

Excepții în C++

Anul Universitar 2023 – 2024, Semestrul II

Programare Orientată pe Obiecte

LABORATOR 8, GRUPA 133

Cuprins

| | |
|--------------------------------------------------------------------------------------|-----------|
| I. Introducere în Templates (Şabloane) în C+..... | 3 |
| 1.1. Definiție și motivație | 3 |
| 1.2. Exemplu introductiv templates (ex01.cpp)..... | 3 |
| 1.3. Discuții finale și concluzii | 4 |
| II. Şabloanele și biblioteca standard STL | 4 |
| 2.1. Legătura dintre şabloane și STL | 4 |
| 2.2. Exemplu de utilizare a şablonelor în STL..... | 5 |
| 2.3. Discuții finale și concluzii | 5 |
| III. Implementarea pas cu pas a unei clase Vector | 6 |
| 3.1. Introducere la vectori ca structuri de date | 6 |
| 3.2. Implementarea unui Vector de <i>int</i> | 6 |
| 3.3. Generalizarea la o clasă <i>Vector<T></i> generică cu template-uri | 8 |
| 3.4. Discuții finale și concluzii | 9 |
| IV. Smart pointers și templates..... | 9 |
| 4.1. Introducere în smart pointers..... | 9 |
| 4.2. Motivație și utilizare | 10 |
| 4.3. Exemplu simplu cu <i>std::unique_ptr</i> | 10 |
| 4.4. Discuții finale și concluzii | 11 |
| V. Implementarea unui unique_ptr simplificat..... | 11 |
| 5.1. Introducere - <i>unique_ptr</i> | 11 |
| 5.2. Motivație..... | 11 |
| 5.3. Implementarea propriu-zisă | 11 |
| 5.4. Discuții finale și concluzii | 12 |
| VI. Modelul ErrorOr<Type> versus costurile gestionării excepțiilor..... | 13 |
| 6.1. Introducere la Gestionarea Erorilor în C++ | 13 |

| | | |
|--------------|----------------------------------------------------------------|-----------|
| 6.2. | Motivația pentru ErrorOr<Type> | 13 |
| 6.3. | Implementarea Simplă a ErrorOr<Type> | 13 |
| 6.4. | Discuții despre Costurile Gestionării Excepțiilor..... | 14 |
| 6.5. | Discuții finale și concluzii | 15 |
| VII. | Template Specialization (Specializarea şablonelor)..... | 15 |
| 7.1. | Introducere în specializarea template-urilor | 15 |
| 7.2. | Motivație și utilizare | 15 |
| 7.3. | Exemplu de specializare de template-uri..... | 15 |
| 7.4. | Discuții finale și concluzii | 16 |
| VIII. | Alte utilizări ale template-urilor în C++..... | 17 |
| 8.1. | Introducere | 17 |
| 8.2. | Variadic templates | 17 |
| 8.3. | Exemplu de variadic template | 17 |
| 8.4. | Concepts (constrângeri) - descriere | 17 |
| 8.5. | Exemplu de utilizare concepts | 18 |
| 8.6. | Concepts în C# (generic constraints) | 18 |
| 8.7. | Discuții finale și concluzii | 19 |

Autor: Wagner Ștefan Daniel

I. Introducere în Templates (Şabloane) în C+

1.1. Definiție și motivație

Şabloanele în C++ sunt o caracteristică puternică care permite programatorilor să scrie cod flexibil și reutilizabil fără a sacrifica performanța.

Motivația principală pentru utilizarea şabloanelor este de a permite funcțiilor și claselor să opereze cu orice tip de date, fără a fi necesară rescrierea codului pentru fiecare tip.

Acest lucru este esențial pentru bibliotecile standard, cum ar fi STL (Standard Template Library), care oferă o gamă largă de structuri de date și algoritmi.

