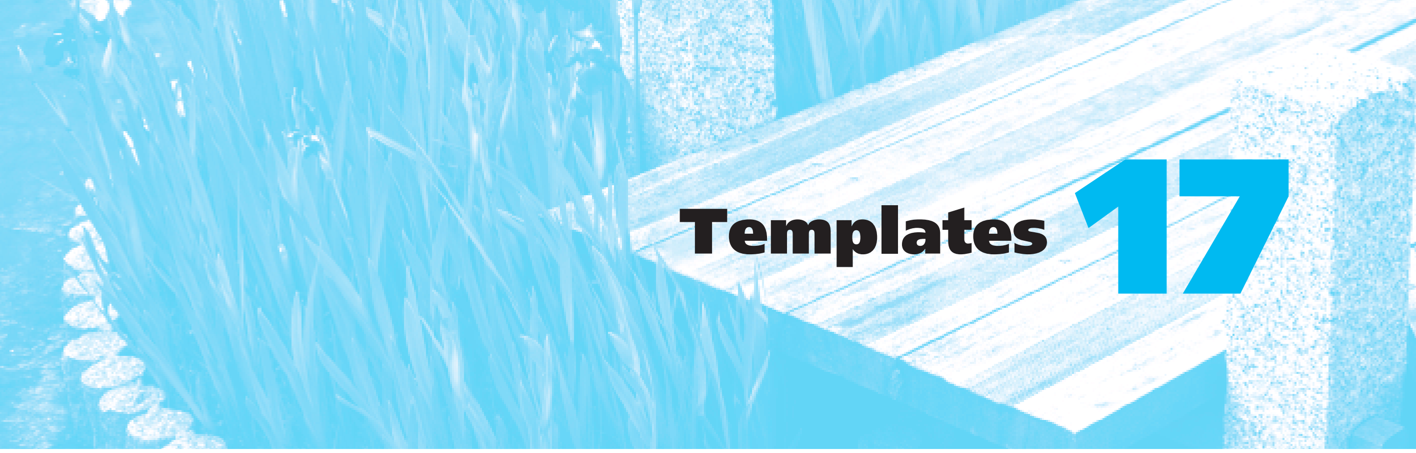

Templates 17

All men are mortal.
Aristotle is a man.
Therefore, Aristotle is mortal.
All X's are Y.
Z is an X.
Therefore, Z is Y.
All cats are mischievous.
Garfield is a cat.
Therefore, Garfield is mischievous.

A SHORT LESSON ON SYLLOGISMS

INTRODUCTION

This chapter discusses C++ templates. Templates allow you to define functions and classes that have parameters for type names. This will allow you to design functions that can be used with arguments of different types and to define classes that are much more general than those you have seen before this chapter.

PREREQUISITES

Section 17.1 uses material from Chapters 2 through 5 and Sections 7.1, 7.2, and 7.3 of Chapter 7. It does not use any material on classes. Section 17.2 uses material from Chapters 2 through 7 and 10 through 12.

17.1 TEMPLATES FOR ALGORITHM ABSTRACTION

Many of our previous C++ function definitions have an underlying algorithm that is much more general than the algorithm we gave in the function definition. For example, consider the function swap_values, which we first discussed in Chapter 5. For reference, we now repeat the function definition:

```
void swap_values(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

Notice that the function swap_values applies only to variables of type *int*. Yet the algorithm given in the function body could just as well be used to swap the values in two variables of type *char*. If we want to also use the function swap_values with variables of type *char*, we can overload the function name by adding the following definition:

```
void swap_values(char& variable1, char& variable2)
{
    char temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

But there is something inefficient and unsatisfying about these two definitions of the swap_values function: They are almost identical. The only difference is that one definition uses the type *int* in three places and the other uses the type *char* in the same three places. Proceeding in this way, if we wanted to have the function swap_values apply to pairs of variables of type *double*, we would have to write a third almost identical function definition. If we wanted to apply swap_values to still more types, the number of almost identical function definitions would be even larger. This would require a good deal of typing and would clutter up our code with lots of definitions that look identical. We should be able to say that the following function definition applies to variables of any type:

```
void swap_values(Type_Of_The_Variables& variable1,
                 Type_Of_The_Variables& variable2)
{
    Type_Of_The_Variables temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

As we will see, something like this is possible. We can define one function that applies to all types of variables, although the syntax is a bit different from what we have shown above. That syntax is described in the next subsection.

Templates for Functions

Display 17.1 shows a C++ template for the function swap_values. This function template allows you to swap the values of any two variables, of any type, as long as the two variables have the same type. The definition and the function declaration begin with the line

```
template<class T>
```

This is often called the **template prefix,** and it tells the compiler that the definition or function declaration that follows is a **template** and that T is a

type parameter. In this context, the word *class* actually means *type*.[1] As we will see, the type parameter T can be replaced by any type, whether the type is a class or not. Within the body of the function definition, the type parameter T is used just like any other type.

The function template definition is, in effect, a large collection of function definitions. For the function template for swap_values shown in Display 17.1, there is, in effect, one function definition for each possible type name. Each of these definitions is obtained by replacing the type parameter T with a type name. For example, the function definition that follows is obtained by replacing T with the type name *double*:

```
void swap_values(double& variable1, double& variable2)
{

    double temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

A template overloads the function name

Another definition for swap_values is obtained by replacing the type parameter T in the function template with the type name *int*. Yet another definition is obtained by replacing the type parameter T with *char*. The one function template shown in Display 17.1 overloads the function name swap_values so that there is a slightly different function definition for every possible type.

The compiler will not literally produce definitions for every possible type for the function name swap_values, but it will behave exactly as if it had produced all those function definitions. A separate definition will be produced for each different type for which you use the template, but not for any types you do not use. Only one definition is generated for a single type regardless of the number of times you use the template for that type. Notice that the function swap_values is called twice in Display 17.1: One time the arguments are of type *int* and the other time the arguments are of type *char*.

