Tratarea Excepțiilor

Mihai Gabroveanu

Ce este o excepție?

- O **excepție** este o eroare care poate să apară la rularea unui program.
- Exemple:
  - încercarea de deschidere a unui fișier ce nu există
  - depășirea limitelor unui tablou
  - încercarea de alocare a unui spațiu de memorie ce depășește dimensiunea heap-ului
  - etc.

Tratarea excepțiilor în C

- Afișarea unui mesaj de eroare și continuarea execuției programului
- Afișarea unui mesaj de eroare și terminarea programului

```
scanf("%d",&n);
if (n>MAX) {
  printf("Depasire valoare maxima");
  exit(1);
 }
```


Tratarea excepțiilor în C

- Testarea valorilor returnate de anumite funcţii, cu privire la posibilele erori ce au apărut pe parcursul execuției acestora
- Exemplu

```
if (scanf("%d %d",&n,&m) != 2) {
   printf("valori invalide");
   exit(1);
}

float *p= (float *) malloc(sizeof(float));
if (p==NULL){
     printf("eroare alocare memorie");
     exit(1);
}
```

Tratarea excepțiilor în C

- Utilizând macrodefiniția assert. Sintaxă

  assert(expresie);

  - Dacă valoarea *expresiei* este *false* atunci se oprește execuția programului și se afișează un mesaj de eroare care indică fișierul și (optional) linia unde este poziționată macrodefiniția.
  - Dacă valoarea *expresiei* este *adevarată* atunci execuția programului se continuă.
  - Avantaj: poate fi inactivată în momentul preprocesării adaugând macrodefiniţia:

    #define NDEBUG

Exemplu: assert

```
#include <assert.h>
#include <iostream>
#define MAX 100
using namespace std;
int main(){
    int t[MAX];
    int i,n;
    cout<<"Introduceti n:";
    cin>>n;
    assert(n<MAX);
    for(i=0;i<n;i++){
        cin>>t[i];
    }
}
```

```
Introduceti n:203
Assertion failed: n<MAX, file
TestAssert.cpp, line 11

This application has requested the
Runtime to terminate it in an unusual
way.
Please contact the application's support
team for more information.
```


Exemplu: Clasa Vector

```
class Vector {
  private:
    float t[MAX];
    int n;
  public:
    Vector(int n);
    int getNrElemente() const;
    void setElement(int  i, float val);
    float getElement(int i) const;
    void citire();
    void afisare();
};
Vector::Vector(int n){
    assert(n<=MAX);
    this->n = n;
    for (int i = 0; i < n; i++){
        this->t[i] = 0;
    }
}
```

```
void Vector::setElement(int i,float val){
    if (i >= 0 && i < n){
        t[i] = val;
    }  else {
        cerr<<"Index invalid: "<<i<<endl;
    }
}

float Vector::getElement(int i) const {
    if (i >= 0 && i < n){
        return t[i];
    } else {
        cerr<<"Index invalid: "<<i<<endl;
        exit(1);
    }
}
```


Exemplu: Clasa Vector (cont.)

```
int Vector::getNrElemente() const {
   return n;
}
void Vector::citire(){
   int i;
   cout<<"Dati elementele vectorului:";
   for (i = 0; i < n; i++){
        cin >> t[i];
   }
}
void Vector::afisare() {
    int i;
    for (i = 0; i < n; i++){
        cout<<t[i]<<" ";
    }
    cout<<endl;
}
```

```
void tratareTerminare (){
     cout<<"Terminare program!"<<endl;
}

int main(){
   int n;
   atexit(tratareTerminare);
   Vector v(2);
   v.setElement(0,4);
   v.setElement(3,9);

   v.afisare();
   getch();
   return 0;
}
```

Se apelează automat la terminarea programului.
Se pot trata eventualele erori

Dezavantajele tratărilor erorilor în C

- Secvența de apel al unei funcții ce returnează un cod e eroare trebuie sa fie însoțită de foarte multe **if**-uri pentru a trata posibilele erori
- Dezavantajul major: codul de tratare a erorilor devine dependent, **cuplat** de logica aplicației

Tratarea excepțiilor în C++

- C++ are un mecanism avansat de tratare a erorilor, codul ce tratează erorile fiind decuplat de logica aplicației:
  - Inițial scriem codul care dorim să se execute, apoi separat,
  - Se scrie codul ce tratează situațiile neprevazute
- O **excepție** în C++ este de regulă un obiect al unei clase ce este generat la apariția unei erori și cuprinde de regulă informații sumare despre cauza erorii.

