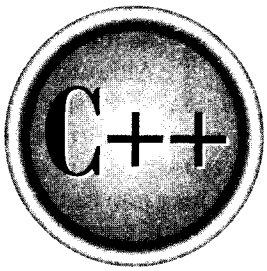
The Complete Reference



Chapter 18

Templates

The template is one of C++'s most sophisticated and high-powered features. Although not part of the original specification for C++, it was added several years ago and is supported by all modern C++ compilers. Using templates, it is possible to create generic functions and classes. In a generic function or class, the type of data upon which the function or class operates is specified as a parameter. Thus, you can use one function or class with several different types of data without having to explicitly recode specific versions for each data type. Both generic functions and generic classes are discussed in this chapter.

Generic Functions

A generic function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter. Through a generic function, a single general procedure can be applied to a wide range of data. As you probably know, many algorithms are logically the same no matter what type of data is being operated upon. For example, the Quicksort sorting algorithm is the same whether it is applied to an array of integers or an array of floats. It is just that the type of the data being sorted is different. By creating a generic function, you can define the nature of the algorithm, independent of any data. Once you have done this, the compiler will automatically generate the correct code for the type of data that is actually used when you execute the function. In essence, when you create a generic function you are creating a function that can automatically overload itself.

A generic function is created using the keyword **template**. The normal meaning of the word "template" accurately reflects its use in C++. It is used to create a template (or framework) that describes what a function will do, leaving it to the compiler to fill in the details as needed. The general form of a template function definition is shown here:

```
template <class Ttype> ret-type func-name(parameter list)
{
   // body of function
}
```

Here, *Ttype* is a placeholder name for a data type used by the function. This name may be used within the function definition. However, it is only a placeholder that the compiler will automatically replace with an actual data type when it creates a specific version of the function. Although the use of the keyword **class** to specify a generic type in a **template** declaration is traditional, you may also use the keyword **typename**.

The following example creates a generic function that swaps the values of the two variables with which it is called. Because the general process of exchanging two values is independent of the type of the variables, it is a good candidate for being made into a generic function.

```
// Function template example.
#include <iostream>
using namespace std;

// This is a function template.
template <class X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}

int main()
{
  int i=10, j=20;
  double x=10.1, y=23.3;
  char a='x', b='z';

  cout << "Original i, j: " << i << ' ' << j << '\n';
  cout << "Original x, y: " << x << ' ' << y << '\n';
  cout << "Original a, b: " << a << ' ' << b << '\n';

  swapargs(i, j); // swap integers
  swapargs(x, y); // swap floats
  swapargs(a, b); // swap chars

  cout << "Swapped i, j: " << i << ' ' << j << '\n';
  cout << "Swapped x, y: " << x << ' ' << y << '\n';
  cout << "Swapped a, b: " << a << ' ' << b << '\n';

  return 0;
}
```

Let's look closely at this program. The line:

```
template <class X> void swapargs(X &a, X &b)
```

tells the compiler two things: that a template is being created and that a generic definition is beginning. Here, **X** is a generic type that is used as a placeholder. After the **template** portion, the function **swapargs( )** is declared, using **X** as the data type of the values that will be swapped. In **main( )**, the **swapargs( )** function is called using three

different types of data: **int**s, **double**s, and **char**s. Because **swapargs( )** is a generic function, the compiler automatically creates three versions of **swapargs( )**: one that will exchange integer values, one that will exchange floating-point values, and one that will swap characters.

Here are some important terms related to templates. First, a generic function (that is, a function definition preceded by a **template** statement) is also called a *template function*. Both terms will be used interchangeably in this book. When the compiler creates a specific version of this function, it is said to have created a *specialization*. This is also called a *generated function*. The act of generating a function is referred to as *instantiating* it. Put differently, a generated function is a specific instance of a template function.

