Chapter 22 - C++ Templates

Outline

22.1 Introduction

- 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

22.2 Class Templates

- 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;`

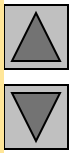
22.2 Class Templates (II)

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

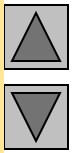
```
template<class T>
MyClass< T >::MyClass(int size)
{
   myArray = new T[size];
}
```

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

```
1   // Fig. 22.3: 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     ~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; } //
20  };
21
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  }
```



Outline



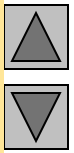
Outline

1.3 push

1.4 pop

```
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     }
40     return false;     // push unsuccessful
41  }
42
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     }
51     return false;     // pop unsuccessful
52  }
53
54  #endif
```

```
55  // Fig. 22.3: fig22_03.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
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     }
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
```



Outline

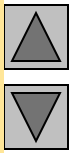
1. Load header

1.1 Initialize `doubleStack`

1.2 Initialize variables

2. Function calls

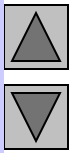
```
80          cout << f << ' ';
81
82       cout << "\nStack is empty. Cannot pop\n";
83
84       Stack< int > intStack;
85       int i = 1;
86       cout << "\nPushing elements onto intStack\n";
87
88       while ( intStack.push( i ) ) { // success true returned
89          cout << i << ' ';
90          ++i;
91       }
92
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   }
```



Outline

2. Function calls

3. Output



Outline

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
```

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;
```

    - Declares 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

22.3 Class Templates and Non-type Parameters (II)

- 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

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

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

22.5 Templates and friends (II)

- `friend void C< T >::f4( X< T > & );`
  - `C<float>::f4( X< float> & )` is a `friend` of `class X<float>` only

- **`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.

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