# IMD0030 – LINGUAGEM DE PROGRAMAÇÃO I

Aula 13 – Programação genérica: templates de funções e templates de classes.
(material baseado nas notas de aula do Prof. Sílvio Sampaio)

INSTITUTO METRÓPOLE DIGITAL
UFRN UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

# Templates de Funções - Contexto

- Vimos anteriormente que podemos definir várias funções com o mesmo nome, mas com assinaturas diferentes
  - **Sobrecarga de funções**
    - **Polimorfismo ad hoc**
- Veremos agora que também podemos criar funções genéricas, capazes de operar com todos os tipos de variáveis
  - **Templates**
    - **Polimorfismo Paramétrico**
    - Ocorre em tempo de compilação

# Template

- Mecanismo de C++ que permite a definição genérica de funções e de classes através de operações usando qualquer tipo de variável

- Templates permitem montar **esqueletos** de funções e de classes que postergam a definição dos tipos de dados para o momento do uso
- Mecanismo bastante interessante para a linguagem **C++** devido a ela ser **tipada**
  - Tipos de dados sempre devem ser declarados
  - A definição de estruturas genéricas é limitada
- Excelente recurso para a construção de bibliotecas
  - Permite criar código extremamente genérico que pode ser reutilizado por muitos programas

# Template de função

- Define uma função genérica (família de funções sobrecarregadas), independente de tipo
  - Recebe qualquer tipo de dado como parâmetro
  - Retorna qualquer tipo de dado
- Os tipos dos parâmetros são definidos no momento da chamada
- Sintaxe:
  - **Clássico - template < class identificador > funcao**
  - **ISO/ANSI - template < typename identificador > funcao**
- A palavra reservada **template** indica ao compilador que o código a seguir é um modelo (template) de função.

# Template

- Exemplo:
  - Dadas as quatro funções sobrecarregadas a seguir

```
4  char   max( char a,   char b )   { return ( a > b ) ? a : b; }
5  int    max( int a,    int b )    { return ( a > b ) ? a : b; }
6  float  max( float a,  float b )  { return ( a > b ) ? a : b; }
7  double max( double a, double b ) { return ( a > b ) ? a : b; }
8
```

  - Podemos em alternativa criar uma função única com **template**

```
4
5  template < typename Tipo >
6  Tipo max( Tipo a, Tipo b ) { return ( a > b ) ? a : b; }
7
```

# Exemplo

- Cálculo do maior valor do conteúdo de duas variáveis

```
6
7   template < typename T >
8   T maximo( T a, T b )
9   {
10      return ( a > b ) ? a : b;
11  }
12
13  int main()
14  {
15      char   a = maximo( 'a', '1' );       // A passagem do tipo dos argumentos
16      int    b = maximo( 58, 15 );         // é feita implicitamente
17      float  c = maximo( 17.2f, 5.46f );
18      double d = maximo( 25.7, 62.3 );
19
20      // Se quisermos forçar o uso de um tipo específico, podemos explicitá-lo
21      double e = maximo< double >( 41, 52.46 );  // Passagem explícita
22
23      return 0;
24  }
25
```

# Especialização de template de função

- Muitas vezes o comportamento genérico de uma função não é capaz de resolver todos os casos necessários

```
1   #include <iostream>
2
3   template < typename T >
4   T maximo ( T a, T b ) { return ( a > b ) ? a : b; }
5
6   int main()
7   {   // Não funciona corretamente para char[]
8       std::cout << maximo( "C++", "Java" ) << std::endl;
9       return 0;
10  }
11
```

- Logo, devemos especializar o template para garantir o seu funcionamento correto para certos tipos específicos

```
3   // Permite que a função maximo seja aplicada corretamente ao tipo char[]
4   template <>
5   char* maximo< char* >( char* a, char* b ) { return ( strcmp( a, b ) > 0 ) ? a : b; }
```

# Exemplo

- Cálculo do maior valor do conteúdo de duas variáveis

```cpp
#include <iostream>
#include <cstring>

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

template <>
char* maximo< char* >( char* a, char* b ) {
    return ( strcmp( a, b ) > 0 ) ? a : b;
}

int main() {
    std::cout << maximo ( 'a', '1' )      << std::endl;
    std::cout << maximo ( 58, 15 )        << std::endl;
    std::cout << maximo ( 17.2f, 5.46f )  << std::endl;
    std::cout << maximo ( 25.7, 62.3 )    << std::endl;
    char string1[] = "C++", string2[] = "Java";
    std::cout << maximo ( string1, string2 ) << std::endl;
    return 0;
}

