Object-Oriented Programming Using MS Visual C++ .NET Chapters 1-12, 14-17 and 22 Draft Version

by T. Grandon Gill

Part One:

Structured Programming in Visual C++ .NET

Chapter 1

Introduction to C++ and Visual Studio .NET

Executive Summary

This chapter provides a quick introduction to C++ variables, functions and the Visual Studio .NET development environment. Since it is assumed that the reader has programmed before (although not necessarily in C++), we do not spend a great deal of time on questions such as why variables are needed or the purpose of functions. Instead, the emphasis is on the mechanics of applying them in C++. The chapter ends with a walkthrough of the bare essentials needed to create and debug simple C++ projects.

The chapter begins with an overview of variables and arrays, along with the primitive C++ data types. During this discussion, the topic of variable scope is introduced, with local and global scope being contrasted. The declaration of arrays and access of array elements is also covered. The next topic covered is the declaration and definition of C++ functions. The final topic of the chapter is the use of Visual Studio .NET. The topic begins with an explanation of terms such as "project", "solution" and "configuration". Next, we create the traditional first C/C++ program, a program that prints "Hello world" on the screen. The concept of a multifile project is then introduced, beginning by bringing in a file of simple input/output functions supplied with the textbook. We then add our own file to the project and create a few simple functions. Finally, we look at some of the tools that are provided for the purposes of debugging projects: breakpoints, code stepping, inspection windows and the call stack.

Learning Objectives

Upon completing this chapter, you should be able to:

- Identify the primitive C++ data types
- Declare and initialize simple variables and arrays
- Explain the components of a function declaration
- Define a simple function that includes a return statement
- Describe the compose, compile, link and load cycle
- Distinguish between the terms solution, project and configuration, as used in Visual Studio .NET.
- Create single file and multifile console project in Visual C++ .NET.
- Use the debugger to step through your code as it runs
- Place breakpoints in your code to pause the debugger
- Use the inspection window to examine the values of variables
- Use the call stack window to see the sequence of functions being executed while your program is paused
- Use the help system to find out more information on the functions you are using

1.1: Scalar Variables and Arrays

The objective of this chapter—and, indeed, the whole of Part One—is to acquaint the reader, who is already familiar with programming in some language (not necessarily C or C++) with the basics of the C++ language and the MS Visual Studio .NET development environment. Our first step in this process is examining how our program can declare and use data, and what types of data are available.

1.1.1: Scalar Variables

A *scalar variable* is the term used to refer to a single unit of data, within a program, that has its own name. We create scalar variables by specifying the data type, followed by a variable name, followed by the ubiquitous semicolon that ends nearly every C++ statement, for example:

```
int myvar;
```

This code serves to define a single variable, an integer, and declares its name to be *myvar*. C++ also gives us the ability to define more than one variable in a given line, for example:

```
int n1,n2,n3;
```

This sets aside memory locations for three integers, giving them the names n1, n2 and n3, respectively. Some programmers believe that is not a good idea to define more than one variable per statement, to prevent confusion and allow for easier commenting.

Initializing Variables
It is also possible to initialize variables when declaring them. For example, the declaration:

```
int n4=273;
```

This declaration sets aside enough room for one integer, gives it the name n4 and places the integer value 273 in the memory we set aside, all in one step.

> *If you do not initialize a variable when you create it, there is generally no guarantee as to what value will be present in the variable. In particular, do not suppose it will be some sensible value, such as 0.*

If a variable is to be initialized when it is created, and then never changed, the **const** modifier can be used to tell the compiler to generate errors if modifications to the variable occur later in the code. For example, the statement:

```
    const int nMyConstant = 32;
```

declares the name nMyConstant to be an integer with the value of 32. That value is then not allowed to change. As a result, a statement later in the program such as:

```
    nMyConstant=31;
```

would produce a compiler error.

In C/C++, the most common use of const will be seen later, in function declarations.

Variable Names
In C/C++, the rules for naming variables are as follows:

- Variable names must begin with a letter or underscore (although the leading underscore is normally reserved for system-defined names, by convention).
- The initial character in a variable name can be followed by any combination of letters, numbers and underscores.
- C++ is *case sensitive*, meaning a variable named X is entirely different from a variable named x.

In addition to the "rules", there are also some common practices for naming variables that make a lot of sense:

- Variable names should be descriptive—a good practice for all but the smallest programs. A variable defined to hold the length of a string, for example, should be named something like length, or nLength, instead of x1.
- Some programmers precede variable names with a few characters indicating what type of variable they are. For example, strBuf (the str implies the variable contains a character string) or nLength (the n means the variable is an integer).
- Many C and C++ programmers name all their variables in lower case, using the underscore—where necessary—to separate words. For example, instead of naming a variable julyhightemp, it would be july_high_temp.
- Many C++ programmers use upper and lower case letters to break up variable names, instead of underscores (which don't always print properly!). For example, JulyHighTemp.

Being consistent in your naming practices can be very helpful in reducing complexity as your program grows. (Having said this, the present text uses a variety of practices—so you will not be surprised by what you encounter in "real world" code).

1.1.2: Arrays

Although the notion of giving a name to each unit of data that we use sounds good at first, it doesn't take a programmer very long to realize it is impractical. Suppose, for example, you wrote a program designed to analyze hourly temperature measurements over a 10-year period. Assuming roughly 8000 hours a year, you would need to come up with 80,000 different names. This would be inconvenient, to put it gently…

Declaring an Array

One approach to reducing the need to create unique names is to create an *array*. An array is a collection of data elements, all of the same type, accessible using a name and an index. We define an array just like a scalar variable—except we follow the name we are declaring by a number, in brackets, specifying the number of elements to be created. For example:

```
int arnTemp[80000];
```

would create an array of 80,000 integers, enough room to hold all the temperature measurements in our previous example. The name, arnTemp, might be used as a way of implying it is an integer array of temperatures—but we could call it anything we wanted to.

Accessing Element Data and Array Storage

Once an array has been declared, the individual elements of it can be accessed using the same name/index notation we used in declaring it. For example, the expression:

```
arnTemp[0]
```

would refer to the first element in our previously declared array. It is very important to note that *every array in C++ is zero-based*. That means arnTemp[1] is actually the second element in the array, arnTemp[2] is the third, and so forth. In Chapter 11, we will discover the benefits of using zero-based indexing. For now, just take it on faith that it isn't done this way solely for the purpose of making it counter-intuitive to novice programmers.

When C++ creates an array, it sets aside memory for the array in a single block. Doing so guarantees that the elements in the array are stored *contiguously*, meaning that element 1 is right next to element 2 is right next to element 3, etc. Storing elements in this way makes it relatively easy for C++ to access individual elements. For example, if the memory address of arnTemp[0] is 1000000, and if we know that integers take up 4 bytes apiece, then we know we can find arnTemp[1] at 1000004 and arnTemp[2] at 1000008, and so forth. This also means that *the most critical piece of information about any array is the location in memory where it starts*. Once we know that, along with size of the data it stores, we could—in theory—locate the position of any element in the array based on its coefficient using the simple formula:

Position = Array Start Address + Element Size * Coefficient

In practice, we will normally let our programming language "do the math" for us.

Initializing an Array
Just as we can initialize a scalar variable when we create it, we can also initialize an array. This is done using braces to create a comma-separated list, such as:

```
int arInit1[5]={1,2,3,5,8};
```

This would create a five-element block of integers, with anInit1[0] being set to 1, anInit1[1] being set to 2, anInit1[2] being set to 3, anInit1[3] being set to 5, and anInit1[4] being set to 8.

Not all elements need to be initialized in the initialization list. For example, the following declaration is also legal:

```
int arInit2[10]={1,2,3,5,8};
```

In this case, the first five elements are initialized to 1,2,3,5,8 (as above) and the remaining elements are zeroed out.

If an initialization list is included, C++ does not require you specify a number of values when you initialize the array—you can just specify []. The compiler will compute the number for you. For example:

```
int arInit3[]={1,2,3,5,8};
```

This will cause the compiler to count the number of initialization elements (5), and then create a five-element array. In effect, it is exactly equivalent to the previous declaration of arInit1. It just allows us to be lazier…

Finally, just as was the case for scalar variables:

- If you create an array without initializing it, the likelihood is that the contents of that array will be whatever garbage happened to be in memory at the time the array was created. As long as you initialize at least one of the arrays elements, however, the entire array will be initialized (to zeros, if values for higher coefficients are not specified).
- If you declare an array as **const**, then any attempts to change its values later in the program will result in a compiler error.

1.2: Primitive Data Types in C++

There are three basic types of built-in data—sometimes referred to as primitive data—that C++ allows us to represent:

- Integers: whole numbers that can either be unsigned (cannot take on a value less than 0) or signed (can represent positive or negative values).

- Integers: whole numbers that can be either positive or negative
- Real numbers: floating point numbers stored internally in a sort of binary scientific notation.
- Characters: a type of integer usually used to hold text information

For all data types, the type declared specifies the amount of memory each data element will hold. In this regard, numeric and textual representations of data are quite different. Written as text, the integer 1,456,783,261 takes up substantially more space than the integer 3. On the computer, however, once you declare a piece of data to be of a given type, the amount of space it takes up is fixed. In other words, it takes the same amount of memory (e.g., 4-bytes) to hold an integer, whether its value is 1,456,783,261 or 3. (The latter memory, however, will probably have a lot more zeros in it than the former). Furthermore, any given block of memory can only hold a finite number of different values. That means that every numeric data type comes with a built-in range of values it can represent. Getting a variable of that data type to hold values outside of its allowable range is impossible. Attempts to do so often lead to a condition called *overflow*, discussed in greater detail in Section 3.4.

1.2.1: Integer Data Types in C/C++

Most of the data you will typically be representing in a C/C++ program will ultimately end up being one type of integer or another. In the design of C and C++, considerable flexibility was left for how such integers could be represented. As a result, the C/C++ standards do not specify what size the various types of integers must be—only their minimum size.

C/C++ provides four basic integer types, which identify the size of the integer:

char
short
int
long

Although **short** and **long** can be used in combination with **int** (i.e., **short int** or **long int**), doing so provides no extra information—so we usually don't bother. In addition to the types, the **signed** and **unsigned** keywords can be added in declaring an integer. **signed** means the integer is can represent either positive or negative numbers, and is the default integer type. **unsigned** means the integer can only represent positive numbers. If you know a value should never be negative (e.g., the value of a count), there are two advantages of storing it in an unsigned data type: 1) you typically double the maximum value that can be represented in the variable, and 2) you never have to worry about the value being negative as the result of an error. If an integer is unsigned, it cannot hold a negative value.

Integer Limits

Whenever you declare an integer, the type of integer you declare determines the amount of memory available to hold values and, correspondingly, the range of values that integer can hold. C++ does not specify how many bytes each integer data type will use. The specific compiler we use therefore determines the sizes that are used by the different types. In Visual Studio .NET, the default sizes and ranges are listed in Table 1.1.

Type	*.NET Size (in bytes)*	*Minimum value*	*Maximum value*	*Comments*
char	1	-128	+127	Two byte characters are on the way, to support Unicode
unsigned char	1	0	+255	
short	2	-32768	+32767	Was the default size for **int** in pre-1995 MS compilers (v. 1.5 and below)
unsigned short	2	0	+65535	
int	4	-2147483648	+2147483647	**int** is currently the same as **long**
unsigned int	4	0	+4294967296	
long	4	-2147483648	+2147483647	8-byte **long** already used in C#. MS provides **_int64** 8-byte integer type.
unsigned long	4	0	+4294967296	

Table 1.1: C/C++ integer data types and their current sizes and ranges

Aside from being familiar with the order of magnitude of the ranges (e.g., its time to get very nervous when your **signed int** values start approaching 2 billion), one the most important things you should take away from Table 1.1 is in the comments column. What it points out is that integer sizes are in a state of flux. Indeed, the "standard" size for every integer type has either: 1) changed within the past decade, or 2) is likely to change in the next decade. As a programmer, this means that you need to think very carefully about any assumptions you make regarding the size of your data types. When you change compilers, you could easily discover you assumptions are no longer correct—and that your program does not run properly as a result. For example, you might save an array of 1-byte character data in one version of your program, then recompile your program using a compiler that assumes 2-byte characters. When you try to load in the previously saved array into your newly compiled program, the difference in assumed sizes causes everything gets screwed up. We will return to this topic when we discuss saving and loading information in files, in Chapters 4 and 11.

Declaring Integer Data

We have already seen some examples of declaring integers and discussed what these declarations accomplish. The following would all be examples of legal declarations in C++:

```
int a1;
unsigned short s1,s2,s3=2;
char buf[80];
long mylong;
long bigarray[5]={1,3,5,7,9};
```

To repeat, for emphasis, what we have already said several times: you must declare a variable or array in C++ before you use it.

Test your understanding 1.1: In each of the declarations above, how much memory are we setting aside in Visual Studio .NET? Could that change based on the compiler we are using?

1.2.2: Real Data Types in C/C++

C/C++ offers three real number data types, only two of which are actually supported in Visual Studio .NET. The properties of these types are presented in Table 1.2.

Type	*.NET Size (in bytes)*	*Range*	*Significant digits*	*Comments*
float	4	+/- 3.4 * 10^{38}	~ 6-7	Not used that much any more
double	8	+/- 1.7 * 10^{308}	~ 15	**double** and **long double** are same in .NET. 10 byte **long double** were used in some earlier versions
long double	8	+/- 1.7 * 10^{308}	~ 15	

Table 1.2: C/C++ real number data types and their current sizes and ranges

Real numbers are internally represented in a form of exponential notation, with a single bit used to indicate whether the number is positive or negative, a collection of bits used to represent the number itself (the mantissa bits), and a certain number of bits used to represent the exponent. Because of this approach to representation, it is rare that a real number (float or double) won't be able to handle a value of a given size (short of programs computing the number of particles in the universe). On the other hand, since a limited number of bits are used to represent each number—the already mentioned mantissa bits—numbers cannot be completely accurate under many circumstances. For example, using a **float** you only get 6 or 7 significant digits. Thus, even a number like the price of a new house could not be represented to the penny with perfect accuracy.

Real Number Programming Issues
The issues concerning real numbers are quite different from those involving the use of integers. Overflow conditions (or underflow—the presence of a non-zero number that is very small, such as 10^{-310}) rarely occur unless a serious logic error is present in the program—unlike the case for integer overflow. Twos complement issues are also irrelevant, as the real number implementation specified by the IEEE standards organizations uses a sign bit. Thus, at first glance, real numbers might seem preferable to integers from a programmer's perspective. They would appear to provide a lot fewer things to worry about.

Actually, real numbers tend to be used a lot less than integers in most programs. There are two reasons for this, one becoming relatively minor, and one quite serious:

- *Performance:* Computations involving real numbers tend to be slower than integer calculations. Although this was once a serious concern, performance issues associated with the use of real numbers have declined significantly over the past decade. While the use of real numbers on the original PC processor could reduce speed significantly—up to a factor of 100—today's architectures have circuitry that performs real number arithmetic built into the CPU. Furthermore, on today's machine it is rarely CPU speed that holds up a program—other components, such as memory and disk access, are much more likely to slow the computer down. Therefore floating point speeds and integer speeds are becoming reasonably comparable.

- *Accuracy:* The accuracy issue is the Achilles heel of real number arithmetic. While 15 digits of precision is very good, there are numbers where it may not be quite enough (e.g., the U.S. GNP in dollars and cents, the Italian GNP denominated in lire). Even worse, rounding errors can occur in arithmetic operations. For example, there is no guarantee that the expressions:

  (X*Y)/Z and X*(Y/Z)

  will be exactly equal, as a result of the rounding that occurs. They will be very, very close—but there's a big difference (in a program) between close and equal.