Generarea şi prinderea excepţiilor

- Mecanismul de tratare a excepţiilor presupune:
  - Suspendarea execuţiei funcţiei curente în momentul în care s-a detectat o eroare, **generarea** unei excepţii care conţine informaţii referitoare la eroare și “*aruncarea*” (**throw**) ei către funcția apelantă;
  - Reluarea execuţiei programului în altă zonă, *prinderea* (**catch**) excepţiei şi executarea acţiunilor necesare tratării erorilor.

Generarea Excepțiilor: throw

- Aruncarea (generarea) unei excepții se face cu ajutorul intrucțiunii throw
- Sintaxa:

```
throw [<expresie>];
```

- Exemple:

```
throw out_of_range("Depasire limite");
throw 1;
```

Generarea Excepțiilor: throw (cont.)

- Mod de execuție:
  - se suspendă execuţia funcţiei curente;
  - se crează un obiect corespunzător valorii expresiei ce urmează cuvântului cheie throw;
  - obiectul creat este 'returnat' către contextul apelant, chiar dacă el nu corespunde tipului valorii de return a funcţiei.
  - înainte de părăsirea funcției toate obiectele create local sunt distruse (se apelează destructorii acestora)

Prinderea excepțiilor: try/catch

- Recepționarea (prinderea) unei excepții se face utilizând blocul de instrucțiuni try/catch
- Sintaxa:

```
try {
    // secvența de intrucțiuni ce poate genera excepții
} catch (TipExceptie1 ex1){
    //tratare excepție de tip TipExceptie1
} catch (TipExceptie2 ex2){
    //tratare excepție de tip TipExceptie2
  . . .
} catch (TipExceptieN exN){
    //tratare excepție de tip TipExceptieN
} catch (...) {
    //tratare orice alta excepție ce poate aparea
}
```

Handlere de tratare a erorilor

Prinderea excepțiilor: try/catch(cont.)

- Mod de execuție:
  - try marchează începutul unui bloc de instrucțiuni care pot genera (arunca) excepții
  - dacă la execuția unei intrucțiuni este aruncată o excepție atunci se abandonează execuția instrucțiunilor din bloc și se încearcă (prinderea) identificarea tipului obiectului aruncat cu unul dintre tipurile specificat în ramurile **catch:** TipExceptie1, TipExceptie2, etc.
  - Dacă se identifică o ramură catch pentru care tipul excepției coincide se va executa secvența de instrucțiuni pentru tratarea acelui tip de excepție
  - În cazul în care tipul excepției aruncate nu coincide cu nici unul din tipurile specificate în ramurile catch și există ramura catch(...) atunci se vor executa instrucțiunile din acel bloc

Exemplu

```
class E {
  public:
    const char* mesaj;
    E(const char* m) : mesaj(m) { }
};
void f(int n){
  switch (n){
    case 1:
      throw 0.5;
      cout<<"Intructiune ce nu se va executa";
    case 2:
    case 3:
      throw E("Eroare n este 2 sau 3");
      cout<<"Intructiune ce nu se va executa";
     case 4:
      throw n;
  }
  cout<<"Terminare funtie f()"<<endl;
}
```

```
int main (){
 int n;
 cin>>n;
 try{
     cout<<"Execut functia f("<<n<<")\n";
     f(n);
     cout<<"Executie cu succes"<<endl;
 } catch (double n){
   cerr<<"Tratare eroare de tip double: "<<n;
 } catch (E e){
   cerr<<"Tratare eroare de tip E: "<<e.mesaj;
 } catch (...){
    cerr<<"Tratare eroare necunoscuta";
 }
  getch();
}
```

Exemplu (cont.)

```
OUTPUT:

n=1
Execut functia f(1)
Tratare eroare de tip double: 0.5 n=1

n=3
Execut functia f(3)
Tratare eroare de tip E: Eroare n este 2 sau 3

n=4
Execut functia f(4)
Tratare eroare necunoscuta

n=5
Execut functia f(5)
Terminare funtie f()
Executie cu succes
```


Potrivirea Excepțiilor

- La aruncarea unei exceptii, se caută cel mai “apropriat” handler în raport cu ordinea în care aceștia apar în cod
- La găsirea unei potriviri, căutarea se opreşte
- Polimorfismul functionează
- E indicat sa prindem o exceptie prin referinţă, pentru a evita apelul constructorului de copiere

Specificarea excepțiilor generate

- Pentru a specifica faptul că o funcție generează numai un anumit set de excepții de anumite tipuri atunci utilizăm următoarea sintaxă:

```
tipr numeFunctie(lista_parametri) throw (TipE1,TipE2, ..., TipEn){
}
```

