Aula 03:

Introdução ao C++ Parte III

ECOP13A - Programação Orientada a Objetos
Prof. André Bernardi
andrebernardi@unifei.edu.br

Universidade Federal de Itajubá

Templates

Um dos recursos de reutilização de software mais poderosos do C++.

Templates de **funções** possibilitam a especificação, através de um único trecho de código, uma gama inteira de **funções relacionadas entre si**, que diferem apenas por detalhes de tipo.

Essa técnica é chamada de Programação **Genérica**.

Templates podem ser também chamados, em alguns livros, de "**gabaritos**". Depende da tradução!

Templates

A partir desta técnica, podemos, por exemplo, escrever um único ***template*** para uma função de ordenação de vetores, que ordene vetores de quaisquer tipos de dados: vetores de int, de float, de strings, etc.

Tudo a partir de **uma única especificação**.

Templates de funções

Quando uma mesma operação é realizada de forma idêntica para vários tipos de dados, ao invés de sobrecarregá-la uma vez para cada tipo, podemos simplesmente definir uma **função genérica** (ou "**template**"), que aceite qualquer tipo de dados para tratamento.

> É baseado nos tipos de argumentos passados para a função (explicitamente ou através de chamada de outra função) que o compilador **gera** funções separadas no código-fonte. Chamamos essas funções geradas pelo compilador de **especializações de template de função**.

Vejamos alguns exemplos de sintaxe:

```
// declara que função será template
template <typename T>
void print_vector(T *v, int sz)
{
    for(int i = 0; i < sz; i++)
        cout << v[i] << " ";
}

// A, assim como T na função anterior, representa qualquer
// tipo de dados que se queira utilizar
template <typename A>
void sort_vector(A vetor[], int tam)
{
    A aux;
    for(int i = 0; i < tam; i++)
        for(int j = 0; j < tam - 1 - i; j++)
            if (vetor[j+1] < vetor[j])
            {
                aux = vetor[j];
                vetor[j] = vetor[j+1];
                vetor[j+1] = aux;
            }
}
```

Definições de templates de função começam sempre com a palavra reservada **template**, seguida por uma lista de **parâmetros de template**.

Os parâmetros (pode haver mais de um) de template são incluídos entre < >, separados por vírgulas e cada um deve ser precedido da palavra **typename** ou **class**.

Estes parâmetros podem ser utilizados para especificar:

- Tipos de argumentos da função;
- Tipos de retorno;
- Tipos de variáveis locais;

```cpp
int main()
{
    // Chamando as funções print_vector e sort_vector <float>
    float vf[10];
    for(int i = 0; i < 10; i++)
        vf[i] = (float)(rand() % 100)/100.0;

    cout << "Vetor de float desordenado: ";
    print_vector(vf, 10);
    cout << endl;
    sort_vector(vf, 10);
    cout << "Vetor de float ordenado: ";
    print_vector(vf, 10);
    cout << endl;

    // Chamando as funções print_vector e sort_vector <int>
    int vi[10];
    for(int i = 0; i < 10; i++)
        vi[i] = rand() % 100;

    cout << "Vetor de int desordenado: ";
    print_vector(vi, 10);
    cout << endl;
    sort_vector(vi, 10);
    cout << "Vetor de int ordenado: ";
    print_vector(vi, 10);
    cout << endl;
}
```

```
Vetor de float desordenado: 0.41 0.67 0.34 0 0.69 0.24 0.78 0.58 0.62 0.64
Vetor de float ordenado: 0 0.24 0.34 0.41 0.58 0.62 0.64 0.67 0.69 0.78
Vetor de int desordenado: 5 45 81 27 61 91 95 42 27 36
Vetor de int ordenado: 5 27 27 36 42 45 61 81 91 95
```

As definições das funções se parecem com a definição de uma função comum, mas podem utilizar os tipos definidos na lista de parâmetros de template.

Na main, pouca coisa muda. No entanto, repare que chamamos uma mesma função com parâmetros de tipos diferentes. E só implementamos uma de cada!

Exemplo 2:

```
1  // Fig. 15.12: maximum.h
2  // Function template maximum header file.
3
4  template < typename T > // or template< class T >
5  T maximum( T value1, T value2, T value3 )
6  {
7     T maximumValue = value1;     // assume value1 is maximum
8
9     // determine whether value2 is greater than maximumValue
10    if ( value2 > maximumValue )
11       maximumValue = value2;
12
13    // determine whether value3 is greater than maximumValue
14    if ( value3 > maximumValue )
15       maximumValue = value3;
16
17    return maximumValue;
18 }
```

Fig. 15.12 | Function template maximum header file.