Observație: pe parcursul acestui laborator vom folosi *template* și *șablon* interschimbabil; deși terminologia în engleză este preferată în programare întotdeauna, este bine să fiți familiarizați cu concepțele în ambele limbi (universal valabil pentru toate materiile, nu numai programare, inclusiv matematicile spre exemplu, dar nu doar).

1.2. Exemplu introductiv templates (ex01.cpp)

Un exemplu simplu de şablon – o funcție care returnează valoarea maximă dintre două valori:

```
#include <iostream>

// template <class T> works as well, but <typename T> is preferred
template<typename T>
T Max(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    // Demonstrating with int
    int iMax = Max(5, 7);
    std::cout << "Maximum (int): " << iMax << std::endl;

    // Demonstrating with double
    double dMax = Max(6.34, 18.523);
    std::cout << "Maximum (double): " << dMax << std::endl;

    // Demonstrating with char
    char cMax = Max('a', 'z');
    std::cout << "Maximum (char): " << cMax << std::endl;

    return 0;
}
```

Acest exemplu ilustrează cum şabloanele permit funcției ***Max()*** să opereze cu orice tip de date, de la întregi și numere floating point până la caractere, și orice alt tip de date care are supraîncarcat ***operator>()*** de comparatie.

Utilizatorul (voi) puteți adapta această funcție inclusiv pentru a determina valoarea maximă dintre două variabile ale căror tipuri pot fi diferite (***template typename<T1, T2>***), cât timp ***operator>()*** de comparație este implementat pentru ***T1*** comparat cu ***T2***, demonstrând astfel universalitatea și flexibilitatea oferită de mecanismul de templates. Funcția ***Max()*** este evident doar un exemplu particular, care poate fi adaptat la rândul lui.

1.3. Discuții finale și concluzii

Utilizarea şabloanelor reduce necesitatea de a supraincarca funcțiile pentru diferite tipuri de date, ceea ce poate duce la o reducere semnificativă a cantității de cod sursă și la o întreținere mai simplă a codului.

De asemenea, şabloanele sunt evaluate la compilare, ceea ce înseamnă că nu există o “penalizare” a performanței la runtime; acesta este un avantaj major față de alte limbi de programare care implementează template-urile (generics) prin mecanisme care pot introduce overhead la runtime.

Aceast prim capitol a introdus conceptul de template-uri în C++ și a oferit un exemplu simplu de utilizare, dar și o idee cum ar putea fi modificat, și evident, generalizat.

În următoarele capitole, vom explora mai în detaliu legătura dintre şabloane și STL, și vom introduce concepte mai complexe legate de şabloane, cum ar fi specializarea şabloanelor și şabloanele pentru smart pointers, dar nu numai 😊.

II. Şabloanele și biblioteca standard STL

2.1. Legătura dintre şabloane și STL

STL este o colecție complexă de clase și funcții (pe care am utilizat-o încă din [Laboratorul 4, capitolul V](#)), cât mai ușor de utilizat, care include containere generice precum vector, list, set, și map (hashTables), printre altele, alături de algoritmi generici pentru sortare, căutare, și manipulare de date, toate implementate utilizând mecanismul de templates.

Referință completă pentru containerele STL: <https://cplusplus.com/reference/stl/>.

Template-urile sunt esențiale în programarea orientată pe obiecte, indiferent de limbajul de programare utilizat, acest motiv și multe altele pe care le vom discuta ulterior (inclusiv [Design Patterns \(link carte\)](#): "Program to an interface, not an implementation.")

Concepțe precum [Domain Driven Design](#) ([link carte](#)), [Dependency Injection](#) ([link carte](#)), Clean Architecture ([link carte](#)), care de asemenea pentru a fi utilizate la adevărata lor capacitate în producție conțin inevitabil templates. Acesta este un alt motiv pentru care este bine să vă punetăți fundamentele, însă din pacate nu avem timp și este mult prea devreme pentru voi să învătați aşa ceva, vă rog să salvați linkurile pentru viitor, nu o să vă pară rău.