```

# Template de função com diversos tipos

- Também é possível criar templates que manipulam mais de um tipo
  - Sintaxe: **template < typename id_1,..., typename id_N > funcao;**
- Exemplo:
  - Divisão de dois números

```
 9  template < typename T, typename U >
10  T divisao( T a, U b ) { return a / b; }
11
12  int main()
13  {
14      double a = divisao( 52.68, 5 );
15      // Geralmente o compilador consegue detectar quais tipos de variáveis usar
16      // Mas caso seja necessário, podemos ajudá-lo indicando os tipos explicitamente
17      double b = divisao< double, int >( 44.18, 10 );
18      cout << b << endl;
19      return 0;
20  }
21
```

# Resumo

- Template é um recurso extremamente poderoso que permite flexibilidade no processo de desenvolvimento de software
  - Melhoria da reusabilidade do código desenvolvido
  - Melhoria substancial da portabilidade e legibilidade do código
  - Maior robustez
  - Menor custo e maior facilidade de manutenção
- Recurso excelente para a construção de bibliotecas de código
  - Um grande número de bibliotecas escritas em linguagem **C++** utilizam templates como base para a sua estrutura

# *Templates* de classes

- Também podemos estender o conceito de criar elementos genéricos com ***templates* também para classes**
- Um *template* de classe pode ser entendido como uma **classe genérica** capaz de utilizar **qualquer tipo de dado**
  - Atributos podem receber qualquer tipo de dado
  - Métodos podem receber, manipular e retornar qualquer tipo de dado
- A principal vantagem de se utilizar *templates* de classe é justamente simplificar a programação por definir uma **família de classes com estrutura semelhante**

# *Templates* de classes

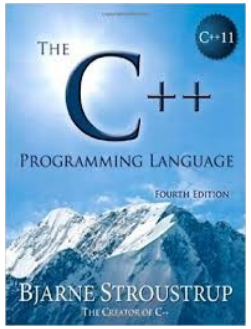
Sintaxe: acréscimo do prefixo de *template* **antes da definição do nome da classe**

```
template<class NomeTipo>
```

Exemplo: classe genérica para representar um par de dois elementos quaisquer

```
template <class T>
class Par {
    private:
        T primeiro;
        T segundo;
    public:
        Par(T a, T b);            // Construtor parametrizado
        T getPrimeiro();          // Retorna o primeiro elemento
        T getSegundo();           // Retorna o segundo elemento
        void setPrimeiro(T v);    // Modifica o primeiro elemento
        void setSegundo(T v);     // Modifica o segundo elemento
};
```

# Implementação em arquivos .h e .cpp distintos




“As usual, the benefits have corresponding weaknesses. For templates, the main problem is that the flexibility and performance come at the cost of **poor separation between the "inside" of a template (its definition) and its interface** (its declaration). When compiling a use of a template, the compiler "looks into" the template and also into the template argument types. It does so to get the information to generate optimal code. To have all the information available, current compilers tend to require that **a template must be fully defined whenever it is used**. That includes all of its member functions and all template functions called from those. Consequently, template writers tend to place template definition in header files. That is not actually required by the standard, but until improved implementations are widely available, we recommend that you do so for your own templates: **place the definition of any template that is to be used in more than one translation unit in a header file**”

# *Templates* de classes

**Não é possível** separar a interface da classe da implementação dos seus métodos em arquivos .h e .cpp distintos

- É necessário implementar toda a informação do *template* em um único arquivo .h
- A implementação dos métodos de um *template* de classe é feita de forma **praticamente idêntica** aos *templates* de função
  - Adicionar o prefixo de *template* de classe **antes da assinatura do método** para que o identificador de tipo torne-se disponível para uso no método
- Pode-se usar o **operador de resolução de escopo (`::`)** para fazer referência aos métodos implementados fora da definição da classe, no mesmo arquivo
  - Adicionar o identificador de tipo **antes do operador de resolução de escopo**

