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 will 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 a group 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.

Self-Review Exercises

14.1 State which of the following statements are *true* and which are *false*. If a statement is *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.

ANS: True.

b) Keywords `typename` and `class` as used with a template type parameter specifically mean "any user-defined class type."

ANS: False. Keywords `typename` and `class` in this context also allow for a type parameter of a built-in type.

c) A function template can be overloaded by another function template with the same function name.

ANS: True.

d) Template parameter names among template definitions must be unique.

ANS: False. Template parameter names among function templates need not be unique.

e) Each member-function definition outside a class template must begin with a template header.

ANS: True.

f) A `friend` function of a class template must be a function-template specialization.

ANS: False. It could be a nontemplate function.

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.

ANS: False. Each class-template specialization will have its own copy of the `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 _____.

ANS: function-template specializations, class-template specializations.

b) All function-template definitions begin with the keyword _____, followed by a list of template parameters to the function template enclosed in _____.

ANS: `template`, angle brackets (< and >).

c) The related functions generated from a function template all have the same name, so the compiler uses _____ resolution to invoke the proper function.

ANS: overloading.

d) Class templates also are called _____ types.

ANS: parameterized.

e) The _____ operator is used with a class-template name to tie each member-function definition to the class template's scope.

ANS: binary scope resolution.

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.

ANS: file.

Solutions

14.3 Write a function template `selectionSort` based on the sort program of Fig. 8.15. Write a driver program that inputs, sorts and outputs an `int` array and a `float` array.

ANS:

```
// Exercise 14.3 solution: Ex14_03.cpp
// This program puts values into an array, sorts the values into
// ascending order, and prints the resulting array.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

// function template for insertionSort
template < typename T >
void insertionSort( T * const array, int size )
{
   T insert; // temporary variable to hold element to insert

   // loop over size elements
   for ( int next = 0; next < size; next++ )
   {
      insert = array[ next ]; // store value in current element
      int moveItem = next; // initialize location to place element

      // search for place to put current element
      while ( moveItem > 0 && array[ moveItem - 1 ] > insert )
      {
         // shift element right one slot
         array[ moveItem ] = array[ moveItem - 1 ];
         moveItem--;
      } // end while

      array[ moveItem ] = insert; // place inserted element
   } // end for
} // end function template insertionSort

// function template for printArray
template < typename T >
void printArray( T * const array, int size )
{
   for ( int i = 0; i < size; i++ )
      cout << setw( 6 ) << array[ i ];
} // end function template printArray

int main()
{
   const int SIZE = 10; // size of array
   int a[ SIZE ] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

   // display int array in original order
   cout << "int data items in original order\n";
```

```
50      printArray( a, SIZE ); // print int array
51      insertionSort( a, SIZE ); // sort int array
52
53      // display int array in sorted order
54      cout << "\nint data items in ascending order\n";
55      printArray( a, SIZE );
56      cout << "\n\n";
57
58      // initialize double array
59      double b[ SIZE ] =
60         { 10.1, 9.9, 8.8, 7.7, 6.6, 5.5, 4.4, 3.3, 2.2, 1.1 };
61
62      // display double array in original order
63      cout << "double point data items in original order\n";
64      printArray( b, SIZE ); // print double array
65      insertionSort( b, SIZE ); // sort double array
66
67      // display sorted double array
68      cout << "\ndouble point data items in ascending order\n";
69      printArray( b, SIZE );
70      cout << endl;
71      return 0;
72   } // end main
```

```
int data items in original order
    10     9     8     7     6     5     4     3     2     1
int data items in ascending order
     1     2     3     4     5     6     7     8     9    10

double point data items in original order
  10.1   9.9   8.8   7.7   6.6   5.5   4.4   3.3   2.2   1.1
double point data items in ascending order
   1.1   2.2   3.3   4.4   5.5   6.6   7.7   8.8   9.9  10.1
```

14.4 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 23–25 of Fig. 14.1). Be sure to test all capabilities of both versions of `printArray`.

