14 Templates

Behind that outside pattern the dim shapes get clearer every day.
It is always the same shape, only very numerous.
—Charlotte Perkins Gilman

Every man of genius sees the world at a different angle from his fellows.
—Havelock Ellis

…our special individuality, as distinguished from our generic humanity.
—Oliver Wendell Holmes, Sr

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.

Outline

14.1 Introduction

In this chapter, we discuss one of C++'s more powerful software reuse features, namely **templates**. **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**.

We might write a single function template for an array-sort function, then have C++ generate separate function-template specializations that will sort `int` arrays, `float` arrays, `string` arrays and so on. We introduced function templates in Chapter 6. We present an additional discussion and example in this chapter.

We might write a single class template for a stack class, then have C++ generate separate class-template specializations, such as a stack-of-`int` class, a stack-of-`float` class, a stack-of-`string` class and so on.

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 tracings 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 template. We also consider the relationships between templates and other C++ features, such as overloading, inheritance, friends and `static` members. The design and details of the template mechanisms discussed here are based on the work of Bjarne Stroustrup as presented in his paper, "Parameterized Types for C++"—published in the *Proceedings of the USENIX C++ Conference* held in Denver, Colorado, in October 1988.

This chapter is only an introduction to templates. Chapter 22, Standard Template Library (STL), presents an in-depth treatment of the template container classes, iterators and algorithms of the STL. Chapter 22 contains dozens of live-code template-based examples illustrating more sophisticated template-programming techniques than those used here.

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 compactly 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. In C, this task can be performed using **macros** created with the preprocessor directive `#define` (see Appendix E, Preprocessor). However, macros can have serious side effects and do not enable the compiler to perform type checking. Function templates provide a compact solution, like macros, but enable full type checking.

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.

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. The function definition follows and appears like any other function definition. 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.

Example: Function Template `printArray`

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 `printArray`; `T` is referred to as a **type template parameter**, or type parameter. You'll see nontype template parameters in Section 14.5.

```
 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
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
```

```
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`.

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. In Fig. 14.1, the compiler creates three printArray specializations—one that expects an int array, one that expects a double array and one that expects a char array. For example, 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 (lines 7–14). The program begins by declaring five-element int array a, seven-element double array b and six-element char array c (lines 22–24, respectively). Then, the program outputs each array by calling printArray—once with a first argument a of type int * (line 29), once with a first argument b of type double * (line 34) and once with a first argument c of type char * (line 39). The call in line 29, for example, causes the compiler to infer that T is int and to instantiate a printArray function-template specialization, for which type parameter T is int. The call in line 34 causes the compiler to infer that T is double and to instantiate a second printArray function-template specialization, for which type parameter T is double. The call in line 39 causes the compiler to infer that T is char and to instantiate a third printArray function-template specialization, for which type parameter T is char. 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.

In this example, the template mechanism saves you from having to write three separate overloaded functions with prototypes

```
void printArray( const int * const, int );
void printArray( const double * const, int );
void printArray( const char * const, int );
```

that all use the same code, except for type T (as used in line 8).

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 generated 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. For example, function template `printArray` of Fig. 14.1 could be overloaded with another `printArray` function template with additional parameters `lowSubscript` and `highSubscript` to specify the portion of the array to output (see Exercise 14.4).

A function template also can be overloaded by providing nontemplate functions with the same function name but different function arguments. For example, function template `printArray` of Fig. 14.1 could be overloaded with a nontemplate version that specifically prints an array of character strings in neat, tabular format (see Exercise 14.5).

The compiler performs a matching process to determine what function to call when a function is invoked. First, the compiler tries to find and use a precise match in which the function names and argument types are consistent with those of the function call. If this fails, the compiler determines whether a function template is available that can be used to generate a function-template specialization with a precise match of function name and argument types. If such a function template is found, the compiler generates and uses the appropriate function-template specialization. If not, the compiler generates an error message. Also, if there are multiple matches for the function call, the compiler considers the call to be ambiguous and the compiler generates an error message.

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 being placed in the stack. However, to instantiate a stack, a data type must be specified. This creates a wonderful opportunity for software reusability. We need the means for describing the notion of a stack generically and instantiating 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.

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.

To produce a variety of class-template specializations you write only one class-template definition. 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. One `Stack` class template, for example, could thus become the basis for creating many `Stack` classes (such as “`Stack` of `double`,” “`Stack` of `int`,” “`Stack` of `char`,” “`Stack` of `Employee`,” etc.) used in a program.

Creating Class Template `Stack< T >`

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. You need not specifically use identifier `T`—any valid identifier can be used. The type of element to be stored on this `Stack` is mentioned generically as `T` throughout the `Stack` class header and member-function definitions. In a moment, we show how `T` becomes associated with a specific type, such as `double` or `int`. 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 (for use in line 44 to create the array that stores the stack elements), and their assignment operators must properly copy objects into the `Stack` (lines 56 and 70).