# *Templates* de classes

Exemplo: classe genérica para representar um par de dois elementos quaisquer

```
template<class T>
class Par {
    private:
        T primeiro;
        T segundo;
    public:
        Par(T a, T b);
        T getPrimeiro();
        T getSegundo();
};

template<class T>
Par<T>::Par(T a, T b) {
    primeiro = a;
    segundo = b;
}
```

```
template<class T>
T Par<T>::getPrimeiro() {
    return primeiro;
}

template<class T>
T Par<T>::getSegundo() {
    return segundo;
}
```

# *Templates* de classes

Exemplo: classe genérica para representar um par de dois elementos quaisquer

```
#include <iostream>
using std::cout;
using std::endl;

#include "par.h"

int main() {
    Par<int> p(1, 2);        // Objeto representando um par ordenado de inteiros

    cout << "Primeiro elemento: " << p.getPrimeiro() << endl;
    cout << "Segundo elemento: " << p.getSegundo() << endl;

    return 0;
}
```

# Especialização de *templates* de classes

- Da mesma maneira que **especializamos** *templates* de funções para operar de maneira distinta sob tipos de dados específicos, é possível fazer isso também com *templates* de classes
- A classe especializada deve **redefinir** todos os atributos e métodos definidos no *template* original para o tipo específico
  - Os métodos deixam de usar a forma de *templates* e passam a ser especializados
- Sintaxe:

```
template<>
class NomeClasse<NomeTipo> {
    // Definicao da classe
};
```

# Especialização de *templates* de classes

Exemplo: classe especializada para representar um par do tipo ***string***

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

template<>
class Par<std::string> {
    private:
        std::string primeiro;
        std::string segundo;
    public:
        Par(std::string a, std::string b);
        std::string getPrimeiro();
        std::string getSegundo();
};
```

# Especialização de *templates* de classes

Exemplo: classe especializada para representar um par do tipo ***string***

```
Par<std::string>::Par(std::string a, std::string b) {
    if (a.length() == 0 || b.length() == 0) {
        std::cout << "String vazia" << std::endl;
        exit(1);
    }
    primeiro = a;
    segundo = b;
}

std::string Par<std::string>::getPrimeiro() {
    return primeiro;
}

std::string Par<std::string>::getSegundo() {
    return segundo;
}
```

# Composição de *templates* de classes

- É possível **aninhar** *templates* de classes, isto é, utilizar um *template* de classe dentro da definição de um outro *template* de classe
- O tipo de dados passado para a composição é passado para o *template* de classe que está sendo utilizado internamente

# Composição de *templates* de classes

Exemplo: classe genérica para representar um par nomeado de dois elementos quaisquer

```
#include <string>
using std::string;

#include "par.h"

template<class U>
class ParNomeado {
    private:
        string nome;
        Par<U> elementos;
    public:
        ParNomeado(string n, U a, U b);
        U getPrimeiro();
        U getSegundo();
};
```

```
template<class U>
ParNomeado<U>::ParNomeado(string n, U a, U b) :
    nome(n), elementos(a, b) {
    // Corpo vazio
}

template<class U>
U ParNomeado<U>::getPrimeiro() {
    return elementos.getPrimeiro();
}

template<class U>
U ParNomeado<U>::getSegundo() {
    return elementos.getSegundo();
}
```

# Composição de *templates* de classes

Exemplo: classe genérica para representar um par de dois elementos quaisquer

```
#include <iostream>
using std::cout;
using std::endl;

#include "parnomeado.h"

int main() {
    // Objeto representando um par ordenado
    // de inteiros com nome "Meu par"
    ParNomeado<int> p("Meu par", 1, 2);

    cout << "Primeiro elemento: " << p.getPrimeiro() << endl;
    cout << "Segundo elemento: " << p.getSegundo() << endl;

    return 0;
}
```

os valores **1** e **2** serão passados ao construtor do *template* de classe **`Par`**

# Referência

http://www.cplusplus.com/doc/oldtutorial/templates/

# Exercício

Implemente um algoritmo de ordenação visto em EDB1 utilizando templates (isto é, dada uma coleção de valores de um tipo arbitrário, fornecidos pelo usuário, seu programa deve ordená-la e retorná-la ao usuário).