Chapter 22 - C++ Templates

Outline

Objectives

- In this chapter, you will learn:
  - 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.

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
  - 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
```

Outline

tstack1.h (Part 1 of 3)

```
// Constructor with default size 10
template< class T >
Stack< T >::Stack( int s )
{
   size = s > 0 ? s : 10;
   top = -1;              // Stack is initially empty
   stackPtr = new T[ size ]; // allocate space for elements
} // end Stack constructor

// Push an element onto the stack
// return 1 if successful, 0 otherwise
template< class T >
bool Stack< T >::push( const T &pushValue )
{
   if ( !isFull() ) {
      stackPtr[ ++top ] = pushValue; // place item in Stack
      return true;  // push successful
   } // end if
   return false;     // push unsuccessful
} // end function template push
```

Outline

tstack1.h (Part 2 of 3)


```
// Pop an element off the stack
template< class T >
bool Stack< T >::pop( T &popValue )
{
   if ( !isEmpty() ) {
      popValue = stackPtr[ top-- ];  // remove item from Stack
      return true;  // pop successful
   } // end if
   return false;     // pop unsuccessful
} // end function template pop

#endif
```

Outline

tstack1.h (Part 3 of 3)

```
// Fig. 22.1: fig22_01.cpp
// Test driver for Stack template
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include "tstack1.h"
```

fig22_01.cpp (Part 1 of 3)


```
int main()
{
   Stack< double > doubleStack( 5 );
   double f = 1.1;
   cout << "Pushing elements onto doubleStack\n";

   while ( doubleStack.push( f ) ) { // success true returned
      cout << f << ' ';
      f += 1.1;
   } // end while

   cout << "\nStack is full. Cannot push " << f
        << "\n\nPopping elements from doubleStack\n";

   while ( doubleStack.pop( f ) )  // success true returned
      cout << f << ' ';

   cout << "\nStack is empty. Cannot pop\n";

   Stack< int > intStack;
   int i = 1;
   cout << "\nPushing elements onto intStack\n";

   while ( intStack.push( i ) ) { // success true returned
      cout << i << ' ';
      ++i;
   } // end while
```

Outline

fig22_01.cpp (Part 2 of 3)


```
   cout << "\nStack is full. Cannot push " << i
        << "\n\nPopping elements from intStack\n";

   while ( intStack.pop( i ) )  // success true returned
      cout << i << ' ';

   cout << "\nStack is empty. Cannot pop\n";
   return 0;
} // end function main
```

```
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
```

Outline

fig22_01.cpp (Part 3 of 3)

Program Output


```
// Fig. 22.2: fig22_02.cpp
// Test driver for Stack template.
// Function main uses a function template to manipulate
// objects of type Stack< T >.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include "tstack1.h"

// Function template to manipulate Stack< T >
template< class T >
void testStack(
   Stack< T > &theStack,   // reference to the Stack< T >
   T value,                // initial value to be pushed
   T increment,            // increment for subsequent values
   const char *stackName ) // name of the Stack < T > object
{
   cout << "\nPushing elements onto " << stackName << '\n';

   while ( theStack.push( value ) ) { // success true returned
      cout << value << ' ';
      value += increment;
   } // end while
```

Outline

fig22_02.cpp (Part 1 of 2)


```
   cout << "\nStack is full. Cannot push " << value
        << "\n\nPopping elements from " << stackName << '\n';

   while ( theStack.pop( value ) )  // success true returned
      cout << value << ' ';

   cout << "\nStack is empty. Cannot pop\n";
} // end function template testStack

int main()
{
   Stack< double > doubleStack( 5 );
   Stack< int > intStack;

   testStack( doubleStack, 1.1, 1.1, "doubleStack" );
   testStack( intStack, 1, 1, "intStack" );

   return 0;
} // end function main
```

Outline

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
```

Outline

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

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 `friend`s (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