Exceptii

Erorile apar datorita greselilor de programare, logica; exceptiile apar datorita unor probleme aparute in timpul rularii (depasire de memorie, date in afara limitelor prescrise).

Problema principala in cazul acestor din urma erori este transmiterea erorii de la codul care gaseste eroarea la codul care trateaza eroarea. In limbajul C (sau in alte limbaje procedurale) aceasta se face cu ajutorul unor valori speciale de intoarcere din functii sau setand anumite flag-uri. Dezavantajele principale ale acestei abordari sunt ca pot exista biblioteci care nu respecta conventiile de intors sau nu intorc niciodata coduri de eroare, si in plus un cod care la fiecare apel de functie trateaza aparitia unei erori se poate transfoma in ceva ilizibil. De asemenea, putem avea situatii in care toata gama valorilor de intors este valida.

Solutia la aceste probleme a fost gasita in limbaje ca Smalltalk, C++, Java sau C# prin tratarea erorilor cu ajutorul unor mecanisme numite exceptii.

Exceptiile sunt mecanisme menite a trata situatiile exceptionale. O situatie exceptionala se intalneste atunci cand este exista o problema care impiedica continuarea metodei curente (nu se poate contiua pentru ca nu exista suficiente informatii la acest nivel pentru a continua). In acest caz se semnaleaza aceasta situatie exceptionala printr-o exceptie unui nivel mai inalt (conceptual) al programului care poate trata exceptia aparuta. Un exemplu foarte simplu este atunci cand se apeleaza o metoda , aceasta in interiorul ei determina o situatie exceptionala (de exemplu functia realizeaza o impartire 0) si anunta aceasta metodei apelante. Aceasta va sti cum sa trateze aceasta exceptie (eventual va apela din nou aceeasi metoda cu alti parametrii etc).

Variantele de tratare a exceptiilor ar fi:

- Abandonarea programului
- Informarea utilizatorului si terminarea programului
- Informarea utilizatorului, astfel incat acesta sa ia masuri pentru continuarea programului
- Actiuni automate de corectare, fara a informa utilizatorul.

De exemplu, headerul C standard **<errno.h>** defineste un mecanism pentru examinarea si atribuirea de valori unei variabile globale, numite **errno**, cu rol de semnalizarea a succesului/esecului ultimei operatii. Intr-un mediu cu mai multe fire de executie, codul de eroare ar putea fi modificat de un alt fir de executie inainte ca firul curent sa aiba sansa de a-l examina

C++ ofera o metoda integrata si sigura pentru tratarea exceptiilor predictibile ce pot apare in timpul rularii programelor.

O exceptie este un obiect care este pasat din zona codului unde a aparut o eroare, secventei de tratare a problemei, functie de tipul exceptiei - transmiterea poate fi facuta prin valoare sau referinta.

Pasii care trebuiesc in general urmati in vederea tratarii exceptiilor in cadrul programelor C++ sunt:

1. se identifica acele zone din program in care se efectueaza o operatie despre care se cunoaste ca ar putea genera o exceptie, si se marcheaza in cadrul unui bloc de tip *try*. In cadrul acestui bloc, se testeaza conditia de aparitie a exceptiei, si in caz pozitiv se semnaleaza aparitia exceptiei prin intermediul cuvantului cheie *throw*;

2. se realizeaza blocuri de tip *catch* pentru a capta exceptiile atunci cand acestea sunt intalnite.

Blocurile *catch* urmeaza un bloc *try*, in cadrul carora sunt tratate exceptiile.

Blocuri *try* sunt create pentru a include codul unde pot apare probleme. Sintaxa pentru *try*:

```
try{
       // cod
      throw TipExceptie;
}
```

Semnalarea exceptiilor in blocurile *try* se face prin instructiunea *throw*, care trebuie executata in interiorul blocului try sau intr-o functie apelata din bloc. Sintaxa pentru *throw*:

```
throw TipExceptie;
```

Sintaxa pentru *catch*:

```
catch (TipExceptie)
{
      // cod tratare exceptie
}
```

Daca *TipExceptie* este "...", este captata orice exceptie aparuta.

Dupa un bloc *try*, pot urma unul sau mai multe blocuri *catch*. Daca exceptia corespunde cu una din declaratiile de tratare a exceptiilor, aceasta este apelata. Daca nu exista definita nici o rutina de tratare a exceptiei, este apelata rutina predefinita, care incheie executia programului in curs. Dupa ce rutina este executata, programul continua cu instructiunea imediat urmatoare blocului *try*.

