Excepții în C++

Anul Universitar 2023 – 2024, Semestrul II

Programare Orientată pe Obiecte

LABORATOR 8, GRUPA 133

Cuprins

Autor: Wagner Ștefan Daniel

I. Motivarea excepțiilor în C++

1.1 Introducere

Excepțiile reprezintă un mecanism în C++ pentru gestionarea erorilor și a situațiilor neașteptate la runtime fără să se întrerupă execuția programului dacă sunt prinse (cu ***catch***) și tratate corespunzător. Prin contrast cu metodele tradiționale bazate pe coduri de eroare, excepțiile oferă o cale mai clară și mai directă de a reacționa la problemele neașteptate care pot apărea în timpul execuției unui program.

1.2 De ce excepții?

Excepțiile permit separarea codului care execută operațiuni de codul care gestionează erorile, facilitând astfel scrierea de programe mai curate și mai ușor de întreținut. Utilizarea excepțiilor asigură că erorile pot fi "aruncate" și "prinse" la diferite niveluri în stiva de apeluri, permițând astfel prinderea centralizată și specifică a diferitelor tipuri de erori, la nivelul corespunzător în stiva de apeluri pentru excepția curentă.

1.3 Avantajele folosirii excepțiilor

- **Claritate și separare**: Codul devine mai ușor de citit și de înțeles, fiind o separare clară între business logic și gestionarea erorilor.
- **Siguranța resurselor**: Excepțiile asigură că resursele sunt eliberate corespunzător, chiar și atunci când apare o eroare, prin utilizarea destructorilor și a blocurilor **try-catch**.
- **Flexibilitate în propagarea erorilor**: Erorile pot fi propagate eficient către nivelurile superioare ale aplicației, unde pot fi gestionate în mod corespunzător, deoarece este posibil ca o clasă/funcție care aruncă o excepție să nu o poată trata imediat la urmatorul nivel în stiva de apeluri, insă în ierarhia de apeluri de funcții una dintre funcții să stie să o trateze.

1.4 Exemplu introductiv excepții (ex01.cpp)

```
#include <iostream>
#include <stdexcept>

int divide(int numarator, int numitor)
{
    if (numitor == 0)
        throw std::runtime_error("Divizie prin zero detectata - nu este
posibila.");

    return numarator / numitor;
}
```

```
int main()
{
    try {
        std::cout << divide(10, 0) << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "A fost prinsa o exceptie: " << e.what() << std::endl;
    }
    return 0;
}
```

În exemplul de mai sus, încercăm să efectuăm o împărțire, care poate genera o excepție în cazul în care numitorul este zero. Utilizând mecanismul de excepții, putem "arunca" o eroare atunci când detectăm condiția problematică și "prinde" eroarea într-un mod controlat, fără a afecta restul programului.

II. *setjmp()* și *longjmp()* vs excepții

2.1 Fundamentele *setjmp()* și *longjmp()* în C

Înainte de apariția mecanismului de excepții în C++, programatorii se bazau pe funcțiile ***setjmp()*** și ***longjmp()*** pentru gestionarea fluxurilor de control neobișnuite în programele C, cum ar fi saltul din interiorul unei funcții în interiorul altei funcții și inapoi, spre deosebire de ***goto***, care permite saltul doar în cadrul funcției curente. Drept urmare, mecanismul descris funcționează fără a respecta ordinea obișnuită a apelurilor de funcții sau bucle.

Exemplu *setjmp()* și *longjmp()* (ex02.cpp)

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf jumpBuffer;

void SecondFunction() {
    printf("Intrare in a doua functie\n");
    longjmp(jumpBuffer, 1);
    printf("Aceasta linie nu va fi executata.\n");
}

void FirstFunction() {
    if (!setjmp(jumpBuffer))
        SecondFunction();
    else
        printf("Retur de la longjmp\n");
}
```