Since C++ does not recognize end-of-line as a statement terminator, the **template** clause of a generic function definition does not have to be on the same line as the function's name. The following example shows another common way to format the **swapargs( )** function.

```
template <class X>
void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

If you use this form, it is important to understand that no other statements can occur between the **template** statement and the start of the generic function definition. For example, the fragment shown next will not compile.

```
// This will not compile.
template <class X>
int i; // this is an error
void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

As the comments imply, the **template** specification must directly precede the function definition.

A Function with Two Generic Types

You can define more than one generic data type in the **template** statement by using a comma-separated list. For example, this program creates a template function that has two generic types.

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

template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
  cout << x << ' ' << y << '\n';
}

int main()
{
  myfunc(10, "I like C++");

  myfunc(98.6, 19L);

  return 0;
}
```

In this example, the placeholder types **type1** and **type2** are replaced by the compiler with the data types **int** and **char** *, and **double** and **long**, respectively, when the compiler generates the specific instances of **myfunc( )** within **main( )**.

Remember *When you create a template function, you are, in essence, allowing the compiler to generate as many different versions of that function as are necessary for handling the various ways that your program calls the function.*

Explicitly Overloading a Generic Function

Even though a generic function overloads itself as needed, you can explicitly overload one, too. This is formally called *explicit specialization*. If you overload a generic function, that overloaded function overrides (or "hides") the generic function relative to that specific version. For example, consider the following revised version of the argument-swapping example shown earlier.

```
// Overriding a template function.
#include <iostream>
using namespace std;
```

```
template <class X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
  cout << "Inside template swapargs.\n";
}

// This overrides the generic version of swapargs() for ints.
void swapargs(int &a, int &b)
{
  int temp;

  temp = a;
  a = b;
  b = temp;
  cout << "Inside swapargs int specialization.\n";
}

int main()
{
  int i=10, j=20;
  double x=10.1, y=23.3;
  char a='x', b='z';

  cout << "Original i, j: " << i << ' ' << j << '\n';
  cout << "Original x, y: " << x << ' ' << y << '\n';
  cout << "Original a, b: " << a << ' ' << b << '\n';

  swapargs(i, j); // calls explicitly overloaded swapargs()
  swapargs(x, y); // calls generic swapargs()
  swapargs(a, b); // calls generic swapargs()

  cout << "Swapped i, j: " << i << ' ' << j << '\n';
  cout << "Swapped x, y: " << x << ' ' << y << '\n';
  cout << "Swapped a, b: " << a << ' ' << b << '\n';

  return 0;
}
```

This program displays the following output.

```
Original i, j: 10 20
Original x, y: 10.1 23.3
Original a, b: x z
Inside swapargs int specialization.
Inside template swapargs.
Inside template swapargs.
Swapped i, j: 20 10
Swapped x, y: 23.3 10.1
Swapped a, b: z x
```

As the comments inside the program indicate, when **swapargs(i, j)** is called, it invokes the explicitly overloaded version of **swapargs( )** defined in the program. Thus, the compiler does not generate this version of the generic **swapargs( )** function, because the generic function is overridden by the explicit overloading.

Recently, a new-style syntax was introduced to denote the explicit specialization of a function. This new method uses the **template** keyword. For example, using the new-style specialization syntax, the overloaded **swapargs( )** function from the preceding program looks like this.

```
// Use new-style specialization syntax.
template<> void swapargs<int>(int &a, int &b)
{
  int temp;

  temp = a;
  a = b;
  b = temp;
  cout << "Inside swapargs int specialization.\n";
}
```

As you can see, the new-style syntax uses the **template<>** construct to indicate specialization. The type of data for which the specialization is being created is placed inside the angle brackets following the function name. This same syntax is used to specialize any type of generic function. While there is no advantage to using one specialization syntax over the other at this time, the new-style is probably a better approach for the long term.

Explicit specialization of a template allows you to tailor a version of a generic function to accommodate a unique situation—perhaps to take advantage of some performance boost that applies to only one type of data, for example. However, as a general rule, if you need to have different versions of a function for different data types, you should use overloaded functions rather than templates.

Overloading a Function Template

In addition to creating explicit, overloaded versions of a generic function, you can also overload the **template** specification itself. To do so, simply create another version of the template that differs from any others in its parameter list. For example:

