Tratarea Excepțiilor

Ce este o excepție?

- O **excepție** este o eroare sau un eveniment neprevăzut ce poate să apară la rularea unui program.
- Exemple:
  - încercarea de a deschide un fişier ce nu există;
  - depăşirea limitelor unui tablou;
  - încercarea de alocare a unui spațiu de memorie ce depăşeşte dimensiunea heap-ului;
  - etc.

Bune practici

- Ce trebuie să avem în vedere într-un program:
  - detectarea unei erori / a tuturor erorilor;
  - transmiterea unor informații ce identifică eroarea către un cod ce se ocupă cu gestionarea unei astfel de situații (*exception handler*);
  - păstrarea stării interne a programului într-o stare validă;
  - evitarea risipei de resurse (de exemplu, *memory leaks*).

Tratarea excepțiilor în C

- Variante de abordare în momentul apariției unei situații de excepție:
  - v1) Afişarea erorilor identificate şi continuarea execuției programului.
  - v2) Afişarea erorilor identificate şi terminarea programului:

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

Tratarea excepțiilor în C

- Pentru identificarea unei situații de excepție trebuie verificate valorile returnate de anumite funcții, 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

- v3) Utilizând macrodefiniția **`assert`**. Sintaxă:

```
#include <assert.h>
assert(expresie);
```

  - Dacă valoarea *expresiei* este falsă atunci se opreşte execuția programului şi se afişează un mesaj de eroare ce indică fişierul şi linia unde este poziționată macrodefiniția.
  - Dacă valoarea *expresiei* este adevărată atunci execuția programului se continuă.
  - Avantaj: poate fi dezactivată (temporar) în momentul preprocesării prin adăugarea declarației:

```
#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];
  }
  return 0;
}
```

```
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 ((0 <= i) && (i < n)) {
    t[i] = val;
  } else {
     cerr << "Index invalid: " << i <<endl;
  }
}

float Vector::getElement(int i) const {
  if ((0 > i) || (i >= n)) {
    cerr << "Index invalid: " << i <<endl;
    exit(1);
  }

  return t[i];
}
```

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();

  return 0;
}
```

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

Dezavantajele tratării erorilor în C

- Secvența de apel al unei funcții ce returnează un cod de eroare trebuie să 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 ce dorim să fie executat,

  apoi, separat,
  - se scrie codul ce tratează situațiile neprevăzute.
- O **excepție** în C++ este un obiect al unei clase ce este *generat / instanțiat* la apariția unei erori şi cuprinde de regulă informații sumare despre cauza erorii.

Generarea şi prinderea excepțiilor

- Mecanismul de tratare a excepțiilor presupune:
  - **Suspendarea** execuției funcției curente în momentul în care s-a detectat o eroare, **generarea** unei excepții ce conține informații referitoare la eroare şi "**aruncarea**" (`throw`) ei către funcția apelantă.
  - **Reluarea** execuției programului în altă zonă a acestuia, *prinderea* (`catch`) excepției şi executarea acțiunilor necesare tratării erorilor.

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 ((0 <= i) && (i < n)){
    t[i] = val;
  } else {
      throw out_of_range("Depasire limite");
  }
}

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

Exemplu (cont.)

```
int main() {
  int n;
  Vector v(2);

  try {
    v.setElement(0, 4);
    v.setElement(3, 9);
  } catch (exception &e) {
      cerr << "Eroare:" << e.what() << endl;
  }

  v.afisare();

  return 0;
}

OUTPUT:
Eroare:Depasire limite
4 0
```

Generarea excepțiilor: `throw`

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

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

- Exemplu:

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

Generarea excepțiilor: `throw`

- Mod de execuție:
  - *se suspendă* execuția funcției curente;
  - *se crează* un obiect corespunzător valorii expresiei ce urmează cuvântului `throw`;
  - obiectul creat este *returnat* către contextul apelant, chiar dacă el nu corespunde tipului valorii de return a funcției;
  - 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 {
  // secventa de instructiuni ce poate genera exceptii
} catch (TipExceptie1 ex1) {
  //tratare exceptie de tip TipExceptie1
} catch (TipExceptie2 ex2) {
  //tratare exceptie de tip TipExceptie2
 …
} catch (TipExceptieN exN) {
  //tratare exceptie de tip TipExceptieN
} catch (…) {
  //tratare orice alta exceptie ce poate sa apara
}
```

