Programação genérica em C++: Templates

Estrutura de Dados Avançada — QXD0015

Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

Universidade Federal do Ceará

1º semestre/2021

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Introdução

Programação genérica

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Programação genérica consiste na criação de funções ou classes que podem ser usadas com tipos de dados diferentes.
  - Exemplo: funções e classes da STL: vector, stack, queue, list, etc.

Programação genérica

- Programação genérica consiste na criação de funções ou classes que podem ser usadas com tipos de dados diferentes.
  - Exemplo: funções e classes da STL: vector, stack, queue, list, etc.

- Ideia: permitir a criação de um algoritmo ou estrutura de dados com comportamento bem definido, cujos dados possuem tipos genéricos que serão posteriormente substituídos (em tempo de compilação) por tipos de dados especificados pelo usuário.

Programação genérica

- Programação genérica consiste na criação de funções ou classes que podem ser usadas com tipos de dados diferentes.
  - Exemplo: funções e classes da STL: vector, stack, queue, list, etc.

- Ideia: permitir a criação de um algoritmo ou estrutura de dados com comportamento bem definido, cujos dados possuem tipos genéricos que serão posteriormente substituídos (em tempo de compilação) por tipos de dados especificados pelo usuário.
  - Deste modo, uma mesma lógica pode ser reutilizada para tipos de dados diferentes e não relacionados sem, contudo, abrir mão da verificação de tipos.

Programação genérica

- Programação genérica consiste na criação de funções ou classes que podem ser usadas com tipos de dados diferentes.
  - Exemplo: funções e classes da STL: vector, stack, queue, list, etc.

- Ideia: permitir a criação de um algoritmo ou estrutura de dados com comportamento bem definido, cujos dados possuem tipos genéricos que serão posteriormente substituídos (em tempo de compilação) por tipos de dados especificados pelo usuário.
  - Deste modo, uma mesma lógica pode ser reutilizada para tipos de dados diferentes e não relacionados sem, contudo, abrir mão da verificação de tipos.

- Em C++ a programação genérica é implementada por meio de templates (em português, alguns chamam de gabaritos)

Standard Template Library (STL)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Considerando a **utilidade do reuso** de software e também a **utilidade das estruturas de dados e algoritmos** utilizados por programadores, a Standard Template Library (STL) foi adicionada à biblioteca padrão C++;

- A STL define **componentes genéricos reutilizáveis** poderosos que **implementam várias estruturas de dados** e algoritmos que processam estas estruturas

- Basicamente, a STL é composta por **contêineres, iteradores e algoritmos**.

Standard Template Library (STL)

- **Contêineres** são templates de estruturas de dados.
  - Possuem métodos associados a eles.

Standard Template Library (STL)

- **Contêineres** são templates de estruturas de dados.
  - Possuem métodos associados a eles.
- **Iteradores** são objetos semelhantes a ponteiros, utilizados para percorrer e manipular os elementos de um contêiner.

Standard Template Library (STL)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Contêineres** são templates de estruturas de dados.
  - Possuem métodos associados a eles.
- **Iteradores** são objetos semelhantes a ponteiros, utilizados para percorrer e manipular os elementos de um contêiner.
- **Algoritmos** são as funções que realizam operações tais como buscar, ordenar e comparar elementos ou contêineres inteiros.
  - Existem aproximadamente 85 algoritmos implementados na STL.
  - A maioria utiliza iteradores para acessar os elementos de contêineres.

