

# IMD0030 – LINGUAGEM DE PROGRAMAÇÃO I

Aula 15 – Programação genérica: templates de funções e templates de classes.

---

# Objetivos da Aula (1)

- Introduzir os conceitos de ponteiro genérico e ponteiro de função
- Introduzir os conceitos de *templates* de funções em **C++**
- Para isso, estudaremos:
  - Uso de parâmetros genéricos em funções
  - Algoritmo da busca binária polimórfica
  - Tipos genéricos
  - *Template*
- Ao final da aula espera-se que o aluno seja capaz de:
  - Criar e usar variáveis de tipo genérico
  - Implementar funções polimórficas em **C**
  - Implementar diferentes tipos de funções genéricas utilizando *templates* em **C++**
  - Identificar o uso de *templates* de função em **C++**

---

## Objetivos da Aula (2)

- Introduzir o conceito de *templates* de classes na linguagem de programação C++
- Para isso, estudaremos:
  - A ideia de tipos genéricos
  - Como implementar *templates* de classes na linguagem de programação C++
- Ao final da aula, espera-se que o aluno seja capaz de:
  - Compreender o conceito de *templates* de classes e como fazer uso dele
  - Implementar *templates* de classes na linguagem de programação C++

# Parte I

Ponteiro genérico e ponteiro de função

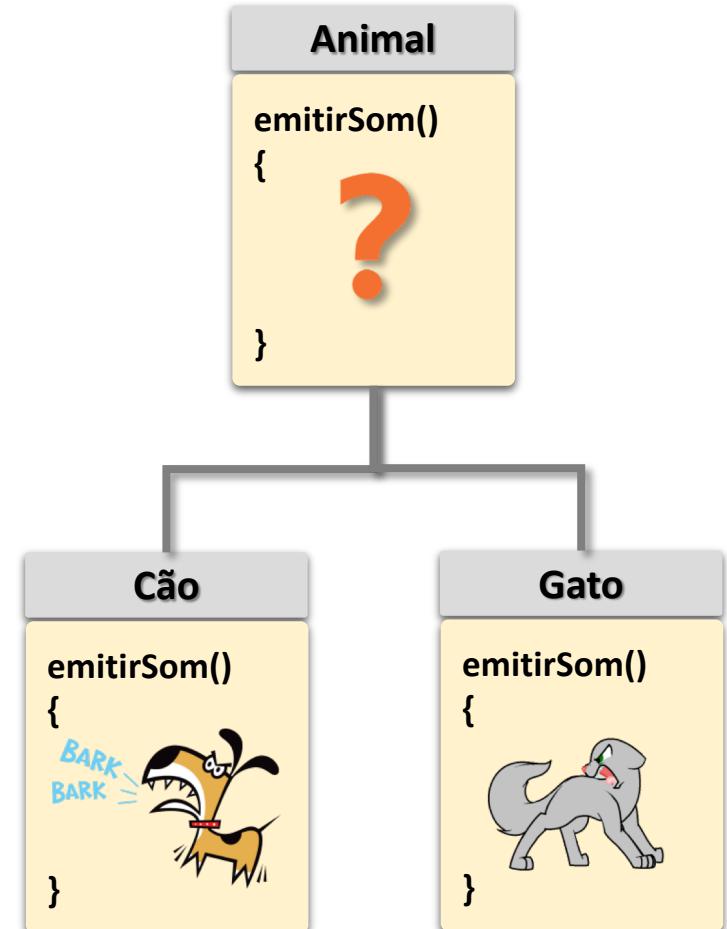
---

# A importância do tipo em ponteiros

- Quando declaramos
  - `int *p`, obtemos um endereço de memória...
  - `float *p`, obtemos um endereço de memória...
  - `struct Data *p`, obtemos um endereço de memória...
- Se tudo é endereço de memória, por que precisamos declarar o tipo de dado que estará armazenado nesse endereço?
  - Útil para "desreferenciar" o endereço (acessar o conteúdo)
    - É necessário saber o formato correto do conteúdo armazenado
  - Útil para realizar aritmética de ponteiros (ex: `p++`, `p--`)
    - É necessário saber o tamanho do dado para realizar o deslocamento necessário
- Mas quando seria útil usar o ponteiro genérico (`void*`) que permite apontar para "qualquer tipo"?

