Linguagem de Programação I

Aula 11 - Templates de funções

Objetivos da aula

- **Introduzir os conceitos de templates de funções em C++**
- Para isto estudaremos:
  - Tipos genéricos
  - Template
- Ao final da aula espera-se que o aluno seja capaz de:
  - Implementar diferentes tipos de funções genéricas utilizando templates em C++
  - Identificar o uso de templates e sobrecarga de função em C++

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

```
1. char   max( char a,   char b )   { return ( a > b ) ? a : b; }
2. int    max( int a,    int b )    { return ( a > b ) ? a : b; }
3. float  max( float a,  float b )  { return ( a > b ) ? a : b; }
4. double max( double a, double b ) { return ( a > b ) ? a : b; }
```

  - Podemos em alternativa criar uma função única com template

```
1. template < class Tipo >
2. Tipo max( Tipo a, Tipo b ) { return ( a > b ) ? a : b; }
```

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 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
- Sintaxes:
  - `template < class identificador > funcao;`
  - `template < typename identificador > funcao;`
- Exemplo:

```
1. template < class Tipo >
2. Tipo max( Tipo a, Tipo b ) { return ( a > b ) ? a : b; }
```

Exemplo

- Cálculo do maior valor do conteúdo de duas variáveis

```
1.  template < class T >
2.  T max( T a, T b )
3.  {
4.      return ( a > b ) ? a : b;
5.  }
6.
7.  int main()
8.  {
9.      char   a = max( 'a', '1' );        // A passagem do tipo dos argumentos
10.     int    b = max( 58, 15 );          // é feita implicitamente
11.     float  c = max( 17.2f, 5.46f );
12.     double d = max( 25.7, 62.3 );
13.
14.     // Se quisermos forçar o uso de um tipo específico, podemos explicitá-lo
15.     double e = max< double >( 41, 52.46 );  // Passagem explícita
16.
17.     return 0;
18. }
```

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 < class T >
4.  T max( T a, T b ) { return ( a > b ) ? a : b; }
5.
6.  int main()
7.  {   // Não funciona corretamente para char[]
8.      std::cout << max( "C++", "Java" ) << std::endl;
9.      return 0;
10. }
```

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

```
1.  // Permite que a função max seja aplicada corretamente ao tipo char[]
2.  template <>
3.  char* max< 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 < class T >
5.  T max( T a, T b ) {
6.      return ( a > b ) ? a : b;
7.  }
8.
9.  template <>
10. char* max< char* >( char* a, char* b ) {
11.     return ( strcmp( a, b ) > 0 ) ? a : b;
12. }
13.
14. int main() {
15.     std::cout << max( 'a', '1' )       << std::endl;
16.     std::cout << max( 58, 15 )         << std::endl;
17.     std::cout << max( 17.2f, 5.46f )   << std::endl;
18.     std::cout << max( 25.7, 62.3 )     << std::endl;
19.     char string1[] = "C++", string2[] = "Java";
20.     std::cout << max( string1, string2 ) << std::endl;
21.     return 0;
22. }
```

Template de função com diversos tipos

- Também é possível criar templates que manipulam diversos tipos
- Sintaxe:
  - `template < class id_1,..., class id_N > funcao;`
- Exemplo:
  - Divisão de dois números

```
1.  template < class T, class U >
2.  T divisao( T a, U b ) { return a / b; }
3.
4.  int main()
5.  {
6.      double a = divisao( 52.68, 5 );
7.
8.      // Geralmente o compilador consegue detectar quais tipos de variáveis usar
9.      // Mas caso seja necessário, podemos ajudá-lo indicando os tipos explicitamente
10.     int b = divisao< double, int >( 44.18, 10 );
11.     return 0;
12. }
```

Revisitando a busca binária

Uso de template de função para tratar diferentes tipos de dados: **Polimorfismo Paramétrico**

```
1.
2.
3.
4.  template <typename Type>
5.  int binarySearch( Type array[], Type value, int low, int high )
6.  {
7.      if( low > high )                          // não há elementos em array
8.          return -1;
9.
10.     int mid  = ( low + high ) / 2;            // índice do meio
11.     int comp = compara( value, array[mid] );  // compara os elementos
12.
13.     if( comp == 0 )                           // são iguais --> achou
14.         return mid;
15.
16.     else if( comp > 0 ) // comp > 0 --> value > array[mid] --> busca a direita
17.         return binarySearch( array, value, mid + 1, high, comp );
18.
19.     else // value < array[mid] --> busca a esquerda
20.         return binarySearch( array, value, low, mid - 1, comp );
21. }
```

Revisitando a busca binária

```
1.  int compara( int a,    int b )    { return a - b; }
2.  int compara( Pessoa a, Pessoa b ) { return a.idade - b.idade; }
3.
4.
5.  Uso de sobrecarga de função para poder tratar diferentes tipos de função de comparação:
6.
7.  Polimorfismo Ad hoc
8.          return -1;
9.
10.     int mid  = ( low + high ) / 2;            // índice do meio
11.     int comp = compara( value, array[mid] ); // compara os elementos
12.
13.     if( comp == 0 )                           // são iguais --> achou
14.         return mid;
15.
16.     else if( comp > 0 ) // comp > 0 --> value > array[mid] --> busca a direita
17.         return binarySearch( array, value, mid + 1, high, comp );
18.
19.     else // value < array[mid] --> busca a esquerda
20.         return binarySearch( array, value, low, mid - 1, comp );
21. }
```

Revisitando a busca binária

- Podemos passar diversos tipos de argumentos como parâmetro, deixando a identificação da função de comparação para o compilador

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

Resumo da aula

- 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