16 Templates

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, which allow you to define functions and classes that have parameters for type names. This enables 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.

Section 16.1 requires only material from Chapters 1 through 5. Section 16.2 uses material from Section 16.1 as well as Chapters 1 through 11 but does not require the material from Chapter 12 through 15. Section 16.3 requires the previous sections as well as Chapter 14 on inheritance and all the chapters needed for Section 16.2. Section 16.3 does mark some member functions as `virtual`. Virtual functions are covered in Chapter 15. However, this use of virtual functions is not essential to the material presented. It is possible to read Section 16.3 ignoring (or even omitting) all occurrences of the keyword `virtual`.

16.1 Function Templates

Many of our previously discussed 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 `swapValues`, which we first discussed in Chapter 4. For reference, we now repeat the function definition:

```
void swapValues(int& variable1, int& variable2)
{
    int temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

Notice that the function `swapValues` 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 `swapValues` with variables of type `char`, we can overload the function name `swapValues` by adding the following definition:

```
void swapValues(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 `swapValues` 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 `swapValues` apply to pairs of variables of type `double`, we would have to write a third almost identical function definition. If we wanted to apply `swapValues` 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 swapValues(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 more complicated than what we have shown above. The proper syntax is described in the next subsection.

SYNTAX FOR FUNCTION TEMPLATES

Display 16.1 shows a C++ template for the function swapValues. 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>
```

template prefix

type parameter

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.

A template overloads the function name

The function template definition is, in effect, a large collection of function definitions. For the function template for swapValues shown in Display 16.1, 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 shown below is obtained by replacing T with the type name double:

```
void swapValues(double& variable1, double& variable2)
{
    double temp;

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

Another definition for swapValues 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 16.1 overloads the function name swapValues 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 swapValues, 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 swapValues is called twice in Display 16.1: One

[1] In fact, the ANSI/ISO standard provides that the keyword typename may be used instead of class in the template prefix. It would make more sense to use the keyword typename rather than class, but everybody uses class, so we will do the same. (It is often true that consistency in coding is more important than optimality.)

Display 16.1 A Function Template

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

//Interchanges the values of variable1 and variable2.
//The assignment operator must work for the type T.
template<class T>
void swapValues(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;
    swapValues(integer1, integer2);
    cout << "Swapped integer values are "
         << integer1 << " " << integer2 << endl;

    char symbol1 = 'A', symbol2 = 'B';
    cout << "Original character values are: "
         << symbol1 << " " << symbol2 << endl;
    swapValues(symbol1, symbol2);
    cout << "Swapped character values are: "
         << symbol1 << " " << symbol2 << endl;
    return 0;
}
```

Compilers still have problems with templates. To be certain that your templates work on the widest selection of compilers, place the template definition in the same file in which it is used and have the template definition precede all uses of the template.

Sample Dialogue

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

time the arguments are of type `int`, and the other time the arguments are of type `char`. Consider the following function call from Display 16.1:

```
swapValues(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

```
swapValues(symbol1, symbol2);
```

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.

A function template may have a function declaration and a definition, just like an ordinary function. You may 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, separate compilation of template definitions and template function declarations is not yet implemented on most compilers, so it is safest to place your template function definition in the file where you invoke the template function, as we did in Display 16.1. In fact, most compilers require that the template function definition appear before the first invocation of the template. You may simply `#include` the file containing your template function definitions prior to calling the template function. Your particular compiler may behave differently; you should ask a local expert about the details.

In the function template in Display 16.1 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 other names can be used.

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

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 function template.

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.

The template program layout that seems to work with the widest selection of compilers is the following: Place the template definition in the same file in which it is used and have the template definition precede all uses (all invocations) of the template. If you want to place your function template definition in a file separate from your application program, you can `#include` the file with the function template definition in the application file.

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`? Choose from the answers listed below.

a. `T` must be a class.

b. `T` must *not* be a class.

c. `T` can be only a type built into the C++ language.

d. `T` can be any type, whether built into C++ or defined by the programmer.

e. `T` can be any kind of type, whether built into C++ or defined by the programmer, but `T` does have some requirements that must be met. (What are they?)