Aceste componente sunt proiectate să fie independente de tipurile de date specifice, ceea ce le permite să fie utilizate cu orice tip de date printr-o simplă specificare de tip și la momentul compilării se vor genera clasele/funcțiile corespunzătoare cu tipurile cerute (**compile-time polymorphism**).

2.2. Exemplu de utilizare a şabloanelor în STL

Un exemplu clasic ilustrativ (pe care deja îl cunoașteți) este utilizarea containerului *std::vector*, care este un template deci poate stoca orice tip de date specificat. Aici vedem cum putem defini un vector pentru int și unul pentru *std::string*:

```
#include <vector>
#include <string>
#include <iostream>

int main() {
    // Example for int vector
    std::vector<int> intVector = {1, 2, 3, 4, 5};
    std::cout << "intVector: ";
    for (int i : intVector) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    // Example for string vector
    std::vector<std::string> stringVector = {"Hello", "World", "from",
    "Templates"};
    std::cout << "stringVector: ";
    for (const std::string& str : stringVector) {
        std::cout << str << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

2.3. Discuții finale și concluzii

Utilizarea şabloanelor în STL nu se limitează doar la containere, ci se extinde și la algoritmi. De exemplu, funcția *std::sort()* poate sorta orice container STL, deoarece implementarea sa utilizează

șabloane pentru a abstractiza tipul elementelor containerului. Această capacitate de a opera cu orice tip de date face din șabloane și STL unelte extrem de puternice în C++.

Prin această capacitate de a generaliza comportamentul pentru orice tip de date, șabloanele permit programatorilor să scrie cod mai curat și mai eficient, reducând redundanța și mărind reutilizabilitatea codului.

III. Implementarea pas cu pas a unei clase Vector

3.1. Introducere la vectori ca structuri de date

Înainte de a aborda implementarea unui vector cu șabloane, este util să începem cu o implementare specifică pentru un tip de date, cum ar fi *int*. Acest lucru va oferi o bază solidă înainte de a generaliza clasa pentru orice tip de date.

3.2. Implementarea unui Vector de *int*

Să începem prin a construi o clasă simplă **IntVector** care gestionează un array dinamic de întregi.

Această clasă va include funcționalități de bază precum adăugarea de elemente și accesarea acestora, dar și un exemplu de utilizare *assert()*.

Funcția *assert()* (definită ca **macro** în header-ul *<cassert>* / *<assert.h>*) primește un parametru de tip ”boolean”, o condiție care dacă este falsă, programul nu poate continua și comportamentul în acest caz este de afișare al unui mesaj de eroare (care conține linia și fișierul, printre altele), și apelarea funcției *abort()*, care tehnic face core dump, pentru a putea studia memoria programului când a fost întâlnită condiția din care programul nu își poate reveni.

Observație (fun fact): ”boolean” impropriu spus, deoarece este o librărie C unde **bool** este implementation defined pre C99 (*_Bool* ca macro, *typedef* pentru *unsigned char*, neportabil), nici keyword-urile *true* și *false*, care sunt definite ca macro-uri). Începând cu C99, a fost inclus în standard header-ul *<stdbool.h>*, care garantează indiferent de compilator portabilitatea pentru *bool* în C. Deși niciunul din aceste lucruri nu este relevant, deoarece orice diferit de zero este *true*, indiferent de tipul de date, și *assert()* este definit ca macro, totuși am inclus observația, deoarece *<assert.h>* este definit încă din standardul C89, și precede portabilitatea tipului de date boolean în C. Mai mult, compilatoarele evaluatează *true* la 1 și *false* la 0, deci este perfect ok să trimitem și unei funcții C un *bool* C++.

