto su ss
    while(ss.good()) { // while(stato == 0)
        ss >> p; // parsing di un Punto p
        if(ss.good()) { cout << "Input corretto di un Punto\n"; break; }
        else if(ss.fail()) {
            cout << "Input non valido!\n";
            ss.clear(ios::goodbit); char c=0;
            // 10 è il codice ASCII del carattere newline
            while(c!=10) { ss.get(c); } // svuota ss sino a newline
            ss.clear(ios::goodbit);
        }
        else ss.clear(ios::failbit);
    }
}

```



Exception handling

From Wikipedia, the free encyclopedia

Exception handling is the process of responding to the occurrence, during computation, of *exceptions* – anomalous or exceptional conditions requiring special processing – often changing the normal flow of program execution. It is provided by specialized programming language constructs or computer hardware mechanisms.

In general, an exception is *handled* (resolved) by saving the current state of execution in a predefined place and switching the execution to a specific subroutine known as an *exception handler*. If exceptions are *continuable*, the handler may later resume the execution at the original location using the saved information. For example, a floating point divide by zero exception will typically, by default, allow the program to be resumed, while an out of memory condition might not be resolvable transparently.

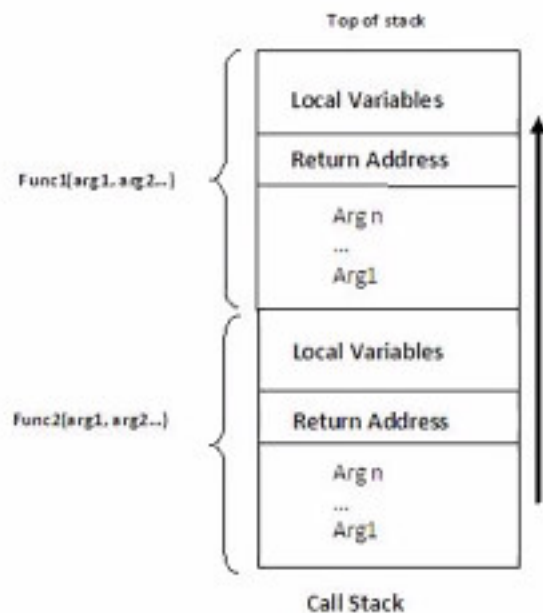
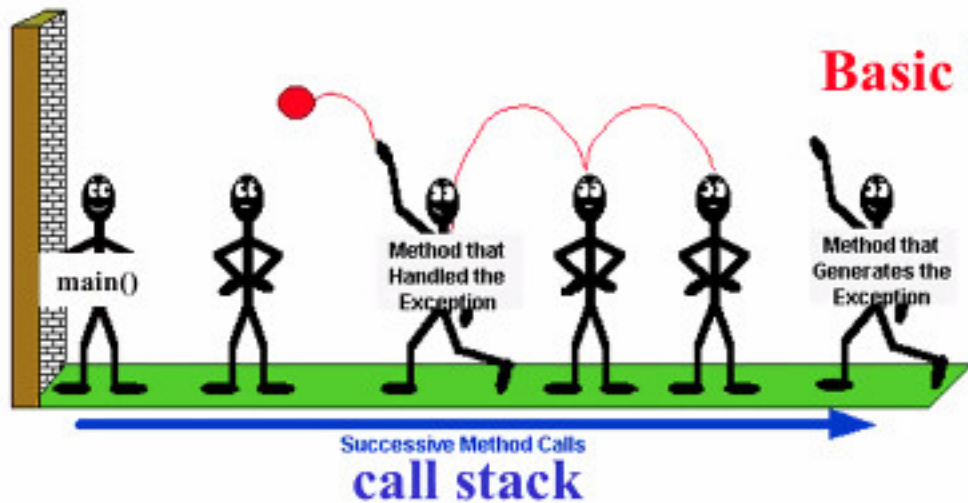
Exception support in programming languages [\[edit\]](#)

See also: *[Exception handling syntax](#)*

Many computer languages have built-in support for exceptions and exception handling. This includes [Actionscript](#), [Ada](#), [BlitzMax](#), [C++](#), [C#](#), [D](#), [ECMAScript](#), [Eiffel](#), [Java](#), [ML](#), [Object Pascal](#) (e.g. [Delphi](#), [Free Pascal](#), and the like), [PowerBuilder](#), [Objective-C](#), [OCaml](#), [PHP](#) (as of version 5), [PL/1](#), [PL/SQL](#), [Prolog](#), [Python](#), [REALbasic](#), [Ruby](#), [Scala](#), [Seed7](#), [Tcl](#), [Visual Prolog](#) and most [.NET](#) languages. Exception handling is commonly not resumable in those languages, and when an exception is thrown, the program searches back through the [stack](#) of function calls until an exception handler is found.



Basic idea



La funzione in cui si verifica la situazione eccezionale solleva (o lancia) una eccezione tramite una **throw**