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 then 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 showStuff(int stuff1, T stuff2, T stuff3);
```

The definition for this function template might be as follows:

```
template<class T>
void showStuff(int stuff1, T stuff2, T stuff3)
{
    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:

```
showStuff(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.

Algorithm Abstraction

As we saw in our discussion of the `swapValues` function, there is a very general algorithm for interchanging the value of two variables that 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.

Example

A Generic Sorting Function

Chapter 5 gave the selection sorting algorithm for sorting an array of values of type `int`. The algorithm was realized in C++ code as the function `sort`, given in Display 5.8. Below we repeat the definitions of this function `sort`:

```
void sort(int a[], int numberUsed)
{
    int indexOfNextSmallest;
    for (int index = 0; index < numberUsed - 1; index++)
    {//Place the correct value in a[index]:
        indexOfNextSmallest =
                     indexOfSmallest(a, index, numberUsed);
        swapValues(a[index], a[indexOfNextSmallest]);
        //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 the above definition of the function `sort` you will see that the base type of the array is never used in any significant way. If we replaced the base type of the array in the function header with the type `double`, we would obtain a sorting function that applies to arrays of values of type `double`. Of course, we also must adjust the helping functions so that they apply to arrays of elements of type `double`. Let's consider the helping functions that are called inside the body of the function `sort`. The two helping functions are `swapValues` and `indexOfSmallest`.

We already saw that `swapValues` can apply to variables of any type for which the assignment operator works, provided we define it as a function template (as in Display 16.1). Let's see if `indexOfSmallest` depends in any significant way on the base type of the array being sorted. The definition of `indexOfSmallest` is repeated below so you can study its details.

```
int indexOfSmallest(const int a[], int startIndex, int numberUsed)
{
    int min = a[startIndex],
        indexOfMin = startIndex;
    for (int index = startIndex + 1; index < numberUsed; index++)
        if (a[index] < min)
        {
            min = a[index];
            indexOfMin = index;
            //min is the smallest of a[startIndex] through a[index]
        }

    return indexOfMin;
}
```

The function indexOfSmallest also does not depend in any significant way on the base type of the array. If we replace the two highlighted instances of the type int with the type double, then we will have changed the function indexOfSmallest 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 need 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 assignment operator and the operator, <, are 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 indexOfSmallest) 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 assigned with the assignment operator and compared using the < operator. In Display 16.2 we have written just such a function template.

Notice that the function template sort shown in Display 16.2 can be used with arrays of values that are not numbers. In the demonstration program, the function template sort is called to sort an array of characters. Characters can be compared using the < operator, which compares characters according to the order of their ASCII numbers (see Appendix 3). Thus, when applied to two uppercase letters, the operato, <, tests to see if the first character comes before the second in alphabetic order. Also, when applied to two lowercase letters, the operator, <, tests to see if the first character 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 16.2 deals only with uppercase letters. In that program an array of 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 the class.)

Our generic sorting function has separated the implementation from the declaration of the sorting function by placing the definition of the sorting function in the file sort.cpp (Display 16.3). However, most compilers do not allow for separate compilation of templates in the usual sense. So, we have separated the implementation from the programmer's point of view, but from the compiler's point of view it looks like everything is in one file. The file sort.cpp is #included in our main file, so it is as if everything were in one file. Note that the include directive for sort.cpp is placed before any invocation of the functions defined by templates. For most compilers this is the only way you can get templates to work.

Display 16.2 A Generic Sorting Function *(part 1 of 3)*

```
//Demonstrates a template function that implements
//a generic version of the selection sort algorithm.
#include <iostream>
using std::cout;
using std::endl;

template<class T>
void sort(T a[], int numberUsed);
```

Display 16.2 A Generic Sorting Function ***(part 2 of 3)***

```cpp
//Precondition: numberUsed <= declared size of the array a.
//The array elements a[0] through a[numberUsed - 1] have values.
//The assignment and < operator work for values of type T.
//Postcondition: The values of a[0] through a[numberUsed - 1] have
//been rearranged so that a[0] <= a[1] <=... <= a[numberUsed - 1].

template<class T>
void swapValues(T& variable1, T& variable2);
//Interchanges the values of variable1 and variable2.
//The assignment operator must work correctly for the type T.

template<class T>
int indexOfSmallest(const T a[], int startIndex, int numberUsed);
//Precondition: 0 <= startIndex < numberUsed. Array elements have values.
//The assignment and < operator work for values of type T.
//Returns the index i such that a[i] is the smallest of the values
//a[startIndex], a[startIndex + 1],..., a[numberUsed - 1].

#include "sort.cpp"
```

This is equivalent to placing the function template definitions in this file at this location.

```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";
```

Display 16.2 A Generic Sorting Function *(part 3 of 3)*

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

    return 0;
}
```

SAMPLE DIALOGUE

```
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
```

Display 16.3 Implementation of the Generic Sorting Function *(part 1 of 2)*

```
// This is the file sort.cpp.