Template de classe std::vector

Exemplo de contêiner — std::vector

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Definido no cabeçalho #include<vector>.
- **Propriedades do contêiner**:
  - Alocação de memória sequencial: elementos armazenados em sequência e acessados pela sua posição.
  - Array dinâmico: permite acesso direto a qualquer elemento. São mais eficientes se inserções e remoções forem feitas apenas no final da sequência. Quando memória estiver esgotada:
    - Aloca maior área sequencial de memória
    - Copia ele mesmo lá
    - Desaloca memória antiga
- Usado quando os dados devem ser ordenados e facilmente acessível.
- Declaração: `std::vector<type> v;`
- Tem iteradores de acesso aleatório

Iteradores

- Objetos que possuem funcionalidade similar a dos ponteiros
  - Apontam para elementos em contêineres
  - Certas operações com iteradores são as mesmas para todos os contêineres
    - `*` desreferência
    - `++` aponta para o próximo elemento
    - `begin()` retorna o iterador do primeiro elemento do contêiner
    - `end()` retorna o iterador do elemento depois do último

std::vector — Iteradores

- std::vector<type>::iterator iterVar;
  - Visita elementos do inicio ao fim.
  - Usa `begin()` para receber ponto de início.
  - Usa `end()` para receber ponto final.

- std::vector<type>::const_iterator iterVar;
  - const_iterator não pode modificar elementos.
  - Usa `cbegin()` para receber ponto de início.
  - Usa `cend()` para receber ponto final.

std::vector — iterator


```
1  #include <iostream> // iterator02.cpp
2  #include <vector>
3
4  int main() {
5      std::vector<int> myvec(5);
6      int i = 0;
7
8      for(auto it = myvec.begin(); it != myvec.end(); ++it)
9          *it = ++i;
10
11     std::cout << "myvec contains: ";
12     for(auto it = myvec.begin(); it != myvec.end(); ++it)
13         std::cout << ' ' << *it;
14     std::cout << std::endl;
15
16     return 0;
17 }
```

O exemplo acima insere os inteiros de 1 a 5 em um `vector<int>` usando um objeto chamado `it` da classe `std::vector<int>::iterator`

std::vector — const_iterator

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream> // iterator03.cpp
2  #include <vector>
3
4  int main() {
5      std::vector<int> mv = {10,20,30,40,50};
6
7      for(auto it = mv.cbegin(); it != mv.cend(); ++it)
8          std::cout << *it << ' ';
9      std::cout << std::endl;
10
11     return 0;
12 }
```

O exemplo acima insere os inteiros de 1 a 5 em um `vector<int>` usando um objeto chamado `it` da classe `std::vector<int>::const_iterator`

std::vector — Iterador Reverso

Universidade Federal do Ceará
Campus Quixadá

Iteradores para acessar elementos em ordem reversa

Iteradores reversos são usados para iterar para trás: incrementá-los move-os em direção ao início do contêiner.

- std::vector<type>::reverse_iterator iterVar;
  - Visita elementos na ordem reversa (fim para o início)
  - Usa `rbegin()` para receber ponto de início na ordem reversa
  - Usa `rend()` para receber ponto final na ordem reversa

std::vector — Iterador Reverso


```
1  #include <iostream> // iterator01.cpp
2  #include <vector>
3
4  int main() {
5      std::vector<int> mv(5);
6      int i = 0;
7
8      for(auto rit = mv.rbegin(); rit != mv.rend(); ++rit)
9          *rit = ++i;
10
11     std::cout << "myvec contains: ";
12     for(auto it = mv.begin(); it != mv.end(); ++it)
13         std::cout << ' ' << *it;
14     std::cout << std::endl;
15
16     return 0;
17 }
```

O exemplo acima insere os inteiros de 1 a 5 em ordem reversa em um `vector<int>` usando um objeto chamado `rit` da classe `std::vector<int>::reverse_iterator`

std::vector — Funções

- `v.push_back(value)`: Adiciona elemento ao final do vector.
- `v.pop_back()`: Remove elemento do final do vector.
- `v.size()`: Devolve número de elementos no vector.
- `v.capacity()`: Devolve a quantidade total de elementos que foi alocada para o vector.
- `v.insert( iterator, value )`: Insere `value` antes do posicionamento do `iterator`.

std::vector — Funções

- `v.erase( iterator )`
  - Remove do vetor o elemento apontado por `iterator`
- `v.erase( iter1, iter2 )`
  - Remove elementos começando do iterador `iter1` até (não incluindo) o iterador `iter2`
- `v.clear()`
  - Apaga todos os elementos em vector, deixando-o vazio.

std::vector — Funções

- `v.front()`, `v.back()`
  - Retorna uma referência para o primeiro e último elemento em std::vector, respectivamente.
- `v[pos]`
  - Retorna uma referência para o elemento na posição `pos`
- `v.at(pos)`
  - Como acima, mas agora checa o intervalo e lança uma exceção caso `pos` não esteja no intervalo válido.

std::vector — Exemplo 1