```

#include <iostream>
#include <cassert>

class IntVector {
private:
    int *data;          // Pointer to int array
    size_t size;        // Number of elements currently in the array
    size_t capacity;   // Maximum capacity (current size)

public:
    IntVector() : data{ nullptr }, size{ 0 }, capacity{ 0 } {}

    void push_back(int value)
    {
        if (size == capacity)
        {
            // We double the capacity because reallocating
            // costs CPU time, and the entire memory segment
            // requires moving, unlike when using realloc(),
            // where there might be space left at the end
            size_t newCapacity = (capacity == 0) ? 2 : capacity * 2;
            int *newData = new int[newCapacity];
            for (size_t i = 0; i < size; ++i)
            {
                newData[i] = data[i];
            }
            delete[] data;
            data = newData;
            capacity = newCapacity;
        }
        data[size++] = value;
    }

    int operator[](size_t index) const
    {
        // We will talk about asserts in more detail verbally
        // and there are links included in the materials
        assert(index < size);
        return data[index];
    }

    ~IntVector() { delete[] data; }
};


```

```

int main()
{
    IntVector vec;
    vec.push_back(10);
    vec.push_back(20);
    // Output: 10
    std::cout << "Element at index 0: " << vec[0] << std::endl;
    // Output: 20
    std::cout << "Element at index 1: " << vec[1] << std::endl;
    return 0;
}

```

După ce am implementat și înțeles *IntVector*, următorul pas este să generalizăm această clasă pentru a lucra cu orice tip de date folosind şabloane. Aceasta implică transformarea clasei *IntVector* într-o clasă *Vector<T>*, în secțiunea următoare.

3.3. Generalizarea la o clasă *Vector<T>* generică cu template-uri

```

#include <iostream>

template<typename T>
class Vector {
private:
    T* data;
    size_t size;
    size_t capacity;
public:
    Vector() : data(nullptr), size(0), capacity(0) {}
    void push_back(T value) {
        // We double the capacity because reallocating
        // costs CPU time, and the entire memory segment
        // requires moving, unlike when using realloc(),
        // where there might be space left at the end
        if (size == capacity) {
            size_t newCapacity = (capacity == 0) ? 2 : capacity * 2;
            T* newData = new T[newCapacity];
            for (size_t i = 0; i < size; ++i) {
                newData[i] = data[i];
            }
            delete[] data;
            data = newData;
            capacity = newCapacity;
        }
        data[size++] = value;
    }
}

```

```

// We will talk about asserts in more detail verbally
// and there are links included in the materials
T operator[](size_t index) const
{
    assert(index < size);
    return data[index];
}

~Vector() { delete[] data; }

int main()
{
    Vector<int> intVec;
    intVec.push_back(10);
    intVec.push_back(20);

    Vector<std::string> stringVec;
    stringVec.push_back("Hello");
    stringVec.push_back("World");
    // Same output - 10 20 - type int
    std::cout << "Int Vector: " << intVec[0] << ", " << intVec[1] << std::endl;
    // Now, we get the classic "Hello world" - type string
    std::cout << "String Vector: " << stringVec[0] << ", " << stringVec[1]
                  << std::endl;
    return 0;
}

```

3.4. Discuții finale și concluzii

Prin generalizarea de la *IntVector* la *Vector<T>*, am demonstrat cum template-urile pot fi utilizate pentru a crea structuri de date flexibile și reutilizabile (evident, asta este o implementare simplificată, însă vă puteți imagina cum sunt implementate containerele STL facând asocierea și cu cursul de Structuri de Date).

IV. Smart pointers și templates

4.1. Introducere în smart pointers

Pointerii “inteligenti” (numiți în continuare strict smart pointers, deoarece este un nume forțat) în C++ sunt un mecanism avansat care ajută la gestionarea resurselor, în special a memoriei alocate dinamic.

Folosirea smart pointers asigură o eliberare automată a memoriei, prevenind memory leaks și alte probleme legate de gestionarea manuală a memoriei.

Librăria care conține implementarea mecanismului de smart pointers în C++ (pentru care trebuie inclus header-ul `<memory>`) pune la dispoziție următoarele tipuri de smart pointers, pe care îi vom discuta în particular mai jos: `std::unique_ptr`, `std::shared_ptr`, și `std::weak_ptr`.

4.2. Motivație și utilizare

Utilizarea smart pointers este motivată de necesitatea de a avea un cod mai sigur și mai ușor de întreținut.

Prin încapsularea unui pointer raw (*Tip**) într-un obiect, smart pointers automatizează gestionarea ciclului de viață al obiectului alocat, inclusiv eliberarea resursei când nu mai este necesară.

4.3. Exemplu simplu cu `std::unique_ptr`

`std::unique_ptr` este un smart pointer care deține exclusivitatea (termen consacrat: **"ownership"**) asupra unei resurse.

O resursă gestionată de `std::unique_ptr` nu poate fi copiată într-un alt `unique_ptr`, transferul de ownership prin intermediul constructorului de mutare sau operatorului de atribuire prin mutare fiind singura opțiune.

Exemplu simplu unde utilizăm `std::unique_ptr` pentru a gestiona un array de întregi:

```
#include <vector>
#include <string>
#include <iostream>

int main() {
    // Example for int vector
    std::vector<int> intVector = {1, 2, 3, 4, 5};
    std::cout << "intVector: ";
    for (int i : intVector) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    // Example for string vector
    std::vector<std::string> stringVector = {"Hello", "World", "from",
"Templates"};
    std::cout << "stringVector: ";
    for (const std::string& str : stringVector) {
        std::cout << str << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

4.4. Discuții finale și concluzii

Utilizarea *shared_ptr* și *weak_ptr* oferă un control mai mare asupra gestionării memoriei în aplicațiile C++ mai complexe, unde obiectele sunt partajate între multiple componente ale sistemului.

Acești smart pointers joacă un rol crucial în prevenirea memory leaks și în gestionarea ciclului de viață al obiectelor.

Această secțiune a abordat conceptul de *smart pointers* și a demonstrat cum aceștia pot fi utilizati alături de template-uri pentru a simplifica gestionarea resurselor în C++, astfel obținem eliberare automată a memoriei când obiectul iese din scop (pur și simplu este destructorul apelat, precum la un obiect normal), spre deosebire de alte limbi OOP, fără overhead-ul unui garbage collector, care costă și timp procesor și necesită comutare între thread-uri și memorie.

V. Implementarea unui unique_ptr simplificat

5.1. Introducere - unique_ptr

Precum am discutat mai sus, **unique_ptr** este un smart pointer din biblioteca standard C++ care deține exclusivitatea asupra unei resurse. Este utilizat pentru gestionarea resurselor alocate dinamic, asigurând că memoria este eliberată automat când **unique_ptr** este distrus sau când ia posesia unei alte resurse.

5.2. Motivație

Implementarea unui **unique_ptr** simplificat ne va ajuta să înțelegem principiile de bază ale RAII (Resource Acquisition Is Initialization), care este esențial pentru gestionarea eficientă a resurselor în C++. Vom începe cu un tip de bază (**int**), pentru a simplifica conceptele.