O exceptie poate fi relansata, dintr-un catch, prin throw. Aceasta face ca exceptia curenta sa fie retransmisa unei secvente try/catch exterioare. Motivul retransmiterii este sa se permita mai multor manipulatori catch, accesul la aceleasi exceptii. La relansare, o exceptie nu va fi preluata de aceeasi instructiune catch, ci se va deplasa spre urmatoarea.

TipExceptie nu este altceva decat instantierea unei clase vide (care determina tipul exceptiei), putand fi declarat ca:

```
class TipExceptie {};
```

Exemplu:

Se semnaleaza o exceptie de tip intreg. Se observa la executie faptul ca dupa semnalarea erorii prin throw, instructiile care urmeaza in blocul try nu se mai executa, controlul fiind predat blocului catch.

Iesirea programului va fi:

Intrare in main

Intrare in blocul try

Exceptia tratata este:10

Terminare main

```
#include <iostream.h>
int main() {
  cout << "Intrare in main" << endl;
```

```
  try {
    cout << "Intrare in blocul try" << endl;
    throw 10; //lansare exceptie, se preda controlul blocului catch
    cout << "Instructiune dupa throw" << endl; //nu se va tipari
  } //terminare bloc try
  catch (int e) { //linie marcata
    cout << "Exceptia tratata este:" << e << endl;
  }
  cout << "Terminare main" << endl;
  return 0;
}
```

Daca linia marcata se inlocuieste cu:

```
catch (float e){
```

programul se va termina anormal, din cauza ca exceptia de tip intreg semnalata in try, nu gaseste un bloc catch pentru tratare.

Exemplu:

Semnalarea exceptiei se face intr-o functie apelata in blocul *try*. Iesirea programului va fi:

Intrare in main
Intrare in blocul try
Intrare in functie cu valoarea:-4
Exceptia tratata este:-4
Terminare main

```
#include <iostream.h>
void functie(int i) {
  cout << "Intrare in functie cu valoarea:" << i << endl;
  if (i)
    throw i;
  cout << "In functie dupa throw" << endl; //nu se va tipari daca se semnaleaza exceptie
}


int main() {
  cout << "Intrare in main" << endl;
  try {
    cout << "Intrare in blocul try" << endl;
    functie(-4);
    functie(0);
    functie(10);
    cout << "Terminare try" << endl;
  } //terminare bloc try
  catch (int e) {
    cout << "Exceptia tratata este:" << e << endl;
  }
  cout << "Terminare main" << endl;
  return 0;
}
```

Exemplu:

In programul de mai jos se defineste clasa Array; se semnaleaza exceptii, obiecte ale clasei vide *BoundaryException* daca indicii sunt in afara domeniului.

De fapt, pentru clasa `BoundaryException`, compilatorul creaza implicit constructor, destructor, copy constructor, copy operator.

A se observa definirea constructorului de copiere, a operatorului de copiere =, dubla supraincarcare pentru operatorul de indexare [] - pentru cazul cand se citeste sau se modifica valoarea unui element de tablou si supraincarea operatorului de iesire << (declarat ca si functie friend).

La rulare se va observa ca dupa initializarea elementelor tabloului (indicii intre 0 si 19), pentru indicele 20 se semnaleaza exceptie, tratata in *catch*, dupa care se iese din functia *main*.

```
#include <iostream.h>

const int DefaultSize = 10;

class Array {
  private:
       int *pType; //adresa tabloului
       int  size;  //numar elemente
  public:
    // constructori
      Array(int value = DefaultSize);
      Array(const Array &anArray);  //tabloul se construieste ca fiind copia
celui parametru
       ~Array() {
        delete [] pType;
      }
    // operatori
      Array& operator= (const Array&);
      int& operator[] (int offset);
      const int& operator[](int offset) const;
    // metode
      int getSize() const {
        return size;
      }
    // functie friend
      friend ostream& operator<< (ostream&, const Array&);
    // se defineste clasa BoundaryException (vida) -obiectele ei vor fi exceptii
      class BoundaryException {};
};

Array::Array(int value): size(value) {
      pType = new int[size];
      for (int i = 0; i < size; i++)
        pType[i] = 0; //se initializeaza elementele cu 0
}

Array& Array::operator= (const Array &anArray) {
     if (this == &anArray)
        return *this;

     delete [] pType;
     size = anArray.getSize();
     pType = new int[size];
     for (int i = 0; i < size; i++)
```

```
       pType[i] = anArray[i];
     return *this;
}

Array::Array(const Array &anArray){
     size = anArray.getSize();
     pType = new int[size];
     for (int i = 0; i < size; i++)
        pType[i] = anArray[i];
}

int& Array::operator[](int offset){
     int size = getSize();
     if (offset >= 0 && offset < size)
        return pType[offset];
     throw BoundaryException ();

     return pType[0]; // nu se ajunge aici, se face return doar pentru a nu
apare eroare in compilare
}

const int& Array::operator[](int offset) const{
     if (offset >= 0 && offset < getSize())
        return pType[offset];
     throw BoundaryException ();

     return pType[0]; // nu se ajunge aici, se face return doar pentru a nu
apare eroare in compilare
}

ostream& operator<< (ostream& output, const Array& theArray){
     for (int i = 0; i < theArray.getSize(); i++)
        output << "[" << i << "] " << theArray[i] << endl;
     return output;
}

void main(void) {
     Array intArray(20);

     try{
        for (int j = 0; j < 30; j++){
           intArray[j] = j;
           cout << "intArray[" << j << "] initializat" << endl;
        }
     }
     catch (Array::BoundaryException) {
        cout << "Indice eronat!\n";
     }

     cout << "Terminare main\n";
}
```

