# 12. Template-uri de funcții. Tratarea excepțiilor

Template-urile (şabloanele) reprezintă una dintre cele mai puternice caracteristici ale limbajului C++. Acestea permit definirea, printr-un singur segment de cod, a unei întregi game de funcții supraîncărcate – template de funcție sau a unei întregi serii de clase – template de clasă. În acest capitol vom discuta cazul funcțiilor. Putem scrie generic, de exemplu, un singur template de funcție pentru ordonarea unui şir de elemente, iar funcția poate fi apelată pentru valori de diverse tipuri: `int`, `double` etc.

Tratarea excepțiilor este o metodă care permite programatorilor să scrie programe mai robuste, mai clare şi cu un grad mai mare toleranță la erori. Programele pot fi concepute astfel încât să detecteze şi să trateze erorile (excepțiile) care pot apărea în timpul rulării, evitând astfel întreruperile neplanificate ale aplicațiilor.

## 12.1 Template-uri de funcții

Funcțiile supraîncărcate realizează, de obicei, operații *similare* pentru diferite tipuri de date. Dacă operațiile sunt *identice* pentru toate tipurile de date, acestea pot fi realizate mai compact folosind *template-uri de funcții*. Programatorul trebuie să scrie doar o singură definiție de template de funcție. Bazându-se pe tipurile argumentelor date explicit sau rezultate din apeluri ale acestor funcții, compilatorul generează funcții separate în codul obiect pentru a manipula corespunzător fiecare tip de apel. În limbajul C, acest lucru poate fi realizat folosind macrouri create prin directiva de preprocesare `#define`. Este, însă, o metodă care poate produce efecte nedorite şi care nu permite compilatorului să facă verificări de tip.

Toate definițiile de template-urile de funcții încep prin cuvântul cheie `template` urmat de lista parametrilor formali de tip ai template-ului de funcție plasată între < >. Fiecare parametru formal de tip trebuie să fie precedat de cuvântul cheie `class` sau de `typename`.

Parametrii formali de tip din definiția unui template de funcție sunt utilizați pentru a specifica tipurile argumentelor funcției, tipul valorii returnate de funcție şi pentru a declara variabile în funcție. Definiția funcției este, astfel, cea a unei funcții obişnuite. Cuvintele cheie `class` sau `typename` folosite pentru a specifica parametrii de tip ai template-ului au semnificația de „orice tip de dată predefinit sau definit prin program”.

Programul de mai jos defineşte funcția `PrintArray` care tipăreşte componentele unor tablouri.

```
#include <iostream>
```

```
<iostream>
using std::cout;
using std::endl;

template< class T >
void PrintArray(const T *array, const int count)
{
  for(int i = 0; i < count; i++)
    cout << array[i] << " ";
  cout << endl;
}

int main()
{
  const int aCount = 5, bCount = 7, cCount = 6;
  int a[aCount] = {1, 2, 3, 4, 5};
  double b[bCount] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
  char c[cCount] = "HELLO";

  cout << "Tabloul a contine: " << endl;
  PrintArray(a, aCount);

  cout << "Tabloul b contine: " << endl;
  PrintArray(b, bCount);

  cout << "Tabloul c contine: " << endl;
  PrintArray(c, cCount);

  return 0;
}
```

Acest program afişează:

```
Tabloul a contine:
1 2 3 4 5
Tabloul b contine:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Tabloul c contine:
H E L L O
```

Funcția template PrintArray declară un singur parametru formal de tip cu numele `T`. Identificatorul `T` poate fi înlocuit cu orice alt identificator valid. Prin `T` se defineşte generic tipul tabloului ale cărui componente urmează să fie tipărite. Atunci când compilatorul detectează în program un apel al funcției `PrintArray`, tipul argumentului generic este înlocuit cu tipul parametrului actual şi se creează o funcție pentru tipărirea tabloului cu acest tip. Noua funcție este apoi compilată.