```
telefonata bolletta::Estrai_Una() {  
    if (Vuota()) throw Ecc_Vuota();  
    telefonata aux = first->info;  
    first = first->next;  
    return aux;  
}
```

```
class Ecc_Vuota {};
```

Nella funzione chiamante:

```
int main() {  
    ...  
    try { b.Estrai_Una(); }  
    catch (Ecc_Vuota e) {  
        cerr << "La bolletta è vuota" << endl;  
        abort(); // definita in stdlib.h  
                // terminazione abnormale di programma  
    }  
    ...  
}
```

function

abort

<stdlib>

C C++11 ?

```
void abort (void);
```

Abort current process

Aborts the current process, producing an abnormal program termination.

The function raises the SIGABRT signal (as if `raise(SIGABRT)` was called). This, if uncaught, causes the program to terminate returning a platform-dependent *unsuccessful termination* error code to the host environment.

parsing di **orario**
in formato
hh:mm:ss


```
istream& operator>>(istream& is,
                    orario& o) {
    // formato di input: hh:mm:ss
    char c; int ore, minuti, secondi;
    string::size_type pos;
    string cifre("0123456789");
    is >> c; // prima cifra delle ore
    pos = cifre.find(c); ore = pos;
    is >> c;
    if (c != ':') {
        // seconda cifra delle ore
        pos = cifre.find(c);
        ore = ore * 10 + pos;
        is >> c; // input di ':'
    } // ho letto le ore e c = ':'
    is >> c; // prima cifra dei minuti
    pos = cifre.find(c);
    minuti = pos;
    is >> c;
    if (c != ':') {
        // seconda cifra dei minuti
        pos = cifre.find(c);
        minuti = minuti * 10 + pos;
        is >> c; // input di ':'
    } // ho letto i minuti e c = ':'
```

```
is >> c; // prima cifra dei secondi
pos = cifre.find(c);
secondi = pos;
is >> c;
if(is && cifre.find(c) != string::npos) {
    // per cin, is == 0 con Ctrl-D
    // seconda cifra secondi
    pos = cifre.find(c);
    secondi = secondi * 10 + pos;
} // ho letto i secondi
else if (is) // carattere non cifra
is.putback(c);
o.sec = ore*3600 + minuti*60 + secondi;
return is;
}
```

parsing di **orario**
in formato
hh:mm:ss

Si possono verificare varie situazioni di errore in questa funzione. Il parser non è “**robusto**”

Definiamo le seguenti classi di eccezioni

```
class err_sint {};      // errore di sintassi  
class fine_file {};    // file finito prematuramente  
class err_ore {};      // ora > 23  
class err_minuti {};   // minuti > 59  
class err_secondi {};  // secondi > 59
```