ANS:

```
1   // Exercise 14.4 solution: Ex14_04.cpp
2   // Using template functions
3   #include <iostream>
4   using std::cout;
5   using std::endl;
6
7   // function template printArray definition
8   // original function from Fig. 14.1
9   template< typename T >
10  void printArray( const T *array, int count )
```

```
{
   // display array
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

   cout << endl;
} // end function printArray

// overloaded function template printArray
// takes upper and lower subscripts to print
template< typename T >
int printArray( T const * array, int size, int lowSubscript,
   int highSubscript )
{
   // check if subscript is negative or out of range
   if ( size < 0 || lowSubscript < 0 || highSubscript >= size )
      return 0;

   // display array
   for ( int i = lowSubscript, count = 0; i <= highSubscript; i++ )
   {
      count++;
      cout << array[ i ] << ' ';
   } // end for

   cout << '\n';
   return count; // number or elements output
} // end overloaded function printArray

int main()
{
   const int ACOUNT = 5; // size of array a
   const int BCOUNT = 7; // size of array b
   const int CCOUNT = 6; // size of array c

   // declare and initialize arrays
   int a[ ACOUNT ] = { 1, 2, 3, 4, 5 };
   double b[ BCOUNT ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
   char c[ CCOUNT ] = "HELLO"; // 6th position for null
   int elements;

   // display array a using original printArray function
   cout << "\nUsing original printArray function\n";
   printArray( a, ACOUNT );

   // display array a using new printArray function
   cout << "Array a contains:\n";
   elements = printArray( a, ACOUNT, 0, ACOUNT - 1 );
   cout << elements << " elements were output\n";

   // display elements 1-3 of array a
   cout << "Array a from positions 1 to 3 is:\n";
   elements = printArray( a, ACOUNT, 1, 3 );
   cout << elements << " elements were output\n";

```

```
66     // try to print an invalid element
67     cout << "Array a output with invalid subscripts:\n";
68     elements = printArray( a, ACOUNT, -1, 10 );
69     cout << elements << " elements were output\n\n";
70
71     // display array b using original printArray function
72     cout << "\nUsing original printArray function\n";
73     printArray( b, BCOUNT );
74
75     // display array b using new printArray function
76     cout << "Array b contains:\n";
77     elements = printArray( b, BCOUNT, 0, BCOUNT - 1 );
78     cout << elements << " elements were output\n";
79
80     // display elements 1-3 of array b
81     cout << "Array b from positions 1 to 3 is:\n";
82     elements = printArray( b, BCOUNT, 1, 3 );
83     cout << elements << " elements were output\n";
84
85     // try to print an invalid element
86     cout << "Array b output with invalid subscripts:\n";
87     elements = printArray( b, BCOUNT, -1, 10 );
88     cout << elements << " elements were output\n\n";
89
90     // display array c using original printArray function
91     cout << "\nUsing original printArray function\n";
92     printArray( c, CCOUNT );
93
94     // display array c using new printArray function
95     cout << "Array c contains:\n";
96     elements = printArray( c, CCOUNT, 0, CCOUNT - 2 );
97     cout << elements << " elements were output\n";
98
99     // display elements 1-3 of array c
100    cout << "Array c from positions 1 to 3 is:\n";
101    elements = printArray( c, CCOUNT, 1, 3 );
102    cout << elements << " elements were output\n";
103
104    // try to display an invalid element
105    cout << "Array c output with invalid subscripts:\n";
106    elements = printArray( c, CCOUNT, -1, 10 );
107    cout << elements << " elements were output" << endl;
108    return 0;
109 } // end main
```

```
Using original printArray function
1 2 3 4 5
Array a contains:
1 2 3 4 5
5 elements were output
Array a from positions 1 to 3 is:
2 3 4
3 elements were output
Array a output with invalid subscripts:
0 elements were output


Using original printArray function
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
7 elements were output
Array b from positions 1 to 3 is:
2.2 3.3 4.4
3 elements were output
Array b output with invalid subscripts:
0 elements were output


Using original printArray function
H E L L O
Array c contains:
H E L L O
5 elements were output
Array c from positions 1 to 3 is:
E L L
3 elements were output
Array c output with invalid subscripts:
0 elements were output
```

14.5 Overload function template `printArray` of Fig. 14.1 with a nontemplate version that specifically prints an array of character strings in neat, tabular, column format.