Datorita faptului ca exceptia este instantierea unei clase, prin derivare pot fi realizate adevarate ierarhii de tratare a exceptiilor. Trebuie avuta insa in vedere posibilitatea de aparitie a unor exceptii chiar in cadrul codului de tratare a unei exceptii, situatii ce trebuie evitate.

O exceptie poate fi transferata handler-ului prin valoare sau prin referinta. Memoria pentru obiectele de tip exceptie este alocata de o maniera nespecificata; unele implementari utilizeaza o

stiva dedicata. Pasarea exceptiei prin referinta asigura comportamentul polimorfic. Pasarea exceptiei prin valoare poate fi costisitoare deoarece obiectul poate fi distrus si reconstituit de cateva ori pana cand este atins handler-ul corespunzator!

Tipul exceptiei determina handler-ul care o va captura si trata. Regulile de potrivire a tipurilor exceptiilor permit doar un set limitat de conversii. Pentru o exceptie E si un handler care captureaza exceptii de tipul T sau T&, potrivirea este valida daca:

- T si E sut de acelasi tip (ignorand const si volatile);
- T este o clasa baza publica a lui E.

O functie care poate genera o exceptie specifica acest lucru in prototipul sau.

Exemplu:

```
class Zerodivide{/**/};
int divide(int, int) throw (Zerodivide);
//poate genera doar exceptii de tipul Zerodivide!
```

O functie care NU genereaza exceptii se declara astfel:

```
bool equals(int, int) throw();
```

O functie precum

```
bool equals(int, int);
```

nu garanteaza nimic in legatura cu exceptiile: poate genera orice tip de exceptie, sau nu va genera exceptii!

O specificare de exceptie nu se poate verifica decat la executie; daca o functie incearca sa genereze o exceptie de un tip nespecificat in prototipul ei, mecanismul de tratare a exceptiilor detecteaza acesta violare si apeleaza functia standard **unexpected()**. Deoarece o violare a specificarii de exceptie este mai degraba un bug, si nu ar trebui sa se produca, comportamentul implicit al functiei unexpected() este sa apeleze functia **terminate()**, care termina programul. Acest comportament poate fi modificat cu functia **set_unexpected()**.

C++ aplica o politica prin care limbajul acorda incredere programatorului; astfel, el ignora un cod care pare sa violeze specificatia de exceptie, precum cel din exemplul urmator:

```
int f(); //f poate genera exceptii de orice tip
void g(int j) throw() { //g nu genereaza exceptii
      int result = f();
}
```

Daca *f* genereaza o exceptie, *g* va viola garantia de a nu genera exceptii, cu toate acestea, acest cod este legal! Desi astfel de situatii pot fi verificate la compilare, verificarea specificatiilor de exceptie se realizeaza numai la executie. Programatorul nu este fortat intr-o situatie de acest gen sa scrie cod **try ...catch** inutil, deoarece *f* poate fi, de exemplu, o functie C din biblioteca standard!

C++ solicita concordanta specificarii exceptiilor in clasele derivate: supraincarcarea unei functii virtuale intr-o clasa derivata trebuie sa aiba o specificare de exceptie care este, cel putin, la fel de restrictiva precum specificarea de exceptie a functiei din clasa de baza.