```
int main()
{
    FirstFunction();
    return 0;
}
```

2.2 Limitările *setjmp()* și *longjmp()*

Deși ***setjmp()*** și ***longjmp()*** oferă un nivel de control al fluxului, ele vin cu semnificative limitări și potențiale probleme, în special în ceea ce privește siguranța și predictibilitatea programului:

- Nu oferă garanții pentru eliberarea corectă a resurselor, deoarece pot sări peste destructorii obiectelor.
- Pot duce la cod dificil de urmărit și de întreținut datorită salturilor neașteptate în execuție.

2.3 Excepțiile ca soluție alternativă

Mecanismul de excepții din C++ a fost introdus pentru a aborda aceste limitări, oferind o metodă mai sigură și mai structurată pentru gestionarea erorilor și a situațiilor neprevăzute. Excepțiile permit separarea clară a codului pentru operațiuni normale de codul de gestionare a erorilor, îmbunătățind lizibilitatea și mentenanța.

- **Eliberarea resurselor**: Excepțiile asigură că destructorii sunt apelați corespunzător, ceea ce permite gestionarea adecvată a resurselor chiar și în cazul erorilor. Altfel spus, programul nu va avea memory leaks, deoarece resursele au fost eliberate corespunzator de destructor.
- **Gestionare centralizată a erorilor**: Prin utilizarea blocurilor ***try*** și ***catch***, excepțiile pot fi gestionate în puncte specifice ale programului, reducând necesitatea de a verifica erorile după fiecare apel de funcție.

2.4 De reținut – *setjmp()*/*longjmp()* vs excepții

În timp ce ***setjmp()*** și ***longjmp()*** pot încă fi folosite în programe C pentru controlul fluxului, excepțiile din C++ oferă o alternativă superioară pentru gestionarea erorilor, datorită siguranței, structurii și clarității pe care le aduc în proiectele software.

III. Structura *try-catch*

3.1 Fundamentele *try-catch*

Mecanismul de gestionare a excepțiilor în C++ se bazează pe blocuri ***try*** și ***catch***. Un bloc ***try*** conține codul ce este susceptibil de a genera excepții, în timp ce blocurile ***catch*** sunt utilizate pentru a prinde și gestiona aceste excepții. Această structură permite separarea clară între logica principală a programului și gestionarea erorilor, facilitând scrierea de cod mai curat și mai ușor de înțeles.

3.2 Utilizarea *try-catch*

Exemplu *try-catch* (ex03.cpp)

```cpp
#include <iostream>
#include <stdexcept>

void PoateGeneraExceptie(bool genereaza) {
    if (genereaza) {
        throw std::runtime_error("Exceptie generata intentionat.");
    }
    std::cout << "Executie fara exceptie." << std::endl;
}

int main()
{
    try {
        PoateGeneraExceptie(true);
    } catch (const std::runtime_error& e) {
        std::cerr << "A fost prinsa o exceptie: " << e.what() << std::endl;
    }
    return 0;
}
```

În exemplul de mai sus, funcția **poateGeneraExceptie** poate arunca o excepție de tip **std::runtime_error**. Blocul **try** înconjoară apelul către această funcție, și dacă o excepție este aruncată, execuția sare la blocul **catch** cel mai apropiat care poate gestiona tipul de excepție aruncat. Acest model permite un tratament elegant și eficient al erorilor.

3.3 Capturarea mai multor tipuri de excepții

C++ permite definirea mai multor blocuri **catch** pentru a trata diferite tipuri de excepții în mod specific. Aceasta este o practică utilă pentru a oferi răspunsuri adecvate la diferite situații de eroare.

```
void FunctieCuExceptii(void)
{
    try {
        // Cod ce poate arunca diferite tipuri de exceptii
    } catch (const std::invalid_argument& e) {
        // Tratarea unei exceptii specifice
    } catch (const std::exception& e) {
        // Tratarea oricarei alte exceptii derivate din std::exception
    }
}
```

3.4 Excepții neparinsate

Dacă o excepție aruncată dintr-un bloc ***try*** nu este prinsă de niciun bloc ***catch*** corespunzător, programul va apela funcția ***std::terminate()***, rezultând în terminarea programului. Acest comportament subliniază importanța acoperirii tuturor posibilelor excepții ce pot fi aruncate de codul din blocul ***try***.

Blocurile ***try-catch*** sunt esențiale în C++ pentru o gestionare robustă a erorilor. Prin utilizarea lor, programatorii pot asigura că aplicațiile pot reacționa adecvat la erori și pot continua să funcționeze, sau pot termina "grațios", chiar și în prezența condițiilor de eroare. Această structură este fundamentală pentru scrierea de software fiabil și ușor de întreținut.

IV. Aruncarea excepțiilor cu *throw*

4.1 Conceptul de *throw*

Instrucțiunea ***throw*** în C++ este folosită pentru a arunca o excepție, semnalând astfel apariția unei condiții excepționale sau a unei erori. Mecanismul ***throw*** permite transmiterea erorilor către un nivel superior în stiva de apeluri, unde pot fi gestionate adecvat prin blocuri ***try-catch***.

4.2 Utilizarea *throw*

Aruncarea unei excepții este adesea însoțită de crearea unei instanțe a unei clase de excepții, care poate fi un obiect derivat din ierarhia standard ***std::exception*** sau o clasă definită de utilizator. Această instanță poate transporta informații detaliate despre eroare, cum ar fi un mesaj de eroare.

Exemplu de utilizare *throw* (ex05.cpp):