# Ponteiro genérico

- Ponteiro genérico (**void\***) é útil como artifício para permitir polimorfismo em linguagem **C++**
  - Polimorfismo permite que diferentes tipos sejam tratados da mesma forma
- Exemplo:
  - O algoritmo da busca binária que implementamos para o tipo **int** na aula de recursividade é o mesmo para outros tipos...
  - Se precisarmos fazer uma busca para os tipos **float**, **char**, **struct Cao** e **struct Gato**, teremos que implementar uma função para cada tipo?



---

# Polimorfismo

- **C** não possui uma forma explícita para fazer polimorfismo
  - Exceto se usarmos uma **união** (**union** visto em ITP) ou um **ponteiro genérico** (**void\***)
- **C++** possui diversas formas de polimorfismo
  - **Ad hoc** (com sobrecarga de funções ou operadores)
  - **Paramétrico** (com o uso de templates, generics e similares)
  - **De subtipos** (com orientação a objetos, através de subclasses)

---

# Solução com ponteiro genérico

- Uma solução para a busca binária seria receber os parâmetros do **arranjo** e do **valor a ser procurado** como **void\***
- Mas, se declararmos a seguinte assinatura para a busca binária...
  - `int binarySearch( void *array, void *value, int tamanho )`
- Como poderemos saber para qual tipo converter (**cast**) os parâmetros de modo a permitir a correta comparação de seus valores na busca binária?
  - 👎 ○ **Solução 1:** passar **mais um parâmetro** para especificar o tipo e tratá-lo com um grande switch-case (um case para cada tipo)
    - Isso seria o equivalente a escrever várias sub-rotinas
  - 👍 ○ **Solução 2:** passar **uma função** para comparar os valores

# Passar uma função como parâmetro?!

- Ideia da solução

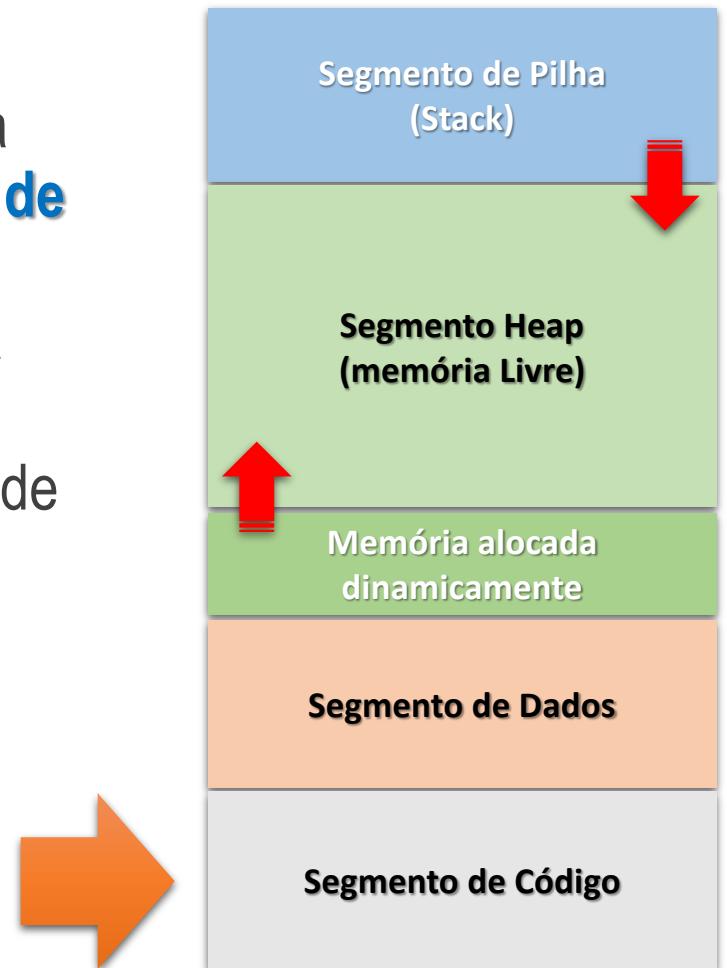
Ponteiro para uma função que irá realizar a comparação

Comparando a chave com o elemento do arranjo, usando a função passada por parâmetro

```
6   int binarySearch( TIPO array[], TIPO value, int low, int high, FUNCTION compare )
7   {
8       if( low > high )                                     // não há elementos em array
9           return -1;
10
11      int mid  = ( low + high ) / 2;                      // índice do meio
12
13      int comp = compare( value, array[mid] );             // compara os elementos
14
15      if( comp == 0 )                                      // são iguais --> achou
16          return mid;
17
18      else if( comp > 0 ) // comp > 0 --> value > array[mid] --> busca a direita
19          return binarySearch( array, value, mid + 1, high, comp );
20
21      else // value < array[mid] --> busca a esquerda
22          return binarySearch( array, value, low, mid - 1, comp );
23
24 }
```

