Chapter 14 Templates

C++ How to Program, Late Objects Version, 7/e

OBJECTIVES

In this chapter you'll learn:

- To use function templates to conveniently create a group of related (overloaded) functions.
- To distinguish between function templates and function-template specializations.
- To use class templates to create groups of related types.
- To distinguish between class templates and class-template specializations.
- To overload function templates.
- To understand the relationships among templates, friends, inheritance and static members.

14.1 Introduction

- Function templates and class templates enable you to specify, with a single code segment, an entire range of related (overloaded) functions—called function-template specializations—or an entire range of related classes—called class-template specializations.
- This technique is called generic programming.
- Note the distinction between templates and template specializations:
  - Function templates and class templates are like stencils out of which we trace shapes.
  - Function-template specializations and class-template specializations are like the separate trac-ings that all have the same shape, but could, for example, be drawn in different colors.
- In this chapter, we present a function template and a class tem-plate.

Software Engineering Observation 14.1

Most C++ compilers require the complete definition of a template to appear in the client source-code file that uses the template. For this reason and for reusability, templates are often defined in header files, which are then `#include`*d into the appropriate client source-code files. For class templates, this means that the member functions are also defined in the header file.*

14.2 Function Templates

- Overloaded functions normally perform *similar or identical operations on different types of data.*
- If the operations are *identical for each type, they can be expressed more com-pactly and conveniently using function templates.*
- Initially, you write a single function-template definition.
- Based on the argument types provided explicitly or inferred from calls to this function, the compiler generates separate source-code functions (i.e., function-template specializations) to handle each function call appropriately.

Error-Prevention Tip 14.1

Function templates, like macros, enable software reuse. Unlike macros, function templates help eliminate many types of errors through the scrutiny of full C++ type checking.

14.2 Function Templates (cont.)

- All function-template definitions begin with keyword `template` followed by a list of template parameters to the function template enclosed in angle brackets (< and >); each template parameter that represents a type must be preceded by either of the interchangeable keywords `class` or `typename`, as in
  - `template<typename T>`
  - Or
    - `template<class ElementType>`
  - Or
    - `template<typename BorderType, typename FillType>`
- The type template parameters of a function-template definition are used to specify the types of the arguments to the function, to specify the return type of the function and to declare variables within the function.
- Keywords `typename` and `class` used to specify function-template parameters actually mean "any fundamental type or user-defined type."

Common Programming Error 14.1

Not placing keyword `class` *or keyword* `typename` *before each type template parameter of a function template is a syntax error.*

14.2 Function Templates (cont.)

- Let's examine function template `printArray` in Fig. 14.1, lines 7–14.
- Function template `printArray` declares (line 7) a single template parameter `T` (`T` can be any valid identifier) for the type of the array to be printed by function `print-Array`; `T` is referred to as a type template parameter, or type parameter.

```
1  // Fig. 14.1: fig14_01.cpp
2  // Using template functions.
3  #include <iostream>
4  using namespace std;
5
6  // function template printArray definition
7  template< typename T >
8  void printArray( const T * const array, int count )
9  {
10    for ( int i = 0; i < count; i++ )
11       cout << array[ i ] << " ";
12
13    cout << endl;
14 } // end function template printArray
15
16 int main()
17 {
18    const int aCount = 5; // size of array a
19    const int bCount = 7; // size of array b
20    const int cCount = 6; // size of array c
21
```

Fig. 14.1 | Function-template specializations of function template `printArray`. (Part 1 of 3.)

```
22      int a[ aCount ] = { 1, 2, 3, 4, 5 };
23      double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
24      char c[ cCount ] = "HELLO"; // 6th position for null
25
26      cout << "Array a contains:" << endl;
27
28      // call integer function-template specialization
29      printArray( a, aCount );
30
31      cout << "Array b contains:" << endl;
32
33      // call double function-template specialization
34      printArray( b, bCount );
35
36      cout << "Array c contains:" << endl;
37
38      // call character function-template specialization
39      printArray( c, cCount );
40  } // end main
```

Fig. 14.1 | Function-template specializations of function template `printArray`. (Part 2 of 3.)

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

Fig. 14.1 | Function-template specializations of function template `printArray`. (Part 3 of 3.)