Consider the following function call from Display 17.1:

```
swap_values(integer1, integer2);
```

When the C++ compiler gets to this function call, it notices the types of the arguments—in this case *int*—and then it uses the template to produce a function definition with the type parameter T replaced with the type name *int*. Similarly, when the compiler sees the function call

```
swap_values(symbol1, symbol2);
```

[1] In fact, the ANSI standard provides that you may use the keyword *typename* instead of *class* in the template prefix. Although we agree that using *typename* makes more sense than using *class*, the use of *class* is a firmly established tradition, and so we use *class* for the sake of consistency with most other programmers and authors.

DISPLAY 17.1 A Function Template

```
//Program to demonstrate a function template.
#include <iostream>
using namespace std;

//Interchanges the values of variable1 and variable2.
template<class T>
void swap_values(T& variable1, T& variable2)
{
      T temp;

      temp = variable1;
      variable1 = variable2;
      variable2 = temp;
}

int main( )
{
    int integer1 = 1, integer2 = 2;
    cout << "Original integer values are "
         << integer1 << " " << integer2 <<endl;
    swap_values(integer1, integer2);
    cout << "Swapped integer values are "
         << integer1 << " " << integer2 <<endl;

    char symbol1 = 'A', symbol2 = 'B';
    cout << "Original character values are "
         << symbol1 << " " << symbol2 <<endl;
    swap_values(symbol1, symbol2);
    cout << "Swapped character values are "
         << symbol1 << " " << symbol2 <<endl;

    return 0;
}
```

Output

```
Original integer values are 1 2
Swapped integer values are 2 1
Original character values are A B
Swapped character values are B A
```

it notices the types of the arguments—in this case *char*—and then it uses the template to produce a function definition with the type parameter T replaced with the type name *char*.

Calling a function template

Notice that you need not do anything special when you call a function that is defined with a function template; you call it just as you would any other function. The compiler does all the work of producing the function definition from the function template.

Notice that in Display 17.1 we placed the function template definition before the `main` part of the program, and we used no template function declaration. A function template may have a function declaration, just like an ordinary function. You may (or may not) be able to place the function declaration and definition for a function template in the same locations that you place function declarations and definitions for ordinary functions. However, many compilers do not support template function declarations and do not support separate compilation of template functions. When these are supported, the details can be messy and can vary from one compiler to another. Your safest strategy is to not use template function declarations and to be sure the function template definition appears in the same file in which it is used and appears before the function template is used.

We said that a function template definition should appear in the same file as the file that uses the template function (that is, the same file as the file that has an invocation of the template function). However, the function template definition can appear via a `#include` directive. You can give the function template definition in one file and then `#include` that file in a file that uses the template function. That is the cleanest and safest general strategy. However, even that may not work on some compilers. If it does not work, consult a local expert.

Although we will not be using template function declarations in our code, we will describe them and give examples of them for the benefit of readers whose compilers support the use of these function declarations.

In the function template in Display 17.1, we used the letter `T` as the parameter for the type. This is traditional but is not required by the C++ language. The type parameter can be any identifier (other than a keyword). `T` is a good name for the type parameter, but sometimes other names may work better. For example, the function template for `swap_values` given in Display 17.1 is equivalent to the following:

```
template<class VariableType>
void swap_values(VariableType& variable1,
                 VariableType& variable2)
{
    VariableType temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

More than one type parameter

It is possible to have function templates that have more than one type parameter. For example, a function template with two type parameters named `T1` and `T2` would begin as follows:

```
template<class T1, class T2>
```

However, most function templates require only one type parameter. You cannot have unused template parameters; that is, each template parameter must be used in your template function.

PITFALL Compiler Complications

VideoNote
Issues Compiling Programs with Templates

Many compilers do not allow separate compilation of templates, so you may need to include your template definition with your code that uses it. As usual, at least the function declaration must precede any use of the template function.

Your safest strategy is not to use template function declarations and to be sure the function template definition appears in the same file in which it is used and appears before the function template is called. However, the function template definition can appear via a `#include` directive. You can give the function template definition in one file and then `#include` that file in a file that uses the template function.

Another common technique is to put your definition and implementation, all in the header file. If you use this technique, then you would only have a header (.h) file and no implementation (.cpp) file. Finally, an alternate approach is to include the implementation (.cpp) file for your template class instead of the header file (.h).

Some C++ compilers have additional special requirements for using templates. If you have trouble compiling your templates, check your manuals or check with a local expert. You may need to set special options or rearrange the way you order the template definitions and the other items in your files. ■

Function Template

The function definition and the function declaration for a function template are each prefaced with the following:

```
template<class Type_Parameter>
```

The function declaration (if used) and definition are the same as any ordinary function declaration and definition, except that the *Type_Parameter* can be used in place of a type.

For example, the following is a function declaration for a function template:

```
template<class T>
void show_stuff(int stuff1, T stuff2, T stuff3);
```

The definition for this function template might be as follows:

```
template<class T>
void show_stuff(int stuff1, T stuff2, T stuff3)
```

(continued)

```
{
    cout << stuff1 << endl
         << stuff2 << endl
         << stuff3 << endl;
}
```

The function template given in this example is equivalent to having one function declaration and one function definition for each possible type name. The type name is substituted for the type parameter (which is T in the example above). For instance, consider the following function call:

```
show_stuff(2, 3.3, 4.4);
```

When this function call is executed, the compiler uses the function definition obtained by replacing T with the type name *double*. A separate definition will be produced for each different type for which you use the template but not for any types you do not use. Only one definition is generated for a specific type regardless of the number of times you use the template.

SELF-TEST EXERCISES

1. Write a function template named `maximum`. The function takes two values of the same type as its arguments and returns the larger of the two arguments (or either value if they are equal). Give both the function declaration and the function definition for the template. You will use the operator < in your definition. Therefore, this function template will apply only to types for which < is defined. Write a comment for the function declaration that explains this restriction.

2. We have used three kinds of absolute value function: `abs`, `labs`, and `fabs`. These functions differ only in the type of their argument. It might be better to have a function template for the absolute value function. Give a function template for an absolute value function called `absolute`. The template will apply only to types for which < is defined, for which the unary negation operator is defined, and for which the constant 0 can be used in a comparison with a value of that type. Thus, the function `absolute` can be called with any of the number types, such as *int*, *long*, and *double*. Give both the function declaration and the function definition for the template.

3. Define or characterize the template facility for C++.

4. In the template prefix

   ```
   template<class T>
   ```

   what kind of variable is the parameter `T`?

a. T must be a class.
b. T must not be a class.
c. T can be only types built into the C++ language.
d. T can be any type, whether built into C++ or defined by the programmer.

Algorithm Abstraction

As we saw in our discussion of the swap_values function, there is a very general algorithm for interchanging the value of two variables, and this more general algorithm applies to variables of any type. Using a function template, we were able to express this more general algorithm in C++. This is a very simple example of *algorithm abstraction*. When we say we are using **algorithm abstraction,** we mean that we are expressing our algorithms in a very general way so that we can ignore incidental detail and concentrate on the substantive part of the algorithm. Function templates are one feature of C++ that supports algorithm abstraction.

PROGRAMMING EXAMPLE A Generic Sorting Function

In Chapter 7 we gave a simple sorting algorithm to sort an array of values of type *int*. The algorithm was realized in C++ code as the function sort, which we gave in Display 7.12. Here we repeat the definition of this function sort:

```
void sort(int a[], int number_used)
{
    int index_of_next_smallest;
    for (int index = 0; index < number_used - 1; index++)
    {//Place the correct value in a[index]:
        index_of_next_smallest =
                 index_of_smallest(a, index, number_used);
        swap_values(a[index], a[index_of_next_smallest]);
        //a[0] <= a[1] <=...<= a[index] are the smallest of
        //the original array elements. The rest of the
        //elements are in the remaining positions.
    }
}
```

If you study this definition of the function sort, you will see that the base type of the array is never used in any significant way. If we replace the base type of the array in the function header with the type *double*, then we would obtain a sorting function that applies to arrays of values of type *double*. Of

Helping functions

course, we also must adjust the helping functions so they apply to arrays of elements of type *double*. So let's consider the helping functions that are called inside the body of the function sort. The two helping functions are swap_values and index_of_smallest.

We already saw that swap_values can apply to variables of any type, provided we define it as a function template (as in Display 17.1). Let's see if index_of_smallest depends in any significant way on the base type of the array being sorted. The definition of index_of_smallest is repeated next so you can study its details.

```
int index_of_smallest(const int a[], int start_index,
                                   int number_used)
{
    int min = a[start_index];
    int index_of_min = start_index;
    for (int index = start_index + 1;
         index < number_used; index++)
    {
        if (a[index] < min)
        {
            min = a[index];
            index_of_min = index;
            //min is the smallest of a[start_index] through
            //a[index]
        }
    }
    return index_of_min;
}
```

The function index_of_smallest also does not depend in any significant way on the base type of the array. If we replaced the two highlighted instances of the type *int* with the type *double*, then we will have changed the function index_of_smallest so that it applies to arrays whose base type is *double*.

To change the function sort so that it can be used to sort arrays with the base type *double*, we only needed to replace a few instances of the type name *int* with the type name *double*. Moreover, there is nothing special about the type *double*. We can do a similar replacement for many other types. The only thing we need to know about the type is that the operator < is defined for that type. This is the perfect situation for function templates. If we replace a few instances of the type name *int* (in the functions sort and index_of_smallest) with a type parameter, then the function sort can sort an array of values of any type provided that the values of that type can be compared using the < operator. In Display 17.2 we have written just such a function template.

Notice that the function template sort shown in Display 17.2 can be used with arrays of values that are not numbers. In the demonstration program in Display 17.3, the function template sort is called to sort an array of characters. Characters can be compared using the < operator. Although the exact meaning of the < operator applied to character values may vary somewhat from one

DISPLAY 17.2 A Generic Sorting Function

```
//This is file sortfunc.cpp

template<class T>
void swap_values(T& variable1, T& variable2)
     <The rest of the definition of swap_values is given in Display 17.1.>

template<class BaseType>
int index_of_smallest(const BaseType a[], int start_index, int number_used)
{
    BaseType min = a[start_index];
    int index_of_min = start_index;

    for (int index = start_index + 1; index < number_used; index++)
        if (a[index] < min)
        {
            min = a[index];
            Index_of_min = index;
            //min is the smallest of a[start_index] through a[index]
        }

    return index_of_min;
}

template<class BaseType>
void sort(BaseType a[], int number_used)
{
int index_of_next_smallest;
for (int index = 0; index < number_used - 1; index++)
   {//Place the correct value in a[index]:
          index_of_next_smallest =
                index_of_smallest(a, index, number_used);
          swap_values(a[index], a[index_of_next_smallest]);
       //a[0] <= a[1] <=...<= a[index] are the smallest of the original array
       //elements. The rest of the elements are in the remaining positions.
   }
}
```

implementation to another, some things are always true about how < orders the letters of the alphabet. When applied to two uppercase letters, the operator < tests to see if the first comes before the second in alphabetic order. Also, when applied to two lowercase letters, the operator < tests to see if the first comes before the second in alphabetic order. When you mix uppercase and lowercase letters, the situation is not so well behaved, but the program shown in Display 17.3 deals only with uppercase letters. In that program, an array of

DISPLAY 17.3 Using a Generic Sorting Function *(part 1 of 2)*

```
//Demonstrates a generic sorting function.
#include <iostream>
using namespace std;

//The file sortfunc.cpp defines the following function:
//template<class BaseType>
//void sort(BaseType a[], int number_used);
//Precondition: number_used <= declared size of the array a.
//The array elements a[0] through a[number_used - 1] have values.
//Postcondition: The values of a[0] through a[number_used - 1] have
//been rearranged so that a[0] <= a[1] <= ... <= a[number_used - 1].