```
 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
```

Fig. 14.2 | Class template `Stack`. (Part 1 of 3.)

```cpp

   // determine whether Stack is empty
   bool isEmpty() const
   {
      return top == -1;
   } // end function isEmpty

   // determine whether Stack is full
   bool isFull() const
   {
      return top == size - 1;
   } // end function isFull

private:
   int size; // # of elements in the Stack
   int top; // location of the top element (-1 means empty)
   T *stackPtr; // pointer to internal representation of the Stack
}; // end class template Stack

// constructor template
template< typename T >
Stack< T >::Stack( int s )
   : size( s > 0 ? s : 10 ), // validate size
     top( -1 ), // Stack initially empty
     stackPtr( new T[ size ] ) // allocate memory for elements
{
   // empty body
} // end Stack constructor template

// push element onto Stack;
// if successful, return true; otherwise, return false
template< typename T >
bool Stack< T >::push( const T &pushValue )
{
   if ( !isFull() )
   {
      stackPtr[ ++top ] = pushValue; // place item on Stack
      return true; // push successful
   } // end if

   return false; // push unsuccessful
} // end function template push

// pop element off Stack;
// if successful, return true; otherwise, return false
template< typename T >
bool Stack< T >::pop( T &popValue )
{
   if ( !isEmpty() )
   {
      popValue = stackPtr[ top-- ]; // remove item from Stack
      return true; // pop successful
   } // end if
```

Fig. 14.2 | Class template `Stack`. (Part 2 of 3.)

```
73
74    return false; // pop unsuccessful
75 } // end function template pop
76
77 #endif
```

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

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 >
```

(lines 40, 51 and 65). 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 `Stack< T >` (lines 41, 52 and 66) to tie each member-function definition to the class template's scope. In this case, the generic class name is `Stack< T >`. 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 (line 44). The statement

```
stackPtr( new T[ size ] );
```

in the `Stack` class-template definition is generated by the compiler in the class-template specialization `Stack<double>` as

```
stackPtr( new double[ size ] );
```

Creating a Driver to Test Class Template ***`Stack< T >`***

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` (line 9). 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 multiple 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
```

Fig. 14.3 | Class template Stack `test` program. (Part 1 of 2.)

```
   cout << "Pushing elements onto doubleStack\n";

   // push 5 doubles onto doubleStack
   while ( doubleStack.push( doubleValue ) )
   {
      cout << doubleValue << ' ';
      doubleValue += 1.1;
   } // end while

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

   // pop elements from doubleStack
   while ( doubleStack.pop( doubleValue ) )
      cout << doubleValue << ' ';

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

   Stack< int > intStack; // default size 10
   int intValue = 1;
   cout << "\nPushing elements onto intStack\n";

   // push 10 integers onto intStack
   while ( intStack.push( intValue ) )
   {
      cout << intValue++ << ' ';
   } // end while