```
// Overload a function template declaration.
#include <iostream>
using namespace std;

// First version of f() template.
template <class X> void f(X a)
{
  cout << "Inside f(X a)\n";
}

// Second version of f() template.
template <class X, class Y> void f(X a, Y b)
{
  cout << "Inside f(X a, Y b)\n";
}

int main()
{
  f(10);     // calls f(X)
  f(10, 20); // calls f(X, Y)

  return 0;
}
```

Here, the template for **f( )** is overloaded to accept either one or two parameters.

Using Standard Parameters with Template Functions

You can mix standard parameters with generic type parameters in a template function. These nongeneric parameters work just like they do with any other function. For example:

```
// Using standard parameters in a template function.
#include <iostream>
using namespace std;
```

```
const int TABWIDTH = 8;

// Display data at specified tab position.
template<class X> void tabOut(X data, int tab)
{
  for(; tab; tab--)
    for(int i=0; i<TABWIDTH; i++) cout << ' ';

  cout << data << "\n";
}

int main()
{
  tabOut("This is a test", 0);
  tabOut(100, 1);
  tabOut('X', 2);
  tabOut(10/3, 3);

  return 0;
}
```

Here is the output produced by this program.

```
This is a test
        100
                X
                        3
```

In the program, the function **tabOut( )** displays its first argument at the tab position requested by its second argument. Since the first argument is a generic type, **tabOut( )** can be used to display any type of data. The **tab** parameter is a standard, call-by-value parameter. The mixing of generic and nongeneric parameters causes no trouble and is, indeed, both common and useful.

Generic Function Restrictions

Generic functions are similar to overloaded functions except that they are more restrictive. When functions are overloaded, you may have different actions performed within the body of each function. But a generic function must perform the same general action for all versions—only the type of data can differ. Consider the overloaded

functions in the following example program. These functions could *not* be replaced by a generic function because they do not do the same thing.

```
#include <iostream>
#include <cmath>
using namespace std;

void myfunc(int i)
{
  cout << "value is: " << i << "\n";
}

void myfunc(double d)
{
  double intpart;
  double fracpart;

  fracpart = modf(d, &intpart);
  cout << "Fractional part: " << fracpart;
  cout << "\n";
  cout << "Integer part: " << intpart;
}

int main()
{
  myfunc(1);
  myfunc(12.2);

  return 0;
}
```

Applying Generic Functions

Generic functions are one of C++'s most useful features. They can be applied to all types of situations. As mentioned earlier, whenever you have a function that defines a generalizable algorithm, you can make it into a template function. Once you have done so, you may use it with any type of data without having to recode it. Before moving on to generic classes, two examples of applying generic functions will be given. They illustrate how easy it is to take advantage of this powerful C++ feature.

A Generic Sort

Sorting is exactly the type of operation for which generic functions were designed. Within wide latitude, a sorting algorithm is the same no matter what type of data is being sorted. The following program illustrates this by creating a generic bubble sort. While the bubble sort is a rather poor sorting algorithm, its operation is clear and uncluttered and it makes an easy-to-understand example. The **bubble( )** function will sort any type of array. It is called with a pointer to the first element in the array and the number of elements in the array.