14.2 Function Templates (cont.)

- When the compiler detects a `printArray` function invocation in the client program (e.g., lines 29, 34 and 39), the compiler uses its overload resolution capabilities to find a definition of function `printArray` that best matches the function call.
- In this case, the only `printArray` function with the appropriate number of parameters is the `printArray` function template (lines 7–14).
- Consider the function call at line 29.
- The compiler compares the type of `printArray`'s first argument (`int *` at line 29) to the `printArray` function template's first parameter (`const T * const` at line 8) and deduces that replacing the type parameter `T` with `int` would make the argument consistent with the parameter.
- Then, the compiler substitutes `int` for `T` throughout the template definition and compiles a `printArray` specialization that can display an array of `int` values.

14.2 Function Templates (cont.)

- The function-template specialization for type `int` is

```
void printArray( const int * const array, int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

   cout << endl;
} // end function printArray
```

- As with function parameters, the names of template parameters must be unique inside a template definition.
- Template parameter names need not be unique across different function templates.
- Figure 14.1 demonstrates function template `printArray`.
- It's important to note that if `T` (line 7) represents a user-defined type (which it does not in Fig. 14.1), there must be an overloaded stream insertion operator for that type; otherwise, the first stream insertion operator in line 11 will not compile.

Common Programming Error 14.2

If a template is invoked with a user-defined type, and if that template uses functions or operators (e.g., ==, +, <=) with objects of that class type, then those functions and operators must be overloaded for the user-defined type. Forgetting to overload such operators causes compilation errors.

Performance Tip 14.1

Although templates offer software-reusability benefits, remember that multiple function-template specializations and class-template specializations are instantiated in a program (at compile time), despite the fact that the templates are written only once. These copies can consume considerable memory. This is not normally an issue, though, because the code generated by the template is the same size as the code you'd have written to produce the separate overloaded functions.

14.3 Overloading Function Templates

- Function templates and overloading are intimately related.
- The function-template specializations gener-ated from a function template all have the same name, so the compiler uses overloading resolution to invoke the proper function.
- A function template may be overloaded in several ways.
  - We can provide other function templates that specify the same function name but different function parameters.
  - We can provide nontemplate func-tions with the same function name but different function arguments.

Common Programming Error 14.3

A compilation error occurs if no matching function definition can be found for a particular function call or if there are multiple matches that the compiler considers ambiguous.

14.4 Class Templates

- It's possible to understand the concept of a "stack" (a data structure into which we insert items at the top and retrieve those items in last-in, first-out order) independent of the type of the items be-ing placed in the stack.
- However, to instantiate a stack, a data type must be specified.
- Wonderful opportunity for software reusability.
- We need the means for describing the notion of a stack generically and instanti-ating classes that are type-specific versions of this generic stack class.
- C++ provides this capability through class templates.

Software Engineering Observation 14.2

Class templates encourage software reusability by enabling type-specific versions of generic classes to be instantiated.

14.4 Class Templates (cont.)

- Class templates are called parameterized types, because they require one or more type parameters to specify how to customize a "generic class" template to form a class-template specialization.
  - Each time an additional class-template specialization is needed, you use a concise, simple notation, and the compiler writes the source code for the specialization you require.

14.4 Class Templates (cont.)

- Note the `Stack` class-template definition in Fig. 14.2.
- It looks like a conventional class definition, except that it's preceded by the header (line 6)
  - `template< typename T >`
- to specify a class-template definition with type parameter `T` which acts as a placeholder for the type of the `Stack` class to be created.
- The type of element to be stored on this `Stack` is men-tioned generically as `T` throughout the `Stack` class header and member-function definitions.
- Due to the way this class template is designed, there are two constraints for nonfundamental data types used with this `Stack`
  - they must have a default constructor
  - their assignment operators must properly copy objects into the `Stack`

```
1  // Fig. 14.2: 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
```

Fig. 14.2 | Class template `Stack`. (Part 1 of 4.)

```
21      // determine whether Stack is empty
22      bool isEmpty() const
23      {
24         return top == -1;
25      } // end function isEmpty
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
```

Fig. 14.2 | Class template `Stack`. (Part 2 of 4.)

```
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
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
```

Fig. 14.2 | Class template `Stack`. (Part 3 of 4.)

```
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
```

Fig. 14.2 | Class template `Stack`. (Part 4 of 4.)

