Laboratório de programação II

Templates

Edson Moreno

edson.moreno@pucrs.br

http://www.inf.pucrs.br/~emoreno

Introdução

- Templates são uma alternativa à sobrecarga de funções, quando estas envolvem lógicas de programas e operações idênticas para vários tipos de dados.

- O uso de templates permite ao programador a implementação de um único código para uma determinada função.
  - C++ gera automaticamente funções template para tratar as chamadas.

- Para definir um template de função, basta usar a palavra-chave template seguida da lista de parâmetros entre os símbolos < e >.
  - Parâmetros devem ser precedidos da palavra-chave typename ou class

Função template

```
template <typename tipoGenerico>  // ou template <class tipoGenerico>
tipoGenerico minhaFuncao(tipoGenerico valor1, tipoGenerico valor2)
{
  // código da função
}

int main()
{
  // meu código
}
```

Classe template

- Templates também pode ser usados para criar classes genéricas.
  - Todos métodos devem ser implementados no arquivo de cabeçalho
    - o compilador gera automaticamente o código quando a palavra-chave template é encontrada

```
/***** minhaClasse.h *****/
```

```
template <class Alpha>
class minhaClasse
{
public:
  minhaClasse();
  void meuMetodo(Alpha var);
private:
  Alpha _var;
};
```

```
template <class T>
minhaClasse<T>::minhaClasse() {
  // minha construtora
}

template <class T>
minhaClasse<T>::meuMetodo(Alpha var) {
  _var = var;
}
```

Exemplo de uma Classe Template

```
template<class ItemType>
class GList
{
 public:
    bool IsEmpty() const;
    bool IsFull() const;
    int  Length() const;
    void Insert( /* in */ ItemType item );
    void Delete( /* in */ ItemType item );
    bool IsPresent( /* in */ ItemType item ) const;
    void SelSort();
    void Print() const;
    GList();                          // Constructor
 private:
    int      length;
    ItemType data[MAX_LENGTH];
};
```

Parâmetro Template

Instanciando uma classe template

- **Argumentos da classe template devem ser explícitos.**

- **Quando uma classe template é instanciada**
  - **O compilador substitui o argumento genérico em toda a classe pelo argumento de tipo informado**

Instanciando uma Classe Template

Para criar uma lista de diferentes tipo

```
// Client code

GList<int> list1;
GList<float> list2;
GList<string> list3;

list1.Insert(356);
list2.Insert(84.375);
list3.Insert("Muffler bolt");
```

template argument

O compilador gera 3 tipos de classe distintos

```
GList_int list1;
GList_float list2;
GList_string list3;
```

Exemplo de substituição

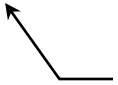
```
class Glist_int
{
public:

void Insert( /* in */ ItemType item );

    void Delete( /* in */ ItemType item );

    bool IsPresent( /* in */ ItemType item ) const;

private:
    int       length;
    ItemType  data[MAX_LENGTH];
};
```



Definição de função membro de uma classe template

```
template<class ItemType>
void GList<ItemType>::Insert( /* in */ ItemType item )
{
    data[length] = item;
    length++;
}
```

```
//after substitution of float
void GList<float>::Insert( /* in */ float item )
{
    data[length] = item;
    length++;
}
```

Classe Template

- A definição de classes template vem precedida de um cabeçalho
  - Tal como `template< typename T >`

- O parâmetro T pode ser usado como um tipo de dado em uma método ou atributo

- Outros parâmetros podem ser especificados usando vírgula para separá-los
  - `template< typename T1, typename T2 >`

Example

- **`Stack.h`**

```
1  // Stack.h
2  // Stack class template.
3  #ifndef STACK_H
4  #define STACK_H
5
6  template< typename T >
7  class Stack
8  {
9  public:
10    Stack( int = 10 ); // default constructor (Stack size 10)
11
12    // destructor
13    ~Stack()
14    {
15       delete [] stackPtr; // deallocate internal space for Stack
16    } // end ~Stack destructor
17
18    bool push( const T& ); // push an element onto the Stack
19    bool pop( T& ); // pop an element off the Stack
20
21    // determine whether Stack is empty
22    bool isEmpty() const
23    {
24       return top == -1;
25    } // end function isEmpty
```

Define uma classe template **`Stack`** com um parâmetro genérico **`T`**

Métodos usando o parâmetro genérico **`T`** na especificação de seus argumentos

Example

- **`Stack.h`**

```
26
27    // determine whether Stack is full
28    bool isFull() const
29    {
30       return top == size - 1;
31    } // end function isFull
32
33 private:
34    int size; // # of elements in the Stack
35    int top; // location of the top element (-1 means empty)
36    T *stackPtr; // pointer to internal representation of the Stack
37 }; // end class template Stack
38
39 // constructor template
40 template< typename T >
41 Stack< T >::Stack( int s )
42    : size( s > 0 ? s : 10 ), // validate size
43      top( -1 ), // Stack initially empty
44      stackPtr( new T[ size ] ) // allocate memory for elements
45 {
46    // empty body
47 } // end Stack constructor template
```

atributo **`stackPtr`** é um ponteiro para um tipo **`T`**

Método implementado fora da classe template inicia com o cabeçalho template equivalente ao da classe

```
48
49 // push element onto Stack;
50 // if successful, return true; otherwise, return false
51 template< typename T >
52 bool Stack< T >::push( const T &pushValue )
53 {
54    if ( !isFull() )
55    {
56       stackPtr[ ++top ] = pushValue; // place item on Stack
57       return true; // push successful
58    } // end if
59
60    return false; // push unsuccessful
61 } // end function template push
62
63 // pop element off Stack;
64 // if successful, return true; otherwise, return false
65 template< typename T >
66 bool Stack< T >::pop( T &popValue )
67 {
68    if ( !isEmpty() )
69    {
70       popValue = stackPtr[ top-- ]; // remove item from Stack
71       return true; // pop successful
72    } // end if
73
74    return false; // pop unsuccessful
75 } // end function template pop
76
77 #endif
```