```

istream& operator>>(istream& is,
                    orario& o) {
    char c; string::size_type pos;
    string cifre("0123456789");
    int ore, minuti, secondi;
    if (!(is >> c)) throw fine_file();
    // prima cifra ore
    pos = cifre.find(c);
    if (pos == string::npos)
        throw err_sint();
    ore = pos;
    if (!(is >> c)) throw fine_file();
    if (c != ':') {
        // seconda cifra ore
        pos = cifre.find(c);
        if (pos == string::npos)
            throw err_sint();
        ore = ore * 10 + pos;
        if (ore > 23) throw err_ore();
        if (!(is >> c))
            throw fine_file();
    }
    // ore lette, c deve essere ':'
    if (c != ':') throw err_sint();
    if (!(is >> c)) throw fine_file();
    // prima cifra minuti
    pos = cifre.find(c);
    if (pos == string::npos)
        throw err_sint();
    minuti = pos;
    if (!(is >> c)) throw fine_file();

```

```

    if (!(is >> c)) throw fine_file();
    if (c != ':') {
        // seconda cifra minuti
        pos = cifre.find(c);
        if (pos == string::npos)
            throw err_sint();
        minuti = minuti * 10 + pos;
        if (minuti > 59) throw err_minuti();
        if (!(is >> c)) throw fine_file();
    }
    // minuti letti, c deve essere ':'
    if (c != ':') throw err_sint();
    if (!(is >> c)) throw fine_file();
    // prima cifra secondi
    pos = cifre.find(c);
    if (pos == string::npos) throw err_sint();
    secondi = pos;
    is >> c;
    if (is && cifre.find(c) != string::npos){
        // per cin, is==0 con Ctrl-D
        // seconda cifra secondi
        pos = cifre.find(c);
        secondi = secondi * 10 + pos;
        if (secondi > 59) throw err_secondi();
    } // ho letto i secondi
    else if (is) // carattere non cifra
        is.putback(c);
    o.sec = ore*3600 + minuti*60 + secondi;
    return is;
}

```

parsing **robusto** di **orario** in formato **hh:mm:ss**

Una funzione esterna che chiede in input da cin due orari da sommare:

```
orario sommaDueOrari() {  
    orario o1,o2;  
    try { cin >> o1; } // può sollevare eccezioni  
    catch (err_sint)  
        {cerr << "Errore di sintassi"; return orario();}  
    catch (fine_file)  
        {cerr << "Errore fine file"; abort();}  
    catch (err_ore)  
        {cerr << "Errore nelle ore"; return orario();}  
    catch (err_minuti)  
        {cerr << "Errore nei minuti"; return orario();}  
    catch (err_secondi)  
        {cerr << "Errore nei secondi"; return orario();}  
    try { cin >> o2; } // può sollevare eccezioni  
    catch (err_sint)  
        {cerr << "Errore di sintassi"; return orario();}  
    catch (fine_file)  
        {cerr << "Errore fine file"; abort();}  
    catch (err_ore)  
        {cerr << "Errore nelle ore"; return orario();}  
    catch (err_minuti)  
        {cerr << "Errore nei minuti"; return orario();}  
    catch (err_secondi)  
        {cerr << "Errore nei secondi"; return orario();}  
    return o1+o2;  
}
```

Utilizziamo un unico blocco try:

```
orario sommaDueOrari() {  
    try {  
        orario o1, o2;  
        cin >> o1 >> o2;  
        return o1 + o2;  
    }  
    catch (err_sint)  
        {cerr << "Errore di sintassi"; return orario();}  
    catch (fine_file)  
        {cerr << "Errore fine file"; abort();}  
    catch (err_ore)  
        {cerr << "Errore nelle ore"; return orario();}  
    catch (err_minuti)  
        {cerr << "Errore nei minuti"; return orario();}  
    catch (err_secondi)  
        {cerr << "Errore nei secondi"; return orario();}  
}
```


Una `throw` può sollevare una espressione di **qualsiasi tipo**.

