

Templates

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP



Introdução

- ▶ Os *templates* ou gabaritos fornecem a base para existência da programação genérica
 - ▶ permite desenvolver componentes de software reutilizável: funções, classes, etc.
- ▶ Permite o uso de diferentes tipos utilizando um único *framework*
- ▶ Os *templates* em C++ permitem **criar uma família de funções e classes de templates** para **executar a mesma operação** com **diferentes tipos** de dados.

Introdução (cont.)

- ▶ Existem **várias funções** de grande importância que são **usadas frequentemente** com **diferentes tipos de dados**.
- ▶ A **limitação dessas funções** é que somente conseguem **trabalhar com um único tipo** de dado.
- ▶ O **mecanismo** dos *templates* de C++ **permite que um tipo ou valor seja um parâmetro** na definição de uma classe ou função

Introdução (cont.)

- ▶ Por exemplo, podemos criar uma função genérica que ordene um vetor

Introdução (cont.)

- ▶ Por exemplo, podemos criar uma função genérica que ordene um vetor
 - ▶ **A linguagem** se encarrega de **criar especializações** que tratarão vetores do tipo `int`, `float`, `string`, etc.

Introdução (cont.)

- ▶ Por exemplo, podemos criar uma função genérica que ordene um vetor
 - ▶ **A linguagem** se encarrega de **criar especializações** que tratarão vetores do tipo `int`, `float`, `string`, etc.
- ▶ Podemos também criar uma **classe genérica** para a estrutura de dados **Pilha**

Introdução (cont.)

- ▶ Por exemplo, podemos criar uma função genérica que ordene um vetor
 - ▶ **A linguagem** se encarrega de **criar especializações** que tratarão vetores do tipo `int`, `float`, `string`, etc.
- ▶ Podemos também criar uma **classe genérica** para a estrutura de dados **Pilha**
 - ▶ **A linguagem** se encarrega de **criar as especializações** pilha de `int`, `float`, `string`, etc.

Introdução (cont.)

- ▶ Criar uma função que soma dos números inteiros

Introdução (cont.)

- ▶ Criar uma função que soma dos números inteiros

```
int Soma(int a, int b)
{
    return a + b;
}
```

Introdução (cont.)

- ▶ Criar uma função que soma dos números inteiros

```
int Soma(int a, int b)
{
    return a + b;
}
```

- ▶ Criar uma função que soma dos *double*

Introdução (cont.)

- ▶ Criar uma função que soma dos números inteiros

```
int Soma(int a, int b)
{
    return a + b;
}
```

- ▶ Criar uma função que soma dos *double*

```
double Soma(double a, double b)
{
    return a + b;
}
```

Introdução (cont.)

- ▶ E se forem *float*? Criar outra função?

Introdução (cont.)

- ▶ E se forem *float*? Criar outra função?
- ▶ É possível criar uma única função que receba qualquer tipo de dado?

Introdução (cont.)

- ▶ E se forem *float*? Criar outra função?
- ▶ É possível criar uma única função que receba qualquer tipo de dado?
- ▶ O ideal seria escrever uma função genérica da forma

Introdução (cont.)

- ▶ E se forem *float*? Criar outra função?
- ▶ É possível criar uma única função que receba qualquer tipo de dado?
- ▶ O ideal seria escrever uma função genérica da forma

```
tipo Soma (tipo a, tipo b)
{
    return a + b;
}
```

Introdução (cont.)

- ▶ Funções *template* de C++ funcionam exatamente assim.
- ▶ Através dos *template* é possível criar funções para um ou mais tipos de dados
- ▶ O tipo dos dados (parâmetros e retorno) é definido durante a compilação

Introdução (cont.)

- ▶ Sintaxe:

- ▶ Formato 1

```
template<class Tipo_1, ..., class Tipo_n>  
Tipo_i nome_função (Tipo_i nome)  
{  
    ...  
}
```

- ▶ Formato 2:

```
template<typename Tipo_1, ..., typename Tipo_n>  
Tipo_i nome_função (Tipo_i nome)  
{  
}
```

Introdução (cont.)

- ▶ A função Soma ficaria:

Introdução (cont.)

- A função Soma ficaria:

```
template <class T>
T Soma (T a, T b)
{
    return a + b;
}
int main()
{
    int a = 10, b = 5;
    double c = 3.4, d = 5.6;
    cout << Soma(a, b);
    cout << Soma(c, d);
    return 0;
}
```

Mostraria na tela:

15
9.0

Introdução (cont.)

- ▶ Em alguns casos o *template* precisa receber vários argumentos

Introdução (cont.)

- ▶ Em alguns casos o *template* precisa receber vários argumentos
- ▶ Funciona de forma semelhante que as funções com parâmetros pré-definidos

Introdução (cont.)