```
#include <iostream>
#include <stdexcept>

void VerificaDivizor(int divizor) {
    if (divizor == 0)
        throw std::invalid_argument("Divizorul nu poate fi zero.");
}
```

```
int main()
{
    try {
        VerificaDivizor(0);
    } catch (const std::invalid_argument& e) {
        std::cerr << "A fost prinsa o exceptie: " << e.what() << std::endl;
    }
    return 0;
}
```

În exemplul de mai sus, funcția **VerificaDivizor()** aruncă o excepție de tip ***std::invalid_argument*** dacă divizorul este zero. Excepția este apoi prinsă în blocul ***catch*** al funcției ***main***, permițând programului să continue să execute alte operații sau să gestioneze eroarea într-un mod controlat.

4.3 Personalizarea claselor de excepții

C++ permite definirea propriilor clase de excepții, oferind astfel posibilitatea de a transporta informații specifice aplicației sau domeniului de problemă. Aceste clase personalizate de excepții ar trebui să derive din ***std::exception*** sau din alte clase de excepții standard pentru a se integra bine în ierarhia de excepții a limbajului C++.

Exemplu - clase de exceptii (ex06.cpp)

```
##include <iostream>
#include <stdexcept>

class EroareMatematica : public std::exception {
public:
    const char* what() const noexcept override {
        return "Eroare matematica generala.";
    }
};
void FunctieRiscanta() {
    throw EroareMatematica();
}
int main()
{
    FunctieRiscanta();
    return 0;
}
```

4.4 Observații suplimentare – personalizarea claselor de excepții

Instrucțiunea **throw** este esențială în gestionarea modernă a erorilor în C++. Prin aruncarea și prinderea excepțiilor, programele pot trata condiții excepționale într-un mod flexibil și structurat, minimizând riscul de erori neanticipate și îmbunătățind robustețea și fiabilitatea software-ului.

V. Echivalentul block-ului *finally* în C++ și gestionarea resurselor

În multe limbaje de programare, blocul **finally** este folosit pentru a executa cod care trebuie rulat indiferent dacă au apărut excepții sau nu, adesea pentru eliberarea resurselor. C++, spre deosebire de limbaje precum Java sau C#, nu oferă un bloc **finally** explicit. Totuși, limbajul furnizează alte mecanisme pentru a asigura o gestionare sigură și eficientă a resurselor, chiar și în prezența excepțiilor.

5.2 RAII: Resource Acquisition Is Initialization

Modelul RAII (Resource Acquisition Is Initialization) este paradigma preferată în C++ pentru gestionarea resurselor. Acest principiu se bazează pe utilizarea obiectelor care achiziționează resurse în constructorul lor și le eliberează în destructor. Datorită garantiei de apelare a destructorilor pentru obiectele automate (locale) la ieșirea din scop (inclusiv când se iese din scop prin aruncarea unei excepții), RAII asigură că toate resursele sunt eliberate corespunzător.

Exemplu RAII (ex07.cpp)

```cpp
#include <iostream>
#include <fstream>

class Fisier {
private:
    std::ofstream fisier;
public:
    Fisier(const std::string& numeFisier) {
        fisier = std::ofstream(numeFisier);
        std::cout << "Fisier deschis." << std::endl;
    }
    void Scrie(const std::string& mesaj) {
        if (!fisier.is_open())
            throw std::runtime_error("Fisierul nu este deschis.");
        fisier << mesaj << std::endl;
    }
    ~Fisier() {
        fisier.close();
        std::cout << "Fisier inchis." << std::endl;
    }
};
```