Example

- **Stack.h**

Example

- **main.cpp**

```
1  // main.cpp
2  // Stack class template test program.
3  #include <iostream>
4
5  using namespace std;
6
7  #include "Stack.h" // Stack class template definition
8
9  int main()
10 {
11    Stack< double > doubleStack( 5 ); // size 5
12    double doubleValue = 1.1;
13
14    cout << "Pushing elements onto doubleStack\n";
15
16    // push 5 doubles onto doubleStack
17    while ( doubleStack.push( doubleValue ) )
18    {
19       cout << doubleValue << ' ';
20       doubleValue += 1.1;
21    } // end while
22
23    cout << "\nStack is full. Cannot push " << doubleValue
24       << "\n\nPopping elements from doubleStack\n";
25
26    // pop elements from doubleStack
27    while ( doubleStack.pop( doubleValue ) )
28       cout << doubleValue << ' ';
```

Main cria uma especialização da classe template **`Stack< double >`** onde o tipo **`double`** é associado ao parâmetro **`T`**

Example

- **`main.cpp`**

```
29
30    cout << "\nStack is empty. Cannot pop\n";
31
32    Stack< int > intStack; // default size 10
33    int intValue = 1;
34    cout << "\nPushing elements onto intStack\n";
35
36    // push 10 integers onto intStack
37    while ( intStack.push( intValue ) )
38    {
39       cout << intValue << ' ';
40       intValue++;
41    } // end while
42
43    cout << "\nStack is full. Cannot push " << intValue
44       << "\n\nPopping elements from intStack\n";
45
46    // pop elements from intStack
47    while ( intStack.pop( intValue ) )
48       cout << intValue << ' ';
49
50    cout << "\nStack is empty. Cannot pop" << endl;
51    return 0;
52 } // end main
```

Mesma **main** cria uma nova especialização da classe template **`Stack< int >`** onde **`int`** é associado ao parâmetro genérico **`T`**

Classe Template e Parâmetros sem Tipo

- Classe templates
  - Parâmetros sem tipo
    - Argumentos padrão
    - Tratados como consts
    - Ex.:

```
template< class T, int elements >
Stack< double, 100 > mostRecentSalesFigures;
```

      - Declara objeto do tipo **`Stack< double, 100> pilha`**

Classe Template e Parâmetros sem Tipo

```
#include <iostream>

using namespace std;

template <typename T, int elementos>
void printArray (T *a) {
     for (int i = 0; i < elementos; i++)
         cout << a [i] << endl;
}

int main(int argc, char *argv[])
{
    int array [] = {1, 2, 3, 4, 5};
    int elementos = 5;

    printArray <int, 5> (array);

    return 0;
}
```

Classe Template e Parâmetros sem Tipo

- Classe templates
  - Parâmetro tipados
    - Tipo padrão
    - Ex.: **`template< class T = string >`**
      - Declara objeto do tipo **`Stack<> pilha;`**

Classe Template e Herança

- Os conceitos associados à herança podem ser utilizados em classes template
  - Além de gerar classes, podemos gerar hierarquias de classes com templates.
  - Cada hieraquia é parametrizada.
  - Funções virtuais são permitidas na hierarquia, ou seja podemos ter polimorfismo.
  - Pode-se ter também classes template abstratas, etc …

Classe template e herança

```
#include <iostream>

template <class T > class Base{
 T tob;
 public:
   Base(T par=0): tob(par) {
   }
   virtual void fpoli( ) {
     cout << "\n base " << tob;
   }
};
```

```
template <class T> class Deriv : public Base {
 T tod;
 public:
   Deriv(T parb=0, T pard=0) : Base(parb), tod(pard) {    }
   void fpoli() {
     cout << "\n deriv " << tod;
   }
};

template <class T> void f(Base<T>* ptb) {
 ptb->fpoli();
}
```

```
int main() {
  Deriv<int> i0;
  i0.fpoli();
  Base<int> i1(1);
  i1.fpoli();
  Base<float> *ptbf = new Base(10.);
  ptbf->fpoli();
  f(ptbf);
  delete ptbf;
  ptbf = new Deriv<float>(20.,30.);
  ptbf->fpoli();
  f(ptbf);
  delete ptbf;
  return 0;
}
```

Exercícios

- A função abaixo só ordena arranjos de elementos do tipo int. Escreva a função genérica correspondente:

```
void ordenaPorSelecao(int a[], int N){
  int i,j,aux,indMenor;
  for(i=0; i<N-1; i++){
  indMenor=i;
  for(j=i+1; j<N; j++)
    if(a[j]<a[indMenor]) indMenor=j;
    aux=a[i]; a[i]=a[indMenor];
    a[indMenor]=aux;
  }
}
```

- Escreva um programa que utiliza a função genérica da questão anterior.