As a result of these two issues, we find two things happening. First, since performance is almost never decisive—and because memory and disk storage are so cheap—we almost always use the **double** in place of the **float**. Second, in applications where consistency counts, such as virtually all applications involving currency, we tend to use integers (e.g., amounts measured in pennies) rather than real numbers.

Declaring Real Number Data
There is no material difference between declaring real numbers and integers. For example, the following lines all define real numbers:

```
float a1;
double d1,d2,d3=2.0;
double buf[4] ={1.0,3.0,5.0};
float f2=125e-2;
```

When writing a real number constant in C++ (e.g., 3.00), it is usually a good idea to put in a decimal point, so the compiler knows that the value is not an integer. Exponential notation can also be used, using an E (or e) followed by an integer indicating the power of 10. For example:

```
1.25, 12.5E-1, 125e-2
```

are equivalent ways of writing the same number. (And 125e-2 is the same as writing the number $125*10^{-2}$).

1.2.3: Text and the char data type

Characters are just 1-byte integers. They can be added, subtracted, multiplied and divided just like any other integers (although, in fairness, they do tend to overflow rather regularly if you use them in this way…). What distinguishes characters from other integers, then, is mainly the way that we most commonly use them: as a convenient means to store text.

Coding Text Values

How do we move from integers to text? We need some sort of a coding scheme, allowing us to interpret numeric values as their text equivalents. Actually, there are three schemes that are, or were, in common use:

- *ASCII (American Standards Committee for Information Interchange) Character Set*: The most commonly used code, consisting of 128 integer to character conversions than include all the English alphabet, upper case and lower case, called ASCII for short. 128 more characters—including non-English letters and drawing characters—are included in the extended ASCII character set, sometimes called the IBM character set (the original character set supported on the IBM PC).
- *EBCDIC (Extended Binary Coded Decimal Interchange Code)*: a system comparable to ASCII utilized by IBM mainframes.
- *Unicode*: a 2-byte character system including numerous non-English letter symbols and Asian word-character symbols. The first 128 characters of Unicode match those of ASCII.

Throughout this text, we will assume ASCII coding—which is the character set traditionally used by C/C++. Memorizing the values of the specific codes is an entirely unnecessary (and futile) exercise—that's what the chart ou'll find in the Visual Studio .NET help system is there for. Certain facts about ASCII are useful to know, however:

- All the characters less than or equal to space (.a.k.a. ' ', 32) are non-printing. These are sometimes called "white" characters.
- All the numeric digit characters are grouped together, stating with '0' (48).
- All the uppercase letters are grouped together in sequence ('A'=65, 'B'=66, etc.).
- All the lowercase letters are grouped together in sequence ('a'=97, 'b'=98, etc.).

From time to time, we make use of these facts in our programming, as will become evident in later chapters.

One of the reasons it makes no sense to learn ASCII is that the C/C++ compiler gives us an easy way to access a letter's ASCII value without looking it up: just put it between apostrophes (single quotes). For example:

> 'A' is the same as writing 65 (or 0x41)
> 'a' is the same as writing 97 (or 0x61)
> '3' is the same as writing 51 (or 0x33)
> ' ' (space) is the same as writing 32 (or 0x20)

Naturally, non-printing characters are a bit harder to write in this way (except for the space, shown above). To help us incorporate these characters in our programs, C/C++ gives us some common escape sequences—which are just sequences of characters beginning with a \ (backslash). Some common sequences are presented in Table 1.3.

'\t' is the tab character, the same as writing 9 (or 0x09)
'\n' is the newline character, the same as writing 10 (or 0x0A)
'\r' is the carriage return character, the same as writing 13 (or 0x0D)
'\0' is the same as writing 0 (or 0x00), the NUL character
'\c' (for any character *c* that is not predefined) refers to c that follows, e.g., '\\\\' is the same as writing \\ '\\"' is the same as writing " '\\'' is the same as writing ' '\\%' is the same as writing % etc.
'\x*hh*' (where *h* represents any hexadecimal digit) can be used for any character whose code we know (e.g., '\x33' is the same as '3' or 51)

Table 1.3: Common escape sequences

Despite the fact that ASCII is used almost universally in ANSI-standard C/C++ programming, programmers need to be aware that Unicode is increasingly being used for data and operating systems, a fact that could cause serious problems if the one byte per character assumption is coded into programs that could be ported from system to system.

The use of apostrophes (a.k.a., single quotes) to represent characters, as opposed to quotation marks (a.k.a. double quotes) is very significant here. To understand why, we need to turn to the subject of NUL terminated character strings.

NUL Terminated Character Strings

Characters don't really become *really* useful until you can group them together, to form names, words, sentences and, ultimately, complete documents. It's pretty obvious that the array, introduced at the beginning of this chapter, can be very helpful to us here. Just put the sequence of characters that form your text into the array. Problem solved, right?

Actually, the pure array solution is not too bad. It is sometimes referred to as *fixed length strings* and is used by many early 3GL languages, such as FORTRAN, and is also common in database contexts (e.g., dBase III+, as we'll see in various chapter examples). It has two disadvantages, however:

1. *Wasted space:* A lot of time, when you store text you don't know how much space you'll need in advance (e.g., how many characters will a person's last name be?). If you're going to store last names in a fixed length array, you need to set aside enough space for the "worst case" scenario—meaning you waste a lot of space when someone named "Lee" comes along.
2. *Need for padding:* Even if you don't care about space, short strings cause yet another problem. If you set aside 80 characters to hold a last name, and the name

happens to be "Lee", what do you do with the remaining 77 characters? Something has to go there—memory can't be empty, it always holds some value. Normally, such a situation is handled by padding the end of the string with blanks (such as the space character, ASCII 32). Doing so, however, means that whenever you access the data in your array (e.g., to print a check), you need to do something to remove those trailing blanks (a process often referred to as trimming) if you want to avoid huge, ungainly spaces.

The way C and structured C++ handle the problem of variable string length is to define what is called a NUL terminated string. The concept is simple: whenever you create a string, you place a NUL character (ASCII value 0) at the end. This NUL character, often written as '\0', should not be confused with the character '0' (ASCII value 48, 0x30). The NUL terminator normally is used for one purpose and one purpose only: to signal the end of string.

Test your understanding 1.2: suppose we have a 10-character array in memory with the following contents:

65	66	67	68	0	0	0	0	0	0

How long is the string that starts at the beginning of the array? If we were to display the string on the console, what characters would appear? (*Hint:* 65 is the 'A' character).

Initializing Strings

We already talked about how to initialize an array using a brace enclosed list, e.g.,

```
int myarray[5]={1,2,3,5,8};
```

We can use the same technique for initializing strings, for example:

```
char mystring1[10]={97,98,99,100};
char mystring2[10]={'a','b','c','d'};
```

This approach is a bit cumbersome, however.

Test your understanding 1.3: Why are the initializations of mystring1 and mystring2 equivalent in the above example. Given what was said about initializing arrays at the beginning of this chapter, why will both of these initializations give us NUL terminated strings?

For this reason, C/C++ gives us an easier way of initializing arrays—using quotation marks (double quotes). For example:

```
char mystring3[10]="abcd";
```

does precisely the same initialization as was done for mystring1 and mystring2. Analogous to normal array initialization, we can also write:

```
char mystring4[]="abcd";
```

In this case, what happens is slightly different. The compiler counts the characters within the quotation marks, discovers there are 4. It then adds 1 to that number (to provide room for the NUL terminator), meaning it needs to set aside room for 5 characters. Knowing this, it then treats the declaration as equivalent to:

```
char mystring4[5]="abcd";
```

and proceeds as it would for the earlier case. What is important to note here is that when you are working with NUL terminated strings, you must always include an NUL at the end—even if you have the option of sizing the array to the precise size you need. There are no walls in memory to keep one array from flowing into another. So, if you leave off the NUL terminator, any of the C/C++ functions that assumes NUL terminated strings (and there are many!) will not stop where your array ends. Instead, they will keep processing characters until a byte containing 0x00 is encountered.

Strings in C++ (using OOP)
NUL terminated strings are incredibly useful, but can also be a real pain in the neck. The problem with them, as we shall see in later chapters, is that a lot of the operators we use—such as assignment (=) and tests (<, >)—don't work with them. Instead, we need to define functions with names only a geek could love (e.g., strcpy, strcmp).

When we start programming using C++ objects, one of the first things we introduce is a **string** class (a *class* is a definition of an object) that remedies a lot of these defects. We will introduce these C++ string objects in Chapter 8. Even with such classes, however, it is not uncommon to use some C/C++ functions (such as the function that does case insensitive comparison) in an OOP application. Thus, understanding NUL terminated strings is important for C++ programmers.

3.1 Section Questions

1. What is the principle benefit of defining an array, as opposed to defining ten individual scalar variables?
2. If we had an array (Test) that starts in memory address 1000, and each element was 4-bytes in size, where would element Test[2] begin in memory?
3. What are the two key pieces of data that we need to know about an array if we want to access data inside that array?
4. What would the values of each of the elements of the following array be after the following definition:

   int MyArray[5]={7,2,1};

5. What's the main difference between a local variable and a variable declared as a function argument?
6. Explain the two main uses of function declaration lines.
7. What is the difference between a signed and unsigned integer? Does the fact that an integer is signed impact the amount of space it takes to hold it?
8. Is it possible to do arithmetic operations on variables of type char?
9. Why do we need a coding system for **char** data, when we don't need one for other types of integers?
10. What is the difference between '1' and 1?
11. What are three different ways of writing the tab character?
12. What purpose does a NUL terminator usually serve in an array of characters?

1.2: Defining a Function

The central organizing construct of structured C++ is the function. The function also plays a central role in the implementation of objects. In this section, we examine how C++ functions are defined and called. Because declaring variables is a critical part of function definition, we begin by examining where data can be declared.

1.2.1: Where Are Variables Declared?

Variables are normally declared in four places:

- Within function declaration lines and prototypes
- Within code blocks, such as the block used to define a function
- External to all code blocks, within a source file
- Within definitions of complex data structures, such as **struct**, **union** and **class** objects

We will delay discussing the fourth type of declaration to Chapter 3, where we introduce the C/C++ structure. Thus, we'll limit ourselves to the first three types of declaration here.

Function Declarations
Two of the most common places where declarations appear are in the argument and return type declarations of a function. Suppose, for example, we wanted to write a function that computed a monthly mortgage payment(a real number) given the following inputs:

- Interest rate (a real number)
- Loan amount (a real number)
- Number of years (a positive integer)

The declaration of that function might look something like the following:

```
double Payment(double dRate,double dAmount,unsigned int nYears);
```

The interpretation of declaration would be as follows:

- *Payment:* the name of the function
- *double dRate,double dAmount,unsigned int nYears:* Declares the number and data types to be used when the function is called
- *double* (preceding Payment): Specifies a return type

The return type tells us what type of value we get back when we call the function. Since we stated that the monthly payment amount was a real number, it makes sense that the Payment() function should return a double.

Given the way that the Payment() function is declared, we know that it would be legal to call it as follows:

```
double dVal=Payment(0.06,10000.00,30);
```

Assuming we defined the function properly somewhere within out program, dVal would presumably contain the amount of the monthly payment for a 6%, 30-year mortgage on $100,000.00 after the line of code executed.

Function declarations can perform two roles. When ended by a semi-colon (as shown previously), the server to tell a program *how* a function is to be called (i.e., what argument types, how many arguments, what return type). When followed by braces containing code, they also represent the start of a function definition. For example:

```
double Payment(double dRate,double dAmount,unsigned int nYears)
{
        // Comment: The code to implement the function would go here
}
```

The { and } represent the start and end of a *code block* in C++. Code blocks are cirtical constructs in C++, because they allow us to group statements together and are critical for many of the constructs presented in Chapter 2. They also lead us to our second type of declarations.

Local Declarations
Any time variables are declared within a code block, they are referred to as local or automatic variables.

> *Any time a variable or array is defined within a code block, it will continue to exist only as long as that code block remains active. As soon as the program leaves the code block, the data will be discarded.*

Where local variables can be declared is one of the major differences between C and structured C++. In C, local variable declarations within a code block must always be the first statements within a code block. C++, on the other hand, is much more flexible—you can declare a local variable pretty much anywhere before you use it. For example, the following code would be legal in C++, but not in C:

```
int x,y;
x=3;
y=4;
int z=x+y; // this would not be legal in C, since the declaration follows code.
```

Global Declarations
It is also possible to declare variables and array outside of any functions. Such variables are often referred to as *global* variables. They are created and initialized when a program starts running, and are not discarded until the program exits.

The natural scope of global variables is called file scope, meaning they can be accessed by any function within the file that is defined after they are declared. Since files are compiled individually, however, creating a global variable or array in one file is not enough to let it be used by another. The problem is not with the variable itself—the problem is that the other file has no idea of how the variable is named or defined.

Global variables are a form of *static* data. What this means is that, once created and initialized, such variables remain in existence until the program stops running. In addition to local and static, a third type of data—dynamic—is provided by C++. This form of data is addressed in a later chapter (Chapter 3).

1.2.3: Components of a Function

As we have just noted, there are two key parts of a function definition:

- *Declaration:* The first portion of the function that specified the function name, number and type of return arguments, and the return value.
- *Body:* A collection of statements, within braces (i.e., { and }) that defines what the function will do.

An example of a function is presented in Example 1.1.

Example 1.1: Typical Function Definition

```
/* Plus(): Takes its two integer arguments, adds them together
   and returns the value */
int Plus(int nArg1,int nArg2)
{
      int nTotal;
      nTotal=nArg1+nArg2;
      return nTotal;
}
```

In this example, the declaration line:

int Plus(int nArg1,int nArg2)

tells us the function is called Plus, takes two arguments (both integers), and returns an integer value. Where a semicolon follows a declaration line (i.e., there is no definition) we have a function prototype. Such prototypes normally appear in header files—where they are used to tell the compiler what to check for when it sees function calls within your code.

The body, the code block that immediately follows the declaration line, defines what the function is supposed to do. Any time a statement beginning with the **return** keyword is encountered within a function body, it must be followed by an expression matching the return type of the function. Since the **int** that precedes Plus() in the declaration line tells us that Plus() is supposed to return an integer, it follows that whatever follows a return statement within the body of Plus() should also be an integer expression.

There is an exception to the rule about expressions following the **return** keyword. In the case a function's return type is specified to be **void**, the function does not return a value. In this case, the return statement is simply written as:

```
return;
```

If a return statement is omitted for a void function, the function simply returns when the end of the body is encountered. For any other function return type, a function without a return statement will produce an error.

1.2.4: Code Blocks and Indentation

We have now used the term code block several times to describe the function body. We also noted that a code block is, itself, a construct used to group C++ statements together. A simplified way of looking at program code in the C++ is as follows:

1. *Expression:* An expression is a legal combination of variable names, operators, keywords and function calls. The *grammar* of a language determines if an expression is legal or not. But in this text, we will mainly emphasize distinguishing legal expressions based on what seems to make sense. For example, of the two expressions:

   ```
   X = (Y+Z)/12
   ) X + / 12 Y Z ) =
   ```

   the first looks a lot more likely to be legal than the second—even though both contain the same set of symbols.
2. *Construct:* A collection of keywords and separators (e.g., (,), {, }) used to group expressions into more complex forms.
3. *Statement:* A statement is either:
   a. A declaration, followed by a semicolon
   b. An expression, followed by a semicolon
   c. A construct

Nature of Code Blocks
A *code block* is just a construct that consists of a collection of statements, contained within braces. It has one very nice property:

> *anywhere the C/C++ language calls for a statement, a code block can be used instead.*

As previously noted, C++ variables can be defined anywhere in the code block. Once defined, however, their existence is tied to the existence of the code block. As soon we exit the code block, they are discarded.