5.3. Implementarea propriu-zisă

Vom implementa o versiune simplificată a **unique_ptr** care gestionează un **int**. Această implementare va include operatorii necesari pentru a demonstra transferul de ownership.

```
#include <iostream>

template<typename T>
class UniquePtr {
private:
    T* ptr;
public:
    explicit UniquePtr(T* p = nullptr) : ptr(p) {}

    ~UniquePtr() { delete ptr; }
```

```

// Deleting copy constructor and assignment operator
// So we will 'steal' the object, making it unique
UniquePtr(const UniquePtr&) = delete;
UniquePtr& operator=(const UniquePtr&) = delete;

// Move constructor
UniquePtr(UniquePtr&& other) noexcept : ptr(other.ptr) {
    other.ptr = nullptr;
}

// operator= for moving the UniquePtr
UniquePtr& operator=(UniquePtr&& other) noexcept
{
    if (this != &other)
    {
        delete ptr;
        ptr = other.ptr;
        other.ptr = nullptr;
    }
    return *this;
}

T& operator*() const { return *ptr; }
T* operator->() const { return ptr; }
bool isNull() const { return ptr == nullptr; }
};

int main()
{
    UniquePtr<int> myPtr{new int(10)};
    std::cout << "Value: " << *myPtr << std::endl;

    UniquePtr<int> movedPtr = std::move(myPtr);
    std::cout << "Moved Value: " << *movedPtr << std::endl;
    std::cout << "Original pointer is null: " << myPtr.isNull() << std::endl;

    return 0;
}

```

5.4. Discuții finale și concluzii

Această implementare ilustrează cum *unique_ptr* gestionează ownership-ul unei resurse. Prin suprascrierea operatorilor de mutare și dezactivarea copierii, ne asigurăm că *unique_ptr* menține un

control strict asupra obiectului pe care îl gestionează, și are un singur proprietar la orice moment din lifetime-ul obiectului.

Utilizarea acestui model în proiecte mai mari poate ajuta la prevenirea memory leaks și la simplificarea codului de gestionare a resurselor.

Această secțiune a demonstrat cum să implementăm un *unique_ptr* simplificat, oferind un exemplu concret al aplicării RAI despre care am discutat în [Laboratorul 8](#), și a gestionării resurselor alocate dinamic fără să fie necesar apelul explicit al destructorilor în C++.

VI. Modelul ErrorOr<Type> versus costurile gestionării exceptiilor

6.1. Introducere la Gestionarea Erorilor în C++

În C++, gestionarea erorilor poate fi realizată prin mai multe mecanisme, inclusivând exceptii și diverse tipuri de returnări de erori. Exceptiile oferă o cale puternică de propagare a erorilor prin stack, dar pot avea un cost semnificativ în termeni de performanță. Alternativa, utilizarea unor structuri - [discriminated unions](#) - cum ar fi **ErrorOr<Type>** (articole în detaliu [aici](#)), poate oferi o soluție mai eficientă și mai predictibilă pentru gestionarea erorilor, mai ales în contexte unde performanța este critică.

6.2. Motivația pentru ErrorOr<Type>

ErrorOr<Type> este un tip de date care poate stoca fie o valoare utilă de un anumit tip **Type**, fie o eroare. Acest model permite funcțiilor să returneze fie un rezultat valid, fie să raporteze o eroare fără a arunca exceptii. Acest lucru este deosebit de util în librării care necesită gestionarea detaliată a erorilor fără overhead-ul asociat cu exceptiile.

6.3. Implementarea Simplă a ErrorOr<Type>

Implementăm un şablon simplu pentru **ErrorOr<Type>**, care encapsulează fie o valoare, fie un mesaj de eroare:

```
#include <iostream>
#include <string>
#include <variant>

template<typename T>
class ErrorOr {
private:
    std::variant<T, std::string> result;

public:
    explicit ErrorOr(T value) : result(value) {}
    explicit ErrorOr(std::string error) : result(error) {}
```

```

bool isError() const
{
    return std::holds_alternative<std::string>(result);
}

T& getValue()
{
    if (isError()) {
        throw std::logic_error("Accessing error as value");
    }
    return std::get<T>(result);
}

std::string getError() const
{
    if (!isError()) {
        throw std::logic_error("Accessing value as error");
    }
    return std::get<std::string>(result);
};

int main()
{
    ErrorOr<int> successResult(42);
    ErrorOr<int> errorResult("An error occurred");

    if (!successResult.isError()) {
        std::cout << "Success result: " << successResult.getValue() <<
std::endl;
    }

    if (errorResult.isError()) {
        std::cout << "Error: " << errorResult.getError() << std::endl;
    }

    return 0;
}