Exemplu:

```
class BaseEx{};
class DerivedEx: public BaseEx{};
```

```
class OtherEx{};
class A{
      public:
      virtual void f() throw (BaseEx);
      virtual void g() throw (BaseEx);
      virtual void h() throw (DerivedEx);
      virtual void i() throw (DerivedEx);
      virtual void j() throw (BaseEx);
};
class D: public A{
      public:
      virtual void f() throw (DerivedEx); //ok
      virtual void f() throw (OtherEx);   //error
      virtual void g() throw (DerivedEx); //ok
      virtual void i() throw (BaseEx);    //error
      virtual void j() throw (BaseEx, OtherEx); //error
};
```

Aceleasi reguli de concordanta se aplica pointerilor la functii: unui pointer la o functie care poseda o specificatie de exceptie i se poate atribui doar o functie care poseda o specificatie de exceptie identica sau mai restrictiva! In consecinta, unui pointer la o functie care nu are specificatie de exceptie nu i se poate atribui o functie care poseda o specificatie de exceptie! Specificarea de exceptie nu se considera parte a tipului functiei; deci nu se pot declara functii care sa difere numai prin tipul de exceptie generat! De asemenea, declararea unui typedef care contine o specificatie de exceptie este o eroare:

```
void f(int) throw(Y);

void f(int) throw(Z); //eroare

typedef void (*PF)(int) throw (Exception); //eroare
```

Generarea unei exceptii dintr-un destructor este o tehnica periculoasa: destructorul poate sa fi fost invocat datorita unei alte exceptii, ce determina iesirea obiectului din scop. In acest caz, mecanismul de tratare invoca *terminate()*. Daca trebuie totusi sa se genereze o exceptie dintr-un destructor, trebuie sa se verifice daca nu cumva este procesata, in mod curent, o exceptie necapturata inca (adica nu s-a intrat inca in handler-ul corespunzator). Se utilizeaza functia standard **uncaught_exception()** din headerul **<stdexcept>**:

```
class FileException{};
File::~File() throw (FileException){
      if (close(file_handler)!=succes){
            if (uncaught_exception() == true) return;
            //este sigur sa se genereze o exceptie
            throw FileException(); //semnalizam eroare
      };
      return; //succes
};
```

Un design mai bun trateaza exceptiile in destructor decat sa le lase sa se propage mai departe!

```
void cleanUp() throw(int);
class C{
      public:
      ~C();
};

C::~C(){
```

```
        try {
              cleanUp();
        }
        catch (int){/*tratarea exceptiei */}
}
```

Pentru un obiect global, constructorul se apeleaza inainte de main(); o exceptie generata de constructor nu poate fi, in acest caz, capturata! Analog, distrugerea unui obiect global se realizeaza dupa terminarea lui main(); o exceptie generata de destructor nu poate fi, in acest caz, capturata!

Limbajul C++ defineste o ierarhie de exceptii standard, derivate din **std::exception**. Se pot captura astfel aceste exceptii cu un singur catch:

```
catch(std::exception& ex){
      //trateaza exceptiile de tipul std::exception si pe cele derivate din ea
}
```

Temă

1. Se cere sa se defineasca o clasa Queue care modeleaza functionarea unei cozi cu elemente de tip char. Dimensiunea cozii este limitata si se specifica la creare. Se semnaleaza exceptie in constructor daca dimensiunea este <=0, daca se incearca extragerea unui element atunci cand coada este vida sau daca coada este plina.

2. Scrieti o functie ce identifica prin metoda cautarii binare daca un cuvant apartine unui tablou de siruri de caractere. Sa se notifice utilizatorul printr-o exceptie situatia in care cuvantul nu este gasit.

3. Scrieti o functie ce determina intr-un arbore binar de cautare cu elemente de tip siruri de caractere, daca un cuvant apartine sau nu respectivei multimi. Sa se notifice utilizatorul printr-o exceptie situatia in care cuvantul nu este gasit.

4. Sa se modifice clasa *Matrice*, prezentata intr-unul din laboratoarele anterioare, punandu-se in evidenta situatiile exceptionale.

5. Scrieti un program care citeste repetat numere intregi pana la introducerea valorii 0. Intregii se vor citi ca siruri de caractere, semnalandu-se exceptii la valori in afara domeniului *int* si la tastarea caracterelor invalide

Bibliografie

http://inf.ucv.ro/~mirel/courses/poo/poo.html

1. *The C++ Programming Language*, B. Stroustrup, Addison-Wesley.
(Cap. 14 Exception handling)