```
// enum di eccezioni invece che classi distinte
enum Errori {ErrSintassi, ErrFineFile, ErrOre,
             ErrMinuti, ErrSecondi};

...

    if (secondi > 59) throw ErrSecondi;
// invece di
// if (secondi > 59) throw err_secondi();

...
```

Flusso del controllo provocato da una throw

Quando in una funzione **F** viene sollevata una eccezione di tipo **T** tramite una istruzione **throw** inizia la **ricerca della clausola catch** in grado di catturarla.

① Se l'espressione **throw** è collocata in un blocco **try** nel **corpo della stessa funzione F**, l'esecuzione abbandona il blocco **try** e vengono esaminate in successione tutte le **catch** associate a tale blocco.

Poco sensato!

Flusso del controllo provocato da una throw

Quando in una funzione **F** viene sollevata una eccezione di tipo **T** tramite una istruzione **throw** inizia la **ricerca della clausola catch** in grado di catturarla.

- ① Se l'espressione **throw** è collocata in un blocco **try** nel **corpo della stessa funzione F**, l'esecuzione abbandona il blocco **try** e vengono esaminate in successione tutte le **catch** associate a tale blocco.
- ② Se si trova un **type match** per una **catch** l'eccezione viene catturata e viene eseguito il codice della **catch**; eventualmente, al termine dell'esecuzione del corpo della **catch** il controllo dell'esecuzione passa al punto di programma che segue l'ultimo blocco **catch**.

Poco sensato!

Flusso del controllo provocato da una throw

Quando in una funzione **F** viene sollevata una eccezione di tipo **T** tramite una istruzione **throw** inizia la ricerca della clausola **catch** in grado di catturarla.

- ① Se l'espressione **throw** è collocata in un blocco **try** nel corpo della stessa funzione **F**, l'esecuzione abbandona il blocco **try** e vengono esaminate in successione tutte le **catch** associate a tale blocco.
- ② Se si trova un type match per una **catch** l'eccezione viene catturata e viene eseguito il codice della **catch**; eventualmente, al termine dell'esecuzione del corpo della **catch** il controllo dell'esecuzione passa al punto di programma che segue l'ultimo blocco **catch**.
- ③ Se non si trova un type match per una **catch** oppure se l'istruzione **throw** non era collocata all'interno di un blocco **try** della stessa funzione **F** la ricerca continua nella **funzione che ha invocato la funzione F**.

Caso più comune

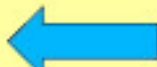
Flusso del controllo provocato da una throw

Quando in una funzione **F** viene sollevata una eccezione di tipo **T** tramite una istruzione **throw** inizia la ricerca della clausola **catch** in grado di catturarla.

- ① Se l'espressione **throw** è collocata in un blocco **try** nel corpo della stessa funzione **F**, l'esecuzione abbandona il blocco **try** e vengono esaminate in successione tutte le **catch** associate a tale blocco.
- ② Se si trova un type match per una **catch** l'eccezione viene catturata e viene eseguito il codice della **catch**; eventualmente, al termine dell'esecuzione del corpo della **catch** il controllo dell'esecuzione passa al punto di programma che segue l'ultimo blocco **catch**.
- ③ Se non si trova un type match per una **catch** oppure se l'istruzione **throw** non era collocata all'interno di un blocco **try** della stessa funzione **F** la ricerca continua nella funzione che ha invocato la funzione **F**.
- ④ Questa **ricerca top-down sullo stack** delle chiamate di funzioni continua fino a che si trova una **catch** che cattura l'eccezione o si arriva alla funzione **main** nel qual caso viene richiamata la funzione di libreria **terminate()** che per default chiama la funzione **abort()** che fa terminare il programma in errore.

Rilanciare un'eccezione

È possibile che una clausola `catch` si accorga di non poter gestire direttamente una eccezione. In tal caso essa può **rilanciare** l'eccezione alla funzione chiamante con una `throw`.