#include "sortfunc.cpp"

int main( )
{
    int i;
    int a[10] = {9, 8, 7, 6, 5, 1, 2, 3, 0, 4};
    cout << "Unsorted integers:\n";
    for (i = 0; i < 10; i++)
        cout << a[i] << " ";
    cout << endl;
    sort(a, 10);
    cout << "In sorted order the integers are:\n";
    for (i = 0; i < 10; i++)
        cout << a[i] << " ";
    cout << endl;

    double b[5] = {5.5, 4.4, 1.1, 3.3, 2.2};
    cout << "Unsorted doubles:\n";
    for (i = 0; i < 5; i++)
        cout << b[i] << " ";
    cout << endl;
    sort(b, 5);
    cout << "In sorted order the doubles are:\n";
    for (i = 0; i < 5; i++)
        cout << b[i] << " ";
    cout << endl;

    char c[7] = {'G', 'E', 'N', 'E', 'R', 'I', 'C'};
    cout << "Unsorted characters:\n";
    for (i = 0; i < 7; i++)
        cout << c[i] << " ";
    cout << endl;
```

Many compilers will allow this function declaration to appear as a function declaration and not merely as a comment. However, including the function declaration is not needed, since the definition of the function is in the file `sortfunc.cpp`*, and so the definition effectively appears before* `main`*.*

(continued)

DISPLAY 17.3 Using a Generic Sorting Function *(part 2 of 2)*

```
        sort(c, 7);
        cout << "In sorted order the characters are:\n";
        for (i = 0; i < 7; i++)
            cout << c[i] << " ";
        cout << endl;
        return 0;
    }
```

Output

```
Unsorted integers:
9 8 7 6 5 1 2 3 0 4
In sorted order the integers are:
0 1 2 3 4 5 6 7 8 9
Unsorted doubles:
5.5 4.4 1.1 3.3 2.2
In sorted order the doubles are:
1.1 2.2 3.3 4.4 5.5
Unsorted characters:
G E N E R I C
In sorted order the characters are:
C E E G I N R
```

uppercase letters is sorted into alphabetical order with a call to the function template `sort`. (The function template `sort` will even sort an array of objects of a class that you define, provided you overload the < operator to apply to objects of that class.)

PROGRAMMING TIP How to Define Templates

When we defined the function template in Display 17.2, we started with a function that sorts an array of elements of type *`int`*. We then created a template by replacing the base type of the array with the type parameter `T`. This is a good general strategy for writing templates. If you want to write a function template, first write a version that is not a template at all but is just an ordinary function. Completely debug the ordinary function and then convert the ordinary function to a template by replacing some type names with a type parameter. There are two advantages to this method. First, when you are defining the ordinary function you are dealing with a much more concrete case, which makes the problem easier to visualize. Second, you have fewer details to check at each stage; when worrying about the algorithm itself, you need not concern yourself with template syntax rules. ■

PITFALL Using a Template with an Inappropriate Type[2]

You can use a template function with any type for which the code in the function definition makes sense. However, all the code in the template function must make sense and must behave in an appropriate way. For example, you cannot use the swap_values template (Display 17.1) with the type parameter replaced by a type for which the assignment operator does not work at all or does not work "correctly."

As a more concrete example, suppose that your program defines the template function swap_values as in Display 17.1. You cannot add the following to your program:

```
int a[10], b[10];
<some code to fill arrays>
swap_values(a, b);
```

This code will not work, because assignment does not work with array types. ■

SELF-TEST EXERCISES

5. Display 7.10 shows a function called search, which searches an array for a specified integer. Give a function template version of search that can be used to search an array of elements of any type. Give both the function declaration and the function definition for the template. (*Hint:* It is almost identical to the function given in Display 7.10.)

6. In Practice Program 8 of Chapter 4 you were asked to overload the abs function so that the name abs would work with several of the built-in types that had been studied at the time. Compare and contrast function overloading of the abs function with the use of templates for this purpose in Self-Test Exercise 2.

[2] The example in this Pitfall section uses arrays. If you have not yet covered arrays (Chapter 7), you should skip this Pitfall section and return after covering arrays.

17.2 TEMPLATES FOR DATA ABSTRACTION

Equal wealth and equal opportunities of culture . . . have simply made us all members of one class.

EDWARD BELLAMY, *Looking Backward: 2000–1887*

As you saw in the previous section, function definitions can be made more general by using templates. In this section, you will see that templates can also make class definitions more general.

Syntax for Class Templates

The syntax for class templates is basically the same as that for function templates. The following is placed before the template definition:

```
template<class T>
```

Type parameter

The type parameter T is used in the class definition just like any other type. As with function templates, the type parameter T represents a type that can be any type at all; the type parameter does not have to be replaced with a class type. As with function templates, you may use any (nonkeyword) identifier instead of T.

For example, the following is a class template. An object of this class contains a pair of values of type T; if T is *int*, the object values are pairs of integers, if T is *char*, the object values are pairs of characters, and so on.

```
//Class for a pair of values of type T:
template<class T>
class Pair
{
public:
    Pair();

    Pair(T first_value, T second_value);

    void set_element(int position, T value);
    //Precondition: position is 1 or 2.
    //Postcondition:
    //The position indicated has been set to value.