```cpp
// erasing from vector
#include <iostream> // exemplo01.cpp
#include <vector>

int main() {
  std::vector<int> myvector;

  // set some values (from 1 to 10)
  for (int i = 1; i <= 10; i++) myvector.push_back(i);

  // erase the 6th element
  myvector.erase (myvector.begin()+5);

  // erase the first 3 elements:
  myvector.erase (myvector.begin(),myvector.begin()+3);

  std::cout << "myvector contains:";
  for (unsigned i = 0; i < myvector.size(); ++i)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';

  return 0;
}
```

std::vector — Exemplo 2


```cpp
#include <iostream> // vector01.cpp
#include <vector>

// std::array remember its length
void printLength(const std::vector<float>& arr) {
    std::cout << "The length is: " << arr.size() << '\n';
}

int main() {
    std::vector<float> array { 9.7, 7.1, 5, 3.3, 1.5 };
    printLength(array);

    return 0;
}
```

std::vector — Exemplo 3

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```cpp
// C++ program to illustrate the capacity function in vector
#include <iostream> // vector02.cpp
#include <vector>
using namespace std;

int main() {
    vector<double> g1;

    for (int i = 1; i <= 5; i++) g1.push_back(i*3.2);

    cout << "Size : " << g1.size();
    cout << "\nCapacity : " << g1.capacity();
    cout << "\nMax_Size : " << g1.max_size();
    // resizes the vector size to 4
    g1.resize(4);
    // prints the vector size after resize()
    cout << "\nSize : " << g1.size();
    cout << "\nCapacity : " << g1.capacity();
    cout << "\nVector elements are: ";
    for (auto it = g1.begin(); it != g1.end(); it++)
        cout << *it << " ";

    return 0;
}
```

Tipos de template

Templates

- Os templates fornecem a base para existência da programação genérica.
  - Permite desenvolver componentes de software reutilizável: funções, classes, etc.
  - Permitem escrever funções ou classes que executam as mesmas operações com diferentes tipos de dados.

Templates

- Os templates fornecem a base para existência da programação genérica.
  - Permite desenvolver componentes de software reutilizável: funções, classes, etc.
  - Permitem escrever funções ou classes que executam as mesmas operações com diferentes tipos de dados.
- Existem basicamente dois tipos de templates:
  - Templates de função
  - Templates de classe

Templates

- 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.

Templates

- 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.

Templates de Função

Exemplo

- Criar uma função que soma dois números inteiros.

```
1 int soma(int a, int b)
2 {
3   return a + b;
4 }
```

Exemplo

- Criar uma função que soma dois números inteiros.

```
1 int soma(int a, int b)
2 {
3   return a + b;
4 }
```

- Criar uma função que soma dois double.

```
1 double soma(double a, double b)
2 {
3   return a + b;
4 }
```

Exemplo

- Criar uma função que soma dois números inteiros.

```
1 int soma(int a, int b)
2 {
3   return a + b;
4 }
```

- Criar uma função que soma dois double.

```
1 double soma(double a, double b)
2 {
3   return a + b;
4 }
```

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

Exemplo

- O ideal seria escrever uma função genérica da forma:

```
1 tipo soma(tipo a, tipo b)
2 {
3   return a + b;
4 }
```

Exemplo

- O ideal seria escrever uma função genérica da forma:

```
1 tipo soma(tipo a, tipo b)
2 {
3   return a + b;
4 }
```

- Templates de funções funcionam exatamente assim.
- Através dos templates é possível criar funções para um ou mais tipos de dados.
- O tipo dos dados (parâmetros e retorno) é definido durante a compilação.

