Escola de Engenharia
Universidade do Minho

Complementos de Programação de Computadores – Aula 6a

Templates em C++

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em P.Deitel e H.Deitel 2010)

Escola de Engenharia
Universidade do Minho

Outline

- 22.1 Introduction
- 22.2 Class Templates
- 22.3 Class Templates and Non-type Parameters
- 22.4 Templates and Inheritance
- 22.5 Templates and friends
- 22.6 Templates and static Members

Escola de Engenharia
Universidade do Minho

Objectives

- To be able to use class templates to create a group of related types
- To be able to distinguish between class templates and template classes
- To understand how to overload template functions
- To understand the relationships among templates, `friend`s, inheritance and `static` members

Escola de Engenharia
Universidade do Minho

22.1 Introduction – Class Templates

- **Templates**
  - Easily create a large range of related functions or classes
  - Function template - the blueprint of the related functions
  - Template function - a specific function *made* from a function template
- **Class templates**
  - Allow type-specific versions of generic classes
- **Format:**

```
template <class T>
class ClassName {
   Definition
 }
```

  - Need not use "`T`", any identifier will work
  - To create an object of the class, type `ClassName < type > myObject;`

    Example: `Stack< double > doubleStack;`

22.2 Class Templates

- **Template class functions**
  - Defined normally, but preceded by `template<class T>`
    - Generic data in class listed as type `T`
  - Binary scope resolution operator used
  - Template class function definition:

```
template<class T>
MyClass< T >::MyClass(int size)
{
   myArray = new T[size];
}
```

    - Constructor definition - creates an array of type `T`


```
1  // Fig. 22.1: tstack1.h
2  // Class template Stack
3  #ifndef TSTACK1_H
4  #define TSTACK1_H
5
6  template< class T >
7  class Stack {
8  public:
9     Stack( int = 10 );    // default constructor (stack size 10)
10    ~Stack() { delete [] stackPtr; } // destructor
11    bool push( const T& ); // push an element onto the stack
12    bool pop( T& );        // pop an element off the stack
13 private:
14    int size;             // # of elements in the stack
15    int top;              // location of the top element
16    T *stackPtr;          // pointer to the stack
17
18    bool isEmpty() const { return top == -1; }      // utility
19    bool isFull() const { return top == size - 1; } // functions
20 }; // end class template Stack
21
```

tstack1.h (Part 1 of 3)

```
22 // Constructor with default size 10
23 template< class T >
24 Stack< T >::Stack( int s )
25 {
26    size = s > 0 ? s : 10;
27    top = -1;               // Stack is initially empty
28    stackPtr = new T[ size ]; // allocate space for elements
29 } // end Stack constructor
30
31 // Push an element onto the stack
32 // return 1 if successful, 0 otherwise
33 template< class T >
34 bool Stack< T >::push( const T &pushValue )
35 {
36    if ( !isFull() ) {
37       stackPtr[ ++top ] = pushValue; // place item in Stack
38       return true;  // push successful
39    } // end if
40    return false;     // push unsuccessful
41 } // end function template push
42
```

tstack1.h (Part 2 of 3)


```
43 // Pop an element off the stack
44 template< class T >
45 bool Stack< T >::pop( T &popValue )
46 {
47    if ( !isEmpty() ) {
48       popValue = stackPtr[ top-- ];  // remove item from Stack
49       return true;  // pop successful
50    } // end if
51    return false;     // pop unsuccessful
52 } // end function template pop
53
54 #endif
```

tstack1.h (Part 3 of 3)

```
55 // Fig. 22.1: fig22_01.cpp
56 // Test driver for Stack template
57 #include <iostream>
58
59 using std::cout;
60 using std::cin;
61 using std::endl;
62
63 #include "tstack1.h"
64
```

fig22_01.cpp (Part 1 of 3)

```
65 int main()
66 {
67    Stack< double > doubleStack( 5 );
68    double f = 1.1;
69    cout << "Pushing elements onto doubleStack\n";
70
71    while ( doubleStack.push( f ) ) { // success true returned
72       cout << f << ' ';
73       f += 1.1;
74    } // end while
75
76    cout << "\nStack is full. Cannot push " << f
77         << "\n\nPopping elements from doubleStack\n";
78
79    while ( doubleStack.pop( f ) )  // success true returned
80       cout << f << ' ';
81
82    cout << "\nStack is empty. Cannot pop\n";
83
84    Stack< int > intStack;
85    int i = 1;
86    cout << "\nPushing elements onto intStack\n";
87
88    while ( intStack.push( i ) ) { // success true returned
89       cout << i << ' ';
90       ++i;
91    } // end while
92
```

fig22_01.cpp (Part 2 of 3)


```
93    cout << "\nStack is full. Cannot push " << i
94         << "\n\nPopping elements from intStack\n";
95
96    while ( intStack.pop( i ) )  // success true returned
97       cout << i << ' ';
98
99    cout << "\nStack is empty. Cannot pop\n";
100    return 0;
101 } // end function main
```

fig22_01.cpp (Part 3 of 3)

