Templates em C++

Restrições e Conceitos em C++

Introdução

Restrições e conceitos foram introduzidos no C++20 para melhorar a expressividade e a segurança dos templates. Eles permitem especificar requisitos que os parâmetros de template devem satisfazer, tornando os templates mais robustos e fáceis de usar.

1. Definição e Sintaxe

- Definição: Restrições são condições que devem ser satisfeitas pelos parâmetros de template. Conceitos são uma forma de expressar essas restrições.
- Sintaxe:

```
template <typename T>
concept Conceito = requires(T a) {
    { a + a } -> std::convertible_to<int>;
};

template <Conceito T>
void funcao(T valor) {
    // Implementação
}
```

2. Conceitos Básicos

Templates em C++

- Definição: Conceitos são usados para definir restrições nos tipos de parâmetros de template.
- Exemplo:

```
template <typename T>
concept Incrementavel = requires(T a) {
    { a++ } -> std::same_as<T&>;
};

template <Incrementavel T>
void incrementar(T& valor) {
    ++valor;
}

int main() {
    int x = 5;
    incrementar(x);  // Válido
    return 0;
}
```

3. Uso de Conceitos em Funções Template

- Definição: Conceitos podem ser usados diretamente na assinatura de funções template para restringir os tipos aceitos.
- Exemplo:

```
template <typename T>
```

```
concept Multiplicavel = requires(T a, T b) {
    { a * b } -> std::convertible_to<T>;
};

template <Multiplicavel T>
T multiplicar(T a, T b) {
    return a * b;
}

int main() {
    std::cout << multiplicar(3, 4) << std::endl;  // Válido
    return 0;
}
```

4. Restrições com requires

- Definição: A cláusula requires é usada para definir restrições mais complexas nos parâmetros de template.
- Exemplo:

```
template <typename T>
concept Ordenavel = requires(T a, T b) {
    { a < b } -> std::convertible_to<bool>;
};

template <typename T>
```

```
requires Ordenavel<T>
bool menor(T a, T b) {
    return a < b;
}

int main() {
    std::cout << menor(1, 2) << std::endl;  // Válido
    return 0;
}
```

5. Conceitos Compostos

- Definição: Conceitos podem ser combinados para criar restrições compostas.
- Exemplo:

```
template <typename T>
concept Igualavel = requires(T a, T b) {
    { a == b } -> std::convertible_to<bool>;
};

template <typename T>
concept OrdenavelIgualavel = Ordenavel<T> && Igualavel<T>;

template <OrdenavelIgualavel T>
bool menorOuIgual(T a, T b) {
    return a < b || a == b;
```

```
}

int main() {
    std::cout << menorOuIgual(1, 2) << std::endl;  // Válido
    return 0;
}
```

6. Restrições em Classes Template

- Definição: Conceitos podem ser usados para restringir parâmetros de template em classes.
- Exemplo:

```
template <typename T>
concept Somavel = requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;
};

template <Somavel T>
class Calculadora {
public:
    T soma(T a, T b) {
        return a + b;
    }
};

int main() {
```

```
    Calculadora<int> calc;
    std::cout << calc.soma(3, 4) << std::endl;  // Válido
    return 0;
}
```

7. Restrições com std::integral e std::floating_point

- Definição: A biblioteca padrão do C++ inclui conceitos pré-definidos como std::integral e std::floating_point.
- Exemplo:

```
#include <concepts>

template <std::integral T>
T dobro(T valor) {
    return valor * 2;
}

int main() {
    std::cout << dobro(5) << std::endl;  // Válido
    return 0;
}
```

8. Benefícios dos Conceitos

- Definição: Conceitos oferecem vários benefícios, incluindo código mais legível e erros de

compilação mais claros.

- Exemplo:

```
// Sem conceitos, o erro de compilação pode ser confuso:
template <typename T>
T soma(T a, T b) {
    return a + b;
}

int main() {
    soma("Hello, ", "World!");  // Erro de compilação confuso

    // Com conceitos, o erro de compilação é mais claro:
    template <typename T>
    concept Somavel = requires(T a, T b) {
        { a + b } -> std::convertible_to<T>;
    };

    template <Somavel T>
    T soma(T a, T b) {
        return a + b;
    }

    soma("Hello, ", "World!");  // Erro de compilação claro
    return 0;
}
```

Templates em C++

Dicas de Boas Práticas

- Uso Consistente de Conceitos: Utilize conceitos para especificar restrições claras e concisas nos parâmetros de template.
- Clareza e Manutenção: Mantenha os conceitos bem documentados para facilitar a leitura e a manutenção do código.
- Verificação de Restrições: Utilize a cláusula requires para definir restrições complexas e garantir que os parâmetros de template atendam aos requisitos necessários.

Esta seção abrange os conceitos sobre restrições e conceitos em C++. Para mais detalhes, consulte a documentação oficial: https://en.cppreference.com/w/cpp/language/constraints