ANS:

```
1  // Exercise 14.5 Solution: Ex14_05.cpp
2  // Using template functions
3  #include <iostream>
4  using std::cout;
5
6  #include <iomanip>
7  using std::setw;
8
9  #include <string>
10 using std::string;
11
12 // function template printArray
13 template< typename T >
14 void printArray( T const * array, int size )
15 {
```

```
16    // display elements of array
17    for ( int i = 0; i < size; i++ )
18       cout << array[ i ] << ' ';
19
20    cout << '\n';
21 } // end function printArray
22
23 // function that prints array of strings in tabular format
24 void printArray( const string stringArray[], int size )
25 {
26    // display elements of array
27    for ( int i = 0; i < size; i++ )
28    {
29       cout << setw( 10 ) << stringArray[ i ];
30
31       if ( ( i + 1 ) % 4 == 0 ) // create rows
32          cout << '\n';
33    } // end for
34
35    cout << '\n';
36 } // end function printArray
37
38 int main()
39 {
40    const int ACOUNT = 5; // size of array a
41    const int BCOUNT = 7; // size of array b
42    const int CCOUNT = 6; // size of array c
43    const int SCOUNT = 8; // size of array strings
44
45    // initialize arrays
46    int a[ ACOUNT ] = { 1, 2, 3, 4, 5 };
47    double b[ BCOUNT ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
48    char c[ CCOUNT ] = "HELLO"; // 6th position for null
49    string strings[ SCOUNT ] =
50       { "one", "two", "three", "four", "five", "six", "seven", "eight" };
51
52    cout << "Array a contains:\n";
53    printArray( a, ACOUNT ); // integer template function
54
55    cout << "\nArray b contains:\n";
56    printArray( b, BCOUNT ); // float template function
57
58    cout << "\nArray c contains:\n";
59    printArray( c, CCOUNT ); // character template function
60
61    cout << "\nArray strings contains:\n";
62    printArray( strings, SCOUNT ); // function specific to string arrays
63    return 0;
64 } // 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

Array strings contains:
       one       two     three      four
      five       six     seven     eight
```

14.6 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` if they are not equal. Use this function template in a program that calls `isEqualTo` only with a variety of built-in 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?

ANS: If the user-defined class does not overload the equality operator (i.e., comment out lines 30–33 in the solution below), compiler would report an error indicating that the class does not define this operator or a conversion to a type acceptable to the predefined operator. Once the class overloads the equality operator, the program should run fine.

```
1  // Exercise 14.6 Solution: Ex14_06.cpp
2  // Combined solution to entire problem
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::ostream;
7
8  // function template isEqualTo
9  template < typename T >
10 bool isEqualTo( const T &arg1, const T &arg2 )
11 {
12    return arg1 == arg2;
13 } // end function isEqualTo
14
15 // class used to demonstrate overloading operators
16 // is necessary when using templates
17 class Complex
18 {
19    friend ostream &operator<<(ostream &, Complex &);
20 public:
21    // constructor for Fraction
22    Complex( int realPart, int iPart )
23       : real( realPart ),
24         imaginary( iPart )
25    {
26       // empty body
```

```
27     } // end Complex constructor
28
29     // Overloaded equality operator. If this is not provided, the
30     // program will not compile.
31     bool operator==( const Complex &right ) const
32     {
33        return real == right.real && imaginary == right.imaginary;
34     } // end overloaded equality operator
35  private:
36     int real; // real part of the complex number
37     int imaginary; // imaginary part of the complex number
38  }; // end class Fraction
39
40  // overloaded << operator
41  ostream &operator<<( ostream &out, Complex &obj )
42  {
43     if ( obj.imaginary > 0 ) // positive imaginary
44        out << obj.real << " + " << obj.imaginary << "i";
45     else if ( obj.imaginary == 0 ) // zero imaginary
46        out << obj.real;
47     else // negative imaginary
48        out << obj.real << " - " << -obj.imaginary << "i";
49
50     return out;
51  } // end overloaded << operator
52
53  int main()
54  {
55     int a; // integers used for
56     int b; // testing equality
57
58     // test if two ints input by user are equal
59     cout << "Enter two integer values: ";
60     cin >> a >> b;
61     cout << a << " and " << b << " are "
62        << ( isEqualTo( a, b ) ? "equal" : "not equal" ) << '\n';
63
64     char c; // chars used for
65     char d; // testing equality
66
67     // test if two chars input by user are equal
68     cout << "\nEnter two character values: ";
69     cin >> c >> d;
70     cout << c << " and " << d << " are "
71        << ( isEqualTo( c, d ) ? "equal" : "not equal" ) << '\n';
72
73     double e; // double values used for
74     double f; // testing equality
75
76     // test if two doubles input by user are equal
77     cout << "\nEnter two double values: ";
78     cin >> e >> f;
79     cout << e << " and " << f << " are "
80        << ( isEqualTo( e, f ) ? "equal" : "not equal") << '\n';
81
```

```
82      Complex g( 10, 5 ); // Complex objects used
83      Complex h( 10, 5 ); // for testing equality
84
85      // test if two Complex objects are equal
86      // uses overloaded << operator
87      cout << "\nThe class objects " << g << " and " << h << " are "
88         << ( isEqualTo( g, h ) ? "equal" : "not equal" ) << '\n';
89      return 0;
90   } // end main
```

```
Enter two integer values: 2 5
2 and 5 are not equal

Enter two character values: a a
a and a are equal

Enter two double values: 2.5 7.5
2.5 and 7.5 are not equal

The class objects 10 + 5i and 10 + 5i are equal
```

14.7 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.