- ▶ Em alguns casos o *template* precisa receber vários argumentos
- ▶ Funciona de forma semelhante que as funções com parâmetros pré-definidos

```
template <typename T1, typename T2>
void print(T1 x, T2 y)
{
    cout << x << " " << y << endl;
}
int main()
{
    char letra = 'C';
    // argumentos ( int, char )
    // identificados automaticamente
    print(5,c);
    return 0;
}
```

Exemplo

Criar a função genérica que permita mostrar o conteúdo de um vetor

Exemplo (cont.)

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

//Definição do template de função printArray
template <typename T>
void printArray(const T *array, int tam){
    for (int i = 0; i < tam; i++)
        cout << array[i] << " "
    cout << endl;
}
```


Exemplo (cont.)

```
int main()
{
    const int ACOUNT = 5; // tamanho do array a
    const int BCOUNT = 7; // tamanho do array b
    const int CCOUNT = 6; // tamanho do array c

    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"; // posição 6 para null

    cout << "O vetor a contém:" << endl;
    // chama a especialização da template de função do tipo inteiro
    printArray( a, ACOUNT );
    ...
}
```

Exemplo (cont.)

```
...  
cout << " O vetor b contém:" << endl;  
// chama a especialização da template de função do tipo double  
printArray( b, BCOUNT );  
  
cout << "O vetor c contém:" << endl;  
// chama a especialização da template de função do tipo caractere  
printArray( c, CCOUNT );  
return 0;  
}
```

Exemplo (cont.)

O vetor a contém:

1 2 3 4 5

O vetor b contém:

1.1 2.2 3.3 4.4 5.5 6.6 7.7

O vetor c contém:

H E L L O

Exemplo (cont.)

- ▶ Quando o **compilador detecta a chamada** a *printArray()*, ele **procura a definição da função**
 - ▶ Neste caso, a **função genérica**;
 - ▶ **Ao comparar os tipos** dos parâmetros, **nota** que há **um tipo genérico**;
 - ▶ Então **deduz qual deverá ser a substituição** a ser feita

Exemplo (cont.)

- O compilador cria três especializações

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

Funções Genéricas

- ▶ Todas as ocorrências de T serão substituídas pelo tipo adequado;
- ▶ Note que, se um **genérico é invocado** com parâmetros que são **tipos definidos pelo programador** e o genérico usa **operadores** estes devem **ser sobrecarregados**
 - ▶ e.g., `==`, `+`, `j=`, etc;
 - ▶ Caso contrário, haverá erro de compilação.

Classes *templates*

- ▶ A ideia de funcionamento é semelhante ao de funções *template*.
- ▶ Vários atributos e métodos serão definidos a partir dos tipos genéricos
- ▶ Uma **classe template implementa** o conceito de uma **classe genérica**.

Classes *templates* (cont.)

► Sintaxe:

```
//Declaração da classe template
template<typename Tipo1, typename Tipo2, ...>
class TCRNome
{
    // Declara atributos
    Tipo1 atributo1;
    Tipo2 atributo2;
public:
    //Declara e define construtor default
    TCRNome() { ... };
    //Declara métodos
    Tipo1 NomeMétodo(Tipo1 obj, ...);
    Tipo2 NomeMétodo(Tipo2 obj, ...);
};
```


Classes *templates* (cont.)

//Definição dos métodos da classe template

```
template<typename Tipo1, typename Tipo2, ... >
Tipo1 TCNome<Tipo1, Tipo2, ... >::NomeMétodo( Tipo1 obj, ... )
{
    ...
}
template<typename Tipo1, typename Tipo2, ... >
Tipo2 TCNome<Tipo1, Tipo2, ... >::NomeMétodo( Tipo2 obj, ... )
{
    ...
}
```

//Utilização da classe template

```
TCNome<Tipo1, Tipo2, ... > nomeObjeto;
```

Classes *templates* (cont.)

- ▶ Exemplo: Criar uma classe Pilha (Stack) de *templates*. A classe deverá ter os seguintes métodos:
 - ▶ `push()`: insere um elemento
 - ▶ `pop()`: remove o último elemento
 - ▶ `print()`: implementado com a sobrecarga do operador `<<`
 - ▶ `isFull()`, `isEmpty()`: estado da pilha

Classes *templates* (cont.)

```
template <class T>
class Stack
{
    T* ptr;
    int size , top;
public:
    Stack(int = 10);
    virtual ~Stack();
    bool push(const T&);
    bool pop(T&);
    void destroy();
    bool isEmpty();
    bool isFull();
    template <class T1>
    friend ostream& operator << (ostream&, const Stack<T1>&);
};
```

Classes *templates* (cont.)

```
template<class T>
Stack<T>::Stack(int n) :
    size(n > 10 ? n : 10),
    top(-1){
    ptr = new T[size];
}

template<class T>
Stack<T>::~~Stack(){ destroy(); }

template<class T>
void Stack<T>::destroy(){
    if (ptr != nullptr) {
        delete [] ptr; ptr = nullptr;
    }
    size = 0;
    top = -1;
}
```

Classes *templates* (cont.)