# Tipo: ponteiro de função

- Como visto anteriormente, o código de máquina um programa encontra-se em uma região da memória chamada **segmento de código**
  - Logo, sempre existe um **endereço de memória** para o início de cada função presente no programa
- O ponteiro que aponta para o endereço de memória do início de uma função chama-se **ponteiro de função**
- Pode-se:
  - passar um ponteiro de função para outra função
  - usar um ponteiro de função como um campo de um registro
  - ou... usá-lo como se usa qualquer outro tipo de dado!!!



# Declarando variável ponteiro de função

- Sintaxe: **TIPO (\*nome\_da\_variavel) ( PARÂMETROS )**
  - Exemplo: Função de impressão

```
1 #include <iostream>
2 int max( int a, int b ) { return a > b ? a : b; }
3 int min( int a, int b ) { return a < b ? a : b; }
4
5 // ponteiros de função são também passados como parâmetro (ou retorno)
6 void imprime( int a, int b, int (*funcao) ( int, int ) )
7 {
8     std::cout << funcao( a, b ) << std::endl;
9 }
10 int main() // declara uma variável (ponteiroFuncao) ponteiro de função que recebe
11 {           // 2 inteiros como parâmetros e retorna 1 inteiro como resultado */
12
13     int (*ponteiroFuncao) ( int, int );
14     ponteiroFuncao = max;           // ponteiroFuncao armazena o endereço de max
15
16     imprime( 5, 9, ponteiroFuncao ); // --> imprime 9
17     imprime( 5, 9, min );         // --> imprime 5
18     return 0;
19 }
20
```

# Usando `typedef` para simplificar

- O uso de ponteiros de função pode se tornar bastante confuso
- É aconselhável usar **`typedef`** para simplificar o código!

```
1 #include <iostream>
2 int max( int a, int b ) { return a > b ? a : b; }
3 int min( int a, int b ) { return a < b ? a : b; }
4
5 /* define um tipo (PonteiroFuncao) que permite apontar para funções que
6    recebem 2 inteiros e retornam 1 */
7 typedef int (*PonteiroFuncao)( int a, int b );
8
9 // usa o tipo para definir o parâmetro funcao
10 void imprime( int a, int b, PonteiroFuncao funcao )
11 {
12     std::cout << funcao( a, b ) << std::endl;
13 }
14
```

- Mas, e o **polimorfismo**? Os parâmetros de **`funcao`** (a variável) não mudam!
  - Solução: usar o ponteiro genérico (**`void*`**) nos parâmetros

# Solução para a busca binária (1)

- Definir funções com a mesma assinatura (e um **typedef** para elas)
- Usar **void\*** nas assinaturas para representar um tipo qualquer
- Em cada função, desreferenciar o conteúdo para o "seu tipo"

```
6 // compara dois valores inteiros
7 int comparaInt( void *a, void *b )
8 {
9     int va = *( (int*) a );
10    int vb = *( (int*) b );
11    return va - vb; // 0 se va = vb, neg. se va < vb ou pos. se va > vb
12 }
13
14 // compara duas pessoas em função da idade
15 int comparaPessoa( void *a, void *b )
16 {
17     Pessoa va = *( (Pessoa*) a );
18     Pessoa vb = *( (Pessoa*) b );
19     return va.idade - vb.idade;
20 }
21
22 typedef int (*CompareFunc) (void*, void*); // tipo para a função de comparação
23
```

# Solução para a busca binária (2)

- A busca polimórfica

```
6  int binarySearch( void *array, void *value, int low, int high,
7                      CompareFunc compare, int size )
8  {
9      if( low > high )
10         return -1;
11
12     int mid = ( low + high ) / 2;
13
14     // é necessário incluir o parâmetro size para poder acessar um
15     // elemento do arranjo sem saber seu tipo
16
17     void *valueArray = array + size * mid; // equivalente a array[mid]
18     int comp = compare( value, valueArray );
19
20     if( comp == 0 )
21         return mid;
22     else if( comp > 0 )
23         return binarySearch( array, value, mid + 1, high, compare, size );
24     else
25         return binarySearch( array, value, low, mid - 1, compare, size );
26 }
27
```