```
// A Generic bubble sort.
#include <iostream>
using namespace std;

template <class X> void bubble(
  X *items,  // pointer to array to be sorted
  int count) // number of items in array
{
  register int a, b;
  X t;

  for(a=1; a<count; a++)
    for(b=count-1; b>=a; b--)
      if(items[b-1] > items[b]) {
        // exchange elements
        t = items[b-1];
        items[b-1] = items[b];
        items[b] = t;
      }
}

int main()
{
  int iarray[7] = {7, 5, 4, 3, 9, 8, 6};
  double darray[5] = {4.3, 2.5, -0.9, 100.2, 3.0};

  int i;

  cout << "Here is unsorted integer array: ";
  for(i=0;  i<7; i++)
```

```
    cout << iarray[i] << ' ';
  cout << endl;

  cout << "Here is unsorted double array: ";
  for(i=0; i<5; i++)
    cout << darray[i] << ' ';
  cout << endl;

  bubble(iarray, 7);
  bubble(darray, 5);

  cout << "Here is sorted integer array: ";
  for(i=0; i<7; i++)
    cout << iarray[i] << ' ';
  cout << endl;

  cout << "Here is sorted double array: ";
  for(i=0; i<5; i++)
    cout << darray[i] << ' ';
  cout << endl;

  return 0;
}
```

The output produced by the program is shown here.

```
Here is unsorted integer array: 7 5 4 3 9 8 6
Here is unsorted double array: 4.3 2.5 -0.9 100.2 3
Here is sorted integer array: 3 4 5 6 7 8 9
Here is sorted double array: -0.9 2.5 3 4.3 100.2
```

As you can see, the preceding program creates two arrays: one integer and one **double**. It then sorts each. Because **bubble( )** is a template function, it is automatically overloaded to accommodate the two different types of data. You might want to try using **bubble( )** to sort other types of data, including classes that you create. In each case, the compiler will create the right version of the function for you.

Compacting an Array

Another function that benefits from being made into a template is called **compact( )**. This function compacts the elements in an array. It is not uncommon to want to remove elements from the middle of an array and then move the remaining elements down so

that all unused elements are at the end. This sort of operation is the same for all types of arrays because it is independent of the type data actually being operated upon. The generic **compact( )** function shown in the following program is called with a pointer to the first element in the array, the number of elements in the array, and the starting and ending indexes of the elements to be removed. The function then removes those elements and compacts the array. For the purposes of illustration, it also zeroes the unused elements at the end of the array that have been freed by the compaction.

```
// A Generic array compaction function.
#include <iostream>
using namespace std;

template <class X> void compact(
  X *items,  // pointer to array to be compacted
  int count, // number of items in array
  int start, // starting index of compacted region
  int end)   // ending index of compacted region
{
  register int i;

  for(i=end+1; i<count; i++, start++)
    items[start] = items[i];

  /* For the sake of illustration, the remainder of
     the array will be zeroed. */
  for( ; start<count; start++) items[start] = (X) 0;
}

int main()
{
  int nums[7] = {0, 1, 2, 3, 4, 5, 6};
  char str[18] = "Generic Functions";

  int i;

  cout << "Here is uncompacted integer array: ";
  for(i=0;  i<7; i++)
    cout << nums[i] << ' ';
  cout << endl;

  cout << "Here is uncompacted string: ";
  for(i=0;  i<18; i++)
```

```
    cout << str[i] << ' ';
  cout << endl;

  compact(nums, 7, 2, 4);
  compact(str, 18, 6, 10);

  cout << "Here is compacted integer array: ";
  for(i=0;  i<7; i++)
    cout << nums[i] << ' ';
  cout << endl;

  cout << "Here is compacted string: ";
  for(i=0;  i<18; i++)
    cout << str[i] << ' ';
  cout << endl;

  return 0;
}
```

This program compacts two different types of arrays. One is an integer array, and the other is a string. However, the **compact( )** function will work for any type of array. The output from this program in shown here.