```

6.4. Discuții despre Costurile Gestionařii Excepđiilor

Excepđiile pot introduce overhead de timp procesor în timpul execuđiei datorită necesităđii de a gestiona [stack unwinding-ul](#) - salvarea regiștrilor și contextul stivei curente pentru a putea ”sări” dintr-un punct în altul în program și a restituî starea iniđială

Spre deosebire de excepții, **ErrorOr<Type>** verifică erorile într-un mod controlat, fără a implica stack unwinding-ul, ceea ce poate duce la performanțe îmbunătățite în anumite scenarii, mai ales în sistemele încorporate sau în aplicațiile unde costul excepțiilor este considerabil.

Pentru mai multe detalii, puteți să vedeați videoclipul următor, și vă recomand să pastrați canalul pe viitor:

[**Don't throw exceptions in C#. Do this instead**](#) - **ErrorOr<Type>** vs excepții + benchmarking, totul este aplicabil la C++, nu contează că este în C# exemplul concret.

6.5. Discuții finale și concluzii

Pattern-ul **ErrorOr<Type>** oferă o alternativă robustă la gestionarea excepțiilor, permitând programatorilor să aibă control complet asupra modului în care sunt gestionate erorile și să optimizeze performanța aplicațiilor.

VII. Template Specialization (Specializarea şablonelor)

7.1. Introducere în specializarea template-urilor

Specializarea şablonelor este o tehnică în C++ care permite programatorilor să definească o implementare diferită a unui şablon pentru un tip de date specific.

Aceasta oferă flexibilitatea de a optimiza sau de a modifica comportamentul unei funcții sau clase şablon pentru cazuri particulare, fără a schimba definiția generală a şablonului.

7.2. Motivație și utilizare

Specializarea este utilă când vrem să tratăm anumite tipuri într-un mod diferit decât în implementarea generică a unui şablon. De exemplu, poate fi nevoie să gestionăm eficient tipuri de date specifice sau să implementăm o logică particulară care este relevantă doar pentru anumite tipuri.

7.3. Exemplu de specializare de template-uri

Începem cu o clasă şablon generică pentru un container care stochează un element și permite accesul la acesta. Ulterior, vom specializa această clasă pentru tipul **char**.

Implementarea generică:

```
template<typename T>
class Container {
private:
    T value;
public:
    Container(T v) : value(v) {}
    void print() const
    {
        std::cout << "Generic Container: " << value << std::endl;
    }
}
```

```
};
```

Specializarea pentru *char*:

```
template<>
class Container<char> {
private:
    char value;
public:
    Container(char v) : value(v) {}

    void print() const
    {
        std::cout << "Char Container: value is a character: "
              << value << std::endl;
    }
};
```

Funcția *main()* pentru a testa comportamentul specializării de template-uri:

```
int main()
{
    Container<int> intContainer(123);
    intContainer.print();

    Container<char> charContainer('A');
    charContainer.print();

    return 0;
}
```

Acest exemplu ilustrează cum specializarea şablonului pentru *char* permite implementarea unei funcții *print()* care tratează *char* diferit de alte tipuri.

Astfel, putem personaliza comportamentul clasei în funcție de tipul de date specificat, îmbunătățind claritatea sau performanța codului unde este necesar, sau când un anume tip de date (sau mai multe) trebuie tratate diferit de toate celelalte.

7.4. Discuții finale și concluzii

Specializarea template-urilor este încă o unealtă puternică oferită de limbajul C++ care permite optimizări specifice și comportamente personalizate pentru tipuri de date specifice, menținând în același timp genericitatea și reutilizabilitatea codului.