În exemplul de mai sus, sunt instanțiate trei funcții `PrintArray`, cu argumente de tip `int`, `double` şi `char`.

Exemplu

Instanțierea pentru tipul de dată `int` este:

```
void PrintArray(const int *array, const int count)
{
  for(int i = 0; i < count; i++)
```

```
        cout << array[i] << " ";
      cout << endl;
    }
```

Fiecare parametru formal de tip din definiția unui template de funcție apare, în mod normal, cel puțin o dată în lista de parametri ai funcției cel puțin o dată. Numele unui parametru formal de tip poate fi folosit o singură dată în lista de parametri ai header-ului unui template.

În programul de mai sus, funcția `PrintArray` este apelată de trei ori cu argumentele `a` de tip `int*`, `b` de tip `double*` şi `c` de tip `char*` pentru primul parametru. Apelul

```
PrintArray(a, aCount);
```

face ca aparițiile lui `T` în funcția `PrintArray` să fie transformate în `int` şi compilatorul instanțiază funcția `PrintArray` pentru `T` de tip `int`. În mod asemănător se întâmplă şi pentru apelurile

```
PrintArray(b, bCount);
PrintArray(c, cCount);
```

În acest program, mecanismul template-urilor face ca programatorul să nu mai fie nevoit să scrie trei funcții supraîncărcate cu prototipurile

```
void PrintArray(const int *, const int);
void PrintArray(const double *, const int);
void PrintArray(const char *, const int);
```

## 12.2 Tratarea excepțiilor

Există mai multe tehnici prin care pot fi tratate erorile care apar în timpul rulării unui program. Cea mai folosită este aceea de a introduce în program secvențe de cod adaptate prevenirii unor posibile situații nedorite. Erorile sunt tratate în locul în care apar în program. Avantajul acestei abordări este că persoana care citeşte codul poate vedea modalitatea de prelucrare a erorii în vecinătatea codului şi poate determina dacă metoda de tratare a excepției este implementată corect. Dezavantajul este că prin această schemă codul este oarecum „poluat” cu secvențe de procesare a erorilor şi devine mult mai greu de citit de un programator care este concentrat pe aplicația însăşi. Această metodă face codul mult mai greu de înțeles şi de menținut.

Tratarea excepțiilor în C++ permite programatorilor să înlăture partea de tratare a excepțiilor din secvența principală de program. Stilul C++ de tratare a excepțiilor permite interceptarea tuturor excepțiilor sau a tuturor excepțiilor de un anumit tip. Aceasta face programul mult mai robust, reducând probabilitatea de întrerupere neplanificată a programului.

Tratarea excepțiilor se foloseşte în situația în care programul poate să îşi continue rularea după ce depăşeşte eroarea care provoacă excepția.

### Tratarea excepțiilor folosind `try`, `throw` şi `catch`

Tratarea excepțiilor în C++ este o metodă care se aplică atunci când funcția care detectează o eroare nu o şi tratează. Ea doar *generează* sau *aruncă* excepția (`throw`). Nu se garantează că excepția va fi şi tratată în afara funcției. Pentru aceasta, trebuie specificată o secvență de cod care *detectează* sau *prinde* excepția şi o *tratează*.

Programatorul trebuie să includă într-un *bloc* `try` codul care ar putea genera o eroare generatoare a unei excepții. Blocul `try` este urmat de unul sau mai multe

*blocuri* `catch`. Fiecare bloc `catch` specifică tipul excepției pe care o poate detecta şi trata. Dacă excepția se potriveşte cu tipul parametrului unuia dintre blocurile `catch`, se execută codul acelui `catch`. Dacă nu este identificat niciun bloc catch care să trateze eroarea, se apelează funcția predefinită `terminate` care la rândul ei apelează în mod implicit funcția predefinită `abort` pentru întreruperea programului.