```
Here is uncompacted integer array: 0 1 2 3 4 5 6
Here is uncompacted string: G e n e r i c   F u n c t i o n s
Here is compacted integer array: 0 1 5 6 0 0 0
Here is compacted string: G e n e r i c t i o n s
```

As the preceding examples illustrate, once you begin to think in terms of templates, many uses will naturally suggest themselves. As long as the underlying logic of a function is independent of the data, it can be made into a generic function.

Generic Classes

In addition to generic functions, you can also define a generic class. When you do this, you create a class that defines all the algorithms used by that class; however, the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.

Generic classes are useful when a class uses logic that can be generalized. For example, the same algorithms that maintain a queue of integers will also work for a queue of characters, and the same mechanism that maintains a linked list of mailing

addresses will also maintain a linked list of auto part information. When you create a generic class, it can perform the operation you define, such as maintaining a queue or a linked list, for any type of data. The compiler will automatically generate the correct type of object, based upon the type you specify when the object is created.

The general form of a generic class declaration is shown here:

```
template <class Ttype> class class-name {
  .
  .
  .
}
```

Here, *Ttype* is the placeholder type name, which will be specified when a class is instantiated. If necessary, you can define more than one generic data type using a comma-separated list.

Once you have created a generic class, you create a specific instance of that class using the following general form:

```
class-name <type> ob;
```

Here, *type* is the type name of the data that the class will be operating upon. Member functions of a generic class are themselves automatically generic. You need not use **template** to explicitly specify them as such.

In the following program, the **stack** class (first introduced in Chapter 11) is reworked into a generic class. Thus, it can be used to store objects of any type. In this example, a character stack and a floating-point stack are created, but any data type can be used.

```
// This function demonstrates a generic stack.
#include <iostream>
using namespace std;

const int SIZE = 10;

// Create a generic stack class
template <class StackType> class stack {
  StackType stck[SIZE]; // holds the stack
  int tos; // index of top-of-stack

public:
  stack() { tos = 0; } // initialize stack
  void push(StackType ob); // push object on stack
```

```
  StackType pop(); // pop object from stack
};

// Push an object.
template <class StackType> void stack<StackType>::push(StackType ob)
{
  if(tos==SIZE) {
    cout << "Stack is full.\n";
    return;
  }
  stck[tos] = ob;
  tos++;
}
// Pop an object.
template <class StackType> StackType stack<StackType>::pop()
{
  if(tos==0) {
    cout << "Stack is empty.\n";
    return 0; // return null on empty stack
  }
  tos--;
  return stck[tos];
}

int main()
{
  // Demonstrate character stacks.
  stack<char> s1, s2;  // create two character stacks
  int i;

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";

  // demonstrate double stacks
  stack<double> ds1, ds2;  // create two double stacks
```

```
  ds1.push(1.1);
  ds2.push(2.2);
  ds1.push(3.3);
  ds2.push(4.4);
  ds1.push(5.5);
  ds2.push(6.6);

  for(i=0; i<3; i++) cout << "Pop ds1: " << ds1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Pop ds2: " << ds2.pop() << "\n";

  return 0;
}
```

As you can see, the declaration of a generic class is similar to that of a generic function. The actual type of data stored by the stack is generic in the class declaration. It is not until an object of the stack is declared that the actual data type is determined. When a specific instance of **stack** is declared, the compiler automatically generates all the functions and variables necessary for handling the actual data. In this example, two different types of stacks are declared. Two are integer stacks. Two are stacks of **doubles**. Pay special attention to these declarations:

```
stack<char> s1, s2;  // create two character stacks
stack<double> ds1, ds2;  // create two double stacks
```

Notice how the desired data type is passed inside the angle brackets. By changing the type of data specified when **stack** objects are created, you can change the type of data stored in that stack. For example, by using the following declaration, you can create another stack that stores character pointers.