14.4 Class Templates (cont.)

- The member-function definitions of a class template are function templates.
- The member-function definitions that appear outside the class template definition each begin with the header
  - `template< typename T >`
- Thus, each definition resembles a conventional function definition, except that the `Stack` element type always is listed generically as type parameter `T`.
- The binary scope resolution operator is used with the class-template name to tie each member-function definition to the class template's scope.
- When `doubleStack` is instantiated as type `Stack<double>`, the `Stack` constructor function-template specialization uses `new` to create an array of elements of type `double` to represent the stack.

14.4 Class Templates (cont.)

- Now, let's consider the driver (Fig. 14.3) that exercises the `Stack` class template.
- The driver begins by instantiating object `doubleStack` of size `5`.
- This object is declared to be of class `Stack< double >` (pronounced "`Stack` of `double`").
- The compiler associates type `double` with type parameter `T` in the class template to produce the source code for a `Stack` class of type `double`.
- Although templates offer software-reusability benefits, remember that mul-tiple class-template specializations are instantiated in a program (at compile time), even though the template is written only once.

```
1  // Fig. 14.3: fig14_03.cpp
2  // Stack class template test program.
3  #include <iostream>
4  #include "Stack.h" // Stack class template definition
5  using namespace std;
6
7  int main()
8  {
9     Stack< double > doubleStack( 5 ); // size 5
10    double doubleValue = 1.1;
11
12    cout << "Pushing elements onto doubleStack\n";
13
14    // push 5 doubles onto doubleStack
15    while ( doubleStack.push( doubleValue ) )
16    {
17       cout << doubleValue << ' ';
18       doubleValue += 1.1;
19    } // end while
20
21    cout << "\nStack is full. Cannot push " << doubleValue
22       << "\n\nPopping elements from doubleStack\n";
23
```

Fig. 14.3 | Class template Stack `test` program. (Part 1 of 3.)

```
24      // pop elements from doubleStack
25      while ( doubleStack.pop( doubleValue ) )
26         cout << doubleValue << ' ';
27
28      cout << "\nStack is empty. Cannot pop\n";
29
30      Stack< int > intStack; // default size 10
31      int intValue = 1;
32      cout << "\nPushing elements onto intStack\n";
33
34      // push 10 integers onto intStack
35      while ( intStack.push( intValue ) )
36      {
37         cout << intValue++ << ' ';
38      } // end while
39
40      cout << "\nStack is full. Cannot push " << intValue
41         << "\n\nPopping elements from intStack\n";
42
43      // pop elements from intStack
44      while ( intStack.pop( intValue ) )
45         cout << intValue << ' ';
46
47      cout << "\nStack is empty. Cannot pop" << endl;
48   } // end main
```

Fig. 14.3 | Class template Stack `test` program. (Part 2 of 3.)

```
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
```

Fig. 14.3 | Class template Stack `test` program. (Part 3 of 3.)

14.4 Class Templates (cont.)

- Line 30 instantiates integer stack `intStack` with the declaration
  - `Stack< int > intStack;`
- Because no size is specified, the size defaults to 10 as specified in the default constructor (Fig. 14.2, line 10).

14.4 Class Templates (cont.)

- Notice that the code in function `main` of Fig. 14.3 is almost identical for both the `double`-`Stack` manipulations in lines 9–28 and the `intStack` manipulations in lines 30–47.
- This presents another opportunity to use a function template.
- Figure 14.4 defines function template `testStack` (lines 10–34) to perform the same tasks as `main` in Fig. 14.3—`push` a series of values onto a `Stack< T >` and `pop` the values off a `Stack< T >`.
- Function template `testStack` uses template parameter `T` (specified at line 10) to represent the data type stored in the `Stack< T >`.
- The function template takes four arguments (lines 12–15)—a reference to an object of type `Stack< T >`, a value of type `T` that will be the first value `push`ed onto the `Stack< T >`, a value of type `T` used to increment the values `push`ed onto the `Stack< T >` and a `string` that represents the name of the `Stack< T >` object for output purposes.
- The output of Fig. 14.4 precisely matches the output of Fig. 14.3.

```
1  // Fig. 14.4: fig14_04.cpp
2  // Stack class template test program. Function main uses a
3  // function template to manipulate objects of type Stack< T >.
4  #include <iostream>
5  #include <string>
6  #include "Stack.h" // Stack class template definition
7  using namespace std;
8
9  // function template to manipulate Stack< T >
10 template< typename T >
11 void testStack(
12    Stack< T > &theStack, // reference to Stack< T >
13    T value, // initial value to push
14    T increment, // increment for subsequent values
15    const string stackName ) // name of the Stack< T > object
16 {
17    cout << "\nPushing elements onto " << stackName << '\n';
18
19    // push element onto Stack
20    while ( theStack.push( value ) )
21    {
22       cout << value << ' ';
23       value += increment;
24    } // end while
```