    T get_element(int position) const;
    //Precondition: position is 1 or 2.
    //Returns the value in the position indicated.
private:
    T first;
    T second;
};
```

Declaring objects

Once the class template is defined, you can declare objects of this class. The declaration must specify what type is to be filled in for T. For example, the following code declares the object score so it can record a pair of integers and declares the object seats so it can record a pair of characters:

```
Pair<int> score;
Pair<char> seats;
```

The objects are then used just like any other objects. For example, the following sets the score to be 3 for the first team and 0 for the second team:

```
score.set_element(1, 3);
score.set_element(2, 0);
```

Defining member functions

The member functions for a class template are defined the same way as member functions for ordinary classes. The only difference is that the member function definitions are themselves templates. For example, the following are appropriate definitions for the member function set_element and for the constructor with two arguments:

```
//Uses iostream and cstdlib:
template<class T>
void Pair<T>::set_element(int position, T value)
{
    if (position == 1)
        first = value;
    else if (position == 2)
        second = value;
    else
    {
        cout << "Error: Illegal pair position.\n";
        exit(1);
    }
}

template<class T>
Pair<T>::Pair(T first_value, T second_value)
       : first(first_value), second(second_value)
{
     //Body intentionally empty.
}
```

Notice that the class name before the scope resolution operator is Pair<T>, not simply Pair.

The name of a class template may be used as the type for a function parameter. For example, the following is a possible declaration for a function with a parameter for a pair of integers:

```
int add_up(const Pair<int>& the_pair);
//Returns the sum of the two integers in the_pair.
```

Class Template Syntax

The class definition and the definitions of the member functions are prefaced with the following:

```
template<class Type_Parameter>
```

The class and member function definitions are then the same as for any ordinary class, except that the *Type_Parameter* can be used in place of a type.

For example, the following is the beginning of a class template definition:

```
template<class T>
class Pair
{
public:
    Pair();
    Pair(T first_value, T second_value);
    void set_element(int position, T value);
        . . .
```

Member functions and overloaded operators are then defined as function templates. For example, the definition of a function definition for the sample class template above could begin as follows:

```
template<class T>
void Pair<T>::set_element(int position, T value)
{
     . . .
```

Note that we specified the type, in this case `int`, that is to be filled in for the type parameter T.

You can even use a class template within a function template. For example, rather than defining the specialized function add_up given above, you could instead define a function template as follows so that the function applies to all kinds of numbers:

```
template<class T>
T add_up(const Pair<T>& the_pair);
//Precondition: The operator + is defined for values of type T.
//Returns the sum of the two values in the_pair.
```

Type Definitions

You can specialize a class template by giving a type argument to the class name, as in the following example:

```
Pair<int>
```

The specialized class name, like `Pair<int>`, can then be used just like any class name. It can be used to declare objects or to specify the type of a formal parameter.

You can define a new class type name that has the same meaning as a specialized class template name, such as `Pair<int>`. The syntax for such a defined class type name is as follows:

```
typedef Class_Name<Type_Argument> New_Type_Name;
```

For example:

```
typedef Pair<int> PairOfInt;
```

The type name `PairOfInt` can then be used to declare objects of type `Pair<int>`, as in the following example:

```
PairOfInt pair1, pair2;
```

The type name `PairOfInt` can also be used to specify the type of a formal parameter.

PROGRAMMING EXAMPLE An Array Class

Display 17.4 contains the interface for a class template whose objects are lists. Since this class definition is a class template, the lists can be lists of items of any type whatsoever. You can have objects that are lists of values of type `int`, or lists of values of type `double`, or lists of objects of type `string`, or lists of items of any other type.

Display 17.5 contains a demonstration program that uses this class template. Although this program does not really do anything much, it does illustrate how the class template is used. Once you understand the syntax details, you can use the class template in any program that needs a list of values. Display 17.6 gives the implementation of the class template.

Notice that we have overloaded the insertion operator << so it can be used to output an object of the class template `GenericList`. To do this, we made the operator << a friend of the class. In order to have a parameter that is of the same type as the class, we used the expression `GenericList<ItemType>` for the parameter type. When the type parameter is replaced by, for example, the type *`int`*, this list parameter will be of type `GenericList<int>`.

A friend

Also note that that the implementation of the overloaded insertion operator << has been placed in the header file rather than the implementation file. This may seem unusual, but it is quite common when using friend functions or operators within a template. Although we are defining << like it is a member of `GenericList`, recall that friend functions really exist outside the class and are part of the namespace. The compiler will have an easy time finding the implementation of << this way when the class is included from other files.

DISPLAY 17.4 Interface for the Class Template `GenericList` *(part 1 of 2)*

```
//This is the header file genericlist.h. This is the interface for the
//class GenericList. Objects of type GenericList can be a list of items
//of any type for which the operators << and = are defined.
//All the items on any one list must be of the same type. A list that
//can hold up to max items all of type Type_Name is declared as follows:
//GenericList<Type_Name> the_object(max);
#ifndef GENERICLIST_H
#define GENERICLIST_H
#include <iostream>
using namespace std;

namespace listsavitch
{
    template<class ItemType>
    class GenericList
    {
    public:
        GenericList(int max);
        //Initializes the object to an empty list that can hold up to
        //max items of type ItemType.
        ~GenericList( );
        //Returns all the dynamic memory used by the object to the freestore.

        int length( ) const;
        //Returns the number of items on the list.

        void add(ItemType new_item);
        //Precondition: The list is not full.
        //Postcondition: The new_item has been added to the list.
```

(continued)

DISPLAY 17.4 Interface for the Class Template `GenericList` *(part 2 of 2)*

```

        bool full( ) const;
        //Returns true if the list is full.

        void erase( );
        //Removes all items from the list so that the list is empty.

        friend ostream& operator <<(ostream& outs,
                                    const GenericList<ItemType>& the_list)
        {
            for (int i = 0; i < the_list.current_length; i++)
                outs << the_list.item[i] << endl;
            return outs;
        }
        //Overloads the << operator so it can be used to output the
        //contents of the list. The items are output one per line.
        //Precondition: If outs is a file output stream, then outs has
        //already been connected to a file.
        //
        //Note the implementation of the overloaded << in the header
        //file! This is commonly done with overloaded friend templates.
        //Since << is a friend it is NOT a member of the class but
        //rather in the namespace, this is the simplest implementation
        //and may make more sense than putting it in genericlist.cpp.
    private:
        ItemType *item; //pointer to the dynamic array that holds the list.
        int max_length; //max number of items allowed on the list.
        int current_length; //number of items currently on the list.
    };
}//listsavitch
#endif //GENERICLIST_H
```

DISPLAY 17.5 Program Using the `GenericList` Class Template *(part 1 of 2)*