Handlere de tratare a erorilor

Prinderea excepțiilor: `try/catch`

- Mod de execuție:
  - **`try`** marchează începutul unui bloc de instrucțiuni ce pot genera (arunca) excepții.
  - Dacă la execuția unei instrucțiuni este aruncată o excepție, atunci se abandonează execuția instrucțiunilor din bloc şi 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`** şi 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 <<"Instructiune ce nu se va executa";
    case 2:
    case 3:
      throw E("Eroare - n este 2 sau 3");
      cout <<"Instructiune ce nu se va executa";
    case 4:
      throw n;
  }
  cout << "Terminare functie f()" << endl;
}
```

```
int main () {
  int n;

  cin >> n;
  try {
    cout << "Executam functia f(" << n << ")\n";
    f(n);
    cout<<"Executie terminata 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";
  }

  return 0;
}
```

Exemplu (cont.)

```
OUTPUT:

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

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

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

n=5
Executam functia f(5)
Terminare functie f()
Executie terminata cu succes
```

Potrivirea excepțiilor

- La aruncarea unei exceptii, se caută cel mai “apropiat” handler în raport cu ordinea în care aceştia apar în cod.
- La găsirea unei potriviri, căutarea se opreşte.
- Polimorfismul funcționează şi în acest caz.
- E indicat să prindem o excepție prin referință sau să folosim pointeri 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, utilizăm sintaxa următoare:

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

- Dacă dorim să specificăm că o funcție *nu generează nici o 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 << "Instructiune ce nu se va executa";
    case 2:
    case 3:
       throw E("Eroare n este 2 sau 3");
       cout << "Instructiune ce nu se va executa";
    case 4:
       throw n;
  }
}
```