Fig. 14.4 | Passing a `Stack` template object to a function template. (Part 1 of 3.)

```
25
26       cout << "\nStack is full. Cannot push " << value
27          << "\n\nPopping elements from " << stackName << '\n';
28
29       // pop elements from Stack
30       while ( theStack.pop( value ) )
31          cout << value << ' ';
32
33       cout << "\nStack is empty. Cannot pop" << endl;
34    } // end function template testStack
35
36    int main()
37    {
38       Stack< double > doubleStack( 5 ); // size 5
39       Stack< int > intStack; // default size 10
40
41       testStack( doubleStack, 1.1, 1.1, "doubleStack" );
42       testStack( intStack, 1, 1, "intStack" );
43    } // end main
```

Fig. 14.4 | Passing a `Stack` template object to a function template. (Part 2 of 3.)

```
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
```

Fig. 14.4 | Passing a `Stack` template object to a function template. (Part 3 of 3.)

14.5 Nontype Parameters and Default Types for Class Templates

- Class template `Stack` of Section 14.4 used only a type parameter in the template header (Fig. 14.2, line 6).
- It's also possible to use non-type template parameters, which can have default arguments and are treated as `const`s.
- For example, the template header could be modified to take an `int elements` parameter as follows:
  - ```
    // nontype parameter elements
    template< typename T, int elements >
    ```
- Then, a declaration such as
  - ```
    Stack< double, 100 > mostRecentSalesFigures;
    ```
- could be used to instantiate (at compile time) a 100-element `Stack` class-template specialization of `double` values named `mostRecentSalesFigures`; this class-template specialization would be of type `Stack< double, 100 >`.

14.5 Nontype Parameters and Default Types for Class Templates (cont.)

- The class definition then might contain a `private` data member with an array declaration such as
  - ```
    // array to hold Stack contents
    T stackHolder[ elements ];
    ```
- A type parameter can specify a default type.
- For example,
  - ```
    // defaults to type string
    template< typename T = string >
    ```
- specifies that a `T` is `string` if not specified otherwise.
- Then, a declaration such as
  - `Stack<> jobDescriptions;`
- could be used to instantiate a `Stack` class-template specialization of `string`s named `jobDescriptions`; this class-template specialization would be of type `Stack< string >`.
- Default type parameters must be the rightmost (trailing) parameters in a template's type-parameter list.

Performance Tip 14.2

When appropriate, specify the size of a container class (such as an array class or a stack class) at compile time (possibly through a nontype template parameter). This eliminates the execution-time overhead of using new *to create the space dynamically.*

Software Engineering Observation 14.3

Specifying the size of a container at compile time avoids the potentially fatal execution-time error if new *is unable to obtain the needed memory.*

14.5 Nontype Parameters and Default Types for Class Templates (cont.)

- If a particular user-defined type will not work with a template or requires customized processing, you can define an explicit specialization of the class template for a particular type.
- Let's assume we want to create an explicit specialization `Stack` for `Employee` objects.
- To do this, form a new class with the name `Stack< Employee >` as follows:

```
template<>
class Stack< Employee >
{
   // body of class definition
};
```

- The `Stack<Employee>` explicit specialization is a complete replacement for the `Stack` class template that is specific to type `Employee`.

14.6 Notes on Templates and Inheritance

- Templates and inheritance relate in several ways:
  - A class template can be derived from a class-template specialization.
  - A class template can be derived from a nontemplate class.
  - A class-template specialization can be derived from a class-template specialization.
  - A nontemplate class can be derived from a class-template specialization.

14.7 Notes on Templates and Friends

- With class templates, friendship can be established between a class template and a global function, a member function of another class (possibly a class-template specialization), or even an entire class (possibly a class-template specialization).

14.8 Notes on Templates and `static` Members

- Each class-template specialization instantiated from a class template has its own copy of each `static` data member of the class template; all objects of that specialization share that one `static` data member.
- In addition, as with `static` data members of nontemplate classes, `static` data members of class-template specializations must be defined and, if necessary, initialized at global namespace scope.
- Each class-template specialization gets its own copy of the class template's `static` member functions.