Program Output

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

```
1  // Fig. 22.2: fig22_02.cpp
2  // Test driver for Stack template.
3  // Function main uses a function template to manipulate
4  // objects of type Stack< T >.
5  #include <iostream>
6
7  using std::cout;
8  using std::cin;
9  using std::endl;
10
11 #include "tstack1.h"
12
13 // Function template to manipulate Stack< T >
14 template< class T >
15 void testStack(
16    Stack< T > &theStack,   // reference to the Stack< T >
17    T value,                // initial value to be pushed
18    T increment,            // increment for subsequent values
19    const char *stackName ) // name of the Stack < T > object
20 {
21    cout << "\nPushing elements onto " << stackName << '\n';
22
23    while ( theStack.push( value ) ) { // success true returned
24       cout << value << ' ';
25       value += increment;
26    } // end while
27
```

fig22_02.cpp (Part 1 of 2)


```
28    cout << "\nStack is full. Cannot push " << value
29         << "\n\nPopping elements from " << stackName << '\n';
30
31    while ( theStack.pop( value ) )  // success true returned
32       cout << value << ' ';
33
34    cout << "\nStack is empty. Cannot pop\n";
35 } // end function template testStack
36
37 int main()
38 {
39    Stack< double > doubleStack( 5 );
40    Stack< int > intStack;
41
42    testStack( doubleStack, 1.1, 1.1, "doubleStack" );
43    testStack( intStack, 1, 1, "intStack" );
44
45    return 0;
46 } // end function main
```

fig22_02.cpp (Part 2 of 2)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Program Output

22.3 Class Templates and Non-type Parameters

- **Can use non-type parameters in templates**
  - Default argument, treated as `const`
- **Example:**

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

    - Defines object of type `Stack< double, 100>`
  - This may appear in the class definition:

```
T stackHolder[ elements ]; //array to hold stack
```

    - Creates array at compile time, rather than dynamic allocation at execution time
- **Classes can be overridden**
  - For template class `Array`, define a class named `Array<myCreatedType>`
  - This new class overrides then class template for `myCreatedType`
  - The template remains for unoverriden types

Escola de Engenharia
Universidade do Minho

22.4 Templates and Inheritance

- A class template can be derived from a template class
- A class template can be derived from a non-template class
- A template class can be derived from a class template
- A non-template class can be derived from a class template

Escola de Engenharia
Universidade do Minho

22.5 Templates and friends

- **Friendships allowed between a class template and**
  - Global function
  - Member function of another class
  - Entire class
- **`friend` functions**
  - Inside definition of class template **X**:
  - `friend void f1();`
    `f1()` a `friend` of all template classes
  - `friend void f2( X< T > & );`
    `f2( X< int > & )` is a `friend` of `X< int >` only. The same applies for `float`, `double`, etc.
  - `friend void A::f3();`
    Member function `f3` of class `A` is a `friend` of all template classes
  - `friend void C< T >::f4( X< T > & );`
    `C<float>::f4( X< float> & )` is a `friend` of `class X<float>` only

Escola de Engenharia
Universidade do Minho

22.5 Templates and friends (II)

- `friend` classes
  - `friend class Y;`
    - Every member function of `Y` a friend with every template class made from `X`
  - `friend class Z<T>;`
    - Class `Z<float>` a `friend` of class `X<float>`, etc.

Escola de Engenharia
Universidade do Minho

22.6 Templates and `static` Members

- **Non-template class**
  - `static` data members shared between all objects
- **Template classes**
  - Each class (`int`, `float`, etc.) has its own copy of `static` data members
  - `static` variables initialized at file scope
  - Each template class gets its own copy of `static` member functions

Escola de Engenharia
Universidade do Minho

Complementos de Programação de Computadores – Aula 6a

Templates em C++

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis1970@gmail.com / lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em P.Deitel e H.Deitel 2010)