Exemplo 2:

```
1 // Fig. 15.13: fig15_13.cpp
2 // Demonstrating function template maximum.
3 #include <iostream>
4 using namespace std;
5
6 #include "maximum.h" // include definition of function template maximum
7
8 int main()
9 {
10     // demonstrate maximum with int values
11     int int1, int2, int3;
12
13     cout << "Input three integer values: ";
14     cin >> int1 >> int2 >> int3;
15
16     // invoke int version of maximum
17     cout << "The maximum integer value is: "
18          << maximum( int1, int2, int3 );
19
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

Fig. 15.13 | Demonstrating function template `maximum`.

Fig. 15.13 | Demonstrating function template maximum.. **(parte1)**

Exemplo 2:

```
20      // demonstrate maximum with double values
21      double double1, double2, double3;
22
23      cout << "\n\nInput three double values: ";
24      cin >> double1 >> double2 >> double3;
25
26      // invoke double version of maximum
27      cout << "The maximum double value is: "
28           << maximum( double1, double2, double3 );
29
30      // demonstrate maximum with char values
31      char char1, char2, char3;
32
33      cout << "\n\nInput three characters: ";
34      cin >> char1 >> char2 >> char3;
35
36      // invoke char version of maximum
37      cout << "The maximum character value is: "
38           << maximum( char1, char2, char3 ) << endl;
39  }
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

Fig. 15.13 | Demonstrating function template `maximum`.

Fig. 15.13 | Demonstrating function template maximum. **(parte2)**

Resumindo...

A definição da função começa com a palavra **template**, seguida da lista de parâmetros de template, entre colchetes angulares (< e >).

```
// função retorna um T e recebe um parâmetro T e um A
template< typename T, typename A>
T example_function(T par1, A par2) {...}
```

A utilização não tem nada diferente do habitual, assim como a própria definição da função, que, no exemplo, contará com dois tipos “genéricos” diferentes: A e T.

Importante:

Caso o template seja chamado com um tipo definido pelo usuário (uma classe), e este template utilize operadores (por exemplo ==, <, >, <<, etc.) com objetos do tipo desta classe, é necessário que os operadores estejam sobrecarregados para este tipo. Caso não estejam, será gerado um erro de compilação.

Exemplo: No caso da impressão, precisaríamos sobrecarregar o operador **<<**. Para a ordenação, o operador **<** de comparação.

Além disso, lembre-se de que, apesar de escrever apenas uma vez, será gerada, no tempo de compilação, **uma versão para cada tipo utilizado**. Cada especialização do template será independente e ocupará seu próprio espaço na memória.

Isso é um problema?
Não! Você teria que escrever o código de qualquer maneira.

Introdução a classe *vector* da biblioteca padrão C++

A classe vector da biblioteca padrão C++ representa uma alternativa mais robusta para *arrays* com muitos recursos que não são fornecidos para *arrays* baseadas em ponteiros no estilo C.

Por exemplo:

- um programa pode facilmente "sair" de qualquer extremidade de um array, porque nem C nem C++ verificam se os índices estão fora do intervalo de um array.
- Dois arrays não podem ser comparados significativamente com operadores de igualdade ou operadores relacionais. Nomes de arrays são simplesmente ponteiros para onde os arrays começam na memória e, é claro, dois arrays diferentes sempre estarão em locais de memória diferentes.

Introdução a classe *vector* da biblioteca padrão C++

- Quando um array é passado para uma função de propósito geral projetada para lidar com arrays de qualquer tamanho, o tamanho do array deve ser passado como um argumento adicional.
- Além disso, um array não pode ser atribuído a outro com o(s) operador(es) de atribuição — nomes de array são ponteiros *const*, então eles não podem ser usados no lado esquerdo de um operador de atribuição.

Para utilizarmos tais recursos que parecem naturais para lidar com arrays, C++ fornece a classe ***vector***.

Introdução a classe *vector* da biblioteca padrão C++

O programa da Fig. 15.14, a seguir, demonstra recursos fornecidos por ela:

- Um vetor pode ser definido para armazenar qualquer tipo de dados usando uma declaração do formato:

  `vector<tipo> nome(tamanho);`

- Após a criação, por padrão, todos os elementos de um objeto **`vector<int>`** são definidos como 0.
- A função membro **`size()`** da classe **`vector`** retorna o número de elementos no vetor no qual é invocado.
- O valor de um elemento de um vetor pode ser acessado ou modificado usando o operador colchetes (**`[]`**).
- Objetos da classe **`vector`** podem ser comparados diretamente com os operadores de igualdade (**`==`**) e desigualdade (**`!=`**).
- O operador de atribuição (**`=`**) também pode ser usado com objetos **`vector`**.