```
int main()
{
    try {
        Fisier jurnal("jurnal.txt");
        jurnal.Scrie("Prima linie in jurnal.");
        throw std::runtime_error("O exceptie generata intentionat.");
    } catch (const std::exception& e) {
        std::cerr << "Exceptie prinsa: " << e.what() << std::endl;
    }
    return 0;
}
```

În exemplul anterior, resursa (fișierul) este achiziționată în constructor și eliberată în destructor, asigurându-se astfel eliberarea resursei chiar dacă o excepție este aruncată.

5.3 Utilizarea smart pointers

O altă abordare comună pentru gestionarea resurselor în C++ este utilizarea smart pointers, cum ar fi ***std::unique_ptr*** și ***std::shared_ptr***, care gestionează automat durata de viață a obiectelor alocate dinamic. Acești pointeri se ocupă de eliberarea memoriei când nu mai este necesară, oferind o alternativă sigură la pointerii raw.

Exemplu utilizare smart pointers alături de excepții (ex08.cpp):

```
#include <iostream>
#include <memory>

void Procesare()
{
    // 'ptr' este un std::unique_ptr<int>
    auto ptr = std::make_unique<int>(42);
    throw std::runtime_error("Exceptie dupa alocare");
    // Memoria alocata este eliberata automat
    // la distrugerea lui 'ptr', chiar daca se arunca o exceptie.
}
int main()
{
    try {
        Procesare();
    } catch (const std::exception& e) {
        std::cerr << "Exceptie prinsa: " << e.what() << std::endl;
    }
    return 0;
}
```

5.4 De reținut – RAII & smart pointers

Deși C++ nu oferă un bloc ***finally***, modelul RAII și smart pointers oferă mecanisme robuste și eficiente pentru gestionarea resurselor, asigurând curățarea resurselor într-un mod sigur, chiar și în fața excepțiilor. Aceste paradigme sunt preferate pentru gestionarea resurselor în C++ și sunt esențiale pentru scrierea de cod robust și sigur din punct de vedere al gestionării resurselor.

VI. Clase de excepții existente în C++

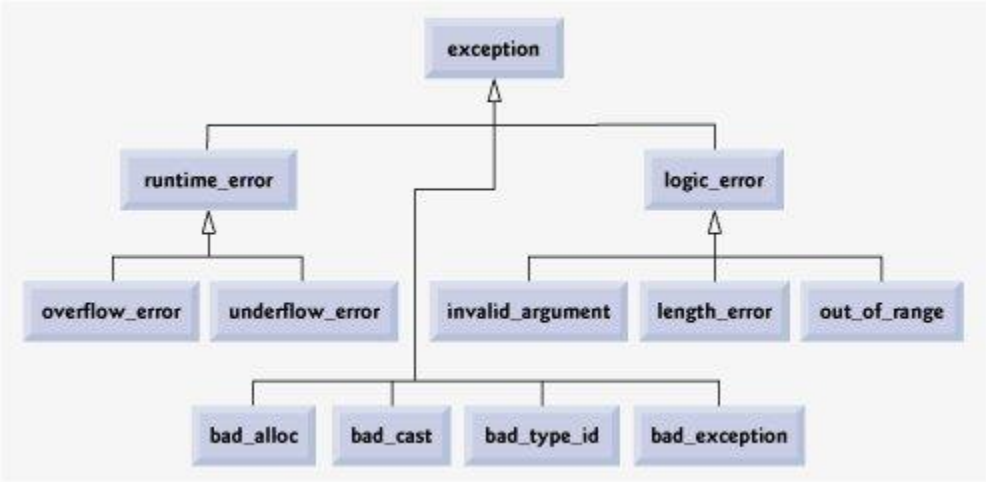
6.1 Introducere la clasele de excepții standard

C++ standardizează gestionarea excepțiilor prin furnizarea unei ierarhii de clase de excepții, cu ***std::exception*** la baza acesteia. Această structură permite programatorilor să trateze și să reacționeze la diferite tipuri de erori într-un mod organizat și eficient.

6.2 Ierarhia claselor de excepții

La baza ierarhiei se află clasa ***std::exception***, care oferă o interfață de bază pentru toate excepțiile standard. Derivate din ***std::exception*** sunt clase care reprezintă categorii specifice de erori, cum ar fi erori de runtime (***std::runtime_error***) și erori logice (***std::logic_error***).

Exemplu de ierarhie:



Sursa: https://flylib.com/books/en/2.253.1/standard_library_exception_hierarchy.html

Fiecare dintre aceste clase derivă din ***std::exception*** și adaugă context suplimentar și/sau specificități legate de tipul de eroare pe care îl reprezintă.

6.3 Utilizarea claselor de excepții standard

Utilizarea claselor de excepții standardizate facilitează comunicarea clară a tipului de eroare care a apărut, permițând gestionarea specifică a diferitelor tipuri de erori predefinite în ierarhia de excepții.

Exemplu de utilizare al claselor de exceptii standard (ex09.cpp)