```
//Program to demonstrate use of the class template GenericList.
#include <iostream>
#include "genericlist.h"
#include "genericlist.cpp"
using namespace std;
using namespace listsavitch;

int main( )
{
    GenericList<int> first_list(2);
    first_list.add(1);
    first_list.add(2);
```

Since `genericlist.cpp` is included, you need compile only this one file (the one with the `main`).

(continued)

DISPLAY 17.5 Program Using the `GenericList` Class Template *(part 2 of 2)*

```
    cout << "first_list = \n"
         << first_list;
    GenericList<char> second_list(10);
    second_list.add('A');
    second_list.add('B');
    second_list.add('C');
    cout << "second_list = \n"
         << second_list;

    return 0;
}
```

Output

```
first_list =
1
2
second_list =
A
B
C
```

DISPLAY 17.6 Implementation of `GenericList` *(part 1 of 2)*

```
//This is the implementation file: genericlist.cpp
//This is the implementation of the class template named GenericList.
//The interface for the class template GenericList is in the
//header file genericlist.h.
#ifndef GENERICLIST_CPP
#define GENERICLIST_CPP
#include <iostream>
#include <cstdlib>
#include "genericlist.h" //This is not needed when used as we are using this file,
                         //but the #ifndef in genericlist.h makes it safe.
using namespace std;

namespace listsavitch
{
    //Uses cstdlib:
    template<class ItemType>
    GenericList<ItemType>::GenericList(int max) : max_length(max),
                                                  current_length(0)
```

(continued)

DISPLAY 17.6 Implementation of `GenericList` *(part 2 of 2)*

```
        {
            item = new ItemType[max];
        }

        template<class ItemType>
        GenericList<ItemType>::~GenericList( )
        {
            delete [] item;
        }

        template<class ItemType>
        int GenericList<ItemType>::length( ) const
        {
            return (current_length);
        }

        //Uses iostream and cstdlib:
        template<class ItemType>
        void GenericList<ItemType>::add(ItemType new_item)
        {
            if ( full( ) )
            {
                cout << "Error: adding to a full list.\n";
                exit(1);
            }
            else
            {
                item[current_length] = new_item;
                current_length = current_length + 1;
            }
        }

        template<class ItemType>
        bool GenericList<ItemType>::full( ) const
        {
            return (current_length == max_length);
        }

        template<class ItemType>
        void GenericList<ItemType>::erase( )
        {
            current_length = 0;
        }
    }//listsavitch
    #endif // GENERICLIST_CPP Notice that we have enclosed all the template
           // definitions in #ifndef. . . #endif.
```

A note is in order about compiling the code from Displays 17.4, 17.5, and 17.6. A safe solution to the compilation of this code is to `#include` the template class definition and the template function definitions before use, as we did. In that case, only the file in Display 17.5 needs to be compiled. Be sure that you use the `#ifndef #define #endif` mechanism to prevent multiple file inclusion of all the files you are going to `#include`.

Also note that that the implementation of the overloaded insertion operator << has been placed in the header file rather than the implementation file. This may seem unusual, but it is quite common when using friend functions or operators within a template. Although we are defining << like it is a member of `GenericList`, recall that friend functions really exist outside the class and are part of the namespace. The compiler will have an easy time finding the implementation of << this way when the class is included from other files.

If you want to separate the implementation of the overloaded friend insertion operator << from the header, then it requires a little bit of extra work. We must make a forward declaration of the << operator which in turn requires a forward declaration of the `GenericList` class. Display 17.7 illustrates the required changes to `genericlist.h` while Display 17.8 illustrates the changes to `genericlist.cpp`, which simply has the additional implementation.

DISPLAY 17.7 Interface for the Class Template `GenericList` Without Implementation *(part 1 of 2)*

```
//This version moves the implementation of the overloaded <<
//to the .cpp file, but requires adding some forward declarations.
#ifndef GENERICLIST_H
#define GENERICLIST_H
#include <iostream>
using namespace std;

namespace listsavitch
{
    template<class ItemType>
    class GenericList;
    //We need a forward declaration of the GenericList template
    //class for the friend header declaration that comes right after it.

    template<class ItemType>
    ostream& operator <<(ostream& outs, const GenericList<ItemType>& the_list);
    //Forward declaration of the friend << for the definition inside the
    //GenericList class below. These must be defined here since << is not
    //a member of the class.

    template<class ItemType>
```

(continued)

DISPLAY 17.7 Interface for the Class Template `GenericList` Without Implementation *(part 2 of 2)*

```
        class GenericList
        {
            The rest of this class is identical to Display 17.4 except the overloaded
            operator below has no implementation code and an additional <>.

                friend ostream& operator << <>(ostream& outs,
                                                    const GenericList<ItemType>& the_list);
                //Overloads the << operator so it can be used to output the
                //contents of the list.
                //Note the <> needed after the operator (or function) name!
               //The implementation is in genericlist.cpp (Display 17.8).
            };
    }//listsavitch
    #endif //GENERICLIST_H
```

DISPLAY 17.8 Implementation of `GenericList` with Overloaded Operator