Classe vector da biblioteca padrão

```
1  // Fig. 15.14: fig15_14.cpp
2  // Demonstrating C++ Standard Library class template vector.
3  #include <iostream>
4  #include <iomanip>
5  #include <vector>
6  using namespace std;
7
8  void outputVector(const vector< int > & ); // display the vector
9  void inputVector( vector< int > &);        // input values into the vector
10
11 int main()
12 {
13    vector< int > integers1( 7 ); // 7-element vector< int >
14    vector< int > integers2( 10 ); // 10-element vector< int >
15
16    // print integers1 size and contents
17    cout << "Size of vector integers1 is " << integers1.size()
18         << "\nvector after initialization:" << endl;
19    outputVector( integers1 );
```

Classe vector

```
21    // print integers2 size and contents
22    cout << "\nSize of vector integers2 is " << integers2.size()
23         << "\nvector after initialization:" << endl;
24    outputVector( integers2 );
25
26    // input and print integers1 and integers2
27    cout << "\nEnter 17 integers:" << endl;
28    inputVector( integers1 );
29    inputVector( integers2 );
30
31    cout << "\nAfter input, the vectors contain:\n"
32         << "integers1:" << endl;
33    outputVector( integers1 );
34    cout << "integers2:" << endl;
35    outputVector( integers2 );
36
37    // use inequality (!=) operator with vector objects
38    cout << "\nEvaluating: integers1 != integers2" << endl;
39
40    if ( integers1 != integers2 )
41       cout << "integers1 and integers2 are not equal" << endl;
42
```

```
43    // create vector integers3 using integers1 as an
44    // initializer; print size and contents
45    vector< int > integers3( integers1 );      // copy constructor
46
47    cout << "\nSize of vector integers3 is " << integers3.size()
48         << "\nvector after initialization:" << endl;
49    outputVector( integers3 );
50
51    // use overloaded assignment (=) operator
52    cout << "\nAssigning integers2 to integers1:" << endl;
53    integers1 = integers2;           // assign integers2 to integers1
54
55    cout << "integers1:" << endl;
56    outputVector( integers1 );
57    cout << "integers2:" << endl;
58    outputVector( integers2 );
59
60    // use equality (==) operator with vector objects
61    cout << "\nEvaluating: integers1 == integers2" << endl;
62
63    if ( integers1 == integers2 )
64       cout << "integers1 and integers2 are equal" << endl;
65
```

Classe vector

Classe vector

```
66    // use square brackets to create rvalue
67    cout << "\nintegers1[5] is " << integers1[ 5 ];
68
69    // use square brackets to create lvalue
70    cout << "\n\nAssigning 1000 to integers1[5]" << endl;
71    integers1[ 5 ] = 1000;
72    cout << "integers1:" << endl;
73    outputVector( integers1 );
74
75    // attempt to use out-of-range index
76    try
77    {
78       cout << "\nAttempt to display integers1.at( 15 )" << endl;
79       cout << integers1.at( 15 ) << endl; // ERROR: out of range
80    }
81    catch ( out_of_range &ex )
82    {
83       cout << "An exception occurred: " << ex.what() << endl;
84    }
85 }
86
```

Classe vector

```
87    // output vector contents
88    void outputVector( const vector< int > &array )
89    {
90       size_t i; // declare control variable
91
92       for ( i = 0; i < ; ++i )
93       {
94          cout << setw( 12 ) << array[ i ];
95
96          if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
97             cout << endl;
98       }
99
100      if ( i % 4 != 0 )
101         cout << endl;
102   }
103
104   // input vector contents
105   void inputVector( vector< int > &array )
106   {
107      for ( size_t i = 0; i < array.size(); ++i )
108         cin >> array[ i ];
109   }
```

```
Size of vector integers1 is 7
vector after initialization:
         0         0         0         0
         0         0         0

Size of vector integers2 is 10
vector after initialization:
         0         0         0         0
         0         0         0         0
         0         0

Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the vectors contain:
integers1:
         1         2         3         4
         5         6         7
integers2:
         8         9        10        11
        12        13        14        15
        16        17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of vector integers3 is 7
vector after initialization:
         1         2         3         4
         5         6         7

Assigning integers2 to integers1:
integers1:
         8         9        10        11
        12        13        14        15
        16        17
integers2:
         8         9        10        11
        12        13        14        15
        16        17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
         8         9        10        11
        12      1000        14        15
        16        17

Attempt to display integers1.at( 15 )
An exception occurred: invalid vector<T> subscript
```

Fig. 15.14 | Demonstrating C++ Standard Library class template vector.

- Uma **exceção** indica um problema que ocorre enquanto um programa é executado. O nome "exceção" sugere que o problema ocorre com pouca frequência — se a "regra" é que uma instrução normalmente é executada corretamente, então o problema representa a "exceção à regra".
- O tratamento de exceções permite que você crie programas tolerantes a falhas que podem resolver exceções.
- Para lidar com uma exceção, coloque qualquer código que possa lançar uma exceção em uma instrução **`try`**.
- O bloco **`try`** contém o código que pode lançar uma exceção, e o bloco **`catch`** contém o código que lida com a exceção se ela ocorrer.
- Quando um bloco **`try`** termina, quaisquer variáveis declaradas no bloco try saem do escopo.
- Um bloco **`catch`** declara um tipo e um parâmetro de exceção. Dentro do bloco catch, você pode usar o identificador do parâmetro para interagir com um objeto de exceção capturado.
- O método **`what`** de um objeto de exceção retorna a mensagem de erro da exceção.

Referências

- https://cplusplus.com/reference/
- C++ How to program – Deitel e Deitel.
- Notas de aula da disciplina Programação Orientada a Objetos, Prof. André Bernardi.