Excepții netratate

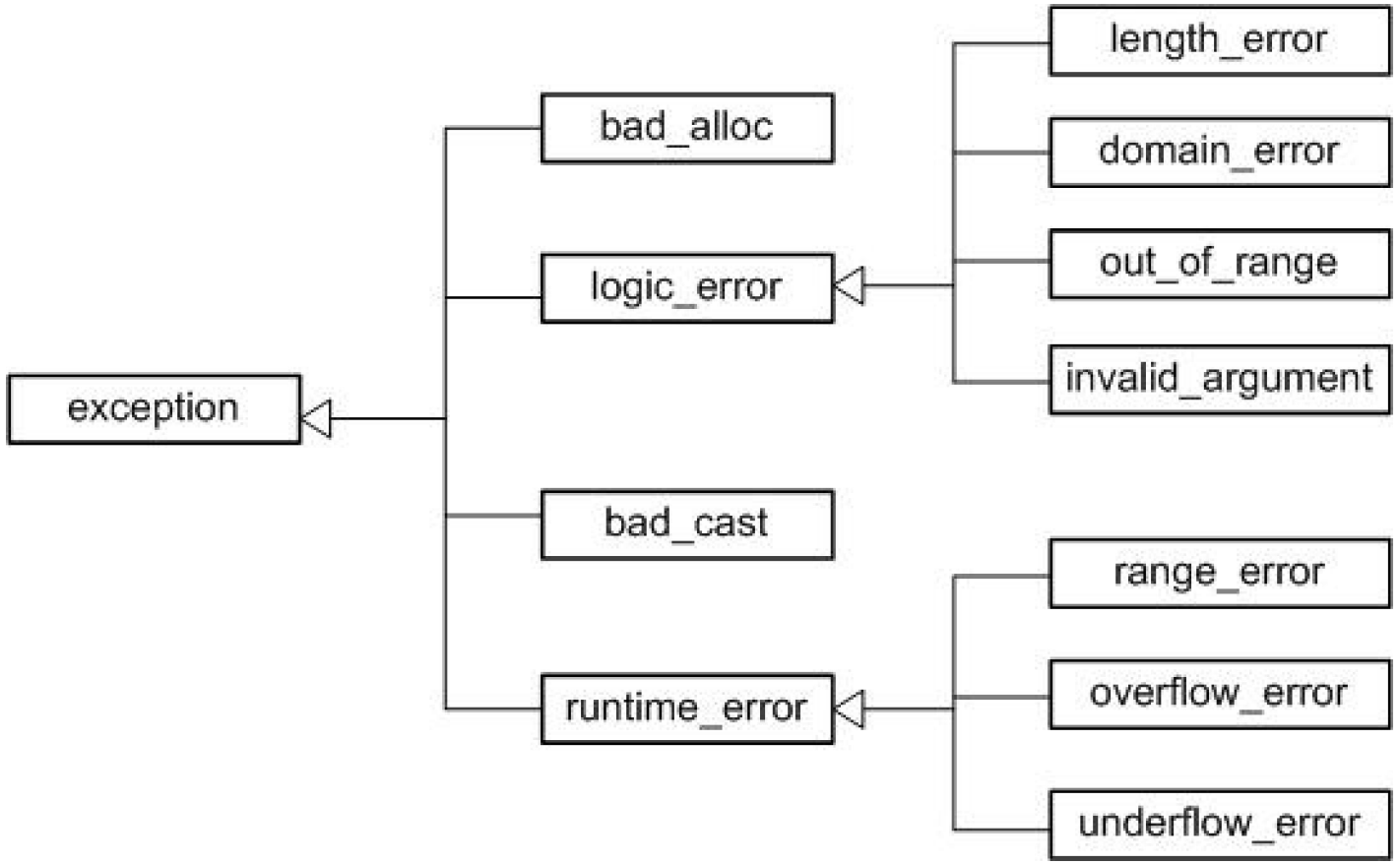
- Dacă un bloc `try / catch` *nu tratează* (*nu prinde*) un anumit tip de excepție, atunci excepția respectivă *este aruncată* către contextul superior.
- Dacă o excepție *nu e tratată* de nici un handler, se va apela funcția **`terminate()`**, ce apelează la rândul său **`abort()`** – ieşirea din program.

Rearuncarea unei excepții

- Dacă se doreşte *rearuncarea* unei excepții prinse către contextul superior atunci se foloseşte clauza **`throw`** fără nici un argument.
- De obicei, atunci când nu suntem interesați de cauza excepției, ci doar dorim eliberarea resurseler, folosim clauza **`catch(…)`** pentru a prinde excepția, eliberăm resursele şi apoi o aruncăm mai departe:

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

Tipuri de excepții predefinite



Tipuri de excepții predefinite

- Din clasa `exception` sunt derivate două clase, `logic_error`, pentru raportarea erorilor în logica internă a programului, ce pot fi prevenite înainte de execuția programului şi `runtime_error`, pentru raportarea erorilor ce pot fi detectate numai în timpul execuției programului.
- Principalele clase derivate din `logic_error` sunt următoarele:
  - `domain_error` – raportează violarea unei precondiții;
  - `invalid_argument` – argument invalid pentru o funcție;
  - `length_error` – un obiect cu o lungime mai mare decât NPOS (valoarea maximă reprezentabilă);
  - `out_of_range` – în afara domeniului;
- `bad_cast` – operație *dynamic_cast* eronată;
- Principalele clase derivate din `runtime_error` sunt următoarele:
  - `range_error` – violarea unei postcondiții;
  - `overflow_error` – operație aritmetică eronată (depăşire);
- `bad_alloc` – eroare de alocare.

Temă

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

Bibliografie

- https://www.linkedin.com/pulse/ariane-5-rocket-launch-failure-do-one-line-computer-code-dahlberg/
- https://softwareengineering.stackexchange.com/questions/149888/what-was-the-historical-impact-of-ariane-5s-flight-501
- http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html
- https://www.modernescpp.com/index.php/c-core-guidelines-rules-to-error-handling
- https://www.bogotobogo.com/cplusplus/exceptions.php