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

   // pop elements from intStack
   while ( intStack.pop( intValue ) )
      cout << intValue << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;
} // end 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
```

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

Lines 15–19 invoke `push` to place the `double` values 1.1, 2.2, 3.3, 4.4 and 5.5 onto `doubleStack`. The `while` loop terminates when the driver attempts to `push` a sixth value onto `doubleStack` (which is full, because it holds a maximum of five elements). Function `push` returns `false` when it's unable to push a value onto the stack.[1]

Lines 25–26 invoke `pop` in a `while` loop to remove the five values from the stack (note, in the output of Fig. 14.3, that the values do `pop` off in last-in, first-out order). When the driver attempts to `pop` a sixth value, the `doubleStack` is empty, so the `pop` loop terminates.

Line 30 instantiates integer stack `intStack` with the declaration

```
    Stack< int > intStack;
```

(pronounced "`intStack` is a `Stack` of `int`"). Because no size is specified, the size defaults to `10` as specified in the default constructor (Fig. 14.2, line 10). Lines 35–38 loop and invoke `push` to place values onto `intStack` until it's full, then lines 44–45 loop and invoke `pop` to remove values from `intStack` until it's empty. Once again, notice in the output that the values `pop` off in last-in, first-out order.

Creating Function Templates to Test Class Template `Stack< T >`

Notice that the code in function `main` of Fig. 14.3 is almost identical for both the `doubleStack` 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. Function `main` (lines 36–43) instantiates an object of type `Stack< double >` called `doubleStack` (line 38) and an object of type `Stack< int >` called `intStack` (line 39) and uses these objects in lines 41 and 42. The compiler infers the type of `T` for `testStack` from the type used to instantiate the function's first argument (i.e., the type used to instantiate `doubleStack` or `intStack`). 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
```

Fig. 14.4 | Passing a `Stack` template object to a function template. (Part 1 of 2.)

1. Class `Stack` (Fig. 14.2) provides the function `isFull`, which you can use to determine whether the stack is full before attempting a push operation. This would avoid the potential error of pushing onto a full stack. As we discuss in Chapter 16, Exception Handling, if the operation cannot be completed, function `push` would "throw an exception." You can write code to "catch" that exception, then decide how to handle it appropriately for the application. The same technique can be used with function `pop` when an attempt is made to pop an element from an empty stack.

```
using namespace std;

// function template to manipulate Stack< T >
template< typename T >
void testStack(
   Stack< T > &theStack, // reference to Stack< T >
   T value, // initial value to push
   T increment, // increment for subsequent values
   const string stackName ) // name of the Stack< T > object
{
   cout << "\nPushing elements onto " << stackName << '\n';

   // push element onto Stack
   while ( theStack.push( value ) )
   {
      cout << value << ' ';
      value += increment;
   } // end while

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

   // pop elements from Stack
   while ( theStack.pop( value ) )
      cout << value << ' ';

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

int main()
{
   Stack< double > doubleStack( 5 ); // size 5
   Stack< int > intStack; // default size 10

   testStack( doubleStack, 1.1, 1.1, "doubleStack" );
   testStack( intStack, 1, 1, "intStack" );
} // end 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
```

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

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:

```
template< typename T, int elements > // nontype parameter 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 >`. The class definition then might contain a `private` data member with an array declaration such as

```
T stackHolder[ elements ]; // array to hold Stack contents
```

In addition, a type parameter can specify a **default type**. For example,

```
template< typename T = string > // defaults to type string
```

might specify that a `Stack` contains `string` objects by default. 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. When one is instantiating a class with two or more default types, if an omitted type is not the rightmost type parameter in the type-parameter list, then all type parameters to the right of that type also must be omitted.

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.

In the exercises, you'll be asked to use a nontype parameter to create a template for our class `Array` from Chapter 11. This template will enable `Array` objects to be instantiated with a specified number of elements of a specified type at compile time, rather than creating space for the `Array` objects at execution time.

In some cases, it may not be possible to use a particular type with a class template. For example, the `Stack` template of Fig. 14.2 requires that user-defined types that will be stored in a `Stack` must provide a default constructor and an assignment operator that properly copies objects. If a particular user-defined type will not work with our Stack 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—it does not use anything from the original class template and can even have different members.

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

We've seen that functions and entire classes can be declared as friends of nontemplate classes. 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).

Throughout this section, we assume that we've defined a class template for a class named X with a single type parameter T, as in:

```
template< typename T > class X
```

Under this assumption, it's possible to make a function f1 a friend of every class-template specialization instantiated from the class template for class X. To do so, use a friendship declaration of the form

```
friend void f1();
```

For example, function f1 is a friend of X< double >, X< string > and X< Employee >, etc.

It's also possible to make a function f2 a friend of only a class-template specialization with the same type argument. To do so, use a friendship declaration of the form

```
friend void f2( X< T > & );
```

For example, if T is a float, function f2(X< float > &) is a friend of class-template specialization X< float > but not a friend of class-template specification X< string >.

You can declare that a member function of another class is a friend of any class-template specialization generated from the class template. To do so, the friend declaration must qualify the name of the other class's member function using the class name and the binary scope resolution operator, as in:

```
        friend void A::f3();
```

The declaration makes member function `f3` of class `A` a friend of every class-template specialization instantiated from the preceding class template. For example, function `f3` of class `A` is a friend of `X< double >`, `X< string >` and `X< Employee >`, etc.

As with a global function, another class's member function can be a friend of only a class-template specialization with the same type argument. A friendship declaration of the form

```
        friend void C< T >::f4( X< T > & );
```

for a particular type `T` such as `float` makes class `C`'s member function

```
        C< float >::f4( X< float > & )
```

a friend function of *only* class-template specialization `X< float >`.

In some cases, it's desirable to make an entire class's set of member functions friends of a class template. In this case, a friend declaration of the form

```
        friend class Y;
```

makes every member function of class `Y` a friend of every class-template specialization produced from the class template `X`.

Finally, it's possible to make all member functions of one class-template specialization friends of another class-template specialization with the same type argument. For example, a friend declaration of the form:

```
        friend class Z< T >;
```

indicates that when a class-template specialization is instantiated with a particular type for `T` (such as `float`), all members of `class Z< float >` become friends of class-template specialization `X< float >`. We use this particular relationship in Chapter 20, Data Structures.

14.8 Notes on Templates and `static` Members

What about `static` data members? Recall that, with a nontemplate class, one copy of each `static` data member is shared among all objects of the class, and the `static` data member must be initialized at global namespace scope.

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.

14.9 Wrap-Up

This chapter introduced one of C++'s most powerful features—templates. You learned how to use function templates to enable the compiler to produce a set of function-template specializations that represent a group of related overloaded functions. We also discussed how to overload a function template to create a specialized version of a function that handles a particular data type's processing in a manner that differs from the other function-template specializations. Next, you learned about class templates and class-template specializations. You saw examples of how to use a class template to create a group of related types that each perform identical processing on different data types. Finally, you learned about some of the relationships among templates, friends, inheritance and `static` members. In the next chapter, we discuss many of C++'s I/O capabilities and demonstrate several stream manipulators that perform various formatting tasks.

Summary

Section 14.1 Introduction

- Templates enable us to specify a range of related (overloaded) functions—called function-template specializations—or a range of related classes—called class-template specializations.

Section 14.2 Function Templates

- To use function-template specializations, you write a single function-template definition. Based on the argument types provided in calls to this function, C++ generates separate specializations to handle each type of call appropriately.
- All function-template definitions begin with the keyword `template` followed by template parameters enclosed in angle brackets (< and >); each template parameter that represents a type must be preceded by keyword `class` or `typename`. Keywords `typename` and `class` used to specify function-template parameters mean "any fundamental type or user-defined type."
- Template-definition template parameters are used to specify the kinds of arguments to the function, the return type of the function and to declare variables in the function.
- 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.

Section 14.3 Overloading Function Templates

- 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. A function template can also be overloaded by providing other nontemplate functions with the same function name, but different function parameters. If both the template and non-template versions match a call, the non-template version will be used.

Section 14.4 Class Templates

- Class templates provide the means for describing a class generically and for instantiating classes that are type-specific versions of this generic class.
- Class templates are called parameterized types; they require type parameters to specify how to customize a generic class template to form a specific class-template specialization.
- To use class-template specializations you write one class template. When you need a new type-specific class, the compiler writes the source code for the class-template specialization.
- A class-template definition looks like a conventional class definition, except that it's preceded by `template< typename T >` (or `template< class T >`) to indicate this is a class-template definition. Type parameter `T` acts as a placeholder for the type of the class to create. The type `T` is mentioned throughout the class definition and member-function definitions as a generic type name.

- Member-function definitions outside a class template each begin with `template<typename T>` (or `template<class T>`). Then, each function definition resembles a conventional function definition, except that the generic data in the class 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.

Section 14.5 Nontype Parameters and Default Types for Class Templates

- It's possible to use nontype parameters in the header of a class or function template.
- You can specify a default type for a type parameter in the type-parameter list.
- An explicit specialization of a class template overrides a class template for a specific type.

Section 14.6 Notes on Templates and Inheritance

- 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.

Section 14.7 Notes on Templates and Friends

- Functions and entire classes can be declared as friends of nontemplate classes. With class templates, friendship arrangements can be declared. 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).

Section 14.8 Notes on Templates and `static` Members

- Each class-template specialization has its own copy of each `static` data member; all objects of that specialization share that `static` data member. Such data members must be defined and, if necessary, initialized at global namespace scope.
- Each class-template specialization gets a copy of the class template's `static` member functions.

Terminology

angle brackets (< and >) 628
`class` keyword in a template type parameter 628
class template 627
class-template definition 632
class-template specialization 627
default type for a type parameter 638
explicit specialization 639
`friend` of a template 639
function template 627
function-template definition 628
function-template specialization 627
generic programming 627
macro 628
member function of a class-template specialization 639
non-type template parameter 638
overloading a function template 631
parameterized type 632
`static` data member of a class template 640
`static` data member of a class-template specialization 640
`static` member function of a class template 640
`static` member function of a class-template specialization 640
template 627
`template` keyword 628
template parameter 628
type parameter 628
type template parameter 628
`typename` keyword 628

Self-Review Exercises

14.1 State which of the following are *true* and which are *false*. If *false*, explain why.

a) The 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.

b) Keywords `typename` and `class` as used with a template type parameter specifically mean "any user-defined class type."
c) A function template can be overloaded by another function template with the same function name.