Since a code block can be substituted anywhere a statement is allowed—and since a code block consists of a collection of statements—it follows that code blocks can be placed within code blocks. This process of including code blocks within code blocks is a form of *nesting*. Thus, the following code structure would be perfectly legal with a C/C++ codeblock:

```
{
        some declarations…
        some statements…
        {
                some more declarations…
                {
                        some more statements…
                        {
                                even more declarations…
                                even more statements…
                        }
                }
        }
}
```

Normally, you don't see a lot of pure code blocks nested within pure code blocks—since the only benefit such a structure would provide would be in controlling the appearance and disappearance of local variables. The other major programming constructs—branches, loops, etc., to be presented in Chapters 5 and 6—all typically use code blocks, however. Thus, such nesting is common. Indeed, it leads to the typical appearance of a C/C++ program—loved by its converts, hated and feared by all others.

Indentation
In the above example, it would be hard not to notice that the nested code blocks are written using indentation to keep them organized. The C++ compiler ignores all such forms of indentation. It cares about things like braces and semicolons, how you choose to display your program is a matter of complete indifference. In fact, if it weren't for preprocessor statements, you could write your entire C/C++ application using a single line of text.

Hopefully, it is self-evident that *could* and *should* are entirely different notions here. Choosing a consistent style of indentation can help you navigate the complexities of C/C++ syntax. Being haphazard about it guarantees you will develop a close and intimate—but not necessarily friendly—relationship with your compiler, since a single brace out of place can generate hundreds of error messages.

Example 1.2: Three common indenting styles

Style 1:

```
function-declaration
{
        statement-or-codeblock
        statement-or-codeblock
        etc.
}
```

Style 2:

```
function-declaration  {
        statement-or-codeblock
        statement-or-codeblock
        etc.
}
```

Style 3:

```
function-declaration
        {
        statement-or-codeblock
        statement-or-codeblock
        etc.
        }
```

There are three common indenting styles, shown in Example 1.2 for a hypothetical function definition, which we refer to here as Style 1, Style 2 and Style 3. Being consistent in indentation is probably the single most important guideline to follow. Being practical doesn't hurt either, however. For example, the author indented code using Style 3 for many years—until the desire not to engage in a constant battle with the Visual Studio built-in editor led to a change to Style 1.

1.2 Section Questions

1. What purpose does a semicolon serve in C++?
2. What is a code block?
3. Will the compiler be impacted by the style of indenting you use when you write your program?
4. If you were to write a C++ program entirely on one line, what impact would likely result when you first compiled it?
5. What is the principle practical justification for adopting indenting style 1 or 2 from Example 1.2?

1.3: Calling a Function

In this section, we examine function calls in more details focusing, in particular, on how arguments are passed into a function.

1.3.1: How Arguments Are Passed

Programming languages offer two ways to pass arguments into a function: by value and by reference. When arguments are passed by value, a copy of the arguments is made—and therefore all changes to the arguments within the function are made to the copy, not the original value. The default behavior in C++ is to pass arguments in this manner.

When arguments are passed by reference, on the other hand, any changes made to the parameters inside the function are reflected in the value of the variable being passed. Unlike C++, most languages pass arguments in this way. (C++ also allows parameters to be passed by reference, to be discussed in Section 1.4.1). There are two main reasons for passing by reference:

- Since a return statement can only specify a single value, reference arguments can be used to allow functions to "return" more complex information—by changing argument values.
- Since passing by reference does not require a copy of the argument to be made, it is often more efficient to pass large amounts of data into functions using references.

As it turns out, the use of pointers (Chapter 3) in C++ allows us to gain many of the advantages of passing by reference. Furthermore, passing arguments by value also offers some valuable benefits:

- It avoids *side-effects*—changes made by a function to variables that aren't directly related to its return value. Functions that only affect their return value tend to be the easiest for a programmer to work with, since you can call them without worrying about any changes they might make to the data in your own code. (This is the same argument that is used against declaring global variables.)
- It allows values to be passed that aren't associated with any variables, such as function calls that return a value of the proper type.

These two benefits are now considered.

Avoiding Side-Effects
The most important implication of passing by value is simple:

> *Any C++ function passing an argument by value cannot change the value of that argument. Any changes made within the function are made to a copy of the argument that is local to the function.*

Although this rule has no exceptions, it does have a loophole so large that a good C++ programmer can—and frequently does—drive a truck through it. While a function cannot change the value of an argument, if you pass it an address, there's nothing to prevent you from changing value of the elements at that address—just so long as you don't change the address itself. In Chapter 3 we pointed out that when we pass an array into a function, what really gets passed is the address where the array starts—the one piece of information you really need. What this implies is simple:

> *When you pass an array into a C/C++ function, you can change the values of the array elements. What you can't change is the address in memory where the array actually starts.*

We will say more on this subject in Chapter 11, when we delve into pointers. For now, it is enough for us to realize scalar variables and arrays are handled quite differently by C++ functions.

Passing Expressions as Arguments
Where arguments are passed by value, it is possible to substitute function calls (or other expressions) for variables in the function arguments. For example, you could call the previously defined function Plus() as follows:

```
int a0;
a0=Plus(Plus(3,4),Plus(5,6));
```

Since Plus(3,4) and Plus(5,6) both return integers, those values can be determined, then used in the top-level call. This type of call, with functions being used as arguments, is not always possible in languages that pass by reference—they might require the code to be written as follows:

```
int a0;
int a1;
int a2;
a1=Plus(3,4);
a2=Plus(5,6);
a0=Plus(a1,a2);
```

Naturally, the second example, which will lead to precisely the same value in a0 as the first example, could also be written in C/C++—and the neat community might well justifiably argue that the second approach is clearer and easier to debug than the first. But it is certainly less concise than the first version.

Sadly—or happily, depending on your perspective—it is pretty common to see lines of code such as:

```
a0=Plus(Plus(3,4),Plus(5,6))
```

in actual C++ programs. What you need (to know to figure out what happens) is this:

Before you can enter the body of a function, all of its argument values must be known.

In our current example, then, our "outer" call to Plus() can't begin work until the two inner calls—Plus(3,4) and Plus(5,6)—have been completed.

Test your understanding 1.4: What would the return value for the call Plus(Plus(3,4),Plus(5,6)) be? Explain the call.

1.3 Section Questions

1. Explain why Plus(3,4) is called before the main (outer) Plus() function is called in the expression: Plus(7,Plus(3,4)).
2. What do we mean when we say that a C++ function cannot change the value of a normal (i.e., non-referenced) argument?
3. What is different about the ways in which scalars and arrays are passed into a function?
4. If a function that has a local variable named *counter* calls another function with also defines a local variable named *counter*, will the two variables interfere with each other?

1.4: C++ Function Extensions

C++ provides three important extensions to the standard C function: the ability to pass arguments by reference, the ability to overload function definitions (declare multiple functions with the same name) and the ability to declare default values for arguments. In this section, we summarize these three capabilities.

1.4.1: Passing Arguments by Reference

The ability to pass function arguments by reference is implemented in most programming languages, but is absent in C. C++ added this capability to the language. Because the array name (and pointer) loophole, mentioned at the end of Section 4.1.3, can be used to accomplish the same purpose as reference arguments, references are not critical for C++ structured programming. That situation changes when the language's OOP capabilities are used—certain types of functions (e.g., copy constructors) require references be used to define arguments. Even if you don't plan to use them, however, it is useful to understand them..

When an argument is passed into a function by reference, the parameter name given in the function declaration becomes an alias for the variable or other expression being passed in. Creating an alias is very different from creating a copy:

- When changes are made to a parameter inside a function that was passed by value, the original variable expression being passed in remains unchanged—since the changes affect only the copy that we made.
- When changes are made to a parameter inside a function that was passed by reference, the value of the variable or expression (e.g., array element) in the code that called the function changes as well.

To understand this distinction, lets contrast the function Plus() that we defined earlier to another function we might define, PlusEquals(). The idea of Plus(), as you may recall, was to return a value equal to its two arguments. For example, Plus(3,4) would return the integer value 7. Our function PlusEquals() on the other hand, is going to do something a bit more sophisticated—it will add the two arguments together, like Plus(), then place the result in the first argument. For example:

```
int X=3,Y=4,Z;

Z=Plus(X,Y); // returns 7, no change to X or Y

PlusEquals(X,Y); // returns 7, X becomes 7

// Equivalent to writing: X=X+Y;
```

The ampersand (&) operator is used in a declaration to identify the fact that a reference parameter is being created, rather than a normal parameter. We can therefore implement our function PlusEquals() as follows:

```
int PlusEquals(int &a1,int a2)
{
        a1=a1+a2;
        return a1;
}
```

The & in front of the declaration of a1 means it is being passed by reference. As a result, the line of code:

```
a1=a1+a2;
```

not only changes a1 inside the function, it also changes the value of what a1 was aliased to. In the block of code preceding it, for example, that was the variable X.

Test your understanding 1.5: In the function PlusEquals(), why was a1 defined as a reference argument, while a2 was not? Could we have defined a2 as a reference.

If a reference is passed, but we do not want its value to be changed inside the function, we can use the **const** qualifier in front of the argument declaration. By using constant references in this way, we can eliminate one of the major criticisms of passing parameters by reference. Indeed, the program safety and memory management benefits that result from using references in places where we would otherwise have used pointers have led some newer languages based on C++ (such as Java and, to a large extent, C#) to eliminate the use of pointers altogether.

1.4.2: Overloaded Functions

Many languages (such as C) require that every function have a unique name. While this policy appears to be a very sensible way of eliminating unnecessary confusion, it can actually create quite a drain on the programmer's mental resources as projects get large and complex.

To understand how this mental drain could occur, let us return to our workhorse Plus() function. As we defined it, the Plus() function takes two integers as arguments, and returns an integer. But, suppose:

- We also wanted a function to add two double values. Our problem here is that the name Plus() is already taken, so we need to choose a new name, and decide to call the function PlusReals(double a1,double a2).
- But now, we want another version to add two floats together. Fine, we define PlusFloats(float a1,float a2)—and rename our PlusReals(double a1,double a2) to PlusDoubles(double a1,double a2), annoyed at ourselves for not having realized the problem in the first place.
- Now it becomes really fun. We decide we want to be able to add a double to an int. So we define PlusDouble2Int(double a1,int a2). Then we realize we want to add an int to a double. So we also define PlusInt2Double(int a1,double a2).

- Then we realize we need to mix int and floats, floats and doubles, etc. so we keep defining away.

The worst of it is, coming up with the names is the easy part. Next, you start using your functions and, every other minute, you find yourself asking the question: what did I call that stupid function? Was it PlusRealToInteger, PlusDoubleToInt, PlusDouble2Int, PlusDouble2Integer? After a full day of this, your head feels as if it's going to explode. When you get home, you find yourself searching the Internet for full time sheep ranching job opportunities in Montana. Another sad casualty of too many darned names...

As it turns out, there is no "computational" reason that function names need to be unique. The purpose of function names is to give the programmer something to work with that is less cumbersome than memory addresses—as we pointed out in Chapter 1. It therefore stands to reason that as long as the compiler can tell our function calls apart, there is no reason they can't share the same name.

C++ provides the capability of different functions sharing the same name. As long as the functions have different arguments—different enough so that the compiler can figure out which version of the function you want when you make a call in your code—you can have as many functions with the same name as you want.

In the case of our Plus() function, then, we could define different versions of the function for all of our special needs. For example, we might have the following prototypes:

```
int Plus(int a1,int a2); // Our original version
double Plus(double a1,double a2);
float Plus(float a1,float a2);
double Plus(double a1,int a2);
double Plus(int a1,double a2);
float Plus(float a1,int a2);
float Plus(int a1,float a2);
double Plus(double a1,float a2);
double Plus(float a1,double a2);
```

While we would still have to define each function, it becomes much easier to work with them. Whenever you want to call your function, you just call Plus(). For example:

```
int ival1=1,ival2=4;
double dval1=3.5, dval2=9.0;
float fval1=6.0,fval2=4.1;
Plus(ival1,ival2);      // calls Plus(int a1,int a2)
Plus(dval1,dval2);      // calls Plus(double a1,double a2)
Plus(ival1,fval2);      // calls Plus(int a1,float a2)
```

This can dramatically reduce the cognitive load you experience while programming.

Test your understanding 1.6: What would each of the three calls to Plus() in the example above probably return? How do you think the definition of the original version Plus(int a1,int a2) would differ from Plus(int a1,double a2)?

It turns out that being able to assign the same name to multiple versions of a function is also critical in implementing the OOP capability called *polymorphism*.

1.4.3: Default Argument Values

In a lot of applications—particularly when you start programming in MS Windows—you find yourself writing functions with lots of arguments (e.g., to specify the frame style, window type, window text). Frequently, however, you find that many of the arguments seem to have the same value, time after time. It can get very tiring filling in these same values, over and over again.

C++ provides a simple way of getting around the problem: the ability to define default values for arguments. Default values are assigned by initializing argument values in the function prototype. (Such initialization needs to be done in the prototype, because any file that calls the function needs to know what default values are assigned.)

To illustrate this using a practical example, suppose we wanted to create a more sophisticated version of PrintRealNumber(), discussed in Chapter 3, Section 3.3.3, called PrintDouble(). The difference between the two functions would be as follows:

- A second argument to PrintDouble(), called nMinLen, would be an integer, specifying the minimum number of characters to be printed. This could be useful, for example, in lining up columns of text. We could also specify that if –1 was specified for nMinLength, we'd just print as many characters as were needed.
- A third argument, called nPrecision, would specify how many decimal digits of precision (digits to the right of the decimal point) should be printed. We could also specify that if –1 was specified for nPrecision, we'd just print as many digits as were needed.

If we were to define the function, e.g.,

```
void PrintDouble(double dVal,int nMinLen,int nPrecision);
```

we would always have to call it with three arguments. Thus, if we just wanted it to print (as in PrintRealNumber()), we'd have to call it as follows:

```
PrintDouble(dVal,-1,-1);
```

using the –1 values to identify we didn't care about its minimum length or how many characters to the right of the decimal were printed.

In C++, however, the ability to define default arguments would allow us to prototype the function as follows:

```
void PrintDouble(double dVal,int nMinLen=-1,int nPrecision=-1);
```

The presence of these arguments would mean the function could be called in three ways:

```
PrintDouble(dVal,10,3); // Both specified (e.g., 10 chars, 3 decimal places)
PrintDouble(dVal); // equivalent  to PrintDouble(dVal, -1,-1)
PrintDouble(dVal,10); // equivalent to PrintDouble(dVal,10,-1);
```

There are certain rules that need to be followed in defining default values:

- As soon as a default value for an argument is specified in the declaration, all arguments to the right of it in the declaration must also have default values.
- When a function with default values is called, values are defaulted from right to left—you are not allowed to skip a value. For example, a call such as PrintDouble(dVal,,3)—presumably an attempt to get three digits of precision without specifying length—is not allowed.
- No variation on any function with default arguments can conflict with an overloaded version of the function having the same name. The compiler does not respond well to such choices.

In the event that any of these three rules are violated, compiler errors will be encountered.

Test your understanding 1.7: If we had named our more sophisticated function DisplayReal(), and included it in a project that also had our SimpleIO files, which of the three rules would be violated, and why?

1.4 Section Questions

1. Explain the principle difference between passing function arguments by value and by reference.

2. What drawback of passing by reference can the **const** specifier address?

3. Is function overloading likely to be most useful in simple or complex programs?

4. If you were designing a function with a lot of default arguments, how would it be best to order the arguments in the function declaration?

5. What do you need to be careful about when defining overloaded functions that also have default arguments?

1.5: Steps in Creating a Program

Before we create our first project, we need to have a basic understanding of the process through which *source code*, the program code that we type in, is transformed into *executable code*, the code that is actually run by our processor. In this section, we provide an overview of the process, which is illustrated in Figure 1.1. We refer to these steps as compose, compile, link and load.

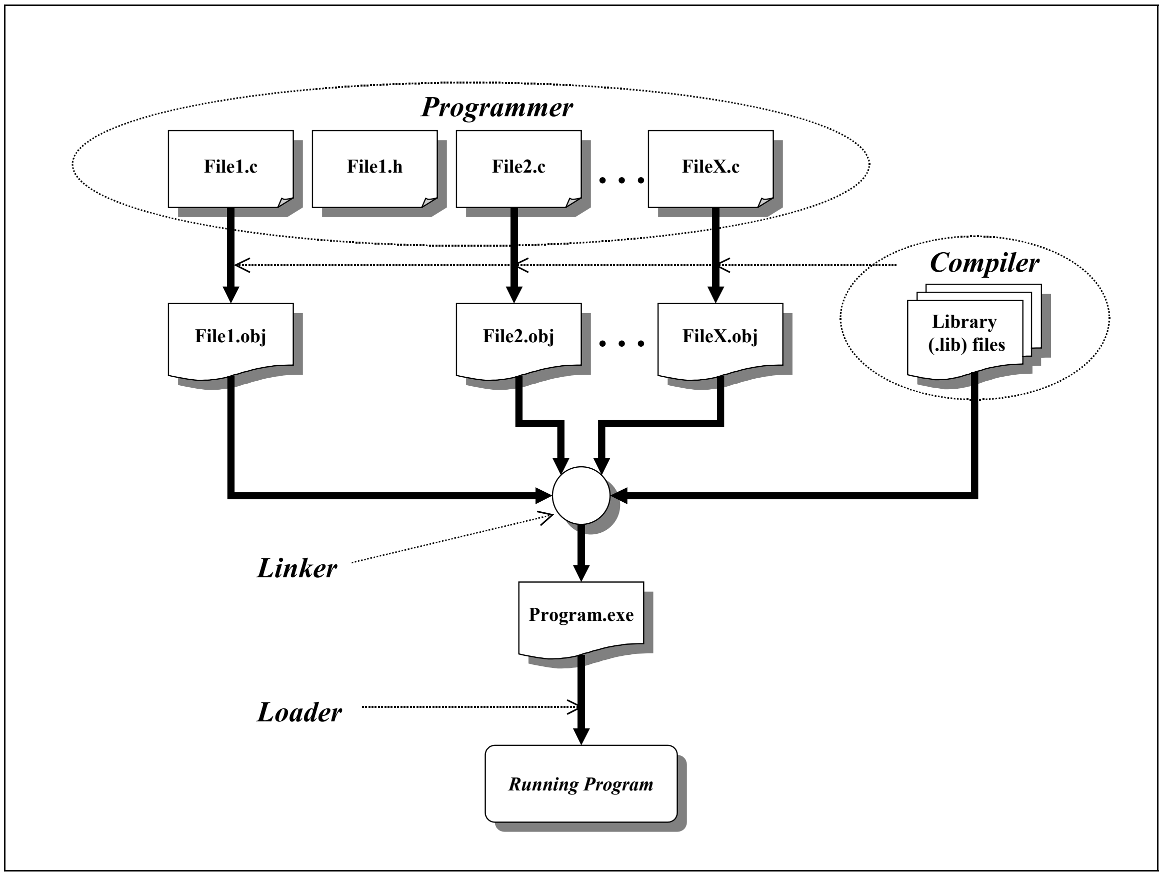


Figure 1.1: Overview of program creation

1.5.1: Compose

The first step in programming an application is, of course, writing the code for the application. As often as not, source code can be written in any text editor (such as MS Notepad). Integrated development environments, such as Visual Studio .NET, will normally provide their own editor. These included editors offer special features—such as keyword highlighting, auto completion and automatic indentation—that make writing code somewhat easier.

It is important to realize that, for all but the simplest program, more than a single file of source code will normally be required. The use of multiple files both helps the programmer organize the project and speeds the compiling process (as only files that have been recently edited need to be recompiled when recreating a program to fix defects or change the code). In Figure 1.1, the collection of source files created by the programmer are illustrated by the row of .c files at the top.

One side effect of using multiple files is that there may be some information that has to be shared across all the files. A common example of such information might be the names and types of individual variables that are used in all of the source files. Another example might be constants that we define, such as the maximum number of characters in a line of text. To ensure such consistency, most 3GLs provide a command that causes a file to be included—effectively pasting the included file into the source file when the compile process begins. In C/C++ such files are usually referred to as header files, illustrated by the .h file in Figure 1.1.

1.5.2: Compile

After a programmer has completed writing one or more source files, each file is compiled separately. The compiling process is intended to check the code for errors and—if no serious errors are found—generate a file containing object code, as illustrated by the .obj files in the second row of Figure 1.1. In addition to the .obj files created from the source code written by the programmer, a typical compiler comes with a collection of pre-written files, called library files, containing useful functions that the programmer can call from his or her programs. The .lib files in the second row of Figure 1.1 illustrate these pre-compiled function libraries.

1.5.3: Linking

The linking process involves taking all the object modules produced by the compiler, along with object modules stored in any library we choose to reference, and assembling them into a single executable program (normally a file with the .exe extension for the programs we'll be creating). In doing so, it assigns the relative memory locations of all variables and functions that are used in the program. (Actual memory locations cannot be assigned until the final stage: loading). This is illustrated by the circle on the third row of Figure 1.1.

1.5.4: Loading

The loading process, normally performed by the operating system, takes the executable file produced by the linker, loads it into memory, then updates the addresses in the program so they reflect actual memory locations (as opposed to the memory locations relative to the start of the program that were established by the linker). After a program is loaded, it can commence running. In MS Windows, for example, double clicking a program or document icon normally invokes the loader on the appropriate application, to start it running.

1.5 Section Questions

1. What are some advantages of having a project consisting of a lot of small source files, rather than a single large one?

2. If you were moving a project, such as that in Figure 1.1, from one computer to another, which files would you have to move, and which could you recreate?

3. What is type of file in Figure 1.1 is library file most like? What are some reasons for distributing code as library files, rather than as source code?

4. If a project only has one source file, do you still need to link the project before you run it? Explain why or why not.

5. When a program is loaded, will it always be loaded in the same location? Why or why not?

1.6: Introduction to Visual Studio .NET

Visual Studio .NET is the newest software development environment in a long series of environments designed by Microsoft. It is built to accommodate programming in multiple languages (Visual Basic, Visual C++ and Visual C#—pronounced C-sharp—being the most tightly integrated) and is particularly targeted towards the creation of applications that are distributed across the Internet.

Visual Studio is an example of an *integrated development environment* (IDE). What distinguishes an IDE from other development tools is the tight coupling between all the components and utilities associated with developing a program.

To understand the benefits of an IDE, it is useful to consider the alternative. In the mid-1980s, for example, if you wanted to program in C you needed to acquire:

- A text editor, such as IBM's Personal Editor, to create your source code
- A compiler, such as the Lattice C compiler (that later became Microsoft C). It was also often useful to have an assembler, such as Microsoft's Macro Assembler, to handle specialized code development, or mixed language programming.
- A linker, normally provided with the operating system
- A debugger, such as the CodeView debugger, that could be used to examine your program while it was running
- If you wanted to program in Windows (late-1980s), you also needed to acquire the Windows SDK (software development kit)—a library of functions needed to write Windows applications.

There were many disadvantages to using these tools. First, they were expensive, easily running upwards of $1000 if Windows programming was your goal (of course, in the late 1980s, few people used the versions of MS Windows that were then available). Second, almost no online help was available, so you typically needed five or six reference manuals on your desk while you programmed. Third, each component had to be run separately. You would start by running your text editor, editing your code, saving your work, then exiting your editor. Then you would run your compiler. If there were errors, you'd have to go back into your editor. If not, you could run the linker. If that worked, you could run the program, etc. Although they seem archaic by today's standards, it is worth being aware that such separate tools existed—and continue to exist. Throughout the .NET C++ technical documentation, for example, references are frequently made to command line compiler switches. For some types of applications and development situations, using the standalone (command line) compiler continues to be recommended.

The experience of programming in an *integrated* development environment is entirely different from working with separate tools. To begin with, all the different components are incorporated into the same tool. In Visual Studio .NET, for example, you have:

- Source code editors that are aware of the language in use and that supply appropriate (yet customizable) indentation and highlighting (e.g., for keywords and comments)
- Online help—tied into Microsoft's MSDN library of technical documentation—that is equivalent to thousands of manuals.
- Intelligent Make facilities that combine compiling and linking, using modification dates of the files to determine what files need to be updated each time a program is rebuilt (i.e., the executable is created).
- Debugging tools tightly coupled with the other development tools. Indeed, the .NET debugger is so sophisticated that it has the ability to make some code changes while a program is actually running.
- Incorporated libraries and code generation Wizards that can be used to facilitate programming in environments that previously required expert-level skills (such as component development).

These tools dramatically increase a programmer's potential productivity. Of course, they also come with a downside—if you insisted on understanding every capability Visual Studio .NET offered, it would probably take you upwards of a year to start using it (and then, only if you were an expert programmer to begin with). So, the best way to proceed is usually to take on a few features at a time.

1.6.1: Projects and Solutions

The primary organizing elements of Visual Studio .NET are the *project* and the *solution*. These elements can be characterized roughly as follows:

- *Project:* A collection of files that, together, can be used to create and debug a single executable *target*. For our purposes, the *target* is always an executable program—although this does not always have to be the case. The files can include a variety of different types, including source code (e.g., .cpp and .h files), object files (.obj) created when source files are compiled, executable files (e.g., .exe, .dll files) produced by the project and a variety of project definition files whose purpose is to keep track of the other files in the project, and other information such as the location of break points.
- *Solution:* A collection of one or more related projects. A solution also contains environmental information, such as the position of each open window. That information is used to ensure that the programmer's *workspace* (the old name for solution) appears just as he or she left it each time it is retrieved.

The incorporation of multiple projects into a single solution is most useful when creating an application out of separate independent components. That programming approach is an advanced topic, far beyond the scope of this textbook. So, for our purposes, projects and solutions will always have a one-to-one correspondence (one project per solution), and we can think of the two as being effectively equivalent.

Real projects can include quite a few files. Even simple projects contain more files than you might expect, when IDE generated files are counted. For this reason, the standard behavior of Visual Studio .NET is to create a new folder for each project. Additional folders will then be created, if necessary, based on the project's *configuration*.

Project *configuration* allows the user to maintain different variations of a target program within the same project. The two most common configurations are Debug and Release. The Debug version of a target contains information that allows the debugger to synchronize the lines of source code with the machine code in the executable file. The Release version has no such information (and the .exe file it produces is therefore almost ten times smaller, on average). Because Debug and Release mode targets must be compiled differently, a separate folder is used to hold the .obj and .exe files for each separate configuration.

For the purposes of this textbook, we will always be using the "Debug" configuration—since there is no reason for us to disable the tool most likely to help us get our programs working. Since "Debug" is the default, that means we will not normally need to adjust the configuration. It is wise to be aware that configuration settings exist, however, for three practical reasons:

- Should you accidentally set your configuration to "Release", you won't be able to use the debugger until you return the setting to "Debug" and rebuild the project.
- If you need to copy your executable program—for example to hand in as part of an assignment or to show friends and family—creating a release mode version may be the only way you can get it small enough to fit on a floppy.
- If you need to find the executable version of your program, you should be aware: a) that it will exist in the subfolder named after the configuration (e.g., the Debug

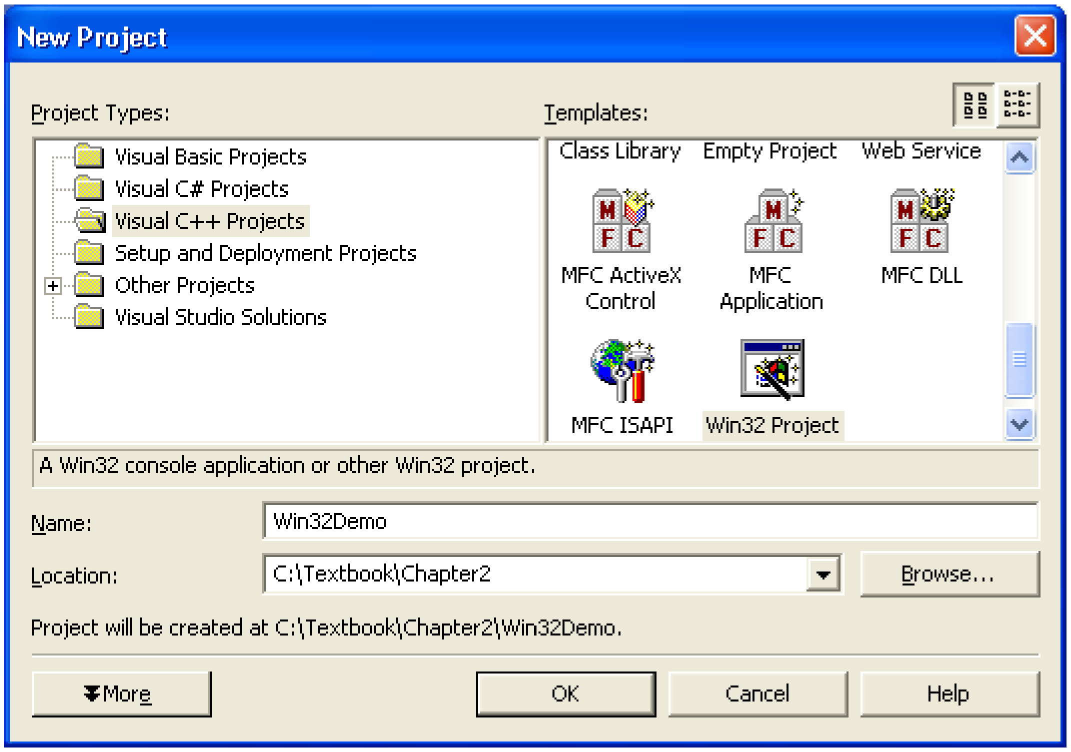
folder), and b) that where multiple configurations exist (e.g., Debug and Release), the version of the program in each subfolder will only be as recent as the last time you built the project while in that particular configuration.

Before leaving the subject of moving your programs around, it is also important to realize that projects can become quite large (even projects that do practically nothing can range from 3 to 10 megabytes in size). In the event that you need to move projects between machines, the *Build|Clean* option in Visual Studio deletes every file that can be recreated by the IDE (i.e., object files, executable files, etc.)—reducing the size of most small projects to a few hundred kilobytes. This is an excellent tool for moving your work between home and work/lab computers, and for sharing code during group projects.

1.6.2: Creating a Console Project

Throughout this text, we will be creating what are called console projects—applications that run in ugly black windows with I/O no more sophisticated than a teletype could manage. Such projects represent the "lowest common denominator" of C/C++ programming, and limiting ourselves in this way allows us to adhere pretty closely to ANSI standards—meaning the code we will write can be reused on almost any hardware/operating system combination that supports a decent (i.e., ANSI compliant) C/C++ compiler. By the time we reach the end of the book (particularly chapter 19, where we demonstrate CGI), we also will discover that this plain vanilla approach to programming can still be used to create some pretty impressive I/O.

Project Type
Creating a new project begins by selecting a project type. This involves two steps. First, you must select the basic project category (Visual C++ Project) from the left hand side of the New Project dialog. Then, from the list of templates that is presented on the right hand side, you select the **Win32 Project** option, as illustrated in Figure 1.2. At that time, you should also give the project a name—which will be used to create a project folder, and also as the default name of the target (which will be a .exe file, throughout this text).



A Win32 console application or other Win32 project.

Name: Win32Demo

Location: C:\Textbook\Chapter2

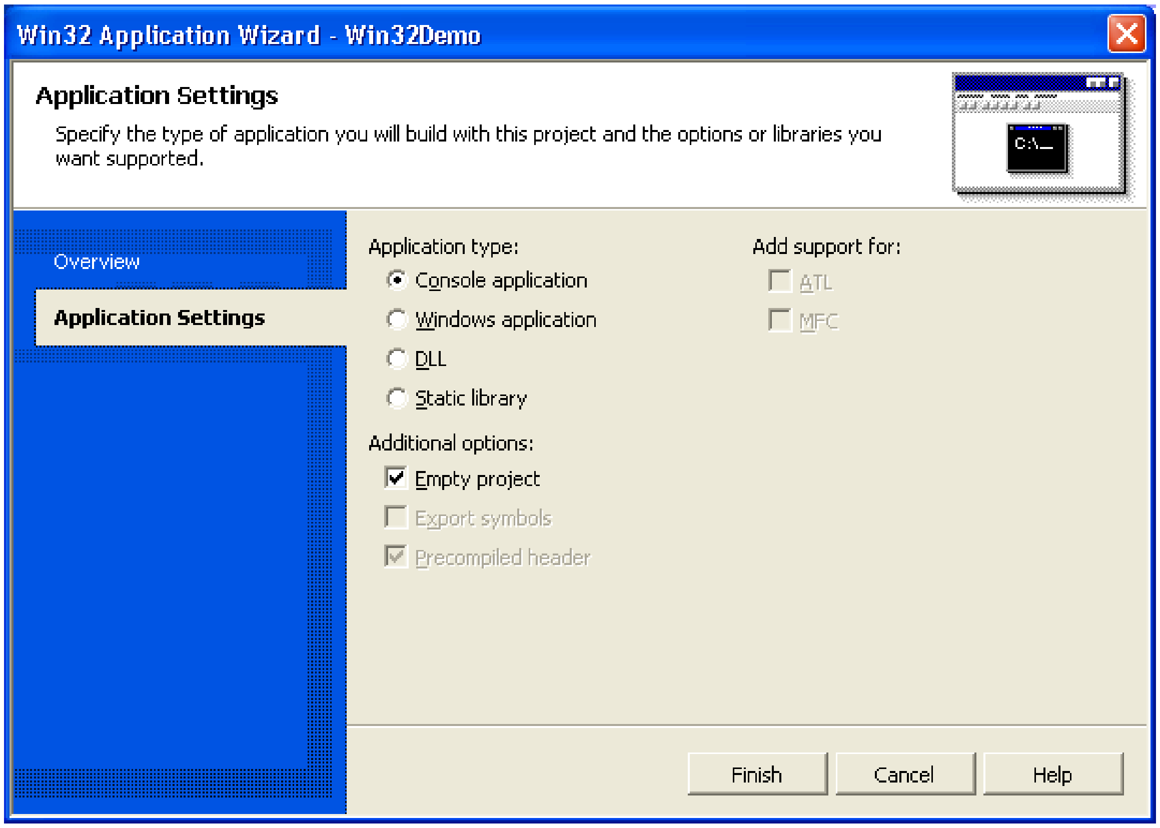
Project will be created at C:\Textbook\Chapter2\Win32Demo.

Figure 1.2: Creating a Win32 Console Project

Upon pressing "OK", you will then be presented with a dialog (the *Win32 Application Wizard*). After choosing "Application Settings" from the left hand panel in the dialog, you'll be given a choice of settings for your project. As illustrated in Figure 1.3, you should set these up as follows:

- *Application Type* should be set to "Console Application"
- "Empty Project" box should be checked.

This will create a project into which you can place your C and C++ files.



Specify the type of application you will build with this project and the options or libraries you want supported.

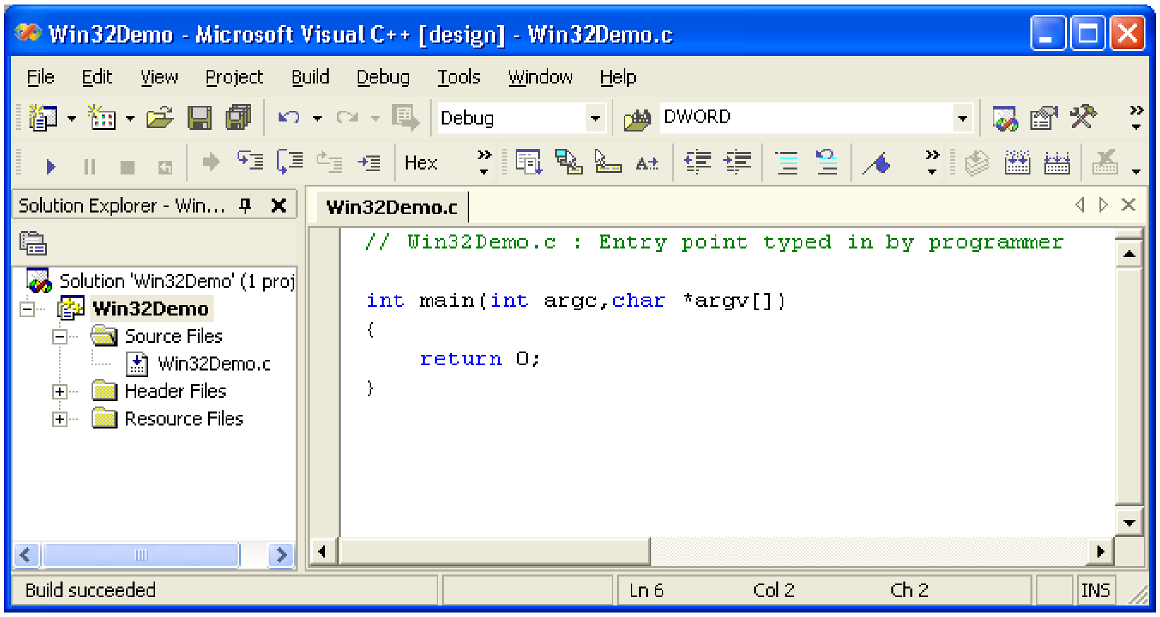
Application type:
- [✓ Console application]
- Windows application
- DLL
- Static library

Add support for:
- ☐ ATL
- ☐ MFC

Additional options:
- ☑ Empty project
- ☐ Export symbols
- ☑ Precompiled header

Figure 1.3: Application Settings Dialog

Once your project has been created, you can right-click the "Source Files" folder below the project icon (highlighted in Figure 1.4) in the Solution Explorer pane, and choose *Add|Add New Item*. You can then choose from the list of file types to create a new source file.



```
// Win32Demo.c : Entry point typed in by programmer

int main(int argc,char *argv[])
{
    return 0;
}
```

Figure 1.4: Adding simple main function

Once your empty file has been added to the project, you can type in your code and comments, as illustrated in Figure 1.4.

1.6 Section Questions

1. What's the difference between a C file, a project and a solution? Why won't the last of these differences be very important in this book?

2. What is a target? What type of target will we be focusing on?

3. What types of tools are incorporated into the Visual Studio .NET IDE?

4. What is the difference between "Debug" mode and "Release" mode?

5. What types of files are placed in the "Debug" and "Release" folders? Could you have the same files in both folders?

6. Why will we need to adjust "Application Settings" every time we create a new project?

1.7: A Simple Project

In this section, we create a project that display's "Hello, World" to the screen in C++. This particular project has a long tradition of being the first C/C++ program taught in almost every course.

1.7.1: Hello World in C++

Walkthrough available in HelloCpp.avi

Creating a C++ console "Hello World" project in Visual Studio is relatively simple. It involves:

1. Generating a project with a skeletal main function
2. Editing the code so it prints "Hello, World".
3. Building the executable file
4. Running the program

We now look at each of these steps.

Generating a project with a skeletal main function
Follow the procedure in Section 1.6.2 to create an empty console project. Add an empty C++ source file, type in the contents as shown in Figure 1.4, and save it as "Hello.cpp".

Editing the code so it prints "Hello, World"
Modify the code so it matches that in Example 1.3. The lines being added are the following:

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

above the line *int main(int argc, char* argv[])*, then

```
cout << "Hello, World " << endl;
```

within the { and }, above the line *return 0;*

Example 1.3: Modified C++ Code

```
// Hello.cpp : Defines the entry point for the console application.

#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
      cout << "Hello, World " << endl;
      return 0;
}
```

Building the Executable File
Because Visual C++ has a sophisticated compiler/linker combination, we can perform our compile and link in a single step. This can be done either using the toolbar, or by selecting *Build*|*Build Solution* from the menu. If you entered your changes properly, in an output box beneath the window you should see the following messages along the lines of those in Example 1.4 (with your paths being different, presumably).

Example 1.4: Build Messages

```
Compiling...
Hello.cpp
Build log was saved at "file://c:
c:\YourFolder\Hello\Hello.cpp(6) : warning C4100: 'argv' : unreferenced formal parameter
c:\YourFolder\Hello\Hello.cpp(6) : warning C4100: 'argc' : unreferenced formal parameter
Linking...
\YourFolder\Hello\Debug\BuildLog.htm"
Hello - 0 error(s), 2 warning(s)


---------------------- Done ----------------------

   Build: 1 succeeded, 0 failed, 0 skipped
```

Messages such as these are what might be termed a mixed signal. On the one hand, it completed compiling and linking your project. On the other hand it warned you about something—and you should *never* ignore a warning.

The particular message in question results from our main() function, the entry point for every C++ console program, which has two arguments: *argc* and *argv*. The compiler noticed, however, that we weren't using those arguments within our main function. It then wondered: why would you define a function with arguments, then not use them? The warnings are just the compiler's way of trying to help us. In doing so, it has also illustrated an important, and nearly universal, law of compiler design: *never issue a message using language that a beginner can understand.* For further information, you can highlight the error code (C4100) then press F1 to bring up a help screen. Doing so will illustrate another important design principle: *when you're just getting started, don't expect the help screens to be of much use.*

> *Hint*: if you want more information on an error, try typing the code into a search engine such as G*oogle* or *Yahoo*. Frequently, more detailed explanations will be found.

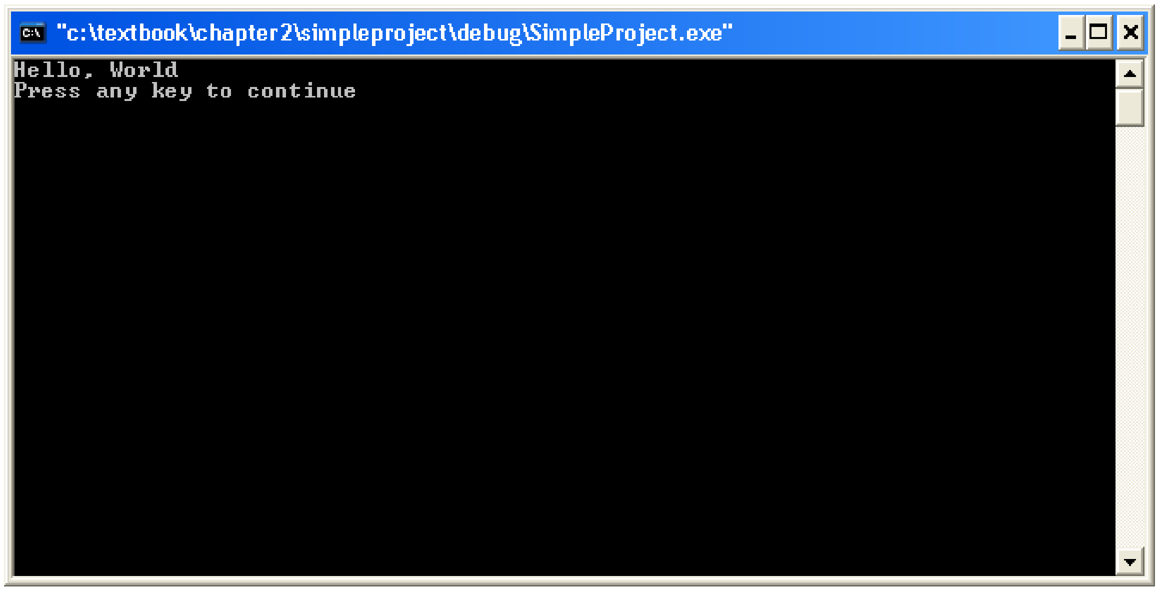
Fortunately, the above message is relatively easy to get rid of. The main() function, it turns out, can also be defined with no arguments. So to make the warnings go away, just change the line that reads *int main(int argc, char* argv[])* to:

```
int main()
```

build the project again and verify no errors and no warnings.

Running the Program

The last step in creating our first program is to run it. There are two ways to run a program, using the debugger and not using the debugger. Common sense would suggest that we need all the help we can get, so the first thing we will do is run it under the debugger, which can be done using *Debug|Start*. If you are quick, your eye might be able to catch the black screen appearing and disappearing, as the program ends. It turns out, if you want to run a console project under the debugger, you need to know something about breakpoints, which we will cover shortly.


"c:\textbook\chapter2\simpleproject\debug\SimpleProject.exe"

```
Hello, World
Press any key to continue
```

Figure 1.5: Console window when program is run without debugging

The other alternative to running the program—an alternative you should almost never choose voluntarily when you're writing your actual programs—is to run without debugger support. This is done using *Debug|Start without debugging*. As it happens, when a console project is run this way, the console window—which we might think of as "the black screen of despair" (so as to distinguish it from the ever-popular "blue screen of death" that appears when we've been *really* bad)—remains visible and the program doesn't end until we press a key, as illustrated in Figure 1.5.

1.7.2: Anatomy of Hello, World

Although Hello, World is a very small program, it is nevertheless a complete program. For this reason, by examining it closely, we can learn a lot about how a C++ program is created.

We now go through the code originally presented in Example 1.3, line-by-line.

Explanation 1: Comments
The first line we encounter in the file is:

```
// Hello.cpp : Defines the entry point for the console application.
```

This line is a comment, meaning it is there for the purposes of informing the programmer, but has no effect on the actual program. There are two ways to define comments, within /* and */ (traditional C comments), such as:

```
/* anything between the comment delimiters,
        no matter how it is spaced, or how many lines it takes
        is a comment */
```

The other approach (which is technically a C++ standard) is the // (slash-slash comment)—as shown above—which comments out the rest of the line.

Explanation 2: Preprocessor Instruction
What should first catch your attention about the next line is that it begins with # (pound) character:

```
#include <iostream>
```

At the beginning of this chapter, we illustrated the concept of file inclusion using an analogy between a list of characters and chapters in a book. The file *iostream* contains similar information—most importantly, it tells the compiler what several dozen input and output functions in the standard libraries are supposed to look like. We need that information in order to run this program because C++ itself doesn't know what the **cout** we refer to in the program is. Without including the iostream file, the compiler would look at the line with the **cout** in it and say: "Hmm, I've never heard of it…"

> *Test your understanding 1.8:* Try removing the #include statement (or commenting the line out) and building the project again. What does the error message tell you?

The # sign in front of the #include directive indicates that we are looking at a preprocessor directive. We will talk more about the preprocessor in a later chapter, but, for the time being, we can think of it as a sophisticated search and replace tool that works on your code before it is given to the compiler. The #include directive tells it to:

- find the file *iostream* by looking in the standard include directories, which are specified deep within the IDE's project settings.
- paste the entire file into the code you wrote, creating a temporary file that is then sent to the compiler.

Other preprocessor directives (most notably, #define) will be explained in later chapters, as they are needed.

Explanation 3: Namespace directive
Right under the preprocessor instruction is a namespace directive:

```
using namespace std;
```

To understand the purpose of this line of code, a little background is necessary. As C++ evolved, problems in keeping names unique began to emerge—especially since C++ allows several versions of a function to be defined using the same name. To help tell different functions and objects with the same name apart, the concept of a namespace was introduced. Using namespaces, it became possible to ensure high levels of uniqueness. For example, the iostream library defines **cout** to be in the standard namespace (**std**). That means its "real" name is **std::cout**, not just **cout**. This prevents any confusion with other **cout** objects that might happen to be hanging around.

Unfortunately, namespaces can be quite a pest. First, they make everything that you need to write longer, since you have to prefix the namespace name to objects. Second, old code—compiled before namespaces were incorporated into most compilers (which happened in the mid-1990s)—often won't work with the new libraries using namespaces.

To reduce these problems, the **using namespace name;** statement can be employed. When present, the compiler takes every function and object that is not otherwise identified and tries prefixing the namespace *name* to it. Thus, since we have the statement **using namespace std;** in our file, whenever the unknown object (**cout**) is encountered, the compiler checks for **std::cout**, which it finds in the iostream library.

While this seems like the type of problem that falls under the heading of "C++ trivia", forgetting to put the **using namespace std** statement into a program is a very common error—one that can easily lead to hundreds of compiler errors, even in a short program. Thus, it is one of the pitfalls that even novice C++ programmers need to be aware of.

Explanation 4: Function definition
The next code comes in the form of a block:

```
int main(int argc, char* argv[])
{
        cout << "Hello, World " << endl;
        return 0;
}
```

The block of code at the end of the file is the definition of the function main(). The main() function is special in C++ because it defines the entry point for our program—every standalone C++ program always starts running in main(). Other than that, however, it is a fairly typical C++ function.

Any function definition comes in two parts, the declaration line and the body. We'll now look at these separately.

Explanation 5: Function declaration line
The declaration line:

```
int main(int argc,char *argv[])
```

is extremely important. The declaration involves three parts, as illustrated in Figure 1.6:

- *Function name:* Declares the name to be used in calling the function.
- *Return type:* After a function completes its activities, it can send a value back to the block of code that called it. In the case of main(), it returns an integer.
- *Arguments:* The values within the parentheses tell us about the arguments being passed into the function. In the case of main, we see it has two arguments. The first argument (argc) is an integer. The second argument (argv) is too complicated to figure out by simple inspection, but will become clear after we have examined pointers and complex arrays in Chapter 3. (FYI, it's an array of character addresses used to hold the arguments typed by the user on the command line)

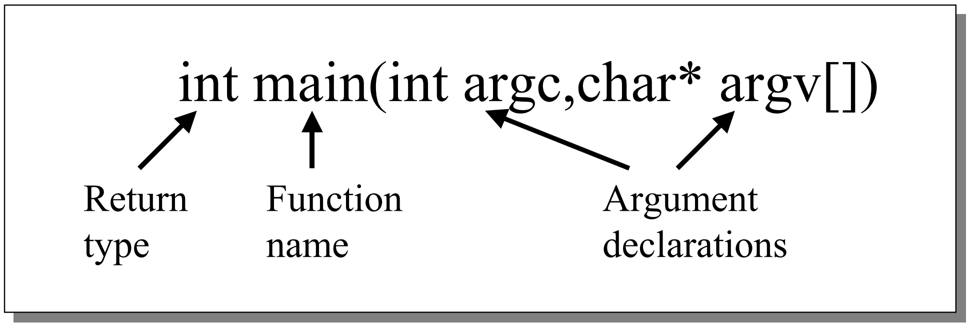


Figure 1.6: Elements of main declaration line

Explanation 6: Definition body
The final piece of our simple C program is the body of main:

```
{
        cout << "Hello, World " << endl;
        return 0;
}
```

The code within the braces identifies what will happen when the function main is called. In this case, the action is very simple:

- First, we send the string "Hello, World" to the standard output object **cout**. The << is sometimes referred to as the output or insertion operator and sending data to **cout** causes that data to be displayed to standard output—usually the console screen. The **endl** is what is called a *manipulator*. When it gets sent to **cout**, a newline character is sent to the screen, causing the cursor to move to the start of the next line. (You can do the same thing by sending a specific character, '\n' to **cout** as well).
- Second, we return the value 0 (the return value of main is sometimes used by the operating system). What is important to note here is that the value being returned (0, which is an integer) matches the type of value we specified in the declaration **int main()**. If it did not, we'd get an error.

In addition to strings, the << operator can also be used to send other data types, such as individual characters, integers and real numbers, to the console. Conversely, to acquire input from the user, you could use:

```
cin >> variable;
```

The >> (extraction) operator is defined to read all the primitive data types fro m the standard input object (**cin**), and can also be used to load strings into arrays—stopping when the first space in the input is encountered.

1.7 Section Questions

1. What is the one advantage of "executing" a console project with the exclamation point button, rather than running it in debug mode?

2. Do the library "include" files that we reference in our programs contain function definitions?

3. What does the fact that the main() function is defined with arguments and a return value imply?

4. What is special about the function main()? Why must every standalone console project have one (and only one) main function?

5. What is cout?

6. What is cin?

1.8: Multifile Projects

Walkthrough available in Multifile.avi

Just as it is hard to imagine a novel, of any length, without chapters, it is hard to imagine a program, of any substance, created using a single source file. Fortunately, the IDE makes multifile projects very straightforward to create.

In this section, we will create a new project—called Multifile—that spans across three files:

1. The first file will contain main().
2. The second a file that is supplied by the text.
3. The third file you will write.

Once complete, we will walk through some of the (very) simple functions we have developed using the debugger, in Section 1.9.

1.8.1: Adding existing files to a project

After you have created a project using the procedure outlined in Section 1.7, out first goal is to add a couple of files to the project. These files, SimpleIO.h and SimpleIO.cpp serve three purposes:

1. They demonstrate the process of bringing existing code into a project
2. They allow us to hide the complexities of I/O for a few chapters
3. They allow us to keep C++ programs entirely structured—allowing us to ignore the fact that some of the things we are working with, such as **cout** are actually objects) until we are ready to focus on how to work with objects.

The actual contents of the files will be discussed in Chapter 4. For now, we will just examine the types of constructs in each file.

Contents of SimpleIO Files
The contents of the SimpleIO.h file are illustrated in Example 1.5.

Example 1.5: Part of SimpleIO.h

```
// SimpleIO.h
#include <stdarg.h>

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

#define MAXLINE 255

// Text Console I/O functions
void InputString(char str[]);
char InputCharacter();
int InputInteger();
double InputReal();
void DisplayString(const char str[]);
void DisplayCharacter(char cVal);
void DisplayInteger(int nVal);
void DisplayReal(double dVal);
```

After our discussion of the Hello project, many of the components of the file should be familiar to us. The line at the top of the file:

// SimpleIO.h

is a comment line, telling us the name of the file. This line is followed by a series of #include statements, identifying libraries we'll be using.

The lines at the bottom of the file are function prototypes. We can tell they are prototypes, as opposed to variable declarations or function definitions, by two characteristics:

1. The presence of parentheses in a declaration indicates a function is being declared
2. The fact that a semicolon follows the declaration—as opposed to a function body within { and }—tells us we are just prototyping the function, not defining it.

The names of the functions should be fairly self-explanatory. The five functions we'll be using in this chapter are:

- *InputInteger()*: waits for the user to type a line, then returns its integer value (or 0, if the user doesn't type an integer).
- *DisplayInteger(int nVal)*: is passed an integer (nVal) as an argument and prints its value to the console screen.
- *DisplayString(const char str[])*: prints a string to the console screen. In this chapter, we'll create strings by placing text within double quotes. Later on, we'll find that there are many other ways of doing so.

- *NewLine()*: Sends a line feed to the console (like the **endl** we already talked about in the "Hello" project).

One line in the SimpleIO.h file represents an instruction not seen in the "Hello" project:

```
#define MAXLINE 255
```

This is an example of a preprocessor instruction (remember, the # symbol) called a macro. #define tells the preprocessor to do a search-and-replace within your source file. In this case, everywhere the symbol MAXLINE is encountered, it replaces it with 255. This is a very convenient way of specifying values that are not going to change during the program. (In SimpleIO.h, MAXLINE is used to represent the maximum number of characters we will accept in a line of text typed in from the user.)

> *Test your understanding 1.9:* What are the advantages of defining a name, such as MAXLINE, and using it in your program, as opposed to just using the number 255?

The SimpleIO.cpp actually defines the functions that we prototype in SimpleIO.h. For example, the definition of DisplayInteger() in the .cpp file is as follows:

```
// Prints an integer
void DisplayInteger(int nVal)
{
    cout << nVal;
}
```

This uses the output operator to send the integer argument value (nVal) to cout.

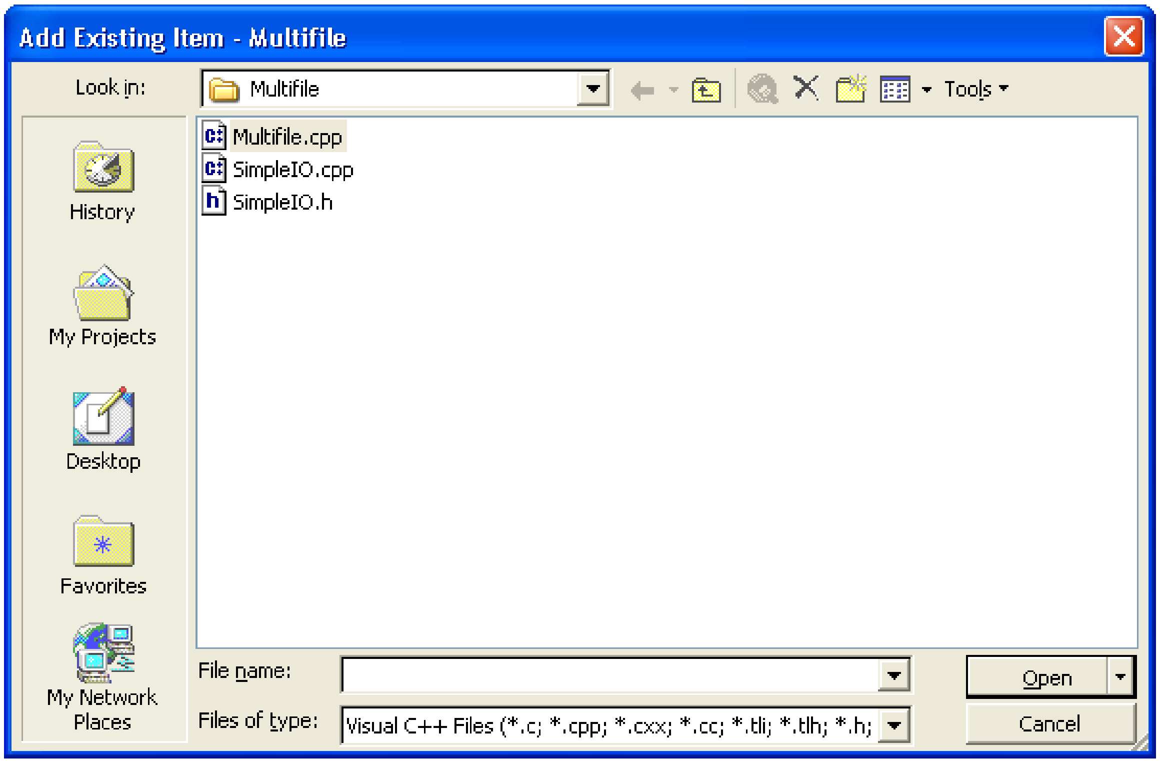
Creating and Adding Existing Files to the Project
The first step in our multifile project walkthrough is to create a C++ project, called Multifile, using the procedures outlined in 1.7.1.

Once created, we need to add our SimpleIO files to the project. The simplest way to do this, and the procedure we will follow throughout this text (to avoid wasting time on folder path problems) will be to follow a two-step procedure:

- First, open Windows Explorer and copy the SimpleIO.h and SimpleIO.cpp files from the CD to the project folder.
- Second, right click the Multifile project name in the Solution Explorer, the select *Add|Add Existing Item* from the menu. You should see a dialog like that in Figure 1.7. Select the two SimpleIO source files and press the "Open" button. These files are now included in the project.

Common Errors: If the files you just copied do not appear in the dialog, the odds are high you forgot step one, or did it improperly. It is important that these files from the CD be physically present in your project folder. If they are not, you might be tempted to browse the "Add Existing Item" dialog until you find them. Try to resist the temptation. If you succumb, will almost certainly get "fatal" errors about missing .h files when you compile. The problem is that while adding files outside the project folder tells the compiler where to find the .cpp files, it doesn't tell it where to look for the associated .h files. Although this is an easy problem to solve (you can add folders as "default" include directories to your project), the simplest approach—for now—is just to put every file we'll be using in the project folder.

Also, just placing files in the project folder does not make them part of the project. You need to add them to the project, as well.



Multifile.cpp
SimpleIO.cpp
SimpleIO.h

Visual C++ Files (*.c; *.cpp; *.cxx; *.cc; *.tli; *.tlh; *.h;

Figure 1.7: Add Existing File Dialog (in C++ example)

The final step of the process is to modify the main function. Change the main function so that it is the same as Example 1.6.

Example 1.6: New Main Function

```
#include "SimpleIO.h"

int main()
{
	int nVal;
	DisplayString("Enter an integer: ");
	nVal=InputInteger();
	DisplayString("You entered the integer ");
	DisplayInteger(nVal);
	NewLine();
	DisplayString("Congratulations. You did it!");
	NewLine();
	return 0;
}
```

Run the program (without the debugger) and see what it does.

Discussion
This initial stage of the multifile project demonstration illustrates a few new concepts. The first can be found in the line:

#include "SimpleIO.h"

This line was different from our earlier *#include <iostream>* because it uses quotation marks instead of < and > around the file name. The difference is significant:

- What an include file name in quotes tells the preprocessor is that the first place it should look for the included file is in the project folder that we created. If it does not find the file there, it then searches the default include folders (the locations of which are set in the compiler).
- Where an include file is between < and >, it tells the preprocessor it should *only* look for the include file in default include folders—it should not check the project folder.

In general, then, you should normally use < and > for standard included libraries, and quotation marks (often called double quotes, to distinguish them from apostrophes) for all other include files—such as the one you will create in the next section.

The code within the main function itself illustrates another new capability of C—how to use newly defined functions. The functions GetInteger(), PrintInteger(), etc. that are used in main are not standard C or C++ functions. Instead, they are functions we incorporated into our project by adding SimpleIO.cpp. Our main function knows how they are supposed to be called because the SimpleIO.h prototypes tell it the number of arguments and types of return values. But the actual definitions of the functions are in the .cpp files.

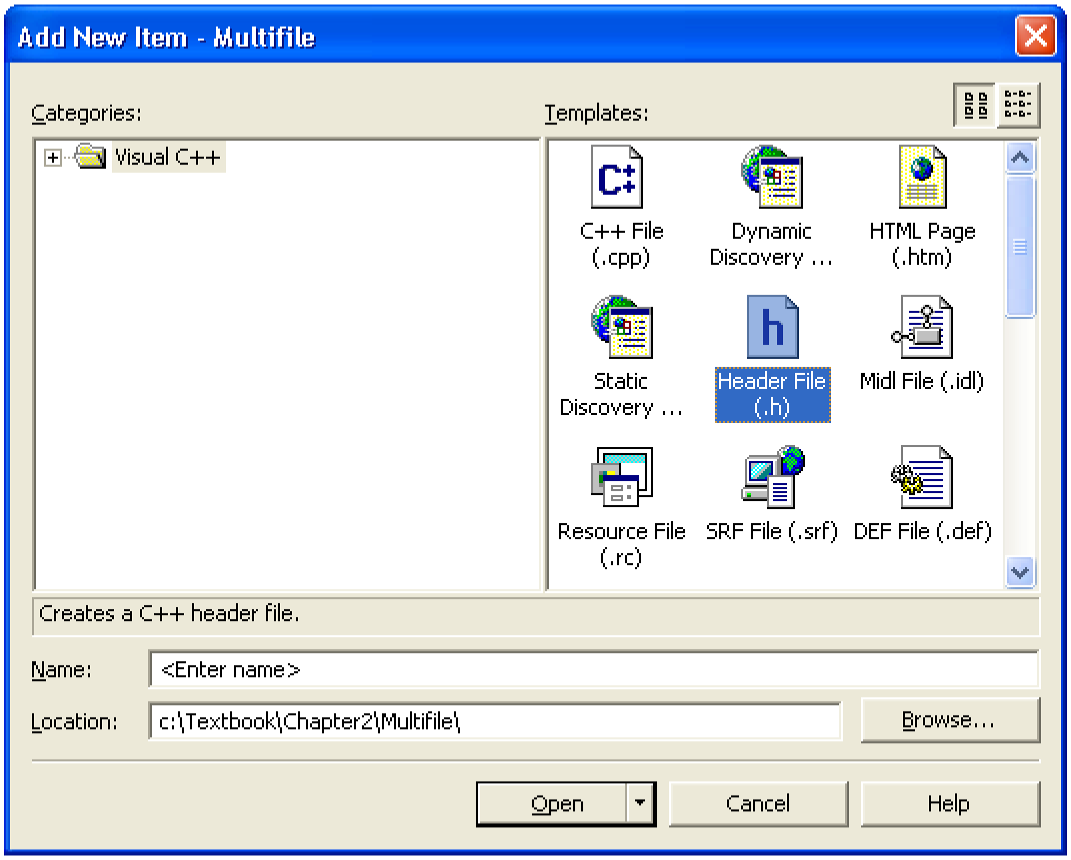
1.8.2: Adding new files to a project

Having learned how to add previously written functions, the next stage of the walkthrough is to learn how to create our own new functions. We will do this in three steps:

- Creating a header file
- Creating a source file
- Calling one of our functions from main

Creating a Header File

As soon as we start defining our own functions, we need to start worrying about creating our own header files. Fortunately, the process of header file creation is straightforward. Right click the Multifile item in Solution Explorer, then select *Add|Add New Item*. In the dialog that appears (see Figure 1.8), select the Header File (.h) icon, type in the name FirstFunc, then press the "Open" button. This will open the empty file FirstFunc.h.



Creates a C++ header file.

Name: <Enter name>

Location: c:\Textbook\Chapter2\Multifile\

Figure 1.8: Add New Item Dialog

Inside FirstFunc.h, type the code listed in Example 1.7.

Example 1.7: FirstFunc.h contents

```
// FirstFunc.h: prototypes for our first C/C++ functions

void TestFunc();
int Plus(int nArg1,int nArg2);
```

Looking at this file, you should be able to tell several things:

- We are going to write two functions, TestFunc() and Plus()
- TestFunc() takes no arguments. The **void** return type tells us it does not return a value.
- Plus() takes two arguments, both integers. It also returns an integer value.

Example 1.8: FirstFunc.cpp

```
// FirstFunc.cpp: Our first function
#include "SimpleIO.h"
#include "FirstFunc.h"

/* TestFunc(): Gets user inputs for Plus() function call,
   then displays results */
void TestFunc()
{
      int n1,n2,n3;
      DisplayString("Enter your first integer: ");
      n1=InputInteger();
      DisplayString("Enter your second integer: ");
      n2=InputInteger();
      n3=Plus(n1,n2);
      DisplayInteger(n1);
      DisplayString(" + ");
      DisplayInteger(n2);
      DisplayString(" = ");
      DisplayInteger(n3);
      NewLine();
      DisplayString("Pretty cool, eh?");
      NewLine();
      return;
}

/* Plus(): Takes its two integer arguments, adds them together
   and returns the value */
int Plus(int nArg1,int nArg2)
{
      int nTotal;
      nTotal=nArg1+nArg2;
      return nTotal;
}
```

Creating a Source File
Follow the process for creating a source file specified in Section 1.7.1. Once you've created your source file, type the code from Example 1.8 into it.

This source code is, once again, relatively simple. Nonetheless, it illustrates some important new concepts. Initially, we see two include directives:

```
#include "SimpleIO.h"
#include "FirstFunc.h"
```

We need both of these because we are calling functions prototyped in both SimpleIO.h (ReadInteger(), for example) and from FirstFunc.h (Plus(), for example). On the other hand, we aren't using any standard library functions in this particular file, so we don't need *#include <stdio.h>* or *#include <iostream>*, as we did in our other files.

There are two lines that are of particular interest in our TestFunc() function. The first is:

```
int n1,n2,n3;
```

The purpose of this line is to define three integers. The definition does two things: 1) it sets aside enough memory to hold three integers and 2) it *declares* that the names n1, n2 and n3 are to be used to identify each of the three locations, respectively. Since we are making this definition within a function, these variables will be local to the function—created when the function is called, eliminated when the function returns (see the discussion of the stack, in Chapter 1, Section 1.2, for an overview of this process).

> *Terminology:* The term *declaration* is used to refer to the process of specifying the data type associated with a given name. The term *definition* is used to refer to a statement that actually creates a variable, or other data element, of a given type in memory. As a result, a definition always incorporates a declaration—since creating a variable requires that we give it name. A declaration, on the other hand, may be made without any variables actually being defined—for example, a function declaration (e.g., Example 1.7).

Our second line of interest is:

```
n3=Plus(n1,n2);
```

The purpose of this line is to call the function—also defined in the FirstFunc.cpp file—called Plus. Being passed into Plus() are the two values in n1 and n2 (which, if you look at the code preceding the call, have values that were input by the user). As you may recall from the function prototype, the Plus() function returns an integer value. It therefore makes sense that this return value will be placed in n3, since we wrote n3=Plus(n1,n2).

The final line of interest is at the end of the function:

```
return;
```

In our earlier main() functions, we saw *return 0;* at the end of the function. The reason no value comes after the return in this case stems from the declaration of the function:

```
void TestFunc()
```

Since, as we said earlier, void here means the function does not return a value, it follows that there should not be any value after the return.

The entire Plus() function is very simple, and matches the previous definition provided in Example 1.1

Modifying main Function

The final step in this demonstration is to modify main so that it calls the TestFunc() function. Your new main() should look like that in Example 1.9.

Example 1.9: main Function Modified to Call TestFunc()

```
#include "SimpleIO.h"
#include "FirstFunc.h"

int main()
{
	int nVal;
	DisplayString("Enter an integer: ");
	nVal=InputInteger();
	DisplayString("You entered the integer ");
	DisplayInteger(nVal);
	NewLine();
	DisplayString("Congratulations. You did it!");
	NewLine();
	TestFunc();
	return 0;
}
```

The modifications are very simple:

- Add *#include "FirstFunc.h"* to the top, since main needs to know what TestFunc() looks like if it is going to call it.
- Add the line *TestFunc();* above the return statement to make the actual call to the function.

You can now build and test the program.

1.8 Section Questions

1. What is the purpose of the SimpleIO files?

2. Why should you copy files into your project folder before you add them, instead of adding them from whatever folder they happen to be located in?

3. When you start creating your own source (.cpp) files, what other type of file are you likely to want to create?

4. What are the two key objectives that are accomplished when you define a variable?

5. Explain the type of error you would have gotten if you had not included FirstFunc.h in Example 1.9.

6. Why does it make sense to keep your main functions in source files by themselves?

1.9: Introduction to Debugging Tools Walkthrough

Walkthrough available in DebugIntro.avi

When you are just starting to learn programming, the debugging capabilities built into the IDE can serve two very useful roles:

1. They can fulfill their stated purpose, which is to help you find bugs in your code
2. They can provide a visual demonstration of the activities being performed as your code executes

This second capability is often overlooked in programming courses. But it can be a very powerful learning tool—one that we will utilize frequently throughout this text.

1.9.1: What is a debugger?

A debugger is not, as its name would seem to imply, a tool that removes bugs from your code. Sadly, the identification and removal of errors is still a task that is pretty much left to the programmer. A debugger, instead, is a collection of tools that can be used to help you locate bugs in your code.

Your typical debugger offers at least four capabilities that can be used to help identify bugs in your code:

- *Breakpoints:* The ability to identify points in your code where the program will pause while it is running. Because modern computers run so quickly, without entering such a paused state it is impossible to use the remaining debugging features.
- *Stepping:* The ability to execute your code line by line. This allows you to watch what happens in your computer one step at a time.
- *Inspection:* The ability to examine the contents of memory (registers too, if you've got a taste for trivia) using the variable names declared in your code.
- *Call Stack:* The ability to examine the collection of functions in the process of being called (see Section 1.1.2 for a further explanation of stacks and their role in modern computers).

We now consider each of these features individually, using the Multifile project we developed in Section 1.8.

1.9.2: Breakpoints

Our early attempts to run the debugger have already underscored our need be able to pause the program: our console screen disappeared before we could read it. To solve that problem, all you need to do is click the mouse on the vertical bar to the left of the *return 0;* statement in main. When you then run the program, you will find it stops there, as illustrated in Figure 1.9 (note the arrow superimposed on the breakpoint dot).

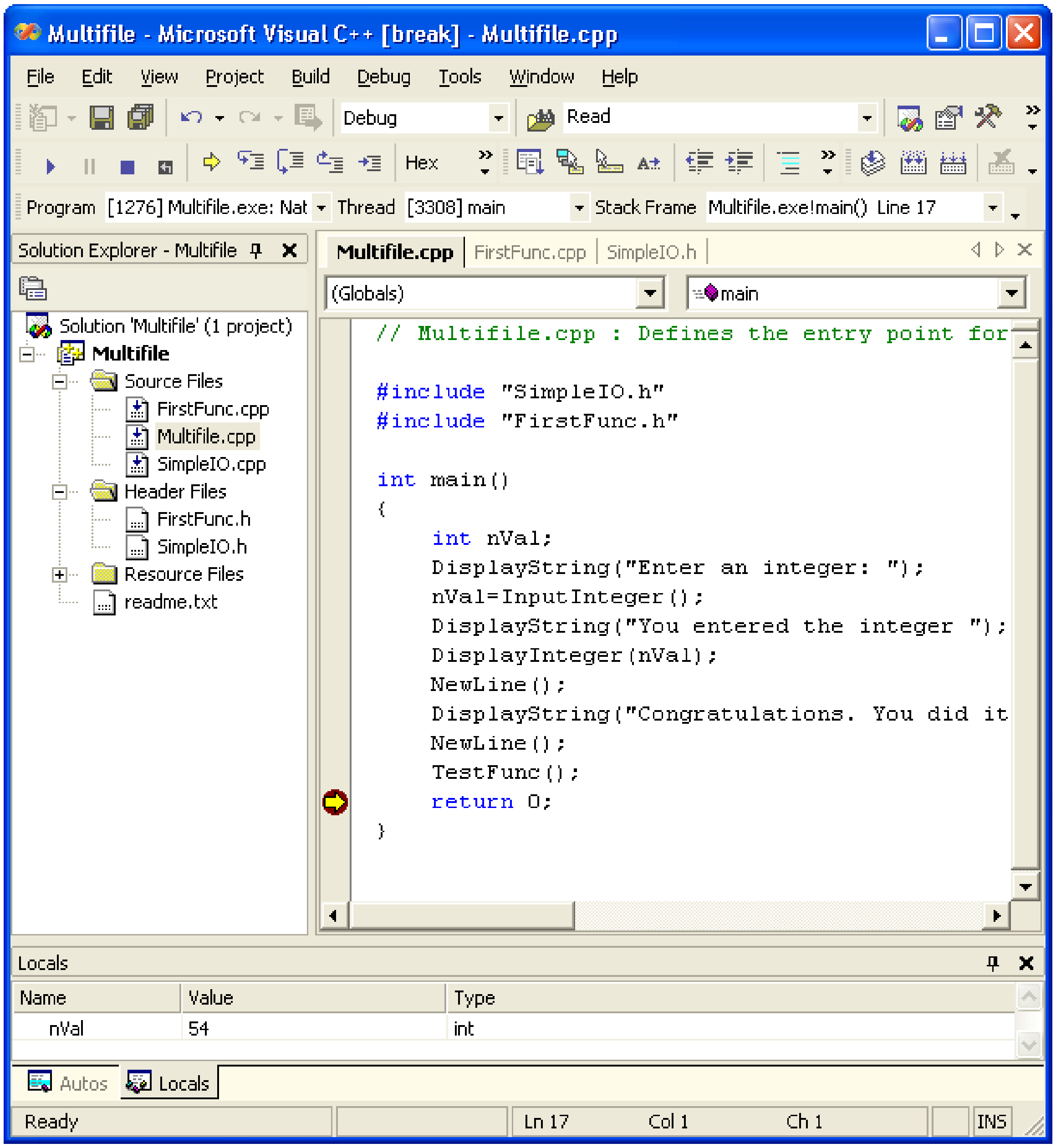
So paused, you can then examine the contents of the console window by clicking on the appropriate button on the Windows task bar (the bar of buttons, normally at the bottom of the screen, used to control what program is active).

Breakpoints can be put on almost any line of code except variable declarations. Their presence will only pause the program if the line of code is encountered while the program is running. What this means will become more apparent in Chapter 5, when we introduce the concept of branching.

Some other useful facts about breakpoints:

- To remove a breakpoint, you can just click the breakpoint a second time.
- You can also deactivate a breakpoint (using a right click), which leaves the breakpoint in place but prevents it from stopping your program.
- It is possible to set *conditional breakpoints*—breakpoints that only stop the program if certain conditions are met. These can be useful, but are beyond what we need to know to get started.
- To resume running after pausing at a breakpoint, you can press the button (or select the same menu item) that you used to start the program running. The

program will then proceed until either: a) it ends (i.e., main() returns) or b) another breakpoint is encountered.



Debug | Read | Hex

Program [1276] Multifile.exe: Nat | Thread [3308] main | Stack Frame Multifile.exe!main() Line 17

```
// Multifile.cpp : Defines the entry point for

#include "SimpleIO.h"
#include "FirstFunc.h"

int main()
{
	int nVal;
	DisplayString("Enter an integer: ");
	nVal=InputInteger();
	DisplayString("You entered the integer ");
	DisplayInteger(nVal);
	NewLine();
	DisplayString("Congratulations. You did it
	NewLine();
	TestFunc();
	return 0;
}
```

Name	Value	Type
nVal	54	int

Ready | Ln 17 | Col 1 | Ch 1 | INS

Figure 1.9: Program paused at break point

1.9.3: Stepping

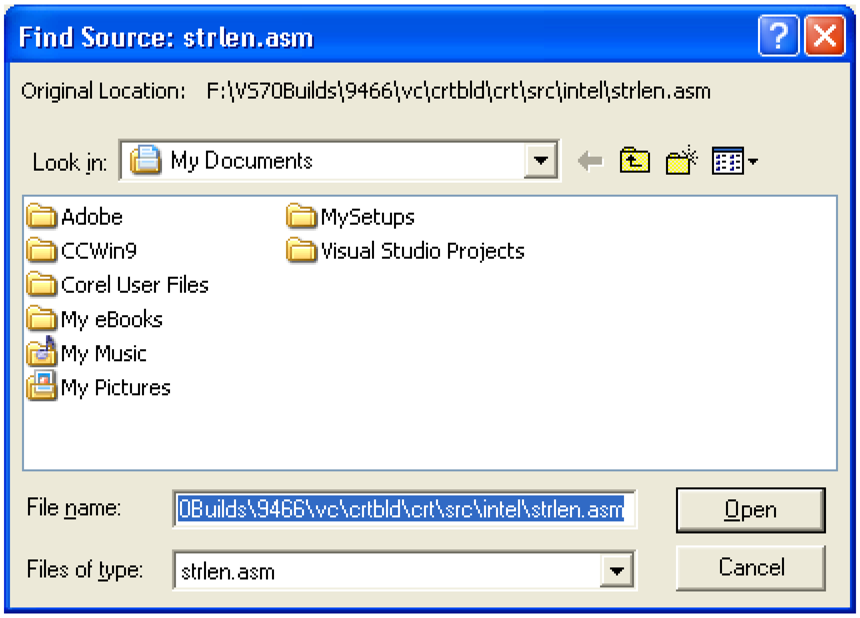
Once you've stopped your code at a breakpoint, it is often convenient to watch what happens as your code executes, line by line. This process is referred to as stepping.

Three forms of stepping are particularly common:

- *Step over:* The program performs the entire line of code, moving to the line below. That means that should a function be called in the line of code (e.g., in the line *n3=Plus(n1,n2);* within TestFunc), the function called will be completed as part of the step—unless there happens to be a breakpoint in the function that causes execution to pause.
- *Step into:* The program steps to whatever code is executed next, going into another function if necessary. For example, stepping into the line *n3=Plus(n1,n2);* would cause us to go to the first line of the Plus() function.
- *Step out of:* Causes the program to run to the point in our code just after the current function returns. This is often used to get us out of functions we accidentally stepped into.

Stepping is very straightforward, and can provide us with useful insights into how our code is running. There are many common coding errors (e.g., missing break statements in a case construct, discussed in Chapter 6) that are easily detected by stepping through the code. Stepping though an entire application is also a very useful approach to testing software, as the stepping process tends to focus the developer's attention.

There is one common problem that virtually everyone discovers within minutes of learning to step through code. That problem is accidentally stepping into a library function. What typically happens is that a dialog box (see Figure 1.10) prompts you that it does not know where the source code for the function is located—and since that code is probably not installed, you find that your only viable alternative is to cancel. Once you cancel, the IDE will offer you the opportunity to look at the disassembly window.


Find Source: strlen.asm

Original Location: F:\VS70Builds\9466\vc\crtbld\crt\src\intel\strlen.asm

Look in: My Documents

Adobe
CCWin9
Corel User Files
My eBooks
My Music
My Pictures
MySetups
Visual Studio Projects

File name: 0Builds\9466\vc\crtbld\crt\src\intel\strlen.asm

Files of type: strlen.asm

Figure 1.10: Accidentally stepping into the strlen() library function

Just say no! If you do not, you'll find yourself deep in assembler code. (Should this happen, however, just "Step Out", then go to *Debug|Windows* and close the Disassembly window).

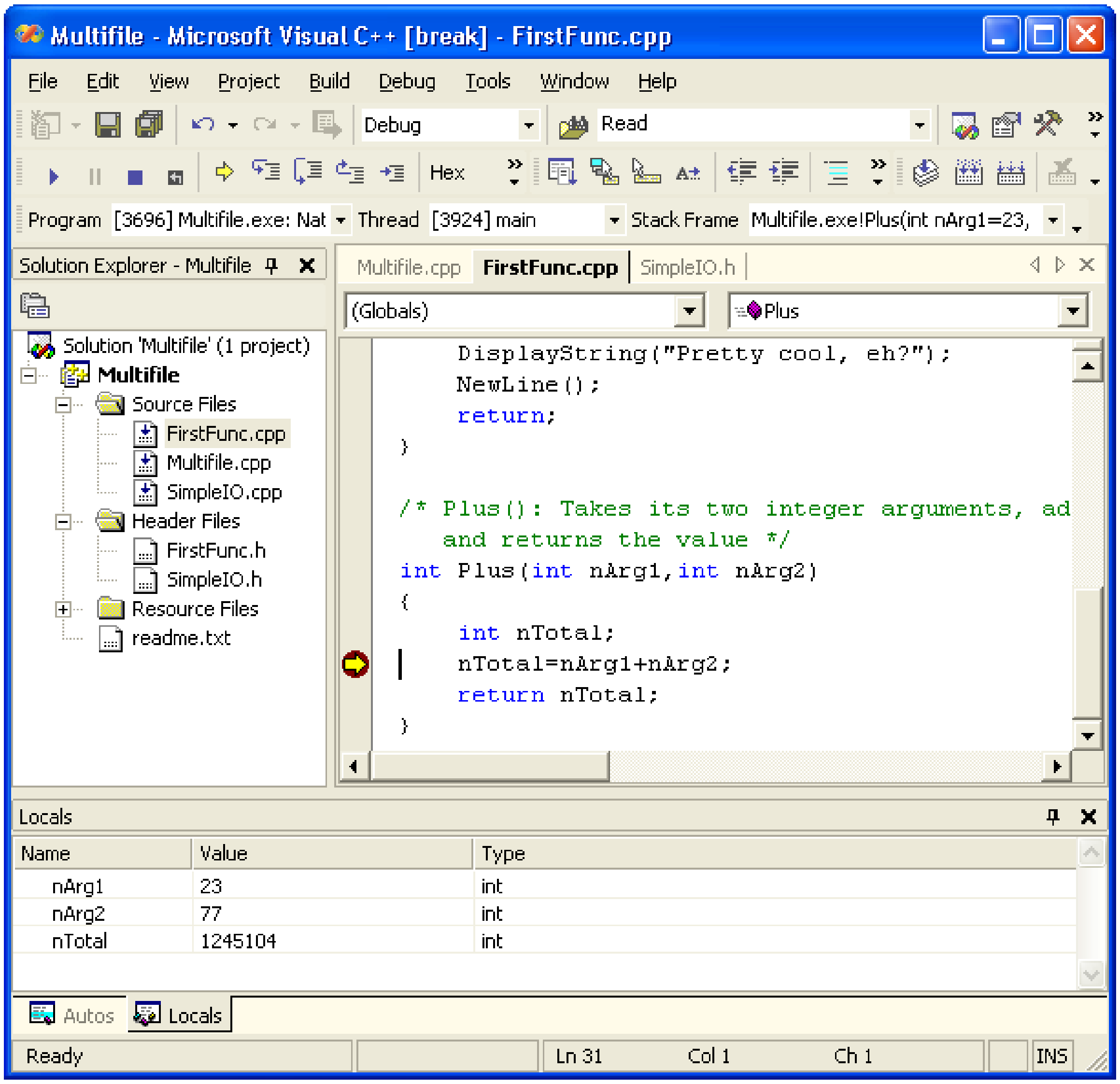
The reason for this bizarre behavior is that the debugger needs access to your source code in order to perform stepping. In installing the IDE, however, the library source code may not be copied—mainly to save space. As a result, the debugger does the only thing that it can do: it takes the machine code it finds and translates it into assembly language—without the benefit of useful things like labels and variable names (a process called disassembly).

1.9.4: Inspection

An inspection window is just a name for a window that provides a list of variables and their values. Inspection is normally tied closely to stepping, as the type of information you are most likely to be interested in, while stepping through your code, is how variable values are changing.

There are many different types of inspection windows. The most commonly used in this text are the "Locals" window and the "Auto" window. These appear by default when your program is paused in the debugger, tabbed so you can switch between them. Should you accidentally close them (a common occurrence, given Visual Studio .NET's huge selection of window types), you can find them under *Debug|Windows*. Because they show values in a running program, they only appear while the program is running under the debugger.

The main difference between the "Locals" and "Auto" inspection windows is that "Locals" displays values for all local variables, whereas "Auto" attempts to figure out the variable values that you are most likely to be interested in. An example of the "Locals" window, paused at a breakpoint within a call to the function Plus() in our Multifile project, is presented in Figure 1.11.


Multifile - Microsoft Visual C++ [break] - FirstFunc.cpp
File Edit View Project Build Debug Tools Window Help
Debug
Read
Hex
Program [3696] Multifile.exe: Nat
Thread [3924] main
Stack Frame Multifile.exe!Plus(int nArg1=23,
Solution Explorer - Multifile
Solution 'Multifile' (1 project)
Multifile
Source Files
FirstFunc.cpp
Multifile.cpp
SimpleIO.cpp
Header Files
FirstFunc.h
SimpleIO.h
Resource Files
readme.txt
Multifile.cpp
FirstFunc.cpp
SimpleIO.h
(Globals)
Plus
Autos
Locals
Ready
Ln 31
Col 1
Ch 1
INS

```
    DisplayString("Pretty cool, eh?");
    NewLine();
    return;
}

/* Plus(): Takes its two integer arguments, ad
   and returns the value */
int Plus(int nArg1,int nArg2)
{
    int nTotal;
    nTotal=nArg1+nArg2;
    return nTotal;
}
```

Locals

Name	Value	Type
nArg1	23	int
nArg2	77	int
nTotal	1245104	int

Figure 1.11: "Locals" window (bottom of screen) while paused at breakpoint

As the yellow arrow on the breakpoint illustrates, we are about to execute the step of code:

nTotal=nArg1+nArg2;

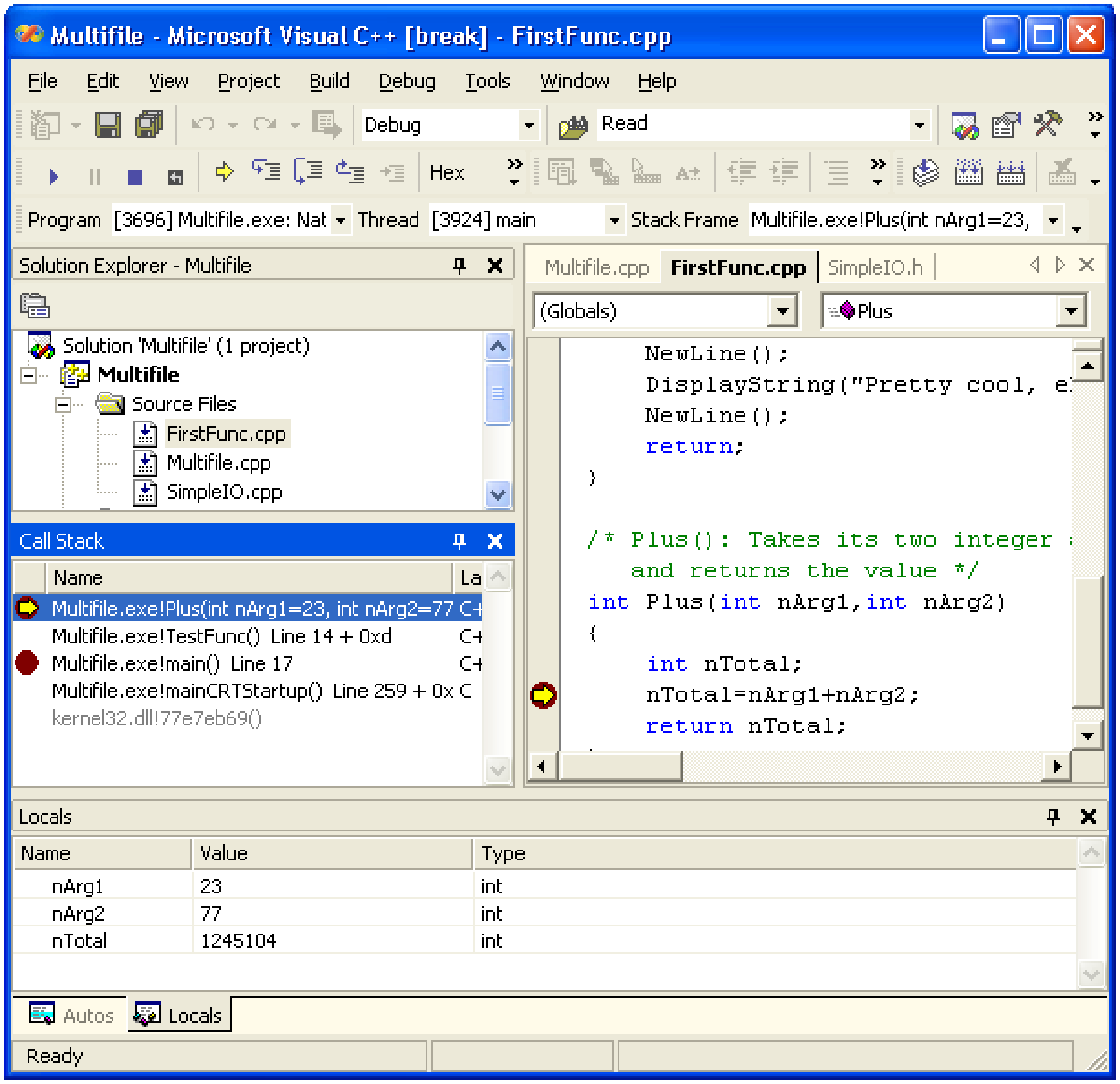
The practical meaning of this is that we have yet to place a value into nTotal (i.e., it has not been initialized). As a result, while nArg1 and nArg2 have values established when the function was called, nTotal is—for our purposes—essentially a random number.

Test your understanding 1.10: What value for nTotal would appear in the Locals window if we were to step over the line of code where we were currently paused?

It is probably worth mentioning that just as you can view variable values in a properties window, it is also possible to change those values. This capability is rarely so useful to a novice programmer that it justifies the high risk of "shooting yourself in the foot" that accompanies using it.

1.9.5: The Call Stack

The final debugging feature to be presented at this time is the call stack, shown in Figure 1.12 (while the program was paused at the same breakpoint in Figure 1.11).


Multifile - Microsoft Visual C++ [break] - FirstFunc.cpp
File Edit View Project Build Debug Tools Window Help
Debug
Read
Hex
Program [3696] Multifile.exe: Nat
Thread [3924] main
Stack Frame Multifile.exe!Plus(int nArg1=23,
Solution Explorer - Multifile
Solution 'Multifile' (1 project)
Multifile
Source Files
FirstFunc.cpp
Multifile.cpp
SimpleIO.cpp
Multifile.cpp FirstFunc.cpp SimpleIO.h
(Globals)
Plus
Call Stack
Locals
Autos Locals
Ready

```
    NewLine();
    DisplayString("Pretty cool, e
    NewLine();
    return;
}

/* Plus(): Takes its two integer 
   and returns the value */
int Plus(int nArg1,int nArg2)
{
    int nTotal;
    nTotal=nArg1+nArg2;
    return nTotal;
```

```
Name                                              La
Multifile.exe!Plus(int nArg1=23, int nArg2=77     C+
Multifile.exe!TestFunc()  Line 14 + 0xd           C+
Multifile.exe!main()  Line 17                     C+
Multifile.exe!mainCRTStartup()  Line 259 + 0x     C
kernel32.dll!77e7eb69()
```

Name	Value	Type
nArg1	23	int
nArg2	77	int
nTotal	1245104	int

Figure 1.12: The call stack (middle pane)

The call stack window provides the programmer with a couple of useful facilities:

- It shows the functions currently being executed, along with the values of their arguments. For example, in Figure 1.12, it tells us:
    - we are in the Plus() function,
    - that was called from TestFunc(),

    - that was called from main(),
    - that was called from mainCRTStartup (a function MS Windows uses to get main() to run in the console window).

  Such information is often useful because, in a complex program, you will not always know how you got to a breakpoint when your program stops.
- It provides a convenient means of navigating between functions. For example, if you were to double click TestFunc() on the call stack, it would move the cursor to your current position in TestFunc() (which would be the line where it calls the Plus() function) and you could look at the values of variables local to TestFunc(), such as n1, n2 and n3.

As was the case with inspection windows, the call stack is only accessible when your program is actually running, and paused in the debugger.

1.10: Review and Questions

1.10.1: Review

Upon completing Chapter 1, you should be able to use the primitive C++ data types, define simple functions, and create a simple, multi-file project in Visual Studio .NET. The most critical concepts presented in the chapter are as follows:

A variable is a particular location in memory where data can be stored and accessed by name. Two types of variables are presented in the chapter. A scalar variable refers to a single location that is associated with a variable name. An array is a collection of locations, each of which can be accessed with a name and an integer offset—specified in the form:

variable-name[integer-offset]

In C/C++, the first element in an array is always at offset 0.

Variables must be declared before they are used. The declaration of a scalar variable takes the form:

variable-type variable-name = *initialization-value*;