```
orario somma() try {  
    orario t1, t2;  
    cin >> t1 >> t2;  
    return t1 + t2;  
}  
catch (err_sint)  
    {cerr << "Errore di sintassi"; return orario();}  
catch (fine_file)  
    {cerr << "Errore fine file"; throw; }   
catch (err_ore)  
    {cerr << "Errore nelle ore"; return orario();}  
catch (err_minuti)  
    {cerr << "Errore nei minuti"; return orario();}  
catch (err_secondi)  
    {cerr << "Errore nei secondi"; return orario();}
```

Houston, do we have a problem?

```
class A {  
public: ~A() {cout << "~A ";}  
};  
  
void F() { A* p = new A[3]; throw 1; delete[] p;};  
  
int main() {  
    try { F(); }  
    catch (int) {cout << "Eccezione int ";}  
    cout << "Fine ";  
}  
// stampa: Eccezione int Fine  
// ovviamente non stampa: ~A ~A ~A
```


Utilizzo di risorse

```
gestore () {  
    risorsa rs; // alloco la risorsa  
    rs.use();  
    ...  
    ... // codice che può sollevare eccezioni  
    ...  
    rs.release(); // non viene eseguita in caso  
                  // di eccezione  
}
```

Se viene sollevata una eccezione e questa non viene catturata all'interno della funzione si esce dalla funzione senza rilasciare la risorsa. Ad esempio, la risorsa è la memoria e quindi si potrebbe provocare **garbage**.



Clausola catch generica

```
gestore () try {  
    risorsa rs;  
    rs.use();  
    ... // codice che può sollevare eccezioni  
    rs.release(); // non viene eseguita in caso  
                  // di eccezione  
}  
  
catch (...) {  
    rs.release();  
    throw; // rilancio l'eccezione al chiamante  
}
```

Match del tipo delle eccezioni

La catch che cattura un'eccezione di tipo **E** è la prima catch incontrata durante la ricerca che abbia un *tipo **T** compatibile con **E***.

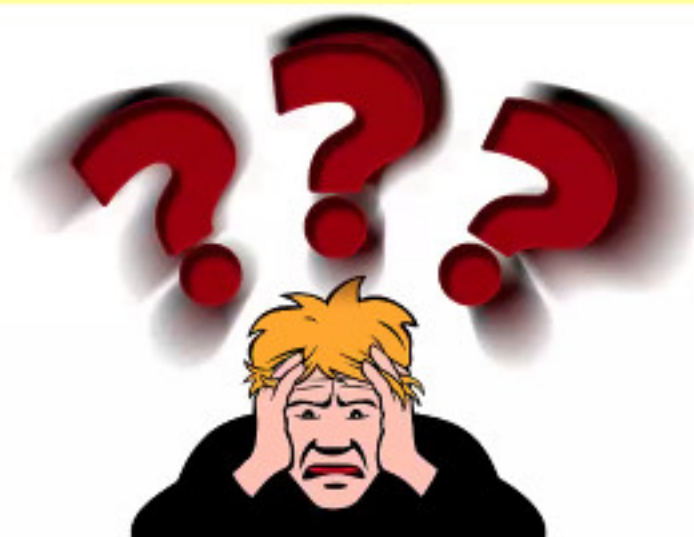
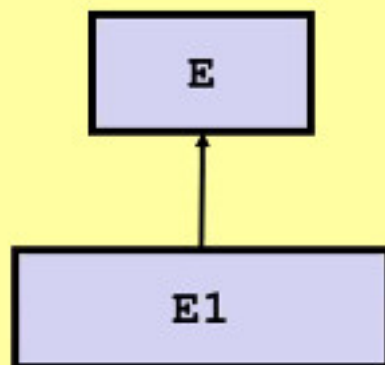
Le regole che definiscono la compatibilità tra il tipo **T** del parametro di una catch non generica ed il tipo **E** dell'eccezione sono le seguenti:

- Il tipo **T** è uguale al tipo **E**;
- Il tipo **E** è un sottotipo di **T**, ovvero:
 - ✓ **E** è un sottotipo derivato pubblicamente da **T**;
 - ✓ **T** è un tipo puntatore **B*** ed **E** è un tipo puntatore **D*** dove **D** è un sottotipo di **B**
 - ✓ **T** è un tipo riferimento **B&** ed **E** è un tipo riferimento **D&** dove **D** è un sottotipo di **B**
- **T** è il tipo **void*** ed **E** è un qualsiasi tipo puntatore
- **Non possono** essere applicate conversioni implicite.