d) Template parameter names among template definitions must be unique.
e) Each member-function definition outside a class template must begin with a template header.
f) A `friend` function of a class template must be a function-template specialization.
g) If several class-template specializations are generated from a single class template with a single `static` data member, each of the class-template specializations shares a single copy of the class template's `static` data member.

14.2 Fill in the blanks in each of the following:
a) Templates enable us to specify, with a single code segment, an entire range of related functions called ________, or an entire range of related classes called ________.
b) All function-template definitions begin with the keyword ________, followed by a list of template parameters to the function template enclosed in ________.
c) The related functions generated from a function template all have the same name, so the compiler uses ________ resolution to invoke the proper function.
d) Class templates also are called ________ types.
e) The ________ operator is used with a class-template name to tie each member-function definition to the class template's scope.
f) As with `static` data members of nontemplate classes, `static` data members of class-template specializations must also be defined and, if necessary, initialized at ________ scope.

Answers to Self-Review Exercises

14.1 a) True. b) False. Keywords `typename` and `class` in this context also allow for a type parameter of a fundamental type. c) True. d) False. Template parameter names among function templates need not be unique. e) True. f) False. It could be a nontemplate function. g) False. Each class-template specialization will have its own copy of the `static` data member.

14.2 a) function-template specializations, class-template specializations. b) `template`, angle brackets (< and >). c) overloading. d) parameterized. e) binary scope resolution. f) global namespace.

Exercises

14.3 *(Selection Sort Function Template)* Write a function template `selectionSort` based on Fig. 8.13. Write a driver program that inputs, sorts and outputs an `int` array and a `float` array.

14.4 *(Print Array Range)* Overload function template `printArray` of Fig. 14.1 so that it takes two additional integer arguments, namely `int lowSubscript` and `int highSubscript`. A call to this function will print only the designated portion of the array. Validate `lowSubscript` and `highSubscript`; if either is out of range or if `highSubscript` is less than or equal to `lowSubscript`, the overloaded `printArray` function should return 0; otherwise, `printArray` should return the number of elements printed. Then modify `main` to exercise both versions of `printArray` on arrays `a`, `b` and `c` (lines 22–24 of Fig. 14.1). Be sure to test all capabilities of both versions of `printArray`.

14.5 *(Function Template Overloading)* Overload function template `printArray` of Fig. 14.1 with a nontemplate version that prints an array of character strings in neat, tabular, column format.

14.6 *(Operator Overloads in Templates)* Write a simple function template for predicate function `isEqualTo` that compares its two arguments of the same type with the equality operator (==) and returns `true` if they are equal and `false` otherwise. Use this function template in a program that calls `isEqualTo` only with a variety of fundamental types. Now write a separate version of the program that calls `isEqualTo` with a user-defined class type, but does not overload the equality operator. What happens when you attempt to run this program? Now overload the equality operator (with the operator function) `operator==`. Now what happens when you attempt to run this program?

14.7 ***(Array Class Template)*** Use an `int` template nontype parameter `numberOfElements` and a type parameter `elementType` to help create a template for the `Array` class (Figs. 11.6–11.7) we developed in Chapter 11. This template will enable `Array` objects to be instantiated with a specified number of elements of a specified element type at compile time.

Write a program with class template `Array`. The template can instantiate an `Array` of any element type. Override the template with a specific definition for an `Array` of `float` elements (`class Array<float>`). The driver should demonstrate the instantiation of an `Array` of `int` through the template and should show that an attempt to instantiate an `Array` of `float` uses the definition provided in `class Array<float>`.

14.8 Distinguish between the terms "function template" and "function-template specialization."

14.9 Explain which is more like a stencil—a class template or a class-template specialization?

14.10 What's the relationship between function templates and overloading?

14.11 Why might you choose to use a function template instead of a macro?

14.12 What performance problem can result from using function templates and class templates?

14.13 The compiler performs a matching process to determine which function-template specialization to call when a function is invoked. Under what circumstances does an attempt to make a match result in a compile error?

14.14 Why is it appropriate to refer to a class template as a parameterized type?

14.15 Explain why a C++ program would use the statement

```
Array< Employee > workerList( 100 );
```

14.16 Review your answer to Exercise 14.15. Explain why a C++ program might use the statement

```
Array< Employee > workerList;
```

14.17 Explain the use of the following notation in a C++ program:

```
template< typename T > Array< T >::Array( int s )
```

14.18 Why might you use a nontype parameter with a class template for a container such as an array or stack?

14.19 Suppose that a class template has the header

```
template< typename T > class Ct1
```

Describe the friendship relationships established by placing each of the following `friend` declarations inside this class template. Identifiers beginning with "`f`" are functions, identifiers beginning with "`C`" are classes, identifiers beginning with "`Ct`" are class templates and `T` is a template type parameter (i.e., `T` can represent any fundamental or class type).

a) `friend void f1();`
b) `friend void f2( Ct1< T > & );`
c) `friend void C2::f3();`
d) `friend void Ct3< T >::f4( Ct1< T > & );`
e) `friend class C4;`
f) `friend class Ct5< T >;`

14.20 Suppose that class template `Employee` has a `static` data member `count`. Suppose that three class-template specializations are instantiated from the class template. How many copies of the `static` data member will exist? How will the use of each be constrained (if at all)?