VIII. Alte utilizări ale template-urilor în C++

8.1. Introducere

Pe lângă funcționalităile de bază ale template-urilor pe care le-am explorat până acum, C++ oferă și alte caracteristici avansate care pot extinde și mai mult utilitatea template-urilor. Acestea includ variadic templates, [template metaprogramming](#) (despre care nu vom discuta, dar vă las referință) și concepts, care permit scrierea unui cod mai expresiv și eficient.

8.2. Variadic templates

Variadic templates permit definirea de funcții și clase care pot accepta un număr variabil de argumente de tipuri diferite (deși sună înfricoșător, este mult mai ușor decât mecanismul C din `<stdarg.h>`).

Acestea sunt utile pentru crearea de funcții sau clase “extrem” de flexibile, care acceptă orice tip de date, cât timp operatorii utilizați de funcție sunt supraîncarcați pentru tipul de date folosit.

8.3. Exemplu de variadic template

Să considerăm o funcție simplă care afișează orice număr de argumente:

```
#include <iostream>

template<typename... Args>
void PrintAll(Args... args)
{
    (std::cout << ... << args) << std::endl;
}

int main()
{
    PrintAll(1, "apple", 3.14, 'c');
    return 0;
}
```

Acest exemplu utilizează un [fold expression](#) pentru a afișa toate argumentele. Acest tip de template este extrem de puternic pentru crearea de biblioteci generice care pot opera cu orice tipuri de date (cu condițiile menționate mai sus legate de operatorii folosiți).

8.4. Concepts (constrângeri) - descriere

Introduse în C++20, concepts sunt o modalitate de a specifica **constrângeri pe tipurile de date folosite în template-uri**, făcând codul mai sigur și mai ușor de înțeles.

8.5. Exemplu de utilizare concepts

Definirea unui concept pentru a verifica dacă un tip de date este un container (în continuare, presupunem că un container este suficient să implementeze iteratorii, însă nu este un exemplu cu adevărat complet, ar trebui constrângerile pentru fiecare operator/metodă folosită:

```
// C++20 is required

// Header for built-in C++ concepts
// But we're defining our own concept
// #include <concepts>

#include <iostream>
#include <vector>
#include <iterator>

template<typename T>
concept IsContainer = requires(T a) {
    std::begin(a);
    std::end(a);
};

template<typename T>
requires IsContainer<T>
void PrintContainer(const T& container) {
    for (const auto& element : container) {
        std::cout << element << " ";
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    PrintContainer(vec);
    return 0;
}
```

Acest exemplu demonstrează cum conceptul *IsContainer* poate fi folosit pentru a restricționa funcția *PrintContainer()* să accepte doar tipuri de date care sunt containere, cu presupunerile de mai sus.

8.6. Concepts în C# (generic constraints)

Deși acesta este un curs POO C++, ”concepts” devine mult mai clar dacă îl vedem și în C#. unde constrângerile sunt mult mai clare:

```
public class MyGenericClass<T> where T : IComparable<T>
{
    public T Data { get; set; }

    public bool IsGreater Than(T value)
    {
        return Data.CompareTo(value) > 0;
    }
}
```

Explicație: C# implementează constrângerile cu ajutorul cuvântului cheie **where**, și exprimă mult mai bine intenția decât C++. Suprîncărcarea operatorilor în C# presupune suprascrierea unor metode definite în clasa părinte, numite explicit.

Deși mecanismul de "concepts" este mai ușor de înțeles și de implementat în C#, este de departe mult mai dezvoltat în C++

8.7. Discuții finale și concluzii

Aceste caracteristici avansate ale template-urilor deschid posibilități vaste pentru optimizarea și generalizarea codului în C++. Folosirea lor 'înțeleaptă' poate duce la crearea de software mai robust și mai eficient, mai ușor scalabil și de menținut, cât mai generic. Prin explorarea acestor tehnici avansate, programatorii pot îmbunătăți semnificativ timpul de dezvoltare și lizibilitatea proiectelor mari de nivel producție în C++.