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

Display 16.3 Implementation of the Generic Sorting Function *(part 2 of 2)*

```
template<class T>
void swapValues(T& variable1, T& variable2)
          <The rest of the definition of swapValues is given in Display 16.1.>

template<class T>
int indexOfSmallest(const T a[], int startIndex, int numberUsed)
{
    T min = a[startIndex];          Note that the type parameter may be
    int indexOfMin = startIndex;    used in the body of the function
                                    definition

    for (int index = startIndex + 1; index < numberUsed; index++)
        if (a[index] < min)
        {
            min = a[index];
            indexOfMin = index;
            //min is the smallest of a[startIndex] through a[index].
        }

    return indexOfMin;
}
```

Tip

How to Define Templates

When we defined the function templates in Display 16.3, 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. Then completely debug the ordinary function, and finally 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

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 makes sense and must behave in an appropriate way. For example, you cannot use the `swapValues` template (Display 16.1) 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 `swapValues` as in Display 16.1. You cannot add the following to your program.

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

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

Self-Test Exercises

5. Display 5.6 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 5.6.)

6. Compare and contrast overloading of a function name with the definition of a function template for the function name.

7. (This exercise is only for those who have already read at least Chapter 6 on structures and classes and preferably also read Chapter 8 on overloading operators.) Can you use the `sort` template function (Display 16.3) to sort an array with base type `DayOfYear` defined in Display 6.4?

8. (This exercise is only for those who have already read Chapter 10 on pointers and dynamic arrays.)

   Although the assignment operator does not work with ordinary array variables, it does work with pointer variables that are used to name dynamic arrays. Suppose that your program defines the template function `swapValues` as in Display 16.1 and contains the following code. What is the output produced by this code?

```
typedef int* ArrayPointer;
ArrayPointer a, b, c;
a = new int[3];
b = new int[3];

int i;
for (i = 0; i < 3; i++)
{
    a[i] = i;
    b[i] = i*100;
}
c = a;

cout << "a contains: ";
for (i = 0; i < 3; i++)
    cout << a[i] << " ";
cout << endl;
```

```
    cout << "b contains: ";
    for (i = 0; i < 3; i++)
        cout << b[i] << " ";
    cout << endl;
    cout << "c contains: ";
    for (i = 0; i < 3; i++)
        cout << c[i] << " ";
    cout << endl;

    swapValues(a, b);
    b[0] = 42;

    cout << "After swapping a and b,\n"
         << "and changing b:\n";
    cout << "a contains: ";
    for (i = 0; i < 3; i++)
        cout << a[i] << " ";
    cout << endl;
    cout << "b contains: ";
    for (i = 0; i < 3; i++)
        cout << b[i] << " ";
    cout << endl;
    cout << "c contains: ";
    for (i = 0; i < 3; i++)
        cout << c[i] << " ";
    cout << endl;
```

16.2 Class Templates

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 details for class templates are basically the same as those 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`, although it is traditional to use `T`.

Display 16.4 (part 1) shows an example of 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.[2]

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 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 `score` to be `3` for the first team and `0` for the second team:

```
score.setFirst(3);
score.setSecond(0);
```

Display 16.4 (Part 1) Class Template Definition

```
//Class for a pair of values of type T:
template<class T>
class Pair
{
public:
    Pair( );
    Pair(T firstValue, T secondValue);
    void setFirst(T newValue);
    void setSecond(T newValue);
    T getFirst( ) const;
    T getSecond( ) const;
private:
    T first;
    T second;
};
```

[2] `Pair` is a template version of the class `intPair` given in Display 8.6. However, since they would not be appropriate for all types `T`, we have omitted the increment and decrement operators.

Display 16.4 (Part 2) Some Sample Member Function Definitions

```
template<class T>
Pair<T>::Pair(T firstValue, T secondValue)
{
    first = firstValue;
    second = secondValue;
}

template<class T>
void Pair<T>::setFirst(T newValue)
{
    first = newValue;
}

template<class T>
T Pair<T>::getFirst( ) const
{
    return first;
}
```

Not all the member functions are shown here.