```
    //This is the implementation file: genericlist.cpp
    //This is the implementation of the class template named GenericList.
    //The interface for the class template GenericList is in the
    //header file genericlist.h.
    #ifndef GENERICLIST_CPP
    #define GENERICLIST_CPP
    #include <iostream>
    #include <cstdlib>
    #include "genericlist.h" //Not needed when used as we are using this file,
                             //but the #ifndef in genericlist.h makes it safe.
    using namespace std;

    namespace listsavitch
    {
            The rest of this file is identical to Display 17.6 except for the
            Implementation of <<.
        template<class ItemType>
        ostream& operator <<(ostream& outs, const GenericList<ItemType>& the_list)
        {
            for (int i = 0; i < the_list.current_length; i++)
                outs << the_list.item[i] << endl;
            return outs;
        }
    }//listsavitch
    #endif // GENERICLIST_CPP Notice that we have enclosed all the template
           // definitions in #ifndef . . . #endif.
```

SELF-TEST EXERCISES

7. Give the definition for the member function `get_element` for the class template `Pair` discussed in the section "Syntax for Class Templates."

8. Give the definition for the constructor with zero arguments for the class template `Pair` discussed in the section "Syntax for Class Templates."

9. Give the definition of a template class called `HeterogeneousPair` that is like the class template `Pair` discussed in the section "Syntax for Class Templates," except that with `HeterogeneousPair` the first and second positions may store values of different types. Use two type parameters `T1` and `T2`; all items in the first position will be of type `T1`, and all items in the second position will be of type `T2`. The single mutator function `set_element` in the template class `Pair` should be replaced by two mutator functions called `set_first` and `set_second` in the template class `HeterogeneousPair`. Similarly, the single accessor function `get_element` in the template class `Pair` should be replaced by two accessor functions called `get_first` and `get_second` in the template class `HeterogeneousPair`.

10. Is the following true or false?

    Friends are used exactly the same for template and nontemplate classes.

CHAPTER SUMMARY

- Using function templates, you can define functions that have a parameter for a type.
- Using class templates, you can define a class with a type parameter for subparts of the class.

Answers to Self-Test Exercises

1. Function Declaration:

```
template<class T>
T maximum(T first, T second);
//Precondition: The operator < is defined for the type T.
//Returns the maximum of first and second.
```

   Definition:

```
template<class T>
T maximum(T first, T second)
{
    if (first < second)
```

```
            return second;
        else
            return first;
    }
```

2. Function Declaration:

```
template<class T>
T absolute(T value);
//Precondition: The expressions x < 0 and -x are defined
//whenever x is of type T.
//Returns the absolute value of its argument.
```

Definition:

```
template<class T>
T absolute(T value)
{
    if (value < 0)
        return -value;
    else
        return value;
}
```

3. Templates provide a facility to allow the definition of functions and classes that have parameters for type names.

4. d. Any type, whether a primitive type (provided by C++) or a type defined by the user (a *class* or *struct* type, an *enum* type, or a defined array type, or *int*, *float*, *double*, etc.).

5. The function declaration and function definition are given here. They are basically identical to those for the versions given in Display 7.10 except that two instances of *int* are changed to `BaseType` in the parameter list.

Function Declaration:

```
template<class BaseType>
int search(const BaseType a[],
           int number_used, BaseType target);
//Precondition: number_used is <= the declared size of a.
//Also, a[0] through a[number_used-1] have values.
//Returns the first index such that a[index] == target,
//provided there is such an index; otherwise, returns -1.
```

Definition:

```
template<class BaseType>
int search(const BaseType a[], int number_used,
           BaseType target)
```

```
{
    int index = 0, found = false;
    while ((!found) && (index < number_used))
    if (target == a[index])
        found = true;
    else
        index++;

    if (found)
        return index;
    else
        return -1;
}
```

6. Function overloading only works for types for which an overloading is provided. Overloading may work for types that automatically convert to some type for which an overloading is provided but may not do what you expect. The template solution will work for any type that is defined at the time of invocation, provided that the requirements for a definition of < are satisfied.

7.
```
//Uses iostream and cstdlib:
template<class T>
T Pair<T>::get_element(int position) const
{
    if (position == 1)
        return first;
    else if (position == 2)
        return second;
    else
    {
        cout << "Error: Illegal pair position.\n";
        exit(1);
    }
}
```

8. There are no natural candidates for the default initialization values, so this constructor does nothing, but it does allow you to declare (uninitialized) objects without giving any constructor arguments.

```
template<class T>
Pair<T>::Pair()
{
//Do nothing.
}
```

9.
```
//Class for a pair of values, the first of type T1
//and the second of type T2:
template<class T1, class T2>
```

```
class HeterogeneousPair
{
public:
    HeterogeneousPair();
    HeterogeneousPair(T1 first_value, T2 second_value);
    void set_first(T1 value);
    void set_second(T2 value);
    T1 get_first() const;
    T2 get_second() const;
private:
    T1 first;
    T2 second;
};
```

The member function definitions are as follows:

```
template<class T1, class T2>
HeterogeneousPair<T1, T2>::HeterogeneousPair( )
{
//Do nothing.
}

template<class T1, class T2>
HeterogeneousPair<T1, T2>::HeterogeneousPair
    (T1 first_value, T2 second_value)
          : first(first_value), second(second_value)
{
    //Body intentionally empty.
}

template<class T1, class T2>
T1 HeterogeneousPair<T1, T2>::get_first() const
{
    return first;
}

template<class T1, class T2>
T2 HeterogeneousPair<T1, T2>::get_second() const
{
    return second;
}

template<class T1, class T2>
void HeterogeneousPair<T1, T2>::set_first(T1 value)
{
    first = value;
}
```