- Dacă dorim să specificăm că o funcție nu generează nicio excepție atunci folosim următoarea sintaxă:

```
tipr numeFunctie(lista_parametri) throw (){
}
```


Exemplu

```
void f(int n) throw (double, E, int){
  switch (n){
    case 1:
      throw 0.5;
      cout<<"Intructiune ce nu se va executa";
    case 2:
    case 3:
      throw E("Eroare n este 2 sau 3");
      cout<<"Intructiune ce nu se va executa";
     case 4:
      throw n;
  }
}
```

Excepții netratate

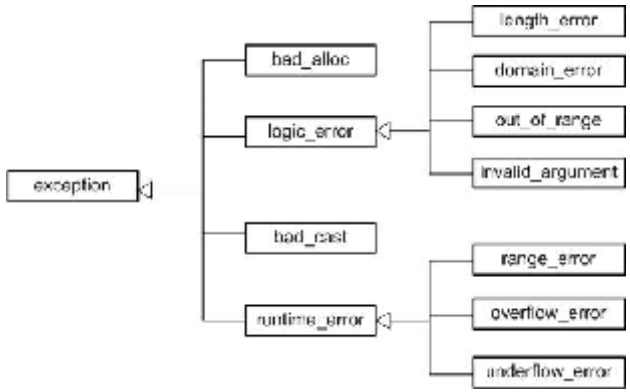
- Dacă un bloc try/catch nu tratează (prinde) un anumit tip de excepție atunci excepția respectivă este aruncată către contextul superior.
- Dacă o excepție nu e tratată de nici un handler, se va apela funcţia terminate(), care apelează la rândul său abort() – ieşirea din program

Re-aruncarea unei excepții

- Dacă se dorește re-aruncarea unei exceptii prinse către contextul superior atunci se folosește clauza throw fără nici un argument.
- De obicei, atunci când nu suntem interesaţi de cauza exceptiei, ci doar dorim eliberarea resurselor, folosim clauza catch(...) pentru a prinde excepția, eliberăm resursele şi apoi o aruncăm mai departe

```
try{
  //instructiuni ce pot genera excepții
} catch(...){
 //eliberare resurse
 throw;
}
```


Tipuri de excepții predefinite



Tipuri de excepții predefinite (cont)

- Din clasa exception sunt derivate două clase, logic_error, pentru raportarea erorilor detectabile înainte de execuţia programului şi runtime_error, pentru raportarea erorilor din timpul execuţiei programului.
- Principalele clase derivate din logic_error sunt următoarele:
  - domain_error – raportează violarea unei precondiţii
  - invalid_argument – argument invalid pentru o funcţie
  - length_error – un obiect cu o lungime mai mare decât NPOS (valoarea maximă reprezentabilă)
  - out_of_range – în afara domeniului
  - bad_cast – operaţie dynamic_cast eronată
- Principalele clase derivate din runtime_error sunt următoarele:
  - range_error – violarea unei postcondiţii
  - overflow_error – operaţie aritmetică eronată (depăşire)
  - bad_alloc – eroare de alocare

Exemplu

```
class Vector {
  private:
    float t[MAX];
    int n;
  public:
    Vector(int n);
    int getNrElemente() const;
    void setElement(int  i, float val);
    float getElement(int i) const;
    void citire();
    void afisare();
};

 Vector::Vector(int n){
    this->n = n;
    if(n>MAX)
      throw out_of_range("Depasire
dimensiune maxima");
    for (int i = 0; i < n; i++){
        this->t[i] = 0;
    }
 }

void Vector::setElement(int  i, float val)
{
    if (i >= 0 && i < n){
       t[i] = val;
    } else {
        throw out_of_range("Depasire
limite");
    }
}
float Vector::getElement(int i) const {
  if (i >= 0 && i < n)
    return t[i];
  else
    throw out_of_range("Depasire limite");
}
```


Exemplu (cont.)

```
int main(){
   int n;
   Vector v(2);
   v.setElement(0,4);
   try{
        v.setElement(3,9);
   } catch (exception &e){
      cerr<<"Eroare:"<<e.what()<<endl;
   }
   v.afisare();
   getch();
   return 0;
}

OUTPUT:
Eroare:Depasire limite
4 0
```


Temă

- Implementați clasa Cantar. Un cântar are o capacitate maximă admisă. Dacă se încearcă cântărirea unui obiect ce depășește cu maxim 10% greutatea maximă admisă se va genera excepția AvertismentDepasireGreutate iar dacă se depășește și această limită se va genera eroarea DepasireGreutate.
- Implementați clasa Stiva implementată sub forma unui tablou alocat dinamic. Gestionați excepțiile ce pot apărea.