```
template<class T>
bool Stack<T>::isEmpty() {
    return (top == -1);
}
```

```
template<class T>
bool Stack<T>::isFull() {
    return (top == size - 1);
}
```

```
template<class T>
bool Stack<T>::push(const T& value) {
    if (!isFull()) {
        ptr[++top] = value;
        return true;
    }
    return false;
}
```

Classes *templates* (cont.)

```
template<class T>
bool Stack<T>::pop(T& value){
    if (!isEmpty()){
        value = ptr[top--];
        return true;
    }
    return false;
}
```

```
template<class T>
ostream& operator << (ostream& out, const Stack<T>& S){
    for (int i = 0; i <= S.top; i++)
        out << S.ptr[i] << " ";
    out << "\n";
    return out;
}
```

Classes *templates* (cont.)

```
int main(){
    Stack<int> V;
    int valor = 10;
    cout << "Inserindo elementos ... \n";
    while ( V.push(valor) ){
        cout << valor++ << " ";
    }
    cout << "\nPilha cheia. Nao foi possivel inserir : "
        << valor << "\nEsvaziando Pilha \n";
    while (V.pop(valor)){
        cout << valor << " ";
    }
    cout << "\nPilha vazia";
    return 0;
}
```

Templates de Funções Sobrecarregadas

- ▶ Função *template* sobrecarregada

- ▶ Funções templates com o mesmo nome (parâmetros diferentes)

```
printArray(const T * array , int tam)  
printArray(const T * array , int tam, int lowsub)
```

- ▶ Funções que não são *templates* com o mesmo nome (argumentos diferentes)

```
printArray(const T * array , int tam)  
printArray(const char * array , int tam)
```


Templates de Funções Sobrecarregadas (cont.)

- ▶ O compilador realiza o processo de identificação de padrão
- ▶ Tenta achar o mesmo padrão do nome da função e dos tipos de argumentos
- ▶ Compilador procura a função mais próxima da função chamada

Herança

A sintaxe para declarar uma classe derivada a partir de uma classe base *template*

```
template < class T1, ... >
class Base{
    // template de tipos de dados e funções
};

template < class T1, ... >
class Derivada : public Base<T1,...>{
    // template de tipos de dados e funções
};
```

Herança

Criar uma classe genérica *Vetor* com os métodos *push_back()*, *clear()*, *size()* e *at()*. Logo, crie uma segunda classe *SortedVetor* que herde de *Vetor* e que insira os dados de forma ordenada

Herança (cont.)

```
template<typename T>
class Vetor{
protected:
    T* vec;
    int n, top;
public:
    Vetor(int = 5);
    void clear();
    virtual ~Vetor();
    bool push_back(T);
    int size();
    T at(int);
};
```

Herança (cont.)

```
template<typename T>
Vetor<T>::Vetor(int n){
    this->n = (n < 5) ? 5 : n;
    vec = new T[n];
    this->top = -1;
}
```

```
template<typename T>
Vetor<T>::~~Vetor(){
    if (vec != nullptr){
        delete [] vec;
        this->top = -1;
        this->n = 0;
    }
}
```

Herança (cont.)

```
template<typename T>
void Vetor<T>::clear() {
    top = -1;
}

template<typename T>
bool Vetor<T>::push_back(T value) {
    if (top < n) {
        vec[++top] = value;
        return true;
    }
    return false;
}
```

Herança (cont.)

```
template<typename T>
T Vetor<T>::at(int pos){
    if (pos >= 0 && pos <= top){
        return vec[pos--];
    }
    else
        throw out_of_range("Fora dos limites");
}

template<typename T>
int Vetor<T>::size(){
    return top+1;
}
```

Herança (cont.)

```
#include "Vetor.h"

template<typename T>
class SortVetor : public Vetor<T>
{
public:
    SortVetor(int = 5);
    virtual ~SortVetor();
    bool push_back(T);
};
```


Herança (cont.)

```
template<typename T>
SortVetor<T>::SortVetor(int tam) : Vetor<T>(tam) {}
```

```
template<typename T>
SortVetor<T>::~~SortVetor() {}
```

```
template<typename T>
bool SortVetor<T>::push_back(T value) {
    if (Vetor<T>::push_back(value)){
        for (int i = this->top-1; i >= 0 && this->vec[i] > value; i--){
            this->vec[i+1] = this->vec[i];
            this->vec[i] = value;
        }
        return true;
    }
    return false;
}
```

Herança (cont.)

```
int main(){
    SortVetor<int> v(6);
    v.push_back(4);
    v.push_back(1);
    v.push_back(9);
    v.push_back(2);
    v.push_back(40);
    for (int i = 0; i < v.size(); i++){
        cout << v.at(i) << " ";
    }
}
```

1 2 4 9 40

FIM