```
template<class T1, class T2>
void HeterogeneousPair<T1, T2>::set_second(T2 value)
{
    second = value;
}
```

10. True.

PRACTICE PROGRAMS

Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.

1. Write a function template for a function that has parameters for a partially filled array and for a value of the base type of the array. If the value is in the partially filled array, then the function returns the index of the first indexed variable that contains the value. If the value is not in the array, the function returns −1. The base type of the array is a type parameter. Notice that you need two parameters to give the partially filled array: one for the array and one for the number of indexed variables used. Also, write a suitable test program to test this function template.

2. Write a template version of the iterative binary search from Display 14.8. Specify and discuss the requirements on the template parameter type.

3. Write a template version of the recursive binary search from Display 14.6. Specify and discuss the requirements on the template parameter type.

PROGRAMMING PROJECTS

Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit www.myprogramminglab.com to complete many of these Programming Projects online and get instant feedback.

1. Rewrite the definition of the class template `GenericList` given in Display 17.4 and Display 17.6 so that it is more general. This more general version has the added feature that you can step through the items on the list in order. One item is always the current item. You can ask for the current item, change the current item to the next item, change the current item to the previous item, start at the beginning of the list by making the first item on the list the current item, and ask for the *n*th item on the list. To do this, you will add the following members: an additional member variable that records the position on the list of the current item, a member function that returns the current item as a value, a member function that makes the next item the current item, a member function that makes the previous item the current item, a member function that makes the first item on the

list the current item, and a member function that returns the *n*th item on the list given `n` as an argument. (Number items as in arrays, so that the first item is the 0th item, the next is item number 1, and so forth.)

Note that there are situations in which some of these function actions are not possible. For example, an empty list has no first item, and there is no item after the last item in any list. Be sure to test for the empty list and handle it appropriately. Be sure to test for the beginning and end of the list and handle these cases appropriately. Write a suitable test program to test this class template.

2. Write a template for a function that has parameters for a list of items and for a possible item on the list. If the item is on the list, then the function returns the position of the first occurrence of that item. If the item is not on the list, the function returns −1. The first position on the list is position `0`, the next is position `1`, and so forth. The type of the items on the list is a type parameter. Use the class template `GenericList` that you defined in Project 1. Write a suitable program to test this function template.

3. Redo Programming Project 3 in Chapter 7, but this time make the function `delete_repeats` a template function with a type parameter for the base type of the array. It would help if you first did the nontemplate version; in other words, it would help if you first did Programming Project 3 in Chapter 7, if you have not already done it.

4. Display 17.3 gives a template function for sorting an array using the selection sort algorithm. Write a similar template function for sorting an array, but this time use the insertion sort algorithm as described in Programming Project 6 of Chapter 7. If you have not already done it, it would be a good idea to first do the nontemplate version; in other words, it would be a good idea to first do Programming Project 6 from Chapter 7.

5. (This project requires that you know what a stack is and how to use dynamic arrays. Stacks are covered in Chapter 14; dynamic arrays are covered in Chapter 9. This is an appropriate project only if you have covered Chapters 9 and 14.)

   Write a template version of a stack class. Use a type parameter for the type of data that is stored in the stack. Use dynamic arrays to allow the stack to grow to hold any number of items.

6. Write a template version of a class that implements a priority queue. Queues are discussed in Chapter 13 and priority queues are discussed in Chapter 18. To summarize, a priority queue is essentially a list of items

that is always ordered by priority. Each item that is added to the list requires an associated priority value. For this problem, make the priority an integer where 0 is the highest priority and larger values are lower in priority. Removing an item from the queue removes the item with the highest priority.

The add function of the priority queue should take a generic type and then an integer priority. In the following example, the generic type is a char and we have added three items to the queue:

```
q.add('X', 10);
q.add('Y', 1);
q.add('Z', 3);
```

The remove function should return and remove from the priority queue the item that has the highest priority. Given the example above, we would expect the following:

```
cout << q.remove();         // Outputs Y  (priority 1)
cout << q.remove();        // Returns Z  (priority 3)
cout << q.remove();        // Returns X  (priority 10)
```

Test your queue on data with priorities in various orders (for example, ascending, descending, mixed). You can implement the priority queue by storing the items using a list(s) of your choice (for example, vector, array, linked list, or GenericList described in this chapter) and then performing a linear search for the item with the lowest integer value in the remove function. In future courses you may study a data structure called a heap that affords a more efficient way to implement a priority queue.

7. Write a template-based class that implements a set of items. A set is a collection of items in which no item occurs more than once. Internally, you may represent the set using the data structure of your choice (for example, list, vector, arrays, etc.). However, the class should externally support the following functions:

VideoNote
Solution to Programming Project 17.7

a. Add a new item to the set. If the item is already in the set then nothing happens.
b. Remove an item from the set.
c. Return the number of items in the set.
d. Determine if an item is a member of the set.
e. Return a pointer to a dynamically created array containing each item in the set. The caller of this function is responsible for deallocating the memory.

Test your class by creating different sets of different data types (for example, strings, integers, or other classes). If you add objects to your set, then you may need to overload the == and != operators for the object's class so your template-based set class can properly determine membership.

8. This project requires that you complete Programming Project 7 from this chapter and Programming Project 8 from Chapter 14. Programming Project 8 asked you to write a program to find all permutations of a set. Modify the program so that it generates permutations given an instance of the template-based set class defined in Programming Project 7. You may wish to also use your template-based set class to help simplify the implementation of the permutation algorithm itself.

   The algorithm requires that you store a set of lists. C++ allows you to create a set of lists with your template-based set class. For example, `myset<vector<T> >` will define a set containing a vector of type T. Be careful to place a space between the last two >'s, or the compiler may get confused. The code `myset<vector<T>>` without a space will likely produce a compiler error unless you are using C++11 or higher.

   Your program should print all permutations of sets of several different sizes and comprised of several different types of data (for example, a set of three integers, a set of four strings, or a set of five doubles).

9. In this chapter we used only a single template class type parameter. C++ allows you to specify multiple type parameters. For example, the following code specifies that the class accepts two type parameters:

```
template<class T, class V>
class Example
{
    ...
}
```

   When creating an instance of the class, we must now specify two data types, such as:

```
Example<int, char> demo;
```

   Create a `Map` class that maps keys to values. The data type for the keys and values should be specified separately using type parameters. The map forms the basis for a simple database. For example, to map from employee ID numbers to employee names, we might use integers for the data type of the keys and strings for the data type of the names. The class should have functions to:

   1. Add a new key/value pair to the map
   2. Set an existing key/value pair to a new value given the key
   3. Delete a key/value pair from the map given the key
   4. Check if a key/value pair exists in the map given the key
   5. Retrieve the value for a key/value pair given the key

   Use any data type you wish to implement the map. Write a main function that tests the class by exercising all of the functions with sample data.