ANS: .

```
1   // Exercise 14.7 Solution: Array.h
2   // Class template Array definition.
3   #ifndef ARRAY1_H
4   #define ARRAY1_H
5
6   #include <iostream>
7   using std::cin;
8   using std::cout;
9
10  template < typename elementType, int numberOfElements >
11  class Array
12  {
13  public:
14     Array(); // default constructor
15     ~Array(); // destructor
16     int getSize() const; // return size
17     bool operator==( const Array & ) const; // compare equal
18     bool operator!=( const Array & ) const; // compare !equal
19     elementType &operator[]( int ); // subscript operator
20     static int getArrayCount(); // return count of arrays instantiated
21     void inputArray(); // input the array elements
22     void outputArray() const; // output the array elements
23  private:
24     elementType elements[ numberOfElements ]; // array of elements
25     static int arrayCount; // # of Arrays instantiated
26  }; // end class Array
```

```
27
28 // define static data member at file scope
29 template < typename elementType, int numberOfElements >
30 int Array< elementType, numberOfElements >::arrayCount = 0; // no objects
31
32 // default constructor for class Array
33 template < typename elementType, int numberOfElements >
34 Array< elementType, numberOfElements >::Array()
35 {
36    arrayCount++; // count one more object
37
38    // initialize array
39    for ( int i = 0; i < numberOfElements; i++ )
40       elements[ i ] = elementType();
41 } // end Array constructor
42
43 // destructor for class Array
44 template < typename elementType, int numberOfElements >
45 Array< elementType, numberOfElements >::~Array()
46 {
47    arrayCount--;
48 } // end Array destructor
49
50 // get the size of the array
51 template < typename elementType, int numberOfElements >
52 int Array< elementType, numberOfElements >::getSize() const
53 {
54    return numberOfElements;
55 } // end function getSize
56
57 // determine if two arrays are equal and return true or false
58 template < class elementType, int numberOfElements >
59 bool Array< elementType, numberOfElements >::
60    operator==( const Array &right ) const
61 {
62    // return false if arrays not equal
63    for ( int i = 0; i < numberOfElements; i++ )
64    {
65       if ( elements[ i ] != right.elements[ i ] )
66          return false;
67    } // end for
68
69    return true; // arrays are equal
70 } // end overloaded == operator
71
72 // determine if two arrays are not equal and return true or false
73 template < typename elementType, int numberOfElements >
74 bool Array< elementType, numberOfElements >::
75    operator!=( const Array &right ) const
76 {
77    // return false if arrays not equal
78    for ( int i = 0; i < numberOfElements; i++ )
79    {
80       if ( elements[ i ] != right.elements[ i ] )
81          return true;
```

```
   } // end for

   return false; // arrays are equal
} // end overloaded != operator

// overloaded subscript operator
template < typename elementType, int numberOfElements >
elementType &Array< elementType, numberOfElements >::
   operator[]( int subscript )
{
   // check for subscript
   assert( 0 <= subscript && subscript < numberOfElements );
   return elements[ subscript ]; // reference return creates lvalue
} // end overloaded subscript operator

// return the number of Array objects instantiated
template < typename elementType, int numberOfElements >
int Array< elementType, numberOfElements >::getArrayCount()
{
   return arrayCount;
} // end function getArrayCount

// input values for entire array.
template < typename elementType, int numberOfElements >
void Array< elementType, numberOfElements >::inputArray()
{
   // get values of array from user
   for ( int i = 0; i < numberOfElements; i++ )
      cin >> elements[ i ];
} // end function inputArray

// Output the array values
template < typename elementType, int numberOfElements >
void Array< elementType, numberOfElements >::outputArray() const
{
   int i;

   // output array
   for ( i = 0; i < numberOfElements; i++ )
   {
      cout << elements[ i ] << ' ';

      // form rows for output
      if ( ( i + 1 ) % 10 == 0 )
         cout << '\n';
   } // end for

   if ( i % 10 != 0 )
      cout << '\n';
} // end function outputArray

#endif
```

```
// Exercise 14.7 Solution: Ex14_07.cpp
```

```
// main.cpp
#include <iostream>
using std::cout;

#include <string>
using std::string;

#include "Array.h"

int main()
{
   Array< int, 5 > intArray; // create intArray object

   // initialize intArray with user input values
   cout << "Enter " << intArray.getSize() << " integer values:\n";
   intArray.inputArray();

   // output intArray
   cout << "\nThe values in intArray are:\n";
   intArray.outputArray();

   Array< string, 7 > stringArray; // create stringArray

   // initialize floatArray with user input values
   cout << "\nEnter " << stringArray.getSize()
      << " one-word string values:\n";
   stringArray.inputArray();

   // output stringArray
   cout << "\nThe values in the stringArray are:\n";
   stringArray.outputArray();
   return 0;
} // end main
```

```
Enter 5 integer values:
99 98 97 96 95

The values in intArray are:
99 98 97 96 95

Enter 7 one-word string values:
one two three four five six seven

The values in the stringArray are:
one two three four five six seven
```