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, Display 16.4 (part 2) shows appropriate definitions for the member functions `setFirst` and `getFirst`, and for the constructor with two arguments for the template class `Pair`. Notice that the class name before the scope resolution operator is `Pair<T>`, not simply `Pair`, but that the constructor name after the scope resolution operator is the simple name `Pair` without any `<T>`.

class templates as parameters

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

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

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

restrictions on the type parameter

Almost all template class definitions have some restrictions on what types can reasonable be substituted for the type parameter (or parameters). Even a straightforward template class like `Pair` does not work well with absolutely all types `T`. The type `Pair<T>` will not be well behaved unless the assignment operator and copy constructor are well behaved for the type `T`, since the assignment operator is used in member function definitions and since there are member functions with call-by-value parameters of type `T`. If `T` involves pointers and dynamic variables, then `T` should also have a suitable destructor. However, these are requirements you might expect a well-behaved class type T to have. So, these requirements are minimal. With other template classes the requirements on the types that can be substituted for a type parameter may be more restrictive.

Class Template Syntax

A class template definition and the definitions of its 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 firstValue, T secondValue);
        . . .
```

Member functions and overloaded operators are then defined as function templates. For example, the definition of the two-argument constructor for the above sample class template would begin as follows:

```
template<class T>
Pair<T>::Pair(T firstValue, T secondValue)
{
     . . .
```

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.

Type Definitions

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 or used anyplace else a type name is allowed.

Self-Test Exercises

9. Give the definition for the default (zero-argument) constructor for the class template `Pair` in Display 16.4.
10. Give the complete definition for the following function, which was discussed in the previous subsection:

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

11. Give the complete definition for the following template function, which was discussed in the previous subsection:

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

Example

An Array Template Class

In Chapter 10 we defined a class for a partially filled array of `double`s (Displays 10.10 and 10.11). In this example, we convert that definition to a template class for a partially filled array of values of any type. The template class `PFArray` has a type parameter T for the base type of the array.

The conversion is routine. We just replace double (when it occurs as the base type of the array) with the type parameter T and convert both the class definition and the member function definitions to template form. The template class definition is given in Display 16.5. The member function template definitions are given in Display 16.6.

namespace

Note that we have placed the template definitions in a namespace. Namespaces are used with templates in the same way as they are used with simple, nontemplate definitions.

separate compilation

A sample application program is given in Display 16.7. Note that we have separated the class template interface, implementation, and application program into three files. Unfortunately, these files cannot be used for the traditional method of separate compilation. Most compilers do not yet accommodate such separate compilation. So, we do the best we can by #include–ing the interface and implementation files in the application file. To the compiler, that makes it look like everything is in one file.

Display 16.5 Interface for the PFArray Template Class *(part 1 of 2)*

```
//This is the header file pfarray.h. This is the interface for the class
//PFArray. Objects of this type are partially filled arrays with base type T.
#ifndef PFARRAY_H
#define PFARRAY_H

namespace PFArraySavitch
{
    template<class T>
    class PFArray
    {
    public:
        PFArray( ); //Initializes with a capacity of 50.

        PFArray(int capacityValue);

        PFArray(const PFArray<T>& pfaObject);

        void addElement(T element);
        //Precondition: The array is not full.
        //Postcondition: The element has been added.

        bool full( ) const; //Returns true if the array is full; false, otherwise.

        int getCapacity( ) const;

        int getNumberUsed( ) const;

        void emptyArray( );
        //Resets the number used to zero, effectively emptying the array.
```

Display 16.5 Interface for the PFArray Template Class *(part 2 of 2)*

```
        T& operator[](int index);
        //Read and change access to elements 0 through numberUsed - 1.

        PFArray<T>& operator =(const PFArray<T>& rightSide);

        virtual ~PFArray( );
    private:
        T *a; //for an array of T.
        int capacity; //for the size of the array.
        int used; //for the number of array positions currently in use.
    };
}// PFArraySavitch
#endif //PFARRAY_H
```

Display 16.6 Implementation for PFArray Template Class *(part 1 of 3)*

```
//This is the implementation file pfarray.cpp.
//This is the implementation of the template class PFArray.
//The interface for the template class PFArray is in the file pfarray.h.