```
stack<char *> chrptrQ;
```

You can also create stacks to store data types that you create. For example, if you want to use the following structure to store address information,

```
struct addr {
  char name[40];
  char street[40];
  char city[30];
  char state[3];
```

```
  char zip[12];
};
```

then to use **stack** to generate a stack that will store objects of type **addr**, use a declaration like this:

```
stack<addr> obj;
```

As the **stack** class illustrates, generic functions and classes are powerful tools that you can use to maximize your programming efforts, because they allow you to define the general form of an object that can then be used with any type of data. You are saved from the tedium of creating separate implementations for each data type with which you want the algorithm to work. The compiler automatically creates the specific versions of the class for you.

An Example with Two Generic Data Types

A template class can have more than one generic data type. Simply declare all the data types required by the class in a comma-separated list within the **template** specification. For example, the following short example creates a class that uses two generic data types.

```
/* This example uses two generic data types in a
   class definition.
*/
#include <iostream>
using namespace std;

template <class Type1, class Type2> class myclass
{
  Type1 i;
  Type2 j;
public:
  myclass(Type1 a, Type2 b) { i = a; j = b; }
  void show() { cout << i << ' ' << j << '\n'; }
};

int main()
{
  myclass<int, double> ob1(10, 0.23);
  myclass<char, char *> ob2('X', "Templates add power.");
```

```
  ob1.show(); // show int, double
  ob2.show(); // show char, char *

  return 0;
}
```

This program produces the following output:

```
10 0.23
X Templates add power.
```

The program declares two types of objects. **ob1** uses **int** and **double** data. **ob2** uses a character and a character pointer. For both cases, the compiler automatically generates the appropriate data and functions to accommodate the way the objects are created.

Applying Template Classes: A Generic Array Class

To illustrate the practical benefits of template classes, let's look at one way in which they are commonly applied. As you saw in Chapter 15, you can overload the [ ] operator. Doing so allows you to create your own array implementations, including "safe arrays" that provide run-time boundary checking. As you know, in C++, it is possible to overrun (or underrun) an array boundary at run time without generating a run-time error message. However, if you create a class that contains the array, and allow access to that array only through the overloaded [ ] subscripting operator, then you can intercept an out-of-range index.

By combining operator overloading with a template class, it is possible to create a generic safe-array type that can be used for creating safe arrays of any data type. This type of array is shown in the following program:

```
// A generic safe array example.
#include <iostream>
#include <cstdlib>
using namespace std;

const int SIZE = 10;

template <class AType> class atype {
  AType a[SIZE];
public:
  atype() {
```

```
    register int i;
    for(i=0; i<SIZE; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Provide range checking for atype.
template <class AType> AType &atype<AType>::operator[](int i)
{
  if(i<0 || i> SIZE-1) {
    cout << "\nIndex value of ";
    cout << i << " is out-of-bounds.\n";
    exit(1);
  }
  return a[i];
}

int main()
{
  atype<int> intob; // integer array
  atype<double> doubleob; // double array

  int i;

  cout << "Integer array: ";
  for(i=0; i<SIZE; i++) intob[i] = i;
  for(i=0; i<SIZE; i++) cout << intob[i] << "  ";
  cout << '\n';

  cout << "Double array: ";
  for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;
  for(i=0; i<SIZE; i++) cout << doubleob[i] << "  ";
  cout << '\n';

  intob[12] = 100; // generates runtime error

  return 0;
}
```

This program implements a generic safe-array type and then demonstrates its use by creating an array of **int**s and an array of **double**s. You should try creating other types of arrays. As this example shows, part of the power of generic classes is that they

allow you to write the code once, debug it, and then apply it to any type of data without having to re-engineer it for each specific application.

Using Non-Type Arguments with Generic Classes

In the template specification for a generic class, you may also specify non-type arguments. That is, in a template specification you can specify what you would normally think of as a standard argument, such as an integer or a pointer. The syntax to accomplish this is essentially the same as for normal function parameters: simply include the type and name of the argument. For example, here is a better way to implement the safe-array class presented in the preceding section. It allows you to specify the size of the array.