# Solução para a busca binária (3)

- Podemos, então, passar as diferentes funções de comparação

```
1 #include <iostream>
2 ...
3 int binarySearch( void *array, void *value, int low, int high,
4                   CompareFunc compare, int size ) { ... }
5 ...
6 int main()
7 {
8     int intArray[] = { 1, 4, 7, 8, 10, 15 };
9     int intValue = 10;
10
11    Pessoa pessoaArray[] = { { "A", 21 }, { "B", 28 }, { "C", 30 } };
12    Pessoa pessoaValue = { "C", 30 };
13
14    if( binarySearch( intArray, &intValue, 0, 5, comparaInt, sizeof( int ) ) >= 0 )
15        std::cout << "Inteiro encontrado!" << std::endl;
16
17    if( binarySearch( pessoaArray, &pessoaValue, 0, 2, comparaPessoa, sizeof( Pessoa ) ) >= 0 )
18        std::cout << "Pessoa encontrada!" << std::endl;
19
20    return 0;
21 }
22
```

---

# Resumo da Parte I

- Polimorfismo é útil para reaproveitar código, definindo funções (ou estruturas) que tratem diferentes tipos de dados
- Ponteiro genérico (**void\***) permite simular polimorfismo em **C**
- Ponteiro de função é útil em diversas utilidades:
  - Programação funcional
    - Funções como elementos de 1<sup>a</sup> classe
  - Programação orientada a eventos (**callbacks**)
    - Passa-se uma função a um serviço (ou módulo), associando-a a um evento
    - Quando o evento ocorre, a função é chamada (callback)
    - Técnica muito usada em interfaces com o usuário (UI)

# Parte II

## 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**
    - Ocorre em tempo de execução
- 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
  - 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 |
```

---

# Template

- 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 sobre carregadas), 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.

# Exemplo

- Cálculo do maior valor do conteúdo de duas variáveis

```
6  template < typename T >
7  T maximo( T a, T b )
8  {
9      return ( a > b ) ? a : b;
10 }
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 {
8     // Não funciona corretamente para char[]
9     std::cout << maximo( "C++", "Java" ) << std::endl;
10    return 0;
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

```
1 #include <iostream>
2 #include <cstring>
3
4 template < typename T >
5 T maximo( T a, T b ) {
6     return ( a > b ) ? a : b;
7 }
8
9 template <>
10 char* maximo< char*>( char* a, char* b ) {
11     return ( strcmp( a, b ) > 0 ) ? a : b;
12 }
13
14 int main() {
15     std::cout << maximo ( 'a', '1' )      << std::endl;
16     std::cout << maximo ( 58, 15 )       << std::endl;
17     std::cout << maximo ( 17.2f, 5.46f )  << std::endl;
18     std::cout << maximo ( 25.7, 62.3 )   << std::endl;
19     char string1[] = "C++", string2[] = "Java";
20     std::cout << maximo ( string1, string2 ) << std::endl;
21     return 0;
22 }
23
```

# 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
20 }
21
```

---

# Resumo da Parte II

- 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

# Parte III

## Templates de Classes

---

# Templates de classes

- Também podemos estender o conceito de criar elementos genéricos com **templates também para classes**
- Um *template* de classe podem 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
};
```

---

# 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 os métodos **no mesmo arquivo** que contém a definição da classe
- 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

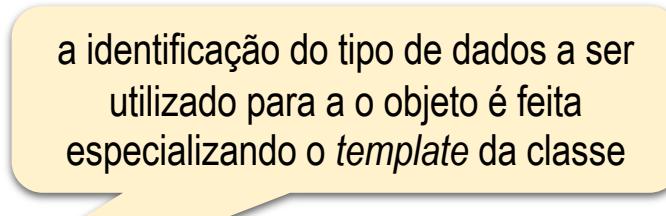
```
#include <iostream>
using std::cout;
using std::endl;

#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;
}
```



a identificação do tipo de dados a ser  
utilizado para o objeto é feita  
especializando o *template* da classe

---

# 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 <cstdlib>
#include <iostream>
#include <string>

template<>
class Par<string> {
    private:
        std::string primeiro;
        std::string segundo;
    public:
        Par(std::string a, std::string b);
        string getPrimeiro();
        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::cerr << "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>::Par(string n, U a, U b) :
    nome(n), elementos(a, b) {
    // Corpo vazio
}

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

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

# 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**

# Alguma Questão?