Dacă într-un bloc `try` nu se generează nicio excepție, programul rulează ignorând blocurile `catch` asociate lui.

Exemplu

```
#include <iostream>
#include <stdexcept>
using std::cin;
using std::cout;
using std::endl;
using std::runtime_error;

double Fractie(int numarator, int numitor)
{
  if(numitor == 0)
    throw runtime_error("Numitorul este 0");
  return static_cast<double>(numarator)/numitor;
}

int main()
{
  int numar1, numar2;
  double rezultat;
  cout << "Introduceti doi intregi (EOF pentru a termina): ";

  while(cin >> numar1 >> numar2)
  {
    try
    {
      rezultat = Fractie(numar1, numar2);
      cout << "Valoarea fractiei este: " << rezultat << endl;
    }
    catch(runtime_error e)
    {
      cout << "A aparut urmatoarea exceptie: "
           << e.what() << endl;
    }

    cout << "Introduceti doi intregi (EOF pentru a termina): ";
  }
  cout << endl;
  return 0;
}
```

Un exemplu de rulare a acestui program este următorul:

```
Introduceti doi intregi (EOF pentru a termina): 3 4
Valoarea fractiei este: 0.75
Introduceti doi intregi (EOF pentru a termina): 5 0
```

```
A aparut urmatoarea exceptie: Numitorul este 0
Introduceti doi intregi (EOF pentru a termina): CTRL-D
```

Programul de mai sus calculează rezultatul împărțirii a două numere întregi. Dacă numitorul este 0, este declanşată o excepție şi se semnalează eroarea.

Programul conține un bloc `try` care cuprinde codul care ar putea genera o excepție. Operația de împărțire a celor două numere este implementată prin funcția `Fractie` care generează o excepție prin instrucțiunea `throw` atunci când numitorul este 0.

Clasa `runtime_error` face parte din biblioteca standard şi este declarată în fişierul header `stdexcept`. Este derivată din clasa de bază `exception` care face parte tot din biblioteca standard, fiind declarată în fişierul header `exception` şi care este folosită pentru a declanşa excepții în C++. Declanşarea excepțiilor cu ajutorul lui `runtime_error` permite adăugarea unui mesaj care poate fi citit prin metoda `what()`. Limbajul C++ oferă o întreagă gamă de clase derivate din `exception`, destinate diverselor tipuri de excepții care se pot declanşa într-un program. La rândul său, programatorul poate să îşi definească propriile sale clase pentru declanşarea excepțiilor, derivându-le, spre exemplu, din `exception` sau din clasele derivate din ea.

În exemplul de mai sus, blocul `try` este urmat imediat de un `catch` care conține codul de tratare a excepției de tip `runtime_error`. Atunci când într-un bloc `try` este generată o excepție, ea va fi tratată de blocul `catch` care este destinat acelui tip particular de excepție. Aşadar, blocul `catch` din exemplu va trata doar excepțiile de tip `runtime_error`.

Dacă în timpul execuției codul din blocul `try` nu generează nicio excepție, toate blocurile `catch` asociate acestui `try` vor fi ignorate, iar execuția cotinuă cu prima instrucțiune care urmează după acestea.

## Generarea unei excepții – `throw`

Cuvântul cheie `throw` se foloseşte pentru a indica faptul că a apărut o excepție. Un `throw` specifică, în mod obişnuit, un operand. Operandul lui `throw` poate fi de orice tip. Dacă este un obiect, îl vom numi *obiect excepție*. Putem folosi, însă, şi obiecte care nu sunt destinate tratării excepțiilor, şi chiar şi expresii de orice tip.

Imediat ce a fost declanțată o excepție, aceasta va fi detectată de cel mai apropiată secvență de cod destinată tratării excepției respective. Secvențele de tratare a excepțiilor generate într-un bloc `try` sunt listate imediat după blocul `try`.