```
class E { public: virtual ~E() {} };  
class E1: public E {};
```

```
void modify(vector<int>& v) {  
    ...  
    if(v.size()==0) throw new E();  
    if(v.size()==1) throw new E1();  
    ...  
}
```

```
void G(vector<int>& v) {  
    try{  
        modify(v);  
    }  
    catch(E* p) {...}  
    catch(E1* q) {...}  
}
```



Comportamenti tipici di una clausola catch sono i seguenti:

- rilanciare un'eccezione
- convertire un tipo di eccezione in un altro, rimediando parzialmente e lanciando un'eccezione diversa
- cercare di ripristinare il funzionamento, in modo che il programma possa continuare dall'istruzione che segue l'ultima catch
- analizzare la situazione che ha causato l'errore, eliminarne eventualmente la causa e riprovare a chiamare la funzione che ha causato originariamente l'eccezione
- esaminare l'errore ed invocare **std::terminate()**

Specifica esplicita delle eccezioni (alla Java)

```
istream& operator>>(istream& is, orario& t)  
    throw(err_sint, fine_file, err_ore, err_secondi,  
          err_minuti) {  
    ...  
}
```

Deprecata da C++11



Specifica delle eccezioni deprecata

Problemi nella specifica delle eccezioni

- **Run-time checking:** il test di conformità delle eccezioni avviene a run-time e non a compile-time, quindi non vi era una garanzia statica di conformità.
- **Run-time overhead:** Run-time checking richiede al compilatore del codice aggiuntivo che potrebbe inficiare alcune ottimizzazioni.
- **Inutilizzabile con i template:** in generale i parametri di tipo dei template non permettono di specificare le eccezioni.

Qt e le eccezioni



Il modulo GUI di Qt non usa le eccezioni. Perché?

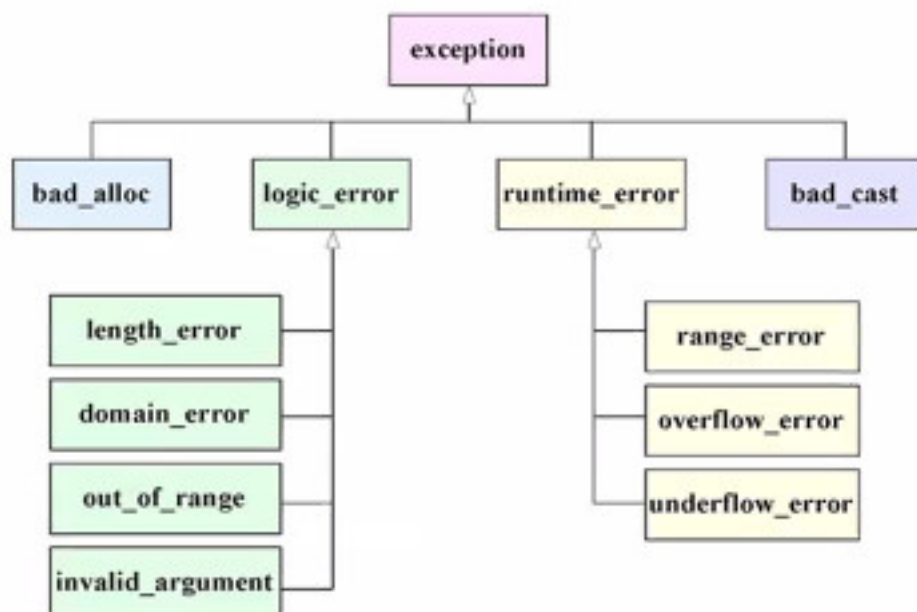
Risposta ufficiale:

“When Qt was started, exceptions were not available for all the compilers that needed to be supported by Qt. Today we are trying to keep the APIs consistent, so modules that have a history of not using exceptions will generally not get new code using exceptions added.

You will notice exceptions are used in some of the new modules of Qt.”

La gerarchia **exception**

Il C++ standard prevede una gerarchia di classi di eccezioni predefinita.

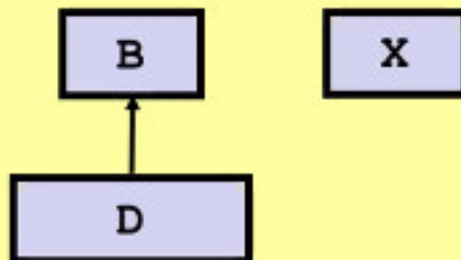


exception è la classe base, da cui derivano **runtime_error** e **logic_error**, da cui derivano parecchie classi.

Se il `dynamic_cast` di un riferimento fallisce allora viene automaticamente lanciata un'eccezione di tipo `bad_cast`.