Template de função — Sintaxe

- Formato 1:

```
1 template< class Tipo1, class Tipo2, ..., class TipoN >
2 Tipo_i nome_funcao(Tipo_i identificador)
3 {
4   ...
5 }
```

- Formato 2:

```
1 template< typename Tipo1, ..., typename TipoN >
2 Tipo_i nome_funcao(Tipo_i identificador)
3 {
4   ...
5 }
```

Template de função — Sintaxe

- Formato 1:

```
1 template< class Tipo1, class Tipo2, ..., class TipoN >
2 Tipo_i nome_funcao(Tipo_i identificador)
3 {
4   ...
5 }
```

- Formato 2:

```
1 template< typename Tipo1, ..., typename TipoN >
2 Tipo_i nome_funcao(Tipo_i identificador)
3 {
4   ...
5 }
```

- As palavras-chave `class` e `typename` são intercambiáveis na definição de um template. Qualquer uma pode ser utilizada.
- No entanto, dê preferência à `typename`, pois `class` já é utilizada para definir classes.

Template de função — Exemplo 1

A função de soma ficaria da seguinte maneira:

```
1  #include <iostream> // prog02.cpp
2  using namespace std;
3
4  template < typename T >
5  T soma(T a, T b)
6  {
7    return a + b;
8  }
9
10 int main()
11 {
12   int a = 4, b = 6;
13   double c = 4.51, d = 7.623;
14   cout << soma(a, b) << endl;
15   cout << soma(c, d) << endl;
16   return 0;
17 }
```

Template de função — Exemplo 2

Em alguns casos o template precisa receber vários argumentos.
Criar uma função que imprima na tela o conteúdo de um array.

Template de função — Exemplo 2

Em alguns casos o template precisa receber vários argumentos.
Criar uma função que imprima na tela o conteúdo de um array.

```
1  #include <iostream> // prog03.cpp
2
3  template < typename Type >
4  void printArray(const Type array[], int n)
5  {
6    for(int i = 0; i < n; ++i)
7      std::cout << array[i] << " ";
8    std::cout << std::endl;
9  }
10
11 int main()
12 {
13   int ivec[5] = {4, 6, 7, 45, 32};
14   double dvec[4] = {3.4, 6.77, 56.8, 2.1};
15   printArray(ivec, 5);
16   printArray(dvec, 4);
17   return 0;
18 }
```

Template de função — Exemplo 2 (cont.)

Universidade Federal do Ceará
Campus Quixadá

```
1 template < typename Type >
2 void printArray(const Type array[], int n)
3 {
4   for(int i = 0; i < n; ++i)
5     std::cout << array[i] << " ";
6   std::cout << std::endl;
7 }
```

- Quando o compilador detecta a chamada a `printArray()`, ele procura a definição da função.
  - Neste caso, ele encontra 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.

Template de função — Exemplo 2 (cont.)


```
1 template < typename Type >
2 void printArray(const Type array[], int n)
3 {
4   for(int i = 0; i < n; ++i)
5     std::cout << array[i] << " ";
6   std::cout << std::endl;
7 }
```

- O compilador cria duas especializações:
  - `void printArray(const int array[], int n);`
  - `void printArray(const double array[], int n);`

- Todas as ocorrências de `Type` serão substituídas pelo tipo adequado.

Funções genéricas e operadores

- Note que, se uma função genérica é invocada com parâmetros que são tipos definidos pelo programador e a função genérica usa operadores, estes devem ser **sobrecarregados**.

- Caso contrário, haverá erro de compilação.

- Exemplos de operadores que são comumente sobrecarregados:
  - operadores aritméticos: $+$, $-$, $*$, $/$
  - operadores relacionais: $>$, $<$, $>=$, $<=$, $!=$, $==$
  - atribuição: $=$
  - input/output: $>>$ e $<<$
  - $()$, etc.