#include "pfarray.h"
#include <iostream>
using std::cout;

namespace PFArraySavitch
{
    template<class T>
    PFArray<T>::PFArray( ) :capacity(50), used(0)
    {
        a = new T[capacity];
    }

    template<class T>
    PFArray<T>::PFArray(int size) :capacity(size), used(0)
    {
        a = new T[capacity];
    }

    template<class T>
    PFArray<T>::PFArray(const PFArray<T>& pfaObject)
      :capacity(pfaObject.getCapacity( )), used(pfaObject.getNumberUsed( ))
    {
        a = new T[capacity];
        for (int i = 0; i < used; i++)
            a[i] = pfaObject.a[i];
```

Note that the T is used before the scope resolution operator, but no T is used for the constructor name

Display 16.6 Implementation for PFArray Template Class *(part 2 of 3)*

```
}

template<class T>
void PFArray<T>::addElement(T element)
{
    if (used >= capacity)
    {
        cout << "Attempt to exceed capacity in PFArray.\n";
        exit(0);
    }
    a[used] = element;
    used++;
}

template<class T>
bool PFArray<T>::full( ) const
{
    return (capacity == used);
}

template<class T>
int PFArray<T>::getCapacity( ) const
{
    return capacity;
}

template<class T>
int PFArray<T>::getNumberUsed( ) const
{
    return used;
}

template<class T>
void PFArray<T>::emptyArray( )
{
    used = 0;
}

template<class T>
T& PFArray<T>::operator[](int index)
{
    if (index >= used)
    {
        cout << "Illegal index in PFArray.\n";
        exit(0);
```

Display 16.6 **Implementation for PFArray Template Class** ***(part 3 of 3)***

```
        }

        return a[index];
    }

    template<class T>
    PFArray<T>& PFArray<T>::operator =(const PFArray<T>& rightSide)
    {
        if (capacity != rightSide.capacity)
        {
            delete [] a;
            a = new T[rightSide.capacity];
        }

        capacity = rightSide.capacity;
        used = rightSide.used;
        for (int i = 0; i < used; i++)
            a[i] = rightSide.a[i];

        return *this;
    }

    template<class T>
    PFArray<T>::~PFArray( )
    {
        delete [] a;
    }
}// PFArraySavitch
```

Display 16.7 **Demonstration Program for Template Class PFArray** ***(part 1 of 3)***

```
//Program to demonstrate the template class PFArray.
#include <iostream>
#include <string>
using std::cin;
using std::cout;
using std::endl;
using std::string;

#include "pfarray.h"
#include "pfarray.cpp"
using PFArraySavitch::PFArray;

int main( )
{
```

Display 16.7 Demonstration Program for Template Class PFArray *(part 2 of 3)*

```
    PFArray<int> a(10);

    cout << "Enter up to 10 nonnegative integers.\n";
    cout << "Place a negative number at the end.\n";
    int next;
    cin >> next;
    while ((next >= 0) && (!a.full( )))
    {
        a.addElement(next);
        cin >> next;
    }
    if (next >= 0)
    {
        cout << "Could not read all numbers.\n";
        //Clear the unread input:
        while (next >= 0)
            cin >> next;
    }

    cout << "You entered the following:\n ";
    int index;
    int count = a.getNumberUsed( );
    for (index = 0; index < count; index++)
        cout << a[index] << " ";
    cout << endl;

    PFArray<string> b(3);

    cout << "Enter three words:\n";
    string nextWord;
    for (index = 0; index < 3; index++)
    {
        cin >> nextWord;
        b.addElement(nextWord);
    }

    cout << "You wrote the following:\n";
    count = b.getNumberUsed( );
    for (index = 0; index < count; index++)
        cout << b[index] << " ";
    cout << endl;
    cout << "I hope you really mean it.\n";

    return 0;
}
```

Display 16.7 **Demonstration Program for Template Class PFArray *(part 3 of 3)***

SAMPLE DIALOGUE

```
Enter up to 10 nonnegative integers.
Place a negative number at the end.
1 2 3 4 5 -1
You entered the following:
1 2 3 4 5
Enter three words:
I love you
You wrote the following:
I love you
I hope you really mean it.
```

FRIEND FUNCTIONS

Friend functions are used with template classes in the same way that they are used with ordinary classes. The only difference is that you must include a type parameter where appropriate.

Self-Test Exercises

12. What do you have to do to make the following function a friend of the template class `PFArray` in Display 16.5?

```
void showData(PFArray<T> theObject);
//Displays the data in theObject to the screen.
//Assumes that << is defined for values of type T.
```

THE vector AND basic_string TEMPLATES

If you have not yet done so, this would be a good time to read Section 7.3 of Chapter 7, which covers the template class `vector`. `vector`

Another predefined template class is the `basic_string` template class. The class `basic_string` is a template class that can deal with strings of elements of any type. The class `basic_string<char>` is the class for strings of characters. The class `basic_string<double>` is the class for strings of numbers of type `double`. The class `basic_string<YourClass>` is the class for strings of objects of the class `YourClass` (whatever that may be). `basic_string`

string

You have already been using a special case of the basic_string template class. The unadorned name string, which we have been using, is an alternate name for the class basic_string<char>. All the member functions you learned for the class string apply and behave similarly for the template class basic_string<T>.

The template class basic_string is defined in the library with header file <string>, and the definition is placed in the std namespace. When using the class basic_string you therefore need the following or something similar near the beginning of your file:

```
#include <string>
using namespace std;
```

or

```
#include <string>
using std::basic_string;
using std::string; //Only if you use the name string by itself
```

16.3 Templates and Inheritance

The ruling ideas of each age have ever been the ideas of its ruling class.

Karl Marx and Friedrich Engels, *The Communist Manifesto*

There is very little new to learn about templates and inheritance. To define a derived template class, you start with a template class (or sometimes a nontemplate class) and derive another template class from it. You do this in the same way that you derive an ordinary class from an ordinary base class. An example should clarify any questions you might have about syntax details.

Example

Template Class for a Partially Filled Array with Backup

Chapter 14 (Displays 14.10 and 14.11) defined the class PFArrayDBak for partially filled arrays of double with backup. We defined it as a derived class of PFArrayD (Displays 14.8 and 14.9). The class PFArrayD was a class for a partially filled array, but it only worked for the base type double. Displays 16.5 and 16.6 converted the class PFArrayD to the template class PFArray so that it would work for any type as the array base type. In this program we will define a template class PFArrayBak for a partially filled array with backup that will work for any type as the array base type. We will define the template class PFArrayBak as a derived class of the template PFArray. We can do this almost automatically by starting with the regular derived class PFArarryDBak and replacing all occurrences of the array base type double with a type parameter T, replacing the class PFArrayD with the template class PFArray, and cleaning up the syntax so it fully conforms to template syntax.

The interface to the template class PFArrayBak is given in Display 16.8. Note that the base class is PFArray<T> with the array parameter, not simply PFArray. If you think about it, you will realize that you need the <T>. A partially filled array of T with backup is a derived class of a partially filled array of T. The T is important, and how it is used is important.

The implementation for the template class PFArrayBak is given in Display 16.9. In what follows we reproduce the first constructor definition in the implementation:

```
template<class T>
PFArrayBak<T>::PFArrayBak( ) : PFArray<T>( ), usedB(0)
{
    b = new T[getCapacity( )];
}
```

Note that, as with any definition of a template class function, it starts with

```
template<class T>
```

Also notice that the base type of the array (given after the new) is the type parameter T. Other details may not be quite as obvious, but do make sense.

Next consider the following line:

```
PFArrayBak<T>::PFArrayBak( ) : PFArray<T>( ), usedB(0)
```

As with any definition of a template class function, the definition has PFArray<T> with the type parameter before the scope resolution operator, but the constructor name is just plain-old PFArrayBak without any type parameter. Also notice that the base class constructor includes the type parameter T in the initialization PFArray<T>( ). This is so that the constructor will match the base type PFArray<T> as given in the following line of the interface:

```
class PFArrayBak : public PFArray<T>
```

A sample program using the template class PFArrayBak is given in Display 16.10.

Display 16.8 Interface for the Template Class PFArrayBak *(part 1 of 2)*

```
//This is the header file pfarraybak.h. This is the interface for the
//template class PFArrayBak. Objects of this type are partially filled
//arrays of any type T. This version allows the programmer to make a backup
//copy and restore to the last saved copy of the partially filled array.
#ifndef PFARRAYBAK_H
#define PFARRAYBAK_H
#include "pfarray.h"

namespace PFArraySavitch
{
```

Display 16.8 Interface for the Template Class PFArrayBak *(part 2 of 2)*

```
    template<class T>
    class PFArrayBak : public PFArray<T>
    {
    public:
        PFArrayBak( );
        //Initializes with a capacity of 50.

        PFArrayBak(int capacityValue);

        PFArrayBak(const PFArrayBak<T>& Object);

        void backup( );
        //Makes a backup copy of the partially filled array.

        void restore( );
        //Restores the partially filled array to the last saved version.
        //If backup has never been invoked, this empties the partially
        //filled array.

        PFArrayBak<T>& operator =(const PFArrayBak<T>& rightSide);

        virtual ~PFArrayBak( );
    private:
        T *b; //for a backup of main array.
        int usedB; //backup for inherited member variable used.
    };

}// PFArraySavitch
#endif //PFARRAY_H
```

Display 16.9 Implementation for the Template Class PFArrayBak *(part 1 of 3)*

```
//This is the file pfarraybak.cpp.
//This is the implementation for the template class PFArrayBak. The
//interface for the template class PFArrayBak is in the file pfarraybak.h.
#include "pfarraybak.h"
#include <iostream>
using std::cout;

namespace PFArraySavitch
{

    template<class T>
    PFArrayBak<T>::PFArrayBak( ) : PFArray<T>( ), usedB(0)
```

Display 16.9 Implementation for the Template Class PFArrayBak *(part 2 of 3)*

```
{
    b = new T[getCapacity( )];
}

template<class T>
PFArrayBak<T>::PFArrayBak(int capacityValue)
               : PFArray<T>(capacityValue), usedB(0)
{
    b = new T[getCapacity( )];
}

template<class T>
PFArrayBak<T>::PFArrayBak(const PFArrayBak<T>& oldObject)
              : PFArray<T>(oldObject), usedB(0)
{
    b = new T[getCapacity( )];
    usedB = oldObject.getNumberUsed();
    for (int i = 0; i < usedB; i++)
        b[i] = oldObject.b[i];
}
template<class T>
void PFArrayBak<T>::backup( )
{
    usedB = getNumberUsed( );
    for (int i = 0; i < usedB; i++)
        b[i] = operator[](i);
}

template<class T>
void PFArrayBak<T>::restore( )
{
    emptyArray( );

    for (int i = 0; i < usedB; i++)
        addElement(b[i]);
}

template<class T>
PFArrayBak<T>& PFArrayBak<T>::operator =(const PFArrayBak<T>& rightSide)
{
    PFArray<T>::operator =(rightSide);

    if (getCapacity( ) != rightSide.getCapacity( ))
    {
        delete [] b;
        b = new T[rightSide.getCapacity( )];
    }
```

Display 16.9 Implementation for the Template Class PFArrayBak *(part 3 of 3)*

```
        usedB = rightSide.usedB;
        for (int i = 0; i < usedB; i++)
            b[i] = rightSide.b[i];

        return *this;
    }

    template<class T>
    PFArrayBak<T>::~PFArrayBak( )
    {
        delete [] b;
    }
}// PFArraySavitch
```

Display 16.10 Demonstration Program for Template Class PFArrayBak *(part 1 of 2)*

```
//Program to demonstrate the template class PFArrayBak.
#include <iostream>
#include <string>
using std::cin;
using std::cout;
using std::endl;
using std::string;

#include "pfarraybak.h"
#include "pfarray.cpp"
#include "pfarraybak.cpp"
using PFArraySavitch::PFArrayBak;

int main( )
{
    int cap;
    cout << "Enter capacity of this super array: ";
    cin >> cap;
    PFArrayBak<string> a(cap);

    cout << "Enter " << cap << " strings\n";
    cout << "separated by blanks.\n";

    string next;
    for (int i = 0; i < cap; i++)
    {
        cin >> next;
        a.addElement(next);
    }
```

Do not forget to include the implementation of the base class template

Display 16.10 Demonstration Program for Template Class PFArrayBak *(part 2 of 2)*

```
    int count = a.getNumberUsed( );
    cout << "The following " << count
         << " strings read and stored:\n";
    int index;
    for (index = 0; index < count; index++)
        cout << a[index] << " ";
    cout << endl;

    cout << "Backing up array.\n";
    a.backup( );

    cout << "Emptying array.\n";
    a.emptyArray( );
    cout << a.getNumberUsed( )
         << " strings are now stored in the array.\n";

    cout << "Restoring array.\n";
    a.restore( );
    count = a.getNumberUsed( );
    cout << "The following " << count
         << " strinss are now stored:\n";
    for (index = 0; index < count; index++)
        cout << a[index] << " ";
    cout << endl;

    cout << "End of demonstration.\n";
    return 0;
}
```

Sample Dialogue

```
Enter capacity of this super array: 3
Enter 3 strings
separated by blanks.
I love you
The following 3 strings read and stored:
I love you
Backing up array.
Emptying array.
0 strings are now stored in the array.
Restoring array.
The following 3 strings are now stored:
I love you
End of demonstration.
```

Self-Test Exercises

13. Is it legal for a derived template class to start as shown below? The template class `TwoDimPFArrayBak` is designed to be a two-dimensional partially filled array with backup.

```
template<class T>
class TwoDimPFArrayBak : public PFArray< PFArray<T> >
{
public:
    TwoDimPFArrayBak( );
```

Note that the space in `< PFArray<T> >` is important, or at least the last space is. If the space between the next-to-last > and the last > is omitted, then the compiler may interpret >> to be the extraction operator used for input in expressions like `cin >> n;` rather than interpreting it as a nested < >.

14. Give the heading for the default (zero-argument) constructor for the class `TwoDimPFArrayBak` given in Self-Test Exercise 13. (Assume all instance variables are initialized in the body of the constructor definition, so you are not being ask to do that.)

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.
- The predefined `vector` and `basic_string` classes are actually template classes.
- You can define a template class that is a derived class of a template base 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. e. 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 type defined array, or int, float, double, etc.), but T must be a type for which the code in the template makes sense. For example, for the swapValues template function (Display 16.1), the type T must have a correctly working assignment operator.

5. The function declaration and function definition are given below. They are basically identical to those for the versions given in Display 5.6 except that two instances of int are changed to T in the parameter list.
   Function declaration:

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

   Definition:

```
template<class T>
int search(const T a[], int numberUsed, T target)
{
    int index = 0;
    bool found = false;
    while ((!found) && (index < numberUsed))
        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 it 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 template function body makes sense for that type.

7. No, you cannot use an array with base type `DayOfYear` with the template function `sort` because the < operator is not defined on values of type `DayOfYear`. (If you overload < , as we discussed in Chapter 8, to give a suitable ordering on values of type `DayOfYear`, then you can use an array with base type `DayOfYear` with the template function `sort`. For example, you might overload < so it means one date comes before the other on the calendar and then sort an array of dates by calendar ordering.)

8.
```
a contains: 0 1 2
b contains: 0 100 200
c contains: 0 1 2
After swapping a and b,
and changing b:
a contains: 0 100 200
b contains: 42 1 2
c contains: 42 1 2
```

Note that before `swapValues(a, b);` `c` is an alias (another name for) the array `a`. After `swapValues(a, b);` `c` is an alias for `b`. Although the values of `a` and `b` are in some sense swapped, things are not as simple as you might have hoped. With pointer variables, there can be side effects of using `swapValues`.

The point illustrated here is that the assignment operator is not as well behaved as you might want on array pointers and so the template `swapValues` does not work as you might want with variables that are pointers to arrays. The assignment operator does not do element by element swapping but merely swaps two pointers. So, the `swapValues` function used with pointers to arrays simply swaps two pointers. It might be best to not use `swapValues` with pointers to arrays (or any other pointers), unless you are very aware of how it behaves on the pointers. The `swapValues` template function used with a type `T` is only as good, or as bad, as the assignment operator is on type `T`.

9. Since the type can be any type at all, 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.
}
```

10.
```
int addUp(const Pair<int>& thePair)
{
    return (thePair.getFirst( ) + thePair.getSecond( ));
}
```

11.
```
template<class T>
T addUp(const Pair<T>& thePair)
{
    return (thePair.getFirst( ) + thePair.getSecond( ));
}
```

12. You add the following to the public section of the template class definition of `PFArray`:

```
friend void showData(PFArray<T> theObject);
//Displays the data in theObject to the screen.
//Assumes that << is defined for values of type T.
```

You also need to add a function template definition of `showData`. One possible definition is as follows:

```
namespace PFArraySavitch
{
    template<class T>
    void showData(PFArray< T > theObject)
    {
        for (int i = 0; i < theObject.used; i++)
            cout << theObject[i] << endl;
    }
}//PFArraySavitch
```

13. Yes, it is perfectly legal. There are other, possibly preferable, ways to accomplish the same thing, but this is legal and not even crazy.

14.
```
template<class T>
TwoDimPFArrayBak<T>::TwoDimPFArrayBak( )
                  : PFArray< PFArray<T> >( )
```