```
class X { public: virtual ~X() {} };
class B { public: virtual ~B() {} };
class D : public B {};

#include<typeinfo>
#include<iostream>
using namespace std;
int main() {
    D d;
    B& b = d; // upcast
    try {
        X& xr = dynamic_cast<X&>(b);
    } catch(bad_cast e) {
        cout << "Cast fallito!" << endl;
    }
}
```



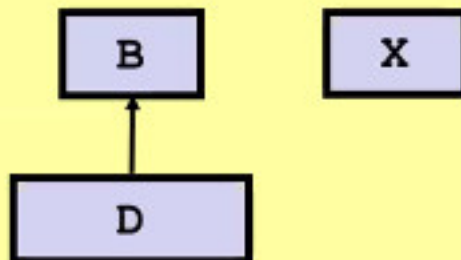
Derivano da **exception** anche le seguenti classi di eccezioni:

- **bad_cast**, le cui eccezioni sono lanciate dal **dynamic_cast** per riferimenti
- **bad_alloc**, lanciata dalla **new** quando lo heap è esaurito (il gestore di default invoca la **terminate()**).
- **bad_typeid**, viene lanciata dall'operatore **typeid** quando ha come argomento un puntatore nullo.

Se il `dynamic_cast` di un riferimento fallisce allora viene automaticamente lanciata un'eccezione di tipo `bad_cast`.

```
class X { public: virtual ~X() {} };
class B { public: virtual ~B() {} };
class D : public B {};

#include<typeinfo>
#include<iostream>
using namespace std;
int main() {
    D d;
    B& b = d; // upcast
    try {
        X& xr = dynamic_cast<X&>(b);
    } catch(bad_cast e) {
        cout << "Cast fallito!" << endl;
    }
}
```



CAUTION

CONSTRUCTIVE

CRITICISM

IN PROGRESS



Turing award



Criticism [\[edit \]](#)

A contrasting view on the safety of exception handling was given by [C.A.R Hoare](#) in 1980, described the [Ada programming language](#) as having "...a plethora of features and notational conventions, many of them unnecessary and **some of them, like exception handling, even dangerous.** [...] Do not allow this language in its present state to be used in applications where reliability is critical[...]. The next rocket to go astray as a result of a programming language error may not be an exploratory space rocket on a harmless trip to Venus: It may be a nuclear warhead exploding over one of our own cities." ^[14]

Citing multiple prior studies by others (1999–2004) and their own results, Weimer and Nacula wrote that a significant problem with **exceptions** is that they **"create hidden control-flow paths that are difficult for programmers to reason about"**.^{[9]:8:27}