Funções genéricas e operadores — Exemplo

Queremos ordenar um array do tipo `Pessoa`. Para isto, a classe `Pessoa` deve ter o operador $>$ (maior que) sobrecarregado.

```
1  #include <iostream> //prog05.cpp
2  using namespace std;
3
4  class Pessoa {
5    private:
6      string nome;
7      int idade;
8    public:
9      Pessoa(string n, int i = 0) // Construtor
10       : nome(n), idade(i)
11     { }
12
13     int getIdade() const { return idade; } // getter
14
15     string getNome() const { return nome; } // getter
16
17     bool operator>(const Pessoa& p) const { // operador >
18       return (this->idade > p.idade);
19     }
20
```

Funções genéricas e operadores — Exemplo (cont.)


```
21
22 template < typename T > // template de funcao: bubblesort
23 void bubblesort(T array[], int n) {
24   for(int i = n-1; i > 0; --i)
25     for(int j = 0; j < i; ++j)
26       if(array[j] > array[j+1])
27         std::swap(array[j], array[j+1]);
28 }
29
30 int main() {
31    Pessoa vec[4] = {{"Pedro", 23}, {"Ana", 34},
32                     {"Jose", 12}, {"Luiza", 10}};
33
34    for(int i = 0; i < 4; i++)
35       cout << vec[i].getIdade() << " "
36            << vec[i].getNome() << endl;
37
38    bubblesort(vec, 4);
```

Exercício

Escreva uma função genérica chamada swap que troca os valores de duas variáveis passadas como argumento. Sua função deve ter dois parâmetros do mesmo tipo. Teste a função com variáveis de tipos `int`, `double` e `string`.

Solução 1

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream> // prog07.cpp
2  using namespace std;
3
4  template < typename T > // usando ponteiros
5  void swap(T *a, T *b) {
6    T aux = *a;
7    *a = *b;
8    *b = aux;
9  }
10
11 int main() {
12   int c = 23, d = 77;
13   double e = 1.1, f = 34.5;
14   swap(&c, &d);
15   swap(&e, &f);
16   cout << c << " " << d << endl;
17   cout << e << " " << f << endl;
18   return 0;
19 }
```

Solução 2

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream> // prog07.cpp
2  using std::cout;
3  using std::endl; // std ja tem uma swap que usa referencia
4
5  template < typename T > // usando referencia: &
6  void swap(T& a, T& b) {
7    T aux = a;
8    a = b;
9    b = aux;
10 }
11
12 int main() {
13   int c = 23, d = 77;
14   double e = 1.1, f = 34.5;
15   swap(c, d);
16   swap(e, f);
17   cout << c << " " << d << endl;
18   cout << e << " " << f << endl;
19   return 0;
20 }
```

Templates de Classe

Exemplo – Classe intArray

A classe `intArray` implementada a seguir encapsula um array de inteiros.

```
1  #ifndef INTARRAY_H // intArray.h
2  #define INTARRAY_H
3
4  #include <cassert>
5
6  class IntArray {
7  private:
8      int m_length;
9      int *m_data;
10
11 public:
12     // Construtor
13     IntArray(int length) {
14         assert(length > 0);
15         m_data = new int[length];
16         m_length = length;
17     }
```

Exemplo – Classe intArray


```
18      // Nao queremos permitir que copias
19      // de IntArray sejam criadas
20      IntArray(const IntArray&) = delete;
21      IntArray& operator=(const IntArray&) = delete;
22
23      ~IntArray() { delete[] m_data; }
24
25      void clear() { m_length = 0; }
26
27      int& operator[](int index)
28      {
29          assert(index >= 0 && index < m_length);
30          return m_data[index];
31      }
32
33      int length() const { return m_length; }
34  };
35
36  #endif
```

Exemplo – Classe intArray


```
37      // Nao queremos permitir que copias
38      // de IntArray sejam criadas
39      IntArray(const IntArray&) = delete;
40      IntArray& operator=(const IntArray&) = delete;
41
42      ~IntArray() { delete[] m_data; }
43
44      void clear() { m_length = 0; }
45
46      int& operator[](int index)
47      {
48          assert(index >= 0 && index < m_length);
49          return m_data[index];
50      }
51
52      int length() const { return m_length; }
53  };
54
55  #endif
```

- E se quiséssemos uma classe para armazenar doubles, strings, etc?