```
#include <iostream>
#include <stdexcept>

void FunctieRiscanta() {
    // Simulam detectarea unei conditii eronate
    throw std::invalid_argument("Argument invalid furnizat.");
}

int main() {
    try {
        FunctieRiscanta();
    } catch (const std::invalid_argument& e) {
        std::cerr << "A fost prinsa o exceptie de tip invalid_argument: " <<
e.what() << std::endl;
    } catch (const std::exception& e) {
        // Prindem orice alt tip de exceptie derivata din std::exception
        std::cerr << "A fost prinsa o exceptie: " << e.what() << std::endl;
    }
    return 0;
}
```

6.4 Definirea propriilor clase de excepții

Pentru erori specifice aplicației, programatorii pot defini propriile clase de excepții, derivând din **std::exception** sau din oricare dintre clasele sale derivate, pentru a se integra în ierarhia de excepții a C++.

Exemplu - clase de excepții personalizate (ex10.cpp)

```
#include <iostream>
#include <stdexcept>

class EroarePersonalizata : public std::runtime_error {
public:
    EroarePersonalizata(const std::string& mesaj) : std::runtime_error(mesaj)
{}
};
```

```
void FunctieOarecare(void) {
    throw EroarePersonalizata("A aparut o eroare personalizata.");
}
int main() { FunctieOarecare(); return 0; }
```

Prin utilizarea claselor de excepții existente în C++ și prin definirea propriilor clase de excepții, programatorii pot crea un sistem robust de gestionare a erorilor, care comunică clar natura problemelor întâmpinate și permite reacții specifice la diferite condiții eronate.

VII. Moștenirea dintr-o clasă de excepții

7.1 Crearea claselor de excepții personalizate

O practică comună și puternică în programarea C++ este definirea propriilor clase de excepții personalizate, derivând din clasele de excepții standard ale limbajului. Acest lucru permite nu doar personalizarea mesajelor de eroare, dar și adăugarea de context sau informații suplimentare specifice domeniului de aplicare.

7.2 De ce să derivăm clase de excepții

- **Specializarea tratării erorilor**: Permite tratamentul specific al diferitelor tipuri de erori, facilitând reacții adecvate la excepții.
- **Îmbogățirea informațiilor despre erori**: Prin adăugarea de membri și metode noi, clasele de excepții personalizate pot transporta mai multe informații despre condițiile de eroare.
- **Integrare în ierarhia de excepții**: Derivând din **std::exception** sau din alte clase de bază standard, excepțiile personalizate se integrează natural în sistemul de gestionare a erorilor din C++.

7.3 Exemplu - definirea unei clase de excepție personalizată (ex11.cpp)

```
#include <iostream>
#include <stdexcept>
#include <string>

// Clasa de exceptie personalizata derivata din std::runtime_error
class EroareDeConexiune : public std::runtime_error {
private:
    int codEroare;
public:
    EroareDeConexiune(const std::string& mesaj, int cod)
        : std::runtime_error(mesaj), codEroare(cod) {}
```

```
    // Metoda pentru obtinerea codului de eroare
    int GetCodEroare() const {
        return codEroare;
    }
};
// Functie exemplu care utilizeaza exceptia personalizata
void ConectareLaServer() {
    // Simulam o eroare de conexiune
    throw EroareDeConexiune("Conexiune esuata la server.", 404);
}

int main() {
    try {
        ConectareLaServer();
    } catch (const EroareDeConexiune& e) {
        std::cerr << "A fost prinsa o eroare de conexiune: " << e.what()
                  << " Cod eroare: " << e.GetCodEroare() << std::endl;
    } catch (const std::exception& e) {
        // Prindem orice alta exceptie
        std::cerr << "Exceptie neasteptata: " << e.what() << std::endl;
    }
    return 0;
}
```

7.4 Moștenirea și ierarhia de clase de excepții

Crearea unei ierarhii de clase de excepții personalizate, pe baza moștenirii, permite gestionarea granulară și structurată a erorilor. Proiectând ierarhii logice și specifice domeniului, dezvoltatorii pot captura și trata eficient diferite condiții excepționale.

7.5 De reținut – moștenirea din ierarhia claselor de excepții C++

Definirea și utilizarea claselor de excepții personalizate, prin moștenirea din clasele de excepții standard, reprezintă o tehnică avansată și extrem de utilă în gestionarea erorilor în programele C++. Aceasta îmbunătățește semnificativ capacitatea de a raporta și trata erorile într-un mod coerent și detaliat, crescând astfel calitatea și fiabilitatea aplicațiilor, precum și scalabilitatea și citibilitatea, fără costuri suplimentare de timp programator (doar cost de timp procesor, vom discuta în detaliu).

VIII. Propagarea excepțiilor

Propagarea excepțiilor este procesul prin care o excepție aruncată, dar neprinsă într-un anumit punct al programului, continuă să se propage în sus în stiva de apeluri, căutând un bloc **catch** corespunzător care să o gestioneze. Acest mecanism permite aplicațiilor să centralizeze gestionarea erorilor în locuri strategice, reducând astfel necesitatea de a trata excepțiile local în fiecare funcție.

8.2 Cum funcționează propagarea excepțiilor

Când o excepție este aruncată cu **throw** și nu este prinsă în blocul **try** curent, execuția programului iese din scopul curent și continuă să se deplaseze în sus pe stiva de apeluri, până când întâlnește un bloc **catch** care poate gestiona tipul de excepție aruncat. Dacă niciun bloc **catch** nu prinde excepția până când aceasta ajunge la **main()**, și **main()** nu o prinde, atunci programul se termină, apelând **std::terminate()**.

Exemplu – propagare excepții (ex12.cpp)

```cpp
#include <iostream>
#include <stdexcept>

void FunctieNivel3(void) {
    throw std::runtime_error("Eroare la nivelul 3");
}
void FunctieNivel2(void) {
    FunctieNivel3();  // Exceptia aruncata aici va fi propagata mai departe
}
void FunctieNivel1(void) {
    try {
        FunctieNivel2();
    } catch (const std::exception& e) {
        std::cout << "Exceptie prinsa in nivelul 1: " << e.what() << std::endl;
        // Exceptia este gestionata aici, oprind propagarea
    }
}
int main()
{
    FunctieNivel1();
    return 0;
}
```

În acest exemplu, **FunctieNivel3()** aruncă o excepție care este propagată prin **FunctieNivel2()**, până când este prinsă și gestionată în **FunctieNivel1()**.

8.4 Gestionarea excepțiilor la diferite nivele

Propagarea excepțiilor permite dezvoltatorilor să decidă la ce nivel al aplicației ar trebui gestionate diferite tipuri de erori. Unele excepții pot necesita tratament imediat, în timp ce altele pot fi mai bine gestionate la un nivel superior, unde deciziile pot fi luate în contextul unei înțelegeri mai largi a stării aplicației.

8.5 De reținut – propagarea și tratarea excepțiilor la diferite nivele

Prin înțelegerea și utilizarea corectă a acestui mecanism, programatorii pot scrie cod mai robust, capabil să gestioneze condiții "speciale" complexe într-un mod organizat și eficient.

IX. Recapitulare și cele mai importante puncte despre excepții în C++

9.1 Importanța excepțiilor

Excepțiile sunt fundamentale în programarea modernă C++, oferind un mecanism robust pentru gestionarea erorilor și a condițiilor excepționale. Prin utilizarea excepțiilor, programele devin mai sigure și mai ușor de întreținut, permițând separarea clară între logica de afaceri și gestionarea erorilor.

9.2 Recapitulare

În acest modul, am acoperit aspecte esențiale ale sistemului de excepții în C++, inclusiv:

- **Motivația pentru utilizarea excepțiilor**: Oferă o abordare structurată pentru gestionarea erorilor, îmbunătățind claritatea și modularitatea codului.
- **Funcțiile setjmp() și longjmp()**: Discutate ca metode anterioare de gestionare a fluxului de control, evidențiind limitările lor în comparație cu sistemul de excepții.
- **Structura try-catch și instrucțiunea throw**: Elementele de bază pentru aruncarea și prinderea excepțiilor, permițând gestionarea eficientă a diferitelor tipuri de erori.
- **Absența blocului finally în C++**: Înlocuit de modelul RAII și de utilizarea smart pointers pentru gestionarea resurselor și asigurarea eliberării corecte în caz de excepție.
- **Clasele de excepții standard și personalizate**: Utilizarea și crearea claselor de excepții pentru a comunica clar tipurile de erori și pentru a transporta informații suplimentare despre eroare.
- **Moștenirea din clase de excepții**: Crearea unei ierarhii de excepții pentru gestionarea detaliată și specifică a erorilor.
- **Propagarea excepțiilor**: Mecanismul prin care excepțiile neprinse se propagă în sus în stiva de apeluri, căutând un bloc **catch** corespunzător.

9.3 Cele mai importante puncte

- **Utilizați excepțiile pentru condiții excepționale**: Excepțiile ar trebui rezervate pentru erori sau alte condiții neobișnuite care necesită tratament special.
- **Gestionați excepțiile la nivelul potrivit**: Capturați și tratați excepțiile la nivelul în care puteți face ceva util cu ele.
- **Curățenia și eliberarea resurselor**: Folosiți RAII și smart pointers pentru a vă asigura că resursele sunt eliberate corespunzător, chiar și în fața excepțiilor.
- **Definiți excepții personalizate când este necesar**: Pentru a comunica erori specifice aplicației sau domeniului de problemă.

9.4 Important de reținut din laboratorul curent

Excepțiile reprezintă o parte integrantă a scrierii de software robust și fiabil în C++. Înțelegând și aplicând corect principiile discutate, veți putea îmbunătăți semnificativ gestionarea erorilor în programele voastre C++.

Aici sunt câteva aspecte suplimentare care merită evidențiate pentru a oferi o înțelegere mai completă a gestionării excepțiilor:

1. **Specificații de excepții**: C++ permite specificarea excepțiilor pe care o funcție le poate arunca prin utilizarea specifier-ului **throw()** în declarația funcției. Cu toate acestea, această practică a fost descurajată și în cele din urmă eliminată începând cu C++17, în favoarea noexcept, care indică dacă o funcție este garantată să nu arunce excepții.
2. **noexcept**: Utilizarea specifierului **noexcept** indică faptul că o funcție nu va arunca nicio excepție. Aceasta poate îmbunătăți performanța, deoarece permite compilatorului să facă optimizări suplimentare, știind că funcția nu va cauza ieșirea din scop prin excepții.
3. **Tratarea excepțiilor în constructori și destructori**: Gestionarea excepțiilor în constructori și destructorii necesită o atenție specială. Aruncarea unei excepții dintr-un destructor trebuie evitată, deoarece poate duce la terminarea programului dacă destructorul este apelat în timpul desfășurării unei alte excepții. Începând cu C++11, destructorii sunt **noexcept** implicit.
4. **Excepții standard derivate din std::exception**: Familiarizara în detaliu cu diferitele tipuri de excepții derivate din **std::exception**, cum ar fi **std::logic_error**, **std::runtime_error**, **std::bad_alloc**, **std::bad_cast**, etc.,
5. **Excepții și moștenire**: Atunci când lucrați cu moștenirea și polimorfismul, este important să înțelegeți cum sunt propagate excepțiile prin ierarhia de clase și cum să proiectați clase de bază și derivate pentru a gestiona corect excepțiile.
6. **Tratarea excepțiilor fără bloc catch**: Explicarea comportamentului programului când o excepție este aruncată dar nu este prinsă de niciun bloc **catch**. Acest lucru include discuția despre funcția **std::terminate()** și posibilitatea de a personaliza comportamentul acesteia prin **std::set_terminate()**.
7. **Tratarea și logarea eroilor**: În contextul unei aplicații mari, poate fi util să discutăm despre strategii pentru logarea și monitorizarea erorilor care apar, facilitând astfel diagnosticarea și remedierea problemelor.