Ca regulă generală pe care este bine să o urmăm atunci când scriem programe, excepțiile trebuie declanşate doar în interiorul blocurilor `try`. O excepție care apare în afara unui `try` poate conduce la terminarea programului.

## Tratarea unei excepții – `catch`

Tratarea excepțiilor se face prin blocuri `catch`. Fiecare astfel de bloc constă din cuvântul cheie `catch` urmat, între paranteze rotunde, de tipul excepției şi, opțional, de un nume de parametru. Între acolade se găseşte codul care tratează excepția. Atunci când este detectată o excepție, se execută codul dintre acolade.

Blocul `catch` defineşte un domeniu de accesibilitate propriu. Dacă parametrul are nume, atunci el poate fi invocat în blocul `catch`. Dacă nu are nume, el este

specificat numai în scopul potrivirii dintre blocul `catch` şi tipul excepției căreia îi este destinat.

O excepție care nu este detectată apelează în mod implicit funcția `terminate()` din biblioteca standard care, la rândul ei, termină programul prin apelul funcției `abort()`. Programatorul poate schimba comportamentul funcției `terminate()` facând ca aceasta să apeleze o altă funcție. Numele noii funcții va fi trimis ca parametru funcției `set_terminate`.

Un `catch` care este urmat între parantezele rotunde de trei puncte tratează toate excepțiile.

Exemplu

```
catch(...)
```

Această opțiune este plasată, de obicei, ca o ultimă variantă într-o serie de blocuri `catch`. Un bloc `try` urmat de mai multe blocuri `catch` are un comportament similar instrucțiunii `switch` care foloseşte o ramură `case` în funcție de valoarea expresiei testate.

## Generarea unei noi excepții în `catch`

Uneori, tratarea unei excepții nu poate fi făcută în blocul catch care a detectat-o şi atunci se poate decide ca excepția să fie transmisă mai departe, lasând tratarea ei pe seama altei secvențe de cod (*rethrowing an exception*). Instrucțiunea

```
throw;
```

lansează din nou excepția.

Exemplu

```
#include <iostream>
using std::cout;
using std::endl;

#include <exception>
using std::exception;

void ThrowException()
{
  //Lanseaza o exceptie si o prinde imediat
  try
  {
    cout << "Functia ThrowException" << endl;
    throw exception(); //Genereaza exceptia
  }
  catch(exception e)
  {
    cout << "Exceptia tratata in ThrowException" << endl;
    throw;
  }
  cout << "Acest text nu este tiparit";
}

int main()
{
  try
  {
```

```
    ThrowException();
    cout << "Acest text nu este tiparit";
  }
  catch(exception e)
  {
    cout << "Exceptia tratata in main" << endl;
  }
  cout << "Programul continua dupa catch in main" << endl;
  return 0;
}
```

Acest program afişează:

```
Functia ThrowException
Exceptia tratata in ThrowException
Exceptia tratata in main
Programul continua dupa catch in main
```

Funcția `ThrowException` este apelată în blocul `try` din `main`. În blocul `try` din funcția `ThrowException` este generată o excepție de tip `exception` care este detectată imediat de blocul `catch` asociat acestui `try`. Aici, excepția este relansată, fiind tratată în blocul `catch` din `main`.

## Specificarea excepțiilor

Lista excepțiilor care pot fi generate de o funcție se poate specifica astfel:

```
int g(double h) throw(a, b, c)
{
  //corpul functiei
}
```

Prin această listă se restrânge gama excepțiilor care pot fi declanşate de funcția `g` la tipurile `a`, `b`, `c`. Dacă funcția generează o altă excepție decât cele listate, se apelează automat funcția `unexpected` din biblioteca standard care, la rândul ei apelează funcția `terminate`. Comportmenul funcției unexpected poate fi modificat asemănător modului în care se intervine asupra funcției `terminate`, dar apelând, de data aceasta, funcția `set_unexpected`.

O funcție pentru care nu există o astfel de listă poate genera orice excepție.