Templates de classe — Sintaxe


```
1  /**
2   * Declaracao da classe template
3   */
4  template <typename Tipo1, typename Tipo2, ...>
5  class TCNome
6  {
7  private:
8    // Declara atributos
9    Tipo1 atributo1;
10   Tipo2 atributo2;
11 public:
12   // Declara e define construtor default
13   TCNome() { ... };
14   // Declara funcoes-membro
15   Tipo1 nomeMetodo(Tipo1 obj, ...);
16   Tipo2 nomeMetodo(Tipo2 obj, ...);
17 };
```

Templates de classe — Sintaxe (cont.)


```
1  // Definicao das funcoes-membro da classe template
2  template<typename Tipo1, typename Tipo2, ...>
3  Tipo1 TCNome<Tipo1, Tipo2, ...>::nomeMetodo(Tipo1 obj,...)
4  {
5    ...
6  }
7  template<typename Tipo1, typename Tipo2, ...>
8  Tipo2 TCNome<Tipo1, Tipo2, ...>::nomeMetodo(Tipo2 obj,...)
9  {
10   ...
11 }
12
13 // Exemplo de utilizacao do template de classe na funcao main
14 TCNome<int, double, ...> nomeObjeto;
```

- Atenção: A sintaxe acima se aplica para os métodos implementados fora da classe. Métodos implementados dentro da classe não precisam ser precedidos pela declaração do template e nem pelo operador de resolução de escopo.

Templates de classe — Exemplo 2

- Criar um template de classe da estrutura de dados Array Estático (Array). A classe deverá ter os seguintes métodos:
  - `Array(int n)`: construtor. Instancia array com n elementos.
  - `size()`: retorna número de elementos no array.
  - `operator[](int i)`: retorna elemento na posição i.

Arquivo Array.h


```
1  #ifndef ARRAY_H // Array.h
2  #define ARRAY_H
3
4  #include <cassert>
5
6  template <typename T>
7  class Array {
8  private:
9      int m_length;
10     T *m_data;
11
12 public:
13     // Construtor
14     Array(int length)
15     {
16         assert(length > 0);
17         m_data = new T[length];
18         m_length = length;
19     }
```

Arquivo Array.h (cont.)

Universidade Federal do Ceará
Campus Quixadá

```
20      // Nao queremos permitir que copias
21      // de IntArray sejam criadas
22      Array(const Array&) = delete;
23      Array& operator=(const Array&) = delete;
24
25      ~Array() { delete[] m_data; }
26
27      void clear() { m_length = 0; }
28
29      T& operator[](int index)
30      {
31          assert(index >= 0 && index < m_length);
32          return m_data[index];
33      }
34
35      int length() const;
```

Arquivo `Array.h` (cont.)

Universidade Federal do Ceará
Campus Quixadá

```
36 // Funcao-membro implementada fora da classe
37 template <typename T>
38 int Array<T>::length() const {
39     return m_length;
40 }
41
42 #endif
```

