Templates and Exceptions

CHAPTER

14

IN THIS CHAPTER

This chapter introduces two advanced C++ features: templates and exceptions. Templates make it possible to use one function or class to handle many different data types. Exceptions provide a convenient, uniform way to handle errors that occur within classes. These features are combined in a single chapter largely for historical reasons: they became part of C++ at the same time. They were not part of the original specification for C++, but were introduced as "Experimental" topics in Ellis and Stroustrup (1990, see Appendix H, "Bibliography"). Subsequently they were incorporated into Standard C++.

The template concept can be used in two different ways: with functions and with classes. We'll look at function templates first, then go on to class templates, and finally to exceptions.

Function Templates

Suppose you want to write a function that returns the absolute value of two numbers. As you no doubt remember from high school algebra, the absolute value of a number is its value without regard to its sign: The absolute value of 3 is 3, and the absolute value of –3 is also 3. Ordinarily this function would be written for a particular data type:

```
int abs(int n)             //absolute value of ints
   {
   return (n<0) ? -n : n;  //if n is negative, return -n
   }
```

Here the function is defined to take an argument of type `int` and to return a value of this same type. But now suppose you want to find the absolute value of a type `long`. You will need to write a completely new function:

```
long abs(long n)           //absolute value of longs
   {
   return (n<0) ? -n : n;
   }
```

And again, for type `float`:

```
float abs(float n)          //absolute value of floats
   {
   return (n<0) ? -n : n;
   }
```

The body of the function is written the same way in each case, but they are completely different functions because they handle arguments and return values of different types. It's true that in C++ these functions can all be overloaded to have the same name, but you must nevertheless

write a separate definition for each one. (In the C language, which does not support overloading, functions for different types can't even have the same name. In the C function library this leads to families of similarly named functions, such as `abs()`, `fabs()`, `fabsl()`, `labs()`, and `cabs()`.

Rewriting the same function body over and over for different types is time-consuming and wastes space in the listing. Also, if you find you've made an error in one such function, you'll need to remember to correct it in each function body. Failing to do this correctly is a good way to introduce inconsistencies into your program.

It would be nice if there were a way to write such a function just once, and have it work for many different data types. This is exactly what function templates do for you. The idea is shown schematically in Figure 14.1.

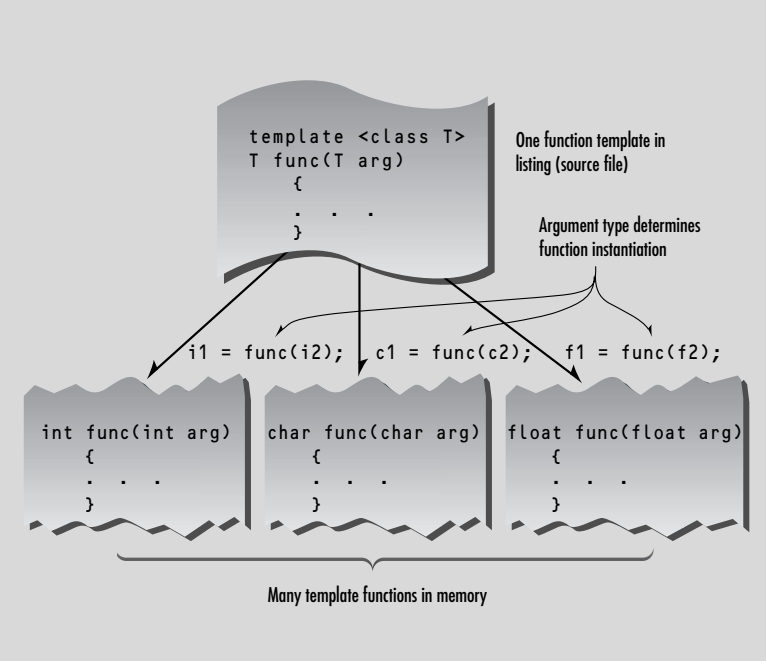

FIGURE 14.1
A function template.

A Simple Function Template

Our first example shows how to write our absolute-value function as a template, so that it will work with any basic numerical type. This program defines a template version of `abs()` and then, in `main()`, invokes this function with different data types to prove that it works. Here's the listing for TEMPABS:

```
// tempabs.cpp
// template used for absolute value function
#include <iostream>
using namespace std;
//-------------------------------------------------------------
template <class T>             //function template
T abs(T n)
    {
    return (n < 0) ? -n : n;
    }
//-------------------------------------------------------------
int main()
    {
    int int1 = 5;
    int int2 = -6;
    long lon1 = 70000L;
    long lon2 = -80000L;
    double dub1 = 9.95;
    double dub2 = -10.15;
                               //calls instantiate functions
    cout << "\nabs(" << int1 << ")=" << abs(int1);  //abs(int)
    cout << "\nabs(" << int2 << ")=" << abs(int2);  //abs(int)
    cout << "\nabs(" << lon1 << ")=" << abs(lon1);  //abs(long)
    cout << "\nabs(" << lon2 << ")=" << abs(lon2);  //abs(long)
    cout << "\nabs(" << dub1 << ")=" << abs(dub1);  //abs(double)
       cout << "\nabs(" << dub2 << ")=" << abs(dub2);  //abs(double)
       cout << endl;
       return 0;
    }
```

Here's the output of the program:

```
abs(5)=5
abs(-6)=6
abs(70000)=70000
abs(-80000)=80000
abs(9.95)=9.95
abs(-10.15)=10.15
```

As you can see, the `abs()` function now works with all three of the data types (`int`, `long`, and `double`) that we use as arguments. It will work on other basic numerical types as well, and it will even work on user-defined data types, provided that the less-than operator (<) and the unary minus operator (-) are appropriately overloaded.

Here's how we specify the `abs()` function to work with multiple data types:

```
template <class T>          //function template
T abs(T n)
   {
   return (n<0) ? -n : n;
   }
```

This entire syntax, with a first line starting with the keyword `template` and the function definition following, is called a *function template*. How does this new way of writing `abs()` give it such amazing flexibility?

Function Template Syntax

The key innovation in function templates is to represent the data type used by the function not as a specific type such as `int`, but by a name that can stand for *any* type. In the preceding function template, this name is `T`. (There's nothing magic about this name; it can be anything you want, like `Type`, or `anyType`, or `FooBar`.) The `template` keyword signals the compiler that we're about to define a function template. The keyword `class`, within the angle brackets, might just as well be called `type`. As we've seen, you can define your own data types using classes, so there's really no distinction between types and classes. The variable following the keyword class (`T` in this example) is called the *template argument*.

Throughout the definition of the template, whenever a specific data type such as `int` would ordinarily be written, we substitute the template argument, `T`. In the `abs()` template this name appears only twice, both in the first line (the function declarator), as the argument type and return type. In more complex functions it may appear numerous times throughout the function body as well.

What the Compiler Does

What does the compiler do when it sees the `template` keyword and the function definition that follows it? Well, nothing right away. The function template itself doesn't cause the compiler to generate any code. It can't generate code because it doesn't know yet what data type the function will be working with. It simply remembers the template for possible future use.

Code generation doesn't take place until the function is actually called (invoked) by a statement within the program. In TEMPABS this happens in expressions like `abs(int1)` in the statement

```
cout << "\nabs(" << int << ")=" << abs(int1);
```

When the compiler sees such a function call, it knows that the type to use is `int`, because that's the type of the argument `int1`. So it generates a specific version of the `abs()` function for type `int`, substituting `int` wherever it sees the name `T` in the function template. This is called *instantiating* the function template, and each instantiated version of the function is called a *template function*. (That is, a *template function* is a specific instance of a *function template*. Isn't English fun?)

The compiler also generates a call to the newly instantiated function, and inserts it into the code where `abs(int1)` is. Similarly, the expression `abs(lon1)` causes the compiler to generate a version of `abs()` that operates on type `long` and a call to this function, while the `abs(dub1)` call generates a function that works on type `double`. Of course, the compiler is smart enough to generate only one version of `abs()` for each data type. Thus, even though there are two calls to the `int` version of the function, the code for this version appears only once in the executable code.

Simplifying the Listing

Notice that the amount of RAM used by the program is the same whether we use the template approach or actually write three separate functions. The template approaches simply saves us from having to type three separate functions into the source file. This makes the listing shorter and easier to understand. Also, if we want to change the way the function works, we need to make the change in only one place in the listing instead of three places.

The Deciding Argument

The compiler decides how to compile the function based entirely on the data type used in the function call's argument (or arguments). The function's return type doesn't enter into this decision. This is similar to the way the compiler decides which of several overloaded functions to call.

Another Kind of Blueprint

We've seen that a function template isn't really a function, since it does not actually cause program code to be placed in memory. Instead it is a pattern, or blueprint, for making many functions. This fits right into the philosophy of OOP. It's similar to the way a class isn't anything concrete (such as program code in memory), but a blueprint for making many similar objects.

Function Templates with Multiple Arguments

Let's look at another example of a function template. This one takes three arguments: two that are template arguments and one of a basic type. The purpose of this function is to search an array for a specific value. The function returns the array index for that value if it finds it, or –1 if it can't find it. The arguments are a pointer to the array, the value to search for, and the size of the array. In `main()` we define four different arrays of different types, and four values to

search for. We treat type `char` as a number. Then we call the template function once for each array. Here's the listing for TEMPFIND:

```
// tempfind.cpp
// template used for function that finds number in array
#include <iostream>
using namespace std;
//-----------------------------------------------------------
//function returns index number of item, or -1 if not found
template <class atype>
int find(atype* array, atype value, int size)
    {
    for(int j=0; j<size; j++)
       if(array[j]==value)
          return j;
    return -1;
    }
//-----------------------------------------------------------
char chrArr[] =   {1, 3, 5, 9, 11, 13};  //array
char ch = 5;                             //value to find
int intArr[] =    {1, 3, 5, 9, 11, 13};
int in = 6;
long lonArr[] =   {1L, 3L, 5L, 9L, 11L, 13L};
long lo = 11L;
double dubArr[] = {1.0, 3.0, 5.0, 9.0, 11.0, 13.0};
double db = 4.0;

int main()
    {
    cout << "\n 5 in chrArray: index=" << find(chrArr, ch, 6);
    cout << "\n 6 in intArray: index=" << find(intArr, in, 6);
    cout << "\n11 in lonArray: index=" << find(lonArr, lo, 6);
    cout << "\n 4 in dubArray: index=" << find(dubArr, db, 6);
      cout << endl;
      return 0;
    }
```

Here we name the template argument `atype`. It appears in two of the function's arguments: as the type of a pointer to the array, and as the type of the item to be matched. The third function argument, the array size, is always type `int`; it's not a template argument. Here's the output of the program:

```
 5 in chrArray: index=2
 6 in intArray: index=-1
11 in lonArray: index=4
 4 in dubArray: index=-1
```

The compiler generates four different versions of the function, one for each type used to call it. It finds a 5 at index 2 in the character array, does not find a 6 in the integer array, and so on.

Template Arguments Must Match

When a template function is invoked, all instances of the same template argument must be of the same type. For example, in `find()`, if the array name is of type `int`, the value to search for must also be of type `int`. You can't say

```
int intarray[] = {1, 3, 5, 7};       //int array
float f1 = 5.0;                      //float value
int value = find(intarray, f1, 4);   //uh, oh
```

because the compiler expects all instances of `atype` to be the same type. It can generate a function

```
find(int*, int, int);
```

but it can't generate

```
find(int*, float, int);
```

because the first and second arguments must be the same type.

Syntax Variation

Some programmers put the `template` keyword and the function declarator on the same line:

```
template<class atype> int find(atype* array, atype value, int size)
   {
   //function body
   }
```

Of course the compiler is happy enough with this format, but we find it more forbidding and less clear than the multiline approach.

More Than One Template Argument

You can use more than one template argument in a function template. For example, suppose you like the idea of the `find()` function template, but you aren't sure how large an array it might be applied to. If the array is too large then type `long` would be necessary for the array size, instead of type `int`. On the other hand, you don't want to use type `long` if you don't need to. You want to select the type of the array size, as well as the type of data stored, when you call the function. To make this possible, you could make the array size into a template argument as well. We'll call it `btype`:

```
template <class atype, class btype>
btype find(atype* array, atype value, btype size)
   {
   for(btype j=0; j<size; j++)   //note use of btype
      if(array[j]==value)
         return j;
   return static_cast<btype>(-1);
   }
```

Now you can use either type `int` or type `long` (or even a user-defined type) for the size, whichever is appropriate. The compiler will generate different functions based not only on the type of the array and the value to be searched for, but also on the type of the array size.

Note that multiple template arguments can lead to many functions being instantiated from a single template. Two such arguments, if there were six basic types that could reasonably be used for each one, would allow the creation of 36 functions. This can take up a lot of memory if the functions are large. On the other hand, you don't instantiate a version of the function unless you actually call it.

Why Not Macros?

Old-time C programmers may wonder why we don't use macros to create different versions of a function for different data types. For example, the `abs()` function could be defined as

```
#define abs(n) ( (n<0) ? (-n) : (n) )
```

This has a similar effect to the class template in TEMPABS, because it performs a simple text substitution and can thus work with any type. However, as we've noted before, macros aren't much used in C++. There are several problems with them. One is that macros don't perform any type checking. There may be several arguments to the macro that should be of the same type, but the compiler won't check whether or not they are. Also, the type of the value returned isn't specified, so the compiler can't tell if you're assigning it to an incompatible variable. In any case, macros are confined to functions that can be expressed in a single statement. There are also other, more subtle, problems with macros. On the whole it's best to avoid them.

What Works?

How do you know whether you can instantiate a template function for a particular data type? For example, could you use the `find()` function from TEMPFIND to find a C-string (type `char*`) in an array of C-strings? To see whether this is possible, check the operators used in the function. If they all work on the data type, you can probably use it. In `find()`, however, we compare two variables using the equal-to (`==`) operator. You can't use this operator with C-strings; you must use the `strcmp()` library function. Thus `find()` won't work on C-strings. However, it does work on the `string` class because that class overloads the `==` operator.

Start with a Normal Function

When you write a template function you're probably better off starting with a normal function that works on a fixed type (`int`, for example). You can design and debug it without having to worry about template syntax and multiple types. Then, when everything works properly, you can turn the function definition into a template and check that it works for additional types.

Class Templates

The template concept can be extended to classes. Class templates are generally used for data storage (container) classes. (We'll see a major example of this in the next chapter, "The Standard Template Library.") Stacks and linked lists, which we encountered in previous chapters, are examples of data-storage classes. However, the examples of these classes that we presented could store data of only a single basic type. The `Stack` class in the STAKARAY program in Chapter 7, "Arrays and Strings," for example, could store data only of type `int`. Here's a condensed version of that class.

```
class Stack
   {
   private:
      int st[MAX];         //array of ints
      int top;             //index number of top of stack
   public:
      Stack();             //constructor
      void push(int var);  //takes int as argument
      int pop();           //returns int value
   };
```

If we wanted to store data of type `long` in a stack, we would need to define a completely new class:

```
class LongStack
   {
   private:
      long st[MAX];         //array of longs
      int top;              //index number of top of stack
   public:
      LongStack();          //constructor
      void push(long var);  //takes long as argument
      long pop();           //returns long value
   };
```

Similarly, we would need to create a new stack class for every data type we wanted to store. It would be nice to be able to write a single class specification that would work for variables of all types, instead of a single basic type. As you may have guessed, class templates allow us to do this. We'll create a variation of STAKARAY that uses a class template. Here's the listing for TEMPSTAK:

```
// tempstak.cpp
// implements stack class as a template
#include <iostream.h>
using namespace std;
const int MAX = 100;            //size of array
//////////////////////////////////////////////////////////////
template <class Type>
class Stack
   {
   private:
      Type st[MAX];             //stack: array of any type
      int top;                  //number of top of stack
   public:
      Stack()                   //constructor
         { top = -1; }
      void push(Type var)       //put number on stack
         { st[++top] = var; }
      Type pop()                //take number off stack
         { return st[top--]; }
   };
//////////////////////////////////////////////////////////////
int main()
   {
   Stack<float> s1;      //s1 is object of class Stack<float>

   s1.push(1111.1F);     //push 3 floats, pop 3 floats
   s1.push(2222.2F);
   s1.push(3333.3F);
   cout << "1: " << s1.pop() << endl;
   cout << "2: " << s1.pop() << endl;
   cout << "3: " << s1.pop() << endl;

   Stack<long> s2;       //s2 is object of class Stack<long>

   s2.push(123123123L);  //push 3 longs, pop 3 longs
   s2.push(234234234L);
   s2.push(345345345L);
   cout << "1: " << s2.pop() << endl;
   cout << "2: " << s2.pop() << endl;
   cout << "3: " << s2.pop() << endl;
   return 0;
   }
```

Here the class Stack is presented as a template class. The approach is similar to that used in function templates. The template keyword and class Stack signal that the entire class will be a template.

```
template <class Type>
class Stack
   {
   //data and member functions using template argument Type
   };
```

A template argument, named Type in this example, is then used (instead of a fixed data type such as int) everyplace in the class specification where there is a reference to the type of the array st. There are three such places: the definition of st, the argument type of the push() function, and the return type of the pop() function.

Class templates differ from function templates in the way they are instantiated. To create an actual function from a function template, you call it using arguments of a specific type. Classes, however, are instantiated by defining an object using the template argument.

```
Stack<float> s1;
```

This creates an object, s1, a stack that stores numbers of type float. The compiler provides space in memory for this object's data, using type float wherever the template argument Type appears in the class specification. It also provides space for the member functions (if these have not already been placed in memory by another object of type Stack<float>). These member functions also operate exclusively on type float. Figure 14.2 shows how a class template and definitions of specific objects cause these objects to be placed in memory.

Creating a Stack object that stores objects of a different type, as in

```
Stack<long> s2;
```

creates not only a different space for data, but also a new set of member functions that operate on type long.

Note that the name of the type of s1 consists of the class name Stack *plus the template argument*: Stack<float>. This distinguishes it from other classes that might be created from the same template, such as Stack<int> or Stack<long>.

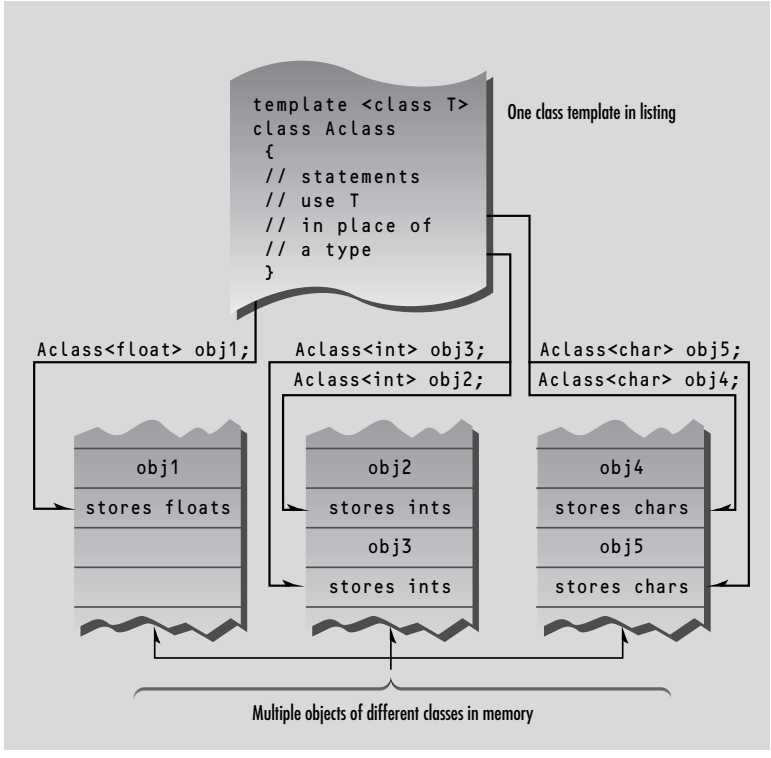

FIGURE 14.2
A class template.

In TEMPSTAK we exercise the `s1` and `s2` stacks by pushing and popping three values on each one and displaying each popped value. Here's the output:

```
1: 3333.3       //float stack
2: 2222.2
3: 1111.1
1: 345345345    //long stack
2: 234234234
3: 123123123
```

In this example the template approach gives us two classes for the price of one, and we could instantiate class objects for other numerical types with just a single line of code.

Class Name Depends on Context

In the TEMPSTAK example, the member functions of the class template were all defined within the class. If the member functions are defined externally (outside of the class specification), we need a new syntax. The next program shows how this works. Here's the listing for TEMPSTAK2:

```
// temstak2.cpp
// implements stack class as a template
// member functions are defined outside the class
#include <iostream>
using namespace std;
const int MAX = 100;
//////////////////////////////////////////////////////////////
template <class Type>
class Stack
   {
   private:
      Type st[MAX];             //stack: array of any type
      int top;                  //number of top of stack
   public:
      Stack();                  //constructor
      void push(Type var);      //put number on stack
      Type pop();               //take number off stack
   };
//////////////////////////////////////////////////////////////
template<class Type>
Stack<Type>::Stack()            //constructor
   {
   top = -1;
   }
//-----------------------------------------------------------
template<class Type>
void Stack<Type>::push(Type var) //put number on stack
   {
   st[++top] = var;
   }
//-----------------------------------------------------------
template<class Type>
Type Stack<Type>::pop()          //take number off stack
   {
   return st[top--];
   }
//-----------------------------------------------------------
int main()
   {
   Stack<float> s1;      //s1 is object of class Stack<float>
```

```
   s1.push(1111.1F);     //push 3 floats, pop 3 floats
   s1.push(2222.2F);
   s1.push(3333.3F);
   cout << "1: " << s1.pop() << endl;
   cout << "2: " << s1.pop() << endl;
   cout << "3: " << s1.pop() << endl;

   Stack<long> s2;       //s2 is object of class Stack<long>

   s2.push(123123123L);  //push 3 longs, pop 3 longs
   s2.push(234234234L);
   s2.push(345345345L);
   cout << "1: " << s2.pop() << endl;
   cout << "2: " << s2.pop() << endl;
   cout << "3: " << s2.pop() << endl;
   return 0;
   }
```

The expression `template<class Type>` must precede not only the class definition, but each externally defined member function as well. Here's how the `push()` function looks:

```
template<class Type>
void Stack<Type>::push(Type var)
   {
   st[++top] = var;
   }
```

The name `Stack<Type>` is used to identify the class of which `push()` is a member. In a normal non-template member function the name `Stack` alone would suffice:

```
void Stack::push(int var)  //Stack() as a non-template function
   {
   st[++top] = var;
   }
```

but for a function template we need the template argument as well: `Stack<Type>`.

Thus we see that the name of the template class is expressed differently in different contexts. Within the class specification, it's simply the name itself: `Stack`. For externally defined member functions, it's the class name plus the template argument name: `Stack<Type>`. When you define actual objects for storing a specific data type, it's the class name plus this specific type: `Stack<float>`, for example.

```
class Stack                        //Stack class specifier
   {   };

void Stack<Type>::push(Type var)  //push() definition
   {   }

Stack<float> s1;                   //object of type Stack<float>
```

You must exercise considerable care to use the correct name in the correct context. It's easy to forget to add the `<Type>` or `<float>` to the `Stack`. The compiler hates it when you get it wrong.

Although it's not demonstrated in this example, you must also be careful of the syntax when a member function returns a value of its own class. Suppose we define a class `Int` that provided safety features for integers, as discussed in Exercise 4 in Chapter 8, "Operator Overloading." If you used an external definition for a member function `xfunc()` of this class that returned type `Int`, you would need to use `Int<Type>` for the return type as well as preceding the scope resolution operator:

```
Int<Type> Int<Type>::xfunc(Int arg)
   {   }
```

The class name used as a type of a function argument, on the other hand, doesn't need to include the `<Type>` designation.

A Linked List Class Using Templates

Let's look at another example where templates are used for a data storage class. This is a modification of our LINKLIST program from Chapter 10, "Pointers," which you are encouraged to reexamine. It requires not only that the `linklist` class itself be made into a template, but that the `link` structure, which actually stores each data item, be made into a template as well. Here's the listing for TEMPLIST:

```
// templist.cpp
// implements linked list as a template
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
template<class TYPE>                  //struct link<TYPE>
struct link                           //one element of list
//within this struct definition 'link' means link<TYPE>
   {
   TYPE data;                         //data item
   link* next;                        //pointer to next link
   };
////////////////////////////////////////////////////////////////
template<class TYPE>                  //class linklist<TYPE>
class linklist                        //a list of links
//within this class definition 'linklist' means linklist<TYPE>
   {
   private:
      link<TYPE>* first;              //pointer to first link
   public:
      linklist()                      //no-argument constructor
         { first = NULL; }            //no first link
```

```
      //note: destructor would be nice; not shown for simplicity
      void additem(TYPE d);           //add data item (one link)
      void display();                 //display all links
   };
////////////////////////////////////////////////////////////////
template<class TYPE>
void linklist<TYPE>::additem(TYPE d)  //add data item

   {
   link<TYPE>* newlink = new link<TYPE>;  //make a new link
   newlink->data = d;                 //give it data
   newlink->next = first;             //it points to next link
   first = newlink;                   //now first points to this
   }
//-------------------------------------------------------------
template<class TYPE>
void linklist<TYPE>::display()        //display all links
   {
   link<TYPE>* current = first;       //set ptr to first link
   while( current != NULL )           //quit on last link
      {
      cout << endl << current->data;  //print data
      current = current->next;        //move to next link
      }
   }
//-------------------------------------------------------------
int main()
   {
   linklist<double> ld; //ld is object of class linklist<double>

   ld.additem(151.5);   //add three doubles to list ld
   ld.additem(262.6);
   ld.additem(373.7);
   ld.display();        //display entire list ld

   linklist<char> lch;  //lch is object of class linklist<char>

   lch.additem('a');    //add three chars to list lch
   lch.additem('b');
   lch.additem('c');
   lch.display();       //display entire list lch
   cout << endl;
   return 0;
   }
```

In `main()` we define two linked lists: one to hold numbers of type `double`, and one to hold characters of type `char`. We then exercise the lists by placing three items on each one with the `additem()` member function, and displaying all the items with the `display()` member function. Here's the output of TEMPLIST:

```
373.7
262.6
151.5
c
b
a
```

Both the `linklist` class and the `link` structure make use of the template argument `TYPE` to stand for any type. (Well, not really any type; we'll discuss later what types can actually be stored.) Thus not only `linklist` but also `link` must be templates, preceded by the line

```
template<class TYPE>
```

Notice that it's not just a class that's turned into a template. Any other programming constructs that use a variable data type must also be turned into templates, as the `link` structure is here.

As before, we must pay attention to how the class (and in this program, a structure as well) are named in different parts of the program. Within its own specification we can use the name of the class or structure alone: `linklist` and `link`. In external member functions, we must use the class or structure name and the template argument: `linklist<TYPE>`. When we actually define objects of type `linklist`, we must use the specific data type that the list is to store:

```
linklist<double> ld;  //defines object ld of class linklist<double>
```

Storing User-Defined Data Types

In our programs so far, we've used template classes to store basic data types. For example, in the TEMPLIST program we stored numbers of type `double` and type `char` in a linked list. Is it possible to store objects of user-defined types (classes) in these same template classes? The answer is yes, but with a caveat.

Employees in a Linked List

Examine the `employee` class in the EMPLOY program in Chapter 9, "Inheritance." (Don't worry about the derived classes.) Could we store objects of type `employee` on the linked list of the TEMPLIST example? As with template functions, we can find out whether a template class can operate on objects of a particular class by checking the operations the template class performs on those objects. The `linklist` class uses the overloaded insertion (`<<`) operator to display the objects it stores:

```
void linklist<TYPE>::display()
   {
   ...
   cout << endl << current->data;  //uses insertion operator (<<)
   ...
   };
```

This is not a problem with basic types, for which the insertion operator is already defined. Unfortunately, however, the `employee` class in the EMPLOY program does not overload this operator. Thus we'll need to modify the `employee` class to include it. To simplify getting employee data from the user, we overload the extraction (>>) operator as well. Data from this operator is placed in a temporary object `emptemp` before being added to the linked list. Here's the listing for TEMLIST2:

```
// temlist2.cpp
// implements linked list as a template
// demonstrates list used with employee class

#include <iostream>
using namespace std;
const int LEN = 80;           //maximum length of names
////////////////////////////////////////////////////////////////
class employee                        //employee class
   {
   private:
      char name[LEN];                 //employee name
      unsigned long number;           //employee number
   public:
      friend istream& operator >> (istream& s, employee& e);
      friend ostream& operator << (ostream& s, employee& e);
   };
//-----------------------------------------------------------
istream& operator >> (istream& s, employee& e)
   {
   cout << "\n   Enter last name: "; cin >> e.name;
   cout << "   Enter number: ";      cin >> e.number;
   return s;
   }
//-----------------------------------------------------------
ostream& operator << (ostream& s, employee& e)
   {
   cout << "\n   Name: " << e.name;
   cout << "\n   Number: " << e.number;
   return s;
   }
```

```
////////////////////////////////////////////////////////////////
template<class TYPE>                  //struct "link<TYPE>"
struct link                           //one element of list
   {
   TYPE data;                         //data item
   link* next;                        //pointer to next link
   };
////////////////////////////////////////////////////////////////
template<class TYPE>                  //class "linklist<TYPE>"
class linklist                        //a list of links
   {
   private:
      link<TYPE>* first;              //pointer to first link
   public:
      linklist()                      //no-argument constructor
         { first = NULL; }            //no first link
      void additem(TYPE d);           //add data item (one link)
      void display();                 //display all links
   };
//--------------------------------------------------------------
template<class TYPE>
void linklist<TYPE>::additem(TYPE d)  //add data item
   {
   link<TYPE>* newlink = new link<TYPE>;  //make a new link
   newlink->data = d;                 //give it data
   newlink->next = first;             //it points to next link
   first = newlink;                   //now first points to this
   }
//--------------------------------------------------------------
template<class TYPE>
void linklist<TYPE>::display()        //display all links
   {
   link<TYPE>* current = first;       //set ptr to first link
   while( current != NULL )           //quit on last link
      {
      cout << endl << current->data;  //display data
      current = current->next;        //move to next link
      }
   }
////////////////////////////////////////////////////////////////
int main()
   {                          //lemp is object of
   linklist<employee> lemp;   //class "linklist<employee>"
   employee emptemp;          //temporary employee storage
   char ans;                  //user's response ('y' or 'n')
```

```
   do
      {
      cin >> emptemp;         //get employee data from user
      lemp.additem(emptemp);  //add it to linked list 'lemp'
      cout << "\nAdd another (y/n)? ";
      cin >> ans;
      } while(ans != 'n');    //when user is done,
   lemp.display();            //display entire linked list
   cout << endl;
   return 0;
   }
```

In `main()` we instantiate a linked list called `lemp`. Then, in a loop, we ask the user to input data for an employee, and we add that employee object to the list. When the user terminates the loop, we display all the employee data. Here's some sample interaction:

```
   Enter last name: Mendez
   Enter number: 1233
Add another(y/n)? y

   Enter last name: Smith
   Enter number: 2344
Add another(y/n)? y

   Enter last name: Chang
   Enter number: 3455
Add another(y/n)? n

   Name: Chang
   Number: 3455

   Name: Smith
   Number: 2344

   Name: Mendez
   Number: 1233
```

Notice that the `linklist` class does not need to be modified in any way to store objects of type `employee`. This is the beauty of template classes: They will work not only with basic types, but with user-defined types as well.

What Can You Store?

We noted that you can tell whether you can store variables of a particular type in a data-storage template class by checking the operators in the member functions of that class. Is it possible to store a string (class `string`) in the `linklist` class in the TEMLIST2 program? Member functions in this class use the insertion (<<) and extraction (>>) operators. These operators work perfectly

well with strings, so there's no reason we can't use this class to store strings, as you can verify yourself. But if any operators exist in a storage class's member function that don't operate on a particular data type, you can't use the class to store that type.

The UML and Templates

Templates (also called *parameterized classes* in the UML) are represented in class diagrams by a variation on the UML class symbol. The names of the template arguments are placed in a dotted rectangle that intrudes into the upper right corner of the class rectangle.

Figure 14.3 shows a UML class diagram for the TEMPSTAK program at the beginning of this chapter.

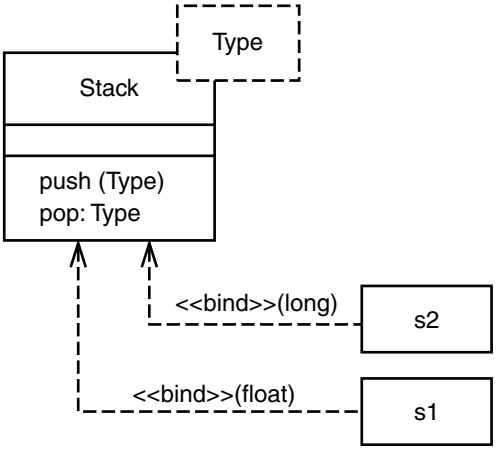

FIGURE 14.3
Template in a UML class diagram.

There's only one template argument here: `Type`. The operations `push()` and `pop()` are shown, with their return types and argument types. (Note that the return type is shown *following* the function name, separated from it by a colon.) The template argument usually shows up in the operation signatures, as `Type` does in `push()` and `pop()`.

This diagram also shows the specific classes that are instantiated from the template class: `s1` and `s2`.

Besides the depiction of templates, Figure 14.3 introduces two new UML concepts: *dependencies* and *stereotypes*.

Dependencies in the UML

A UML *dependency* is a relationship between two elements such that a change in the independent one may cause a change in the dependent one. The dependent one depends on, or uses, the independent one, so a dependency is sometimes called a *using* relationship. Here the template class is the independent element, and classes instantiated from it are dependent elements.

A dependency is shown by a dotted line with an arrow pointing to the independent element. In Figure 14.3 the instantiated classes `s1` and `s2` are dependent on template class `Stack`, because if `Stack` were to change, the instantiated classes would probably be affected.

Dependency is a very broad concept and applies to many situations in the UML. In fact, association, generalization, and the other relationships we've already seen are kinds of dependencies. However, they are important enough to be depicted in a specific way in UML diagrams.

One common dependency arises when one class uses another class as an argument in one of its operations.

Stereotypes in the UML

A stereotype is a way of specifying additional detail about a UML element. It's represented by a word in guillemets (double-angle brackets).

For example, the dotted lines in Figure 14.3 represent dependencies, but they don't tell you what kind of dependency it is. The stereotype `<<bind>>` specifies that the independent element (the template class) instantiates the dependent element (the specific class) using the specified parameters, which are shown in parentheses following the stereotype. That is, it says that `Type` will be replaced by `float` or `long`.

The UML defines many stereotypes as elements of the language. Each one applies to a specific UML element: some to classes, some to dependencies, and so on. You can also add your own.

Exceptions

Exceptions, the second major topic in this chapter, provide a systematic, object-oriented approach to handling errors generated by C++ classes. Exceptions are errors that occur at runtime. They are caused by a wide variety of exceptional circumstance, such as running out of memory, not being able to open a file, trying to initialize an object to an impossible value, or using an out-of-bounds index to a vector.

Why Do We Need Exceptions?

Why do we need a new mechanism to handle errors? Let's look at how the process was handled in the past. C-language programs often signal an error by returning a particular value from the function in which it occurred. For example, disk-file functions often return

NULL or 0 to signal an error. Each time you call one of these functions you check the return value:

```
if( somefunc() == ERROR_RETURN_VALUE )
   //handle the error or call error-handler function
else
   //proceed normally
if( anotherfunc() == NULL )
   //handle the error or call error-handler function
else
   //proceed normally
if( thirdfunc() == 0 )
   //handle the error or call error-handler function
else
   //proceed normally
```

One problem with this approach is that every single call to such a function must be examined by the program. Surrounding each function call with an `if...else` statement, and adding statements to handle the error (or call an error-handler routine), requires a lot of code and makes the listing convoluted and hard to read.

The problem becomes more complex when classes are used, since errors may take place without a function being explicitly called. For example, suppose an application defines objects of a class:

```
SomeClass obj1, obj2, obj3;
```

How will the application find out if an error occurred in the class constructor? The constructor is called implicitly, so there's no return value to be checked.

Things are complicated even further when an application uses class libraries. A class library and the application that makes use of it are often created by separate people: the class library by a vendor and the application by a programmer who buys the class library. This makes it even harder to arrange for error values to be communicated from a class member function to the program that's calling the function. The problem of communicating errors from deep within class libraries is probably the most important problem solved by exceptions. We'll return to this topic at the end of this section.

Old-time C programmers may remember another approach to catching errors: the `setjmp()` and `longjmp()` combination of functions. However, this approach is not appropriate for an object-oriented environment because it does not properly handle the destruction of objects.

Exception Syntax

Imagine an application that creates and interacts with objects of a certain class. Ordinarily the application's calls to the class member functions cause no problems. Sometimes, however, the

application makes a mistake, causing an error to be detected in a member function. This member function then informs the application that an error has occurred. When exceptions are used, this is called *throwing* an exception. In the application we install a separate section of code to handle the error. This code is called an *exception handler* or *catch block*; it *catches* the exceptions thrown by the member function. Any code in the application that uses objects of the class is enclosed in a *try block*. Errors generated in the try block will be caught in the catch block. Code that doesn't interact with the class need not be in a try block. Figure 14.4 shows the arrangement.

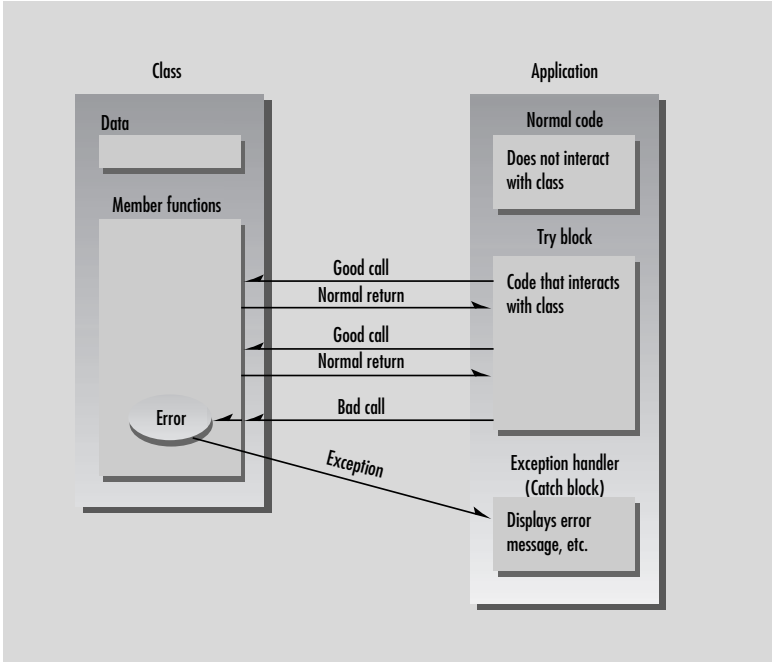

FIGURE 14.4
The exception mechanism.

The exception mechanism uses three new C++ keywords: `throw`, `catch`, and `try`. Also, we need to create a new kind of entity called an `exception class`. XSYNTAX is not a working program, but a skeleton program to show the syntax.

```
// xsyntax.cpp
// not a working program
////////////////////////////////////////////////////////////////
class AClass                    //a class
   {
```

```
   public:
   class AnError                 //exception class
      {
      };
   void Func()                   //a member function
      {
      if( /* error condition */ )
         throw AnError();        //throw exception
      }
   };
//////////////////////////////////////////////////////////////
int main()                      //application
   {
   try                          //try block
      {
      AClass obj1;              //interact with AClass objects
      obj1.Func();              //may cause error
      }
   catch(AClass::AnError)       //exception handler
      {                         //(catch block)
      //tell user about error, etc.
      }
   return 0;
   }
```

We start with a class called `AClass`, which represents any class in which errors might occur. An exception class, `AnError`, is specified in the public part of `AClass`. In `AClass`'s member functions we check for errors. If we find one, we throw an exception, using the keyword `throw` followed by the constructor for the error class:

```
throw AnError();  //'throw' followed by constructor for AnError class
```

In the `main()` part of the program we enclose any statements that interact with `AClass` in a try block. If any of these statements causes an error to be detected in an `AClass` member function, an exception will be thrown and control will go to the catch block that immediately follows the try block.

A Simple Exception Example

Let's look at a working program example that uses exceptions. This example is derived from the STAKARAY program in Chapter 7, which created a stack data structure in which integer data values could be stored. Unfortunately, this earlier example could not detect two common errors. The application program might attempt to push too many objects onto the stack, thus exceeding the capacity of the array, or it might try to pop too many objects off the stack, thus obtaining invalid data. In the XSTAK program we use an exception to handle these two errors.

```
// xstak.cpp
// demonstrates exceptions
#include <iostream>
using namespace std;
const int MAX = 3;              //stack holds 3 integers
////////////////////////////////////////////////////////////////
class Stack
   {
   private:
      int st[MAX];              //array of integers
      int top;                  //index of top of stack
   public:
      class Range               //exception class for Stack
         {                      //note: empty class body
         };

      Stack()                   //constructor
         { top = -1; }

      void push(int var)
         {
         if(top >= MAX-1)       //if stack full,
            throw Range();      //throw exception
         st[++top] = var;       //put number on stack
         }
      int pop()
         {
         if(top < 0)            //if stack empty,
            throw Range();      //throw exception
         return st[top--];      //take number off stack
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Stack s1;

   try
      {
      s1.push(11);
      s1.push(22);
      s1.push(33);
//    s1.push(44);                        //oops: stack full
      cout << "1: " << s1.pop() << endl;
      cout << "2: " << s1.pop() << endl;
      cout << "3: " << s1.pop() << endl;
      cout << "4: " << s1.pop() << endl;  //oops: stack empty
      }
```

```
catch(Stack::Range)                    //exception handler
   {
   cout << "Exception: Stack Full or Empty" << endl;
   }

cout << "Arrive here after catch (or normal exit)" << endl;
return 0;
}
```

Note that we've made the stack small so that it's easier to trigger an exception by pushing too many items.

Let's examine the features of this program that deal with exceptions. There are four of them. In the class specification there is an exception class. There are also statements that throw exceptions. In the `main()` part of the program there is a block of code that may cause exceptions (the try block), and a block of code that handles the exception (the catch block).

Specifying the Exception Class

The program first specifies an exception class within the `Stack` class:

```
class Range
   {   //note: empty class body
   };
```

Here the body of the class is empty, so objects of this class have no data and no member functions. All we really need in this simple example is the class name, `Range`. This name is used to connect a throw statement with a catch block. (The class body need not always be empty, as we'll see later.)

Throwing an Exception

In the `Stack` class an exception occurs if the application tries to pop a value when the stack is empty or tries to push a value when the stack is full. To let the application know that it has made such a mistake when manipulating a `Stack` object, the member functions of the `Stack` class check for these conditions using `if` statements, and throw an exception if they occur. In XSTAK the exception is thrown in two places, both using the statement

```
throw Range();
```

The `Range()` part of this statement invokes the implicit constructor for the `Range` class, which creates an object of this class. The `throw` part of the statement transfers program control to the exception handler (which we'll examine in a moment).

The `try` Block

All the statements in `main()` that might cause this exception—that is, statements that manipulate `Stack` objects—are enclosed in braces and preceded by the `try` keyword:

```
try
   {
   //code that operates on objects that might cause an exception
   }
```

This is simply part of the application's normal code; it's what you would need to write even if you weren't using exceptions. Not all the code in the program needs to be in a try block; just the code that interacts with the `Stack` class. Also, there can be many try blocks in your program, so you can access `Stack` objects from different places.

The Exception Handler (Catch Block)

The code that handles the exception is enclosed in braces, preceded by the `catch` keyword, with the exception class name in parentheses. The exception class name must include the class in which it is located. Here it's `Stack::Range`.

```
catch(Stack::Range)
   {
   //code that handles the exception
   }
```

This construction is called the *exception handler*. It must immediately follow the try block. In XSTAK the exception handler simply prints an error message to let the user know why the program failed.

Control "falls through" the bottom of the exception handler, so you can continue processing at that point. Or the exception handler may transfer control elsewhere, or (often) terminate the program.

Sequence of Events

Let's summarize the sequence of events when an exception occurs:

1. Code is executing normally outside a try block.
2. Control enters the try block.
3. A statement in the try block causes an error in a member function.
4. The member function throws an exception.
5. Control transfers to the exception handler (catch block) following the try block.

That's all there is to it. Notice how clean the resulting code is. Any of the statements in the try block could cause an exception, but we don't need to worry about checking a return value for each one, because the try-throw-catch arrangement handles them all automatically. In this particular example we've deliberately created two statements that cause exceptions. The first

```
s1.push(44);  //pushes too many items
```

causes an exception if you remove the comment symbol preceding it, and the second

```
cout << "4: " << s1.pop() << endl;  //pops item from empty stack
```

causes an exception if the first statement is commented out. Try it each way. In both cases the same error message will be displayed:

```
Stack Full or Empty
```

Multiple Exceptions

You can design a class to throw as many exceptions as you want. To show how this works, we'll modify the XSTAK program to throw separate exceptions for attempting to push data on a full stack and attempting to pop data from an empty stack. Here's the listing for XSTAK2:

```
// xstak2.cpp
// demonstrates two exception handlers
#include <iostream>
using namespace std;
const int MAX = 3;              //stack holds 3 integers
////////////////////////////////////////////////////////////////
class Stack
   {
   private:
      int st[MAX];              //stack: array of integers
      int top;                  //index of top of stack
   public:
      class Full { };           //exception class
      class Empty { };          //exception class
//--------------------------------------------------------------
      Stack()                   //constructor
         { top = -1; }
//--------------------------------------------------------------
      void push(int var)        //put number on stack
         {
         if(top >= MAX-1)       //if stack full,
            throw Full();       //throw Full exception
         st[++top] = var;
         }
```

```
//--------------------------------------------------------------
      int pop()                  //take number off stack
         {
         if(top < 0)             //if stack empty,
            throw Empty();       //throw Empty exception
         return st[top--];
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Stack s1;

   try
      {
      s1.push(11);
      s1.push(22);
      s1.push(33);
//    s1.push(44);                          //oops: stack full
      cout << "1: " << s1.pop() << endl;
      cout << "2: " << s1.pop() << endl;
      cout << "3: " << s1.pop() << endl;
      cout << "4: " << s1.pop() << endl;  //oops: stack empty
      }
   catch(Stack::Full)
      {
      cout << "Exception: Stack Full" << endl;
      }
   catch(Stack::Empty)
      {
      cout << "Exception: Stack Empty" << endl;
      }
   return 0;
   }
```

In XSTAK2 we specify two exception classes:

```
class Full  {  };
class Empty  {  };
```

The statement

```
throw Full();
```

is executed if the application calls `push()` when the stack is already full, and

```
throw Empty();
```

is executed if `pop()` is called when the stack is empty.

A separate catch block is used for each exception:

```
try
   {
   //code that operates on Stack objects
   }
catch(Stack::Full)
   {
   //code to handle Full exception
   }
catch(Stack::Empty)
   {
   //code to handle Empty exception
   }
```

All the catch blocks used with a particular try block must immediately follow the try block. In this case each catch block simply prints a message: "Stack Full" or "Stack Empty". Only one catch block is activated for a given exception. A group of catch blocks, or a *catch ladder*, operates a little like a `switch` statement, with only the appropriate section of code being executed. When an exception has been handled, control passes to the statement following all the catch blocks. (Unlike a switch statement, you don't need to end each catch block with a break. In this way catch blocks act more like functions.)

Exceptions with the `Distance` Class

Let's look at another example of exceptions, this one applied to the infamous `Distance` class from previous chapters. A `Distance` object has an integer value for feet and a floating-point value for inches. The inches value should always be less than 12.0. A problem with this class in previous examples has been that it couldn't protect itself if the user initialized an object with an inches value of 12.0 or greater. This could lead to trouble when the class tried to perform arithmetic, since the arithmetic routines (such as `operator +()`) assumed inches would be less than 12.0. Such impossible values could also be displayed, thus confounding the user with dimensions like 7'–15".

Let's rewrite the `Distance` class to use an exception to handle this error, as shown in XDIST:

```
// xdist.cpp
// exceptions with Distance class
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                    //English Distance class
   {
```

```
   private:
      int feet;
      float inches;
   public:
      class InchesEx { };         //exception class
//--------------------------------------------------------------
      Distance()                  //constructor (no args)
         { feet = 0; inches = 0.0; }
//--------------------------------------------------------------
      Distance(int ft, float in)  //constructor (two args)
         {
         if(in >= 12.0)           //if inches too big,
            throw InchesEx();     //throw exception
         feet = ft;
         inches = in;
         }
//--------------------------------------------------------------
      void getdist()              //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         if(inches >= 12.0)       //if inches too big,
            throw InchesEx();     //throw exception
         }
//--------------------------------------------------------------
      void showdist()             //display distance
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   try
      {
      Distance dist1(17, 3.5);    //2-arg constructor
      Distance dist2;             //no-arg constructor
      dist2.getdist();            //get distance from user
                                  //display distances
      cout << "\ndist1 = ";  dist1.showdist();
      cout << "\ndist2 = ";  dist2.showdist();
      }
   catch(Distance::InchesEx)      //catch exceptions
      {
      cout << "\nInitialization error: "
              "inches value is too large.";
      }
   cout << endl;
   return 0;
   }
```

We install an exception class called `InchesEx` in the `Distance` class. Then, whenever the user attempts to initialize the inches data to a value greater than or equal to 12.0, we throw the exception. This happens in two places: in the two-argument constructor, where the programmer may make an error supplying initial values, and in the `getdist()` function, where the user may enter an incorrect value at the *Enter inches* prompt. We could also check for negative values and other input mistakes.

In `main()` all interaction with `Distance` objects is enclosed in a try block, and the catch block displays an error message.

In a more sophisticated program, of course, you might want to handle a user error (as opposed to a programmer error) differently. It would be more user-friendly to go back to the beginning of the try block and give the user a chance to enter a another distance value.

Exceptions with Arguments

What happens if the application needs more information about what caused an exception? For instance, in the XDIST example, it might help the programmer to know what the bad inches value actually was. Also, if the same exception is thrown by different member functions, as it is in XDIST, it would be nice to know which of the functions was the culprit. Is there a way to pass such information from the member function, where the exception is thrown, to the application that catches it?

You can answer this question if you remember that throwing an exception involves not only transferring control to the handler, but also creating an object of the exception class by calling its constructor. In XDIST, for example, we create an object of type `InchesEx` when we throw the exception with the statement

```
throw InchesEx();
```

If we add data members to the exception class, we can initialize them when we create the object. The exception handler can then retrieve the data from the object when it catches the exception. It's like writing a message on a baseball and throwing it over the fence to your neighbor. We'll modify the XDIST program to do this. Here's the listing for XDIST2:

```
// xdist2.cpp
// exceptions with arguments
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                    //English Distance class
   {
```

```
   private:
      int feet;
      float inches;
   public:
//--------------------------------------------------------------
      class InchesEx              //exception class
          {
          public:
             string origin;       //for name of routine
             float iValue;        //for faulty inches value

             InchesEx(string or, float in)  //2-arg constructor
                {
                origin = or;      //store string
                iValue = in;      //store inches
                }
          };                      //end of exception class
//--------------------------------------------------------------
      Distance()                  //constructor (no args)
         { feet = 0; inches = 0.0; }
//--------------------------------------------------------------
      Distance(int ft, float in)  //constructor (two args)
         {
         if(in >= 12.0)
            throw InchesEx("2-arg constructor", in);
         feet = ft;
         inches = in;
         }
//--------------------------------------------------------------
      void getdist()              //get length from user
          {
          cout << "\nEnter feet: ";  cin >> feet;
          cout << "Enter inches: ";  cin >> inches;
         if(inches >= 12.0)
            throw InchesEx("getdist() function", inches);
          }
//--------------------------------------------------------------
      void showdist()             //display distance
          { cout << feet << "\'-" << inches << '\"'; }
   };
//////////////////////////////////////////////////////////////
int main()
   {
   try
      {
```

```
      Distance dist1(17, 3.5);    //2-arg constructor
      Distance dist2;             //no-arg constructor
      dist2.getdist();            //get value
                                  //display distances
      cout << "\ndist1 = ";  dist1.showdist();
      cout << "\ndist2 = ";  dist2.showdist();
      }
   catch(Distance::InchesEx ix)   //exception handler
      {
      cout << "\nInitialization error in " << ix.origin
           << ".\n   Inches value of " << ix.iValue
           << " is too large.";
      }
   cout << endl;
   return 0;
   }
```

There are three parts to the operation of passing data when throwing an exception: specifying the data members and a constructor for the exception class, initializing this constructor when we throw an exception, and accessing the object's data when we catch the exception. Let's look at these in turn.

Specifying Data in an Exception Class

It's convenient to make the data in an exception class public so it can be accessed directly by the exception handler. Here's the specification for the new `InchesEx` exception class in XDIST2:

```
class InchesEx               //exception class
   {
   public:
      string origin;         //for name of routine
      float iValue;          //for faulty inches value

      InchesEx(string or, float in)  //2-arg constructor
         {
         origin = or;        //put string in object
         iValue = in;        //put inches value in object
         }
   };
```

There are public variables for a `string` object, which will hold the name of the member function being called, and a type `float`, for the faulty inches value.

Initializing an Exception Object

How do we initialize the data when we throw an exception? In the two-argument constructor for the `Stack` class we say

```
throw InchesEx("2-arg constructor", in);
```

and in the `getdist()` member function for `Stack` it's

```
throw InchesEx("getdist() function", inches);
```

When the exception is thrown, the handler will display the string and inches values. The string will tell us which member function is throwing the exception, and the value of `inches` will report the faulty inches value detected by the member function. This additional data will make it easier for the programmer or user to figure out what caused the error.

Extracting Data from the Exception Object

How do we extract this data when we catch the exception? The simplest way is to make the data a public part of the exception class, as we've done here. Then in the catch block we can declare `ix` as the name of the exception object we're catching. Using this name we can refer to its data in the usual way, using the dot operator:

```
catch(Distance::InchesEx ix)
   {
   //access 'ix.origin' and 'ix.iValue' directly
   }
```

We can then display the value of `ix.origin` and `ix.iValue`. Here's some interaction with XDIST2, when the user enters too large a value for inches:

```
Enter feet: 7
Enter inches: 13.5

Initialization error in getdist() function.
   Inches value of 13.5 is too large.
```

Similarly, if the programmer changes the definition of `dist1` in `main()` to

```
Distance dist1(17, 22.25);
```

the resulting exception will cause this error message:

```
Initialization error in 2-arg constructor.
   Inches value of 22.25 is too large.
```

Of course we can make whatever use of the exception arguments we want, but they generally carry information that helps us diagnose the error that triggered the exception.

The `bad_alloc` Class

Standard C++ contains several built-in exception classes. The most commonly used is probably `bad_alloc`, which is thrown if an error occurs when attempting to allocate memory with `new`.

(This exception was called `xalloc` in earlier versions of C++. At the time of this book's publication, this older approach is still used in Microsoft Visual C++.) If you set up the appropriate `try` and `catch` blocks, you can make use of `bad_alloc` with very little effort. Here's a short example, BADALLOC, that shows how it's used:

```
// badalloc.cpp
// demonstrates bad_alloc exception
#include <iostream>
using namespace std;

int main()
   {
   const unsigned long SIZE = 10000;    //memory size
   char* ptr;                           //pointer to memory

   try
      {
      ptr = new char[SIZE];             //allocate SIZE bytes
      }
   catch(bad_alloc)                     //exception handler
      {
      cout << "\nbad_alloc exception: can't allocate memory.\n";
      return(1);
      }
   delete[] ptr;                        //deallocate memory
   cout << "\nMemory use is successful.\n";
   return 0;
   }
```

Put all the statements that use `new` in a try block. The catch block that follows handles the exception, often by displaying an error message and terminating the program.

Exception Notes

We've shown only the simplest and most common approach to using exceptions. We won't go into further detail, but we'll conclude with a few thoughts about exception usage.

Function Nesting

The statement that causes an exception need not be located directly in the try block; it can also be in a function that is called by a statement in the try block. (Or in a function called by a function that is called by a statement in the try block, and so on.) So you only need to install a try block on the program's upper level. Lower-level functions need not be so encumbered, provided they are called directly or indirectly by functions in the try block. (However, it is sometimes useful for the intermediate-level functions to add their own identifying data to the exception and rethrow it to the next level.)

Exceptions and Class Libraries

An important problem solved by exceptions is that of errors in class libraries. A library routine may discover an error, but typically it doesn't know what to do about it. After all, the library routine was written by a different person at a different time than was the program that called it. What the library routine needs to do is pass the error along to whatever program called it, saying in effect, "There's been an error. I don't know what you want to do about it, but here it is." The calling program can thus handle the error as it sees fit.

The exception mechanism provides this capability because exceptions are transmitted up through nested functions until a catch block is encountered. The throw statement may be in a library routine, but the catch block can be in the program that knows how to deal with the error.

If you're writing a class library, you should cause it to throw exceptions for anything that could cause problems to the program using it. If you're writing a program that uses a class library, you should provide try and catch blocks for any exceptions that it throws.

Not for Every Situation

Exceptions should not be used for every kind of error. They impose a certain overhead in terms of program size and (when an exception occurs) time. For example, exceptions should probably not be used for user input errors (such as inserting letters into numerical input) that are easily detectable by the program. Instead the program should use normal decisions and loops to check the user's input and ask the user to try again if necessary.

Destructors Called Automatically

The exception mechanism is surprisingly sophisticated. When an exception is thrown, a destructor is called automatically for any object that was created by the code up to that point in the try block. This is necessary because the application won't know which statement caused the exception, and if it wants to recover from the error, it will (at the very least) need to start over at the top of the try block. The exception mechanism guarantees that the code in the try block will have been "reset," at least as far as the existence of objects is concerned.

Handling Exceptions

After you catch an exception, you will sometimes want to terminate your application. The exception mechanism gives you a chance to indicate the source of the error to the user, and to perform any necessary clean-up chores before terminating. It also makes clean-up easier by executing the destructors for objects created in the try block. This allows you to release system resources, such as memory, that such objects may be using.

In other cases you will not want to terminate your program. Perhaps your program can figure out what caused the error and correct it, or the user can be asked to input different data. When this is the case, the try and catch blocks are typically embedded in a loop, so control can be

returned to the beginning of the try block (which the exception mechanism has attempted to restore to its initial state).

If there is no exception handler that matches the exception thrown, the program is unceremoniously terminated by the operating system.

Summary

Templates allow you to generate a family of functions, or a family of classes, to handle different data types. Whenever you find yourself writing several identical functions that perform the same operation on different data types, you should consider using a function template instead. Similarly, whenever you find yourself writing several different class specifications that differ only in the type of data acted on, you should consider using a class template. You'll save yourself time and the result will be a more robust and more easily maintained program that is also (once you understand templates) easier to understand.

Exceptions are a mechanism for handling C++ errors in a systematic, OOP-oriented way. An exception is typically caused by a faulty statement in a try block that operates on objects of a class. The class member function discovers the error and throws an exception, which is caught by the program using the class, in exception-handler code following the try block.

Questions

Answers to these questions can be found in Appendix G.

1. A template provides a convenient way to make a family of
   a. variables.
   b. functions.
   c. classes.
   d. programs.
2. A template argument is preceded by the keyword _____.
3. True or false: Templates automatically create different versions of a function, depending on user input.
4. Write a template for a function that always returns its argument times 2.
5. A template class
   a. is designed to be stored in different containers.
   b. works with different data types.
   c. generates objects which must all be identical.
   d. generates classes with different numbers of member functions.

6. True or false: There can be more than one template argument.
7. Creating an actual function from a template is called _____ the function.
8. Actual code for a template function is generated when
   a. the function declaration appears in the source code.
   b. the function definition appears in the source code.
   c. a call to the function appears in the source code.
   d. the function is executed at runtime.
9. The key concept in the template concept is replacing a _____ with a name that stands for _____.
10. Templates are often used for classes that _____ _____.
11. An exception is typically caused by
    a. the programmer who writes an application's code.
    b. the creator of a class who writes the class member functions.
    c. a runtime error.
    d. an operating system malfunction that terminates the program.
12. The C++ keywords used with exceptions are _____, _____, and _____.
13. Write a statement that throws an exception using the class `BoundsError`, which has an empty body.
14. True or false: Statements that might cause an exception must be part of a catch block.
15. Exceptions are thrown
    a. from the catch block to the try block.
    b. from a throw statement to the try block.
    c. from the point of the error to a catch block.
    d. from a throw statement to a catch block.
16. Write the specification for an exception class that stores an error number and an error name. Include a constructor.
17. True or false: A statement that throws an exception does not need to be located in a try block.
18. The following are errors for which an exception would typically be thrown:
    a. An excessive amount of data threatens to overflow an array.
    b. The user presses the Ctrl+C key combination to terminate the program.
    c. A power failure shuts down the system.
    d. `new` cannot obtain the requested memory.

19. Additional information sent when an exception is thrown may be placed in
    a. the `throw` keyword.
    b. the function that caused the error.
    c. the catch block.
    d. an object of the exception class.
20. True or false: A program can continue to operate after an exception has occurred.
21. If we're talking about dependencies, the template class is the _____ element and the instantiated class is the _____ element.
22. A template class is shown in the UML as
    a. an ordinary class with something added.
    b. a dashed line.
    c. a rectangle with a dashed outline.
    d. none of the above.
23. True or false: A dependency is a kind of association.
24. A stereotype gives _____ about a UML element.

Exercises

Answers to starred exercises can be found in Appendix G.

*1. Write a template function that returns the average of all the elements of an array. The arguments to the function should be the array name and the size of the array (type `int`). In `main()`, exercise the function with arrays of type `int`, `long`, `double`, and `char`.

*2. A queue is a data-storage device. It's like a stack, except that, instead of being last-in-first-out, it's first-in-first-out, like the line at a bank teller's window. If you put in 1, 2, 3, you get back 1, 2, 3 in that order.

    A stack needs only one index to an array (`top` in the STAKARAY program in Chapter 7). A queue, on the other hand, must keep track of two indexes to an array: one to the tail, where new items are added, and one to the head, where old items are removed. The tail follows the head through the array as items are added and removed. If either the tail or the head reaches the end of the array, it is reset to the beginning.

    Write a class template for a queue class. Assume that the programmer using the queue won't make any mistakes, like exceeding the capacity of the queue or trying to remove an item when the queue is empty. Define several queues of different data types and insert and remove data from them.

*3. Add exceptions to the queue template in Exercise 2. Throw two exceptions: one if the capacity of the queue is exceeded, the other if the program tries to remove an item from an empty queue. One way to handle this is to add a new data member to the queue: a count of the number of items currently in the queue. Increment the count when you insert an item, and decrement it when you remove an item. Throw an exception if this count exceeds the capacity of the queue, or if it becomes less than 0.

   You might try making the `main()` part of this exercise interactive, so the user can put values on a queue and take them off. This makes it easier to exercise the queue. Following an exception, the program should allow the user to recover from a mistake without corrupting the contents of the queue.

4. Create a function called `swaps()` that interchanges the values of the two arguments sent to it. (You will probably want to pass these arguments by reference.) Make the function into a template, so it can be used with all numerical data types (`char`, `int`, `float`, and so on). Write a `main()` program to exercise the function with several types.
5. Create a function called `amax()` that returns the value of the largest element in an array. The arguments to the function should be the address of the array and its size. Make this function into a template so it will work with an array of any numerical type. Write a `main()` program that applies this function to arrays of various types.
6. Start with the `safearay` class from the ARROVER3 program in Chapter 8. Make this class into a template, so the safe array can store any kind of data. In `main()`, create safe arrays of at least two different types, and store some data in them.
7. Start with the `frac` class and the four-function fraction calculator of Exercise 7 in Chapter 8. Make the `frac` class into a template so it can be instantiated using different data types for the numerator and denominator. These must be integer types, which pretty much restricts you to `char`, `short`, `int`, and `long` (unless you develop an integer type of your own). In `main()`, instantiate a class `frac<char>` and use it for the four-function calculator. Class `frac<char>` will take less memory than `frac<int>`, but won't be able to handle large fractions.
8. Add an exception class to the ARROVER3 program in Chapter 8 so that an out-of-bounds index will trigger the exception. The catch block can print an error message for the user.
9. Modify the exception class in Exercise 8 (adapted from ARROVER3) so that the error message in the catch block reports the value of the index that caused the exception.
10. There are various philosophies about when to use exceptions. Refer to the ENGLERR program from Chapter 12, "Streams and Files." Should user-input errors be exceptions? For this exercise, let's assume so. Add an exception class to the `Distance` class in that program. (See also the XDIST and XDIST2 examples in this chapter.) Throw an exception in all the places where ENGLERR displayed an error message. Use an argument to the exception constructor to report where the error occurred and the specific cause of the error

(inches not a number, inches out of range, and so on). Also, throw an exception when an error is found within the `isint()` function (nothing entered, too many digits, nondigit character, integer out of range). Question: If it throws exceptions, can `isint()` remain an independent function?

You can insert both the try block and the catch block within the `do` loop so that after an exception you go back to the top of the loop, ready to ask the user for more input.

You might also want to throw an exception in the two-argument constructor, in case the programmer initializes a `Distance` value with its inches member out of range.

11. Start with the STRPLUS program in Chapter 8. Add an exception class, and throw an exception in the one-argument constructor if the initialization string is too long. Throw another in the overloaded + operator if the result will be too long when two strings are concatenated. Report which of these errors has occurred.
12. Sometimes the easiest way to use exceptions is to create a new class of which an exception class is a member. Try this with a class that uses exceptions to handle file errors. Make a class `dofile` that includes an exception class and member functions to read and write files. A constructor to this class can take the filename as an argument and open a file with that name. You may also want a member function to reset the file pointer to the beginning of the file. Use the REWERR program in Chapter 12 as a model, and write a `main()` program that provides the same functionality, but does so by calling on members of the `dofile` class.