14.8 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 >`.

14.9 Distinguish between the terms "function template" and "function-template specialization."
ANS: A function template is used to instantiate function-template specializations.

14.10 Which is more like a stencil—a class template or a class-template specialization? Explain your answer.

ANS: A class template can be viewed as a stencil from which class-template specializations can be created. A class-template specialization can be viewed as a stencil from which objects of that class can be created. So, in a way, both can be viewed as stencils.

14.11 What is the relationship between function templates and overloading?

ANS: Function templates create function-template specializations—these are overloaded versions of a function. The main difference is at compile time, where the compiler automatically creates the code for the template functions from the function template rather than the programmer having to write the code.

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

ANS: A macro is a text substitution done by the preprocessor. 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.

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

ANS: There can be a proliferation of code in the program due to many copies of code generated by the compiler; this can occupy significant memory.

14.14 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?

ANS: 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, there is an error.

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

ANS: When creating specializations of a class template, it is necessary to provide a type (or possibly several types) to complete the definition of the new type being declared. For example, when creating an "array of integers" from an `Array` class template, the type `int` can be provided to the class template to complete the definition of an array of integers.

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

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

ANS: When creating class-template specializations from a class template, it is necessary to provide a type (or possibly several types) to complete the definition of the new type being declared. For example, when creating an "array of `Employees`" from an `Array` class template, the type `Employee` is provided to the class template to complete the definition of an array of `Employees`.

14.17 Review your answer to Exercise 14.16. Why might a C++ program use the statement

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

ANS: Declares an `Array` object to store `Employees`. The default constructor is used.

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

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

ANS: This notation is used to begin the definition of the `Array( int )` constructor for the class template `Array< T >`.

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

ANS: To specify at compile time the size of the container class object being declared.

14.20 Describe how to provide an explicit specialization of a class template.

14.21 Describe the relationship between class templates and inheritance.

14.22 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();`

ANS: Function `f1` is a `friend` of every class template specialization instantiated from class template `Ct1`.

b) `friend void f2( Ct1< T > & );`

ANS: Function `f2` receives a parameter that is a reference to a specialization of class template `Ct1`. The function is a `friend` of only that specialization. For example, if the parameter type is `Ct1< int > &`, then function `f2` is a `friend` of the specialization `Ct1< int >`.

c) `friend void C2::f3();`

ANS: Member function `f3` of class `C2` is a `friend` of every class template specialization instantiated from class template `Ct1`.

d) `friend void Ct3< T >::f4( Ct1< T > & );`

ANS: Member function `f4` of a given specialization of class template `Ct3` receives a parameter that is a reference to a specialization of class template `Ct1` with the same template argument. The function is a `friend` of only that specialization. For example, if the parameter type is `Ct1< int > &`, then function `f4` is a `friend` of the specialization `Ct1< int >`.

e) `friend class C4;`

ANS: Every member function of class `C4` is a `friend` of every class template specialization instantiated from the class template `Ct1`.

f) `friend class Ct5< T >;`

ANS: For a given class template specialization `Ct5< T >`, every member function is a `friend` of the specialization of class template `Ct1` with the same template argument. For example, if `T` is `int`, every member function of class template specialization `Ct5< int >` is a `friend` of the class template specialization `Ct1< int >`.

14.23 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)?

ANS: 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 initialized at file scope.