Arquivo main2.cpp

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream>
2  #include "Array.h"
3
4  int main() {
5    Array<int> iArray(7);
6    Array<double> dArray(12);
7
8    for (int i = 0; i < iArray.length(); ++i)
9    {
10     iArray[i] = i;
11     dArray[i] = i + 0.5;
12   }
13
14   for (int j = iArray.length()-1; j >= 0; --j)
15     std::cout << iArray[j] << "\t" << dArray[j] << '\n';
16
17   return 0;
18 }
```

Em que arquivo colocar um template de classe?

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Com classes que não são templates, o procedimento comum é colocar a declaração da classe em um arquivo de cabeçalho (.h) e a implementação de cada função-membro em um arquivo de código (.cpp) com nome semelhante.
  - Dessa maneira, o arquivo .cpp é compilado como um arquivo de projeto separado.
- No entanto, com templates, isso não funciona.
- A declaração e a implementação de uma classe template devem ambos estar no arquivo de cabeçalho (.h)

Em que arquivo colocar um template de classe?

- Para que o compilador use o template de classe, ele deve ver a definição da classe (não apenas a declaração) e o tipo do dado usado para instanciar o template.

- Quando `Array.h` é incluído no `main.cpp`, a declaração da classe é copiada no `main.cpp`. Quando o compilador vê que precisamos das instâncias `Array<int>` e `Array<double>`, ele as instancia e compila como parte do `main.cpp`.

- No entanto, ao compilar `Array.cpp` separadamente, ele esquece que precisamos de um `Array<int>` e um `Array<double>`. Assim, obtemos um link error, porque o compilador não consegue encontrar uma definição para `Array<int>::size()` ou `Array<double>::size()`.

Templates de classe — Exemplo 1

- Criar um template de classe da estrutura de dados Pilha (Stack). A classe deverá ter os seguintes métodos:
  - `push()`: insere um elemento no topo da pilha.
  - `pop()`: remove o elemento do topo.
  - `top()`: retorna o elemento no topo.
  - `empty()`: retorna se pilha está vazia.
  - `full()`: retorna se pilha está cheia.
  - `size()`: retorna número de elementos na pilha.

Arquivo Stack.h

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #ifndef MYSTACK_H
2  #define MYSTACK_H
3
4  template <typename T>
5  class Stack {
6  public:
7      class StackException {}; // Classe de Excecao
8      Stack(int n); // construtor
9      ~Stack(); // destrutor
10     void push(const T& item);
11     void pop();
12     const T& top() const;
13     int size() const;
14     bool empty() const;
15     bool full() const;
16 private:
17     int itop; // indice do elemento no topo
18     int _size; // numero de elementos na pilha
19     T *array; // ponteiro para o array de elementos
20 };
```

Arquivo Stack.h (cont.)


```
21 template <typename T>
22 Stack<T>::Stack(int n)
23     : itop(0), _size(n > 0 ? n : 10)
24 {
25     array = new T[n];
26 }
27
28 template <typename T>
29 void Stack<T>::push(const T& item) {
30     if(full()) throw StackException{};
31     array[itop] = item;
32     ++itop;
33 }
34
35 template <typename T>
36 void Stack<T>::pop() {
37     if(empty()) throw StackException{};
38     --itop;
39 }
```

Arquivo Stack.h (cont.)

Universidade Federal do Ceará
Campus Quixadá

```
40 template <typename T>
41 const T& Stack<T>::top() const {
42     if(empty()) throw StackException{};
43     return array[itop - 1];
44 }
45
46 template <typename T>
47 bool Stack<T>::empty() const {
48     return (itop == 0);
49 }
50
51 template <typename T>
52 bool Stack<T>::full() const {
53     return (itop == _size);
54 }
```

Arquivo Stack.h (cont.)

Universidade Federal do Ceará
Campus Quixadá

```
55 template <typename T>
56 int Stack<T>::size() const {
57     return _size;
58 }
59
60 template <typename T>
61 Stack<T>::~Stack() {
62     delete[] array;
63 }
64
65 #endif
```

Arquivo main.cpp


```
1  #include <iostream>
2  #include "Stack.h"
3
4  int main() {
5      Stack<int> ipilha(15); // instancia pilha de int
6      Stack<double> dpilha(14); // instancia pilha de double
7
8      for(int i = 1; i <= 13; i++) {
9          ipilha.push(i);
10         dpilha.push(i / 33.0);
11     }
12
13     while( !ipilha.empty() ) {
14         std::cout << ipilha.top() << " ";
15         ipilha.pop();
16     }
17     std::cout << std::endl;
18     while( !dpilha.empty() ) {
19         std::cout << dpilha.top() << " ";
20         dpilha.pop();
21     }
22     return 0;
23 }
```

Exercícios

Exercícios

- Reescreva o seu código para árvores AVL e árvores rubro-negras como templates de classe.

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM