Programação Orientada a Objetos

Templates em C++

1. Introdução aos Templates

Templates são uma das principais características que diferenciam C++ de outras linguagens. Eles permitem **programação genérica**, ou seja, você pode escrever funções ou classes que **funcionam para diferentes tipos de dados**, evitando repetição de código. Em vez de criar uma função específica para cada tipo (como `int`, `float` ou `string`), um template possibilita definir a lógica uma vez e aplicá-la a qualquer tipo.

Templates são utilizados tanto para **funções** quanto para **classes** e, a partir de C++11 e C++20, eles foram estendidos para incluir **templates variádicos** e **conceitos**, trazendo ainda mais flexibilidade e controle.

---

2. Vantagens e Desvantagens

Vantagens

- **Reuso de Código:** Você não precisa duplicar o código para cada tipo.
- **Eficiência:** Como o template é resolvido em tempo de compilação, o código gerado é altamente otimizado.
- **Flexibilidade:** Templates podem ser combinados com classes e funções para criar estruturas e algoritmos complexos.

Desvantagens

- **Erros Complexos:** Os erros de compilação envolvendo templates podem ser difíceis de entender e depurar.
- **Aumento do Tempo de Compilação:** Cada vez que um template é instanciado com um novo tipo, o compilador precisa gerar uma nova versão do código.
- **Tamanho do Executável:** Muitos templates podem resultar em crescimento do binário.

---

3. Templates de Função

Templates de função permitem que você crie funções genéricas que podem ser instanciadas para diferentes tipos. Eles são usados para implementar **algoritmos reutilizáveis**, como funções de comparação ou cálculo.

Sintaxe de um Template de Função

```
template <typename T>
T nome_da_funcao(T parametro);
```

A palavra-chave `typename` pode ser substituída por `class`, ambas têm o mesmo significado no contexto de templates.

Exemplo: Função genérica de comparação

```cpp
#include <iostream>
using namespace std;

template <typename T>
T max_value(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << max_value(3, 7) << endl;       // Inteiros
    cout << max_value(3.14, 2.71) << endl; // Doubles
    cout << max_value('a', 'z') << endl;   // Caracteres
    return 0;
}
```

Explicação do Código

1. **Definição do Template:** A função `max_value` é declarada usando o parâmetro de tipo genérico `T`.
2. **Instanciação Automática:** No momento da chamada da função, o compilador infere o tipo de `T` com base nos argumentos fornecidos.
3. **Reuso:** A mesma função é usada para diferentes tipos sem necessidade de duplicação.

Quando usar templates de função?

- Quando o mesmo algoritmo pode ser aplicado a diferentes tipos de dados.
- Para criar **funções utilitárias** (como comparação, ordenação, etc.) reutilizáveis.

---

4. Templates de Classe

Templates de classe permitem definir uma **estrutura de dados genérica** que pode armazenar e operar sobre qualquer tipo de dado. Essa técnica é usada amplamente na STL (Standard Template Library), como em `std::vector`, `std::map` e `std::set`.

Sintaxe de Template de Classe

```cpp
template <typename T>
class NomeDaClasse {
    T atributo;
public:
    NomeDaClasse(T valor) : atributo(valor) {}
    void exibir() const;
};
```

Exemplo: Classe `Pair`

```cpp
#include <iostream>
using namespace std;

template <typename T, typename U>
class Pair {
    T first;
    U second;
public:
    Pair(T f, U s) : first(f), second(s) {}
    void display() const {
        cout << "(" << first << ", " << second << ")" << endl;
    }
};

int main() {
    Pair<int, double> p1(1, 3.14);
    p1.display();

    Pair<string, char> p2("Hello", 'A');
    p2.display();

    return 0;
}
```

Explicação do Código

1. **Dois Parâmetros de Tipo:** `T` e `U` são parâmetros de tipo diferentes, permitindo que a classe `Pair` armazene tipos variados.
2. **Construtor:** O construtor recebe dois valores e os armazena nos atributos.
3. **Display:** A função `display()` imprime os valores armazenados.

Uso comum de templates de classe:

- **Contêineres genéricos** (como `std::vector` e `std::map`).
- **Classes de utilidade** que precisam armazenar diferentes tipos de dados.

---

5. Template Especializado

A **especialização completa** permite definir uma implementação específica para um tipo particular, caso a versão genérica não seja adequada.

Exemplo: Função `square` especializada para `char`

```cpp
#include <iostream>
using namespace std;

template <typename T>
T square(T x) {
    return x * x;
}

// Especialização para caracteres
template <>
char square(char c) {
    cout << "Não faz sentido elevar um char ao quadrado!" << endl;
    return c;
}

int main() {
    cout << square(5) << endl;    // Inteiro
    cout << square(3.14) << endl; // Double
    cout << square('A') << endl;  // Char (especializado)
    return 0;
}
```

Explicação do Código

- **Especialização Completa:** A função `square` foi especializada para o tipo `char`, evitando multiplicação sem sentido.
- **Comportamento Diferenciado:** A versão especializada apenas imprime uma mensagem e retorna o caractere.

---

6. Template Parcialmente Especializado

A **especialização parcial** é útil quando queremos personalizar apenas parte do comportamento, dependendo de alguns dos parâmetros de tipo.

Exemplo: Classe `Container` parcialmente especializada

```cpp
#include <iostream>
using namespace std;

template <typename T1, typename T2>
class Container {
    T1 data1;
    T2 data2;
public:
    Container(T1 d1, T2 d2) : data1(d1), data2(d2) {}
    void display() const {
        cout << data1 << " e " << data2 << endl;
    }
```

```cpp
};

// Especialização parcial para parâmetros iguais
template <typename T>
class Container<T, T> {
    T data;
public:
    Container(T d) : data(d) {}
    void display() const {
        cout << "Valor: " << data << endl;
    }
};

int main() {
    Container<int, double> c1(1, 2.5);
    c1.display();

    Container<int, int> c2(42);
    c2.display();

    return 0;
}
```

Explicação do Código

- **Especialização Parcial:** Quando os dois tipos são iguais (`T, T`), usamos uma implementação diferente.
- **Código mais conciso:** A versão especializada armazena apenas um valor em vez de dois.

---

7. Templates Variádicos

O que são templates variádicos?

Os **templates variádicos** permitem que você crie funções e classes que aceitem **um número indefinido de parâmetros**. Essa funcionalidade foi introduzida no **C++11** e é muito útil quando você não sabe com antecedência quantos argumentos um usuário vai fornecer.

Os templates variádicos usam três pontos (`...`) como **expansor de parâmetros**. Isso pode ser aplicado tanto em funções quanto em classes, permitindo a manipulação de listas de argumentos variáveis.

Como os templates variádicos funcionam?

- Um **template variádico** define um número variável de argumentos com `typename... Args`.
- Em cada etapa da execução, ele expande um argumento e recursivamente chama a si mesmo até que todos os argumentos sejam processados.

Exemplo: Função variádica de impressão

```cpp
#include <iostream>
using namespace std;

template <typename T>
void print(T value) {
    cout << value << endl;
}

template <typename T, typename... Args>
void print(T first, Args... args) {
    cout << first << " ";
    print(args...);
}

int main() {
    print(1, 2.5, "Hello", 'A');  // Saída: 1 2.5 Hello A
    return 0;
}
```

Explicação:

- O template **recursivamente expande** os argumentos até que a lista fique vazia.
- Quando há apenas um argumento, ele chama a função não variádica `print(T value)`.

Exemplo Prático: Somatório com Templates Variádicos

```cpp
template <typename... Args>
auto sum(Args... args) {
    return (args + ...);  // Redução com dobragem à esquerda
}

int main() {
    cout << sum(1, 2, 3, 4, 5) << endl;  // Saída: 15
    return 0;
}
```

Explicação:

- A expressão `(args + ...)` é um exemplo de **fold expression**, introduzida no C++17. Ela aplica a operação `+` em todos os argumentos fornecidos.

---

8. Template Template Parameters

O que são Template Template Parameters?

Templates em C++ podem receber **outros templates como parâmetros**. Isso adiciona um nível extra de **generatividade**, permitindo criar **estruturas de dados compostas** e genéricas. Esse recurso é útil para criar

contêineres complexos, como listas de listas ou vetores de mapas.

Sintaxe de Template Template Parameters

```
template <template <typename, typename> class Container>
void display(Container<int, allocator<int>> c) {
    for (const auto& element : c) {
        cout << element << " ";
    }
    cout << endl;
}
```

Exemplo: Vetor Genérico

```
#include <iostream>
#include <vector>
using namespace std;

template <template <typename, typename> class Container>
void display(Container<int, allocator<int>> c) {
    for (auto& el : c) {
        cout << el << " ";
    }
    cout << endl;
}

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

Explicação:

- A função `display` recebe um **template de contêiner** como argumento e imprime os elementos.

---

9. Conceitos e Constraints (C++20)

O que são conceitos?

Os **conceitos** foram introduzidos no C++20 para **restringir os tipos aceitos por templates**. Antes do C++20, era difícil controlar os tipos que podiam ser usados em templates, levando a erros de compilação complexos.

Um **conceito** é uma restrição lógica para tipos de template. Ele pode ser usado para garantir que certos tipos tenham as características necessárias, como implementar uma função específica ou ser um tipo integral.

Exemplo: Uso de Conceitos

```
#include <iostream>
#include <concepts>
using namespace std;

template <typename T>
concept Integral = std::is_integral_v<T>;

template <Integral T>
T add(T a, T b) {
    return a + b;
}

int main() {
    cout << add(3, 5) << endl;  // Saída: 8
    // cout << add(3.5, 2.1) << endl;  // Erro de compilação
    return 0;
}
```

Explicação:

- `Integral` é um conceito que garante que apenas tipos inteiros sejam aceitos.
- Se um tipo não satisfizer o conceito, ocorre um **erro de compilação**.

Quando usar conceitos?

- Para **validar tipos** de entrada de templates.
- Para melhorar a **leitura e compreensão** do código.
- Para evitar **erros difíceis de depurar**.

---

10. Casos de Uso Reais e Práticos

1. Bibliotecas Matemáticas

- A biblioteca **Eigen** para álgebra linear usa templates para definir vetores e matrizes genéricos.

2. Estruturas de Dados Genéricas

- Contêineres como `std::vector`, `std::map` e `std::set` usam templates para manipular diferentes tipos de dados.

3. Funções Utilitárias

- O **algoritmo de ordenação** `std::sort` é um template genérico que pode ordenar qualquer tipo de dado que suporte comparação.

Exemplo Prático: Uso do `std::vector` com Templates

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};
    for (int n : numbers) {
        cout << n << " ";
    }
    return 0;
}
```

11. Conclusão

Templates são uma das funcionalidades mais importantes do C++. Eles fornecem **reuso de código**, **flexibilidade** e **eficiência** ao permitir a criação de funções e classes genéricas. O uso de **templates variádicos** e **conceitos** expande ainda mais as possibilidades, tornando o C++ uma linguagem altamente poderosa para desenvolvimento moderno.