```
// Demonstrate non-type template arguments.
#include <iostream>
#include <cstdlib>
using namespace std;

// Here, int size is a non-type argument.
template <class AType, int size> class atype {
  AType a[size]; // length of array is passed in size
public:
  atype() {
    register int i;
    for(i=0; i<size; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Provide range checking for atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
  if(i<0 || i> size-1) {
    cout << "\nIndex value of ";
    cout << i << " is out-of-bounds.\n";
    exit(1);
  }
  return a[i];
}

int main()
{
```

```
  atype<int, 10> intob;       // integer array of size 10
  atype<double, 15> doubleob; // double array of size 15

  int i;

  cout << "Integer array: ";
  for(i=0; i<10; i++) intob[i] = i;
  for(i=0; i<10; i++) cout << intob[i] << "  ";
  cout << '\n';

  cout << "Double array: ";
  for(i=0; i<15; i++) doubleob[i] = (double) i/3;
  for(i=0; i<15; i++) cout << doubleob[i] << "  ";
  cout << '\n';

  intob[12] = 100; // generates runtime error

  return 0;
}
```

Look carefully at the template specification for **atype**. Note that **size** is declared as an **int**. This parameter is then used within **atype** to declare the size of the array **a**. Even though **size** is depicted as a "variable" in the source code, its value is known at compile time. This allows it to be used to set the size of the array. **size** is also used in the bounds checking within the **operator[ ]( )** function. Within **main( )**, notice how the integer and floating-point arrays are created. The second parameter specifies the size of each array.

Non-type parameters are restricted to integers, pointers, or references. Other types, such as **float**, are not allowed. The arguments that you pass to a non-type parameter must consist of either an integer constant, or a pointer or reference to a global function or object. Thus, non-type parameters should themselves be thought of as constants, since their values cannot be changed. For example, inside **operator[ ]( )**, the following statement is not allowed.

```
size = 10; // Error
```

Since non-type parameters are treated as constants, they can be used to set the dimension of an array, which is a significant, practical benefit.

As the safe-array example illustrates, the use of non-type parameters greatly expands the utility of template classes. Although the information contained in the non-type argument must be known at compile-time, this restriction is mild compared with the power offered by non-type parameters.

Using Default Arguments with Template Classes

A template class can have a default argument associated with a generic type. For example,

```
template <class X=int> class myclass { //...
```

Here, the type **int** will be used if no other type is specified when an object of type **myclass** is instantiated.

It is also permissible for non-type arguments to take default arguments. The default value is used when no explicit value is specified when the class is instantiated. Default arguments for non-type parameters are specified using the same syntax as default arguments for function parameters.

Here is another version of the safe-array class that uses default arguments for both the type of data and the size of the array.

```
// Demonstrate default template arguments.
#include <iostream>
#include <cstdlib>
using namespace std;

// Here, AType defaults to int and size defaults to 10.
template <class AType=int, int size=10> class atype {
  AType a[size]; // size of array is passed in size
public:
  atype() {
    register int i;
    for(i=0; i<size; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Provide range checking for atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
  if(i<0 || i> size-1) {
    cout << "\nIndex value of ";
    cout << i << " is out-of-bounds.\n";
    exit(1);
  }
```

```
  return a[i];
}

int main()
{
  atype<int, 100> intarray;  // integer array, size 100
  atype<double> doublearray; // double array, default size
  atype<> defarray;          // default to int array of size 10

  int i;

  cout << "int array: ";
  for(i=0; i<100; i++) intarray[i] = i;
  for(i=0; i<100; i++) cout << intarray[i] << "  ";
  cout << '\n';

  cout << "double array: ";
  for(i=0; i<10; i++) doublearray[i] = (double) i/3;
  for(i=0; i<10; i++) cout << doublearray[i] << "  ";
  cout << '\n';

  cout << "defarray array: ";
  for(i=0; i<10; i++) defarray[i] = i;
  for(i=0; i<10; i++) cout << defarray[i] << "  ";
  cout << '\n';

  return 0;
}
```

Pay close attention to this line:

```
template <class AType=int, int size=10> class atype {
```

Here, **AType** defaults to type **int**, and **size** defaults to **10**. As the program illustrates, **atype** objects can be created three ways:

- explicitly specifying both the type and size of the array
- explicitly specifying the type, but letting the size default to 10
- letting the type default to **int** and the size default to 10

The use of default arguments—especially default types—adds versatility to your template classes. You can provide a default for the type of data most commonly used while still allowing the user of your classes to specialize them as needed.

Explicit Class Specializations

As with template functions, you can create an explicit specialization of a generic class. To do so, use the **template<>** construct, which works the same as it does for explicit function specializations. For example:

```
// Demonstrate class specialization.
#include <iostream>
using namespace std;

template <class T> class myclass {
  T x;
public:
  myclass(T a) {
    cout << "Inside generic myclass\n";
    x = a;
  }
  T getx() { return x; }
};

// Explicit specialization for int.
template <> class myclass<int> {
  int x;
public:
  myclass(int a) {
    cout << "Inside myclass<int> specialization\n";
    x = a * a;
  }
  int getx() { return x; }
};

int main()
{
  myclass<double> d(10.1);
  cout << "double: " << d.getx() << "\n\n";

  myclass<int> i(5);
```

```
  cout << "int: " << i.getx() << "\n";

  return 0;
}
```

This program displays the following output:

```
Inside generic myclass
double: 10.1

Inside myclass<int> specialization
int: 25
```

In the program, pay close attention to this line:

```
template <> class myclass<int> {
```

It tells the compiler that an explicit integer specialization of **myclass** is being created. This same general syntax is used for any type of class specialization.

Explicit class specialization expands the utility of generic classes because it lets you easily handle one or two special cases while allowing all others to be automatically processed by the compiler. Of course, if you find that you are creating too many specializations, you are probably better off not using a template class in the first place.

The typename and export Keywords

Recently, two keywords were added to C++ that relate specifically to templates: **typename** and **export**. Both play specialized roles in C++ programming. Each is briefly examined.

The **typename** keyword has two uses. First, as mentioned earlier, it can be substituted for the keyword **class** in a template declaration. For example, the **swapargs( )** template function could be specified like this:

```
template <typename X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
```

```
  a = b;
  b = temp;
}
```

Here, **typename** specifies the generic type **X**. There is no difference between using **class** and using **typename** in this context.

The second use of **typename** is to inform the compiler that a name used in a template declaration is a type name rather than an object name. For example,

```
typename X::Name someObject;
```

ensures that **X::Name** is treated as a type name.

The **export** keyword can precede a **template** declaration. It allows other files to use a template declared in a different file by specifying only its declaration rather than duplicating its entire definition.

The Power of Templates

Templates help you achieve one of the most elusive goals in programming: the creation of reusable code. Through the use of template classes you can create frameworks that can be applied over and over again to a variety of programming situations. For example, consider the **stack** class. When first shown in Chapter 11, it could only be used to store integer values. Even though the underlying algorithms could be used to store any type of data, the hard-coding of the data type into the **stack** class severely limited its application. However, by making **stack** into a generic class, it can create a stack for any type of data.

Generic functions and classes provide a powerful tool that you can use to amplify your programming efforts. Once you have written and debugged a template class, you have a solid software component that you can use with confidence in a variety of different situations. You are saved from the tedium of creating separate implementations for each data type with which you want the class to work.

While it is true that the template syntax can seem a bit intimidating at first, the rewards are well worth the time it takes to become comfortable with it. Template functions and classes are already becoming commonplace in programming, and this trend is expected to continue. For example, the STL (Standard Template Library) defined by C++ is, as its name implies, built upon templates. One last point: although templates add a layer of abstraction, they still ultimately compile down to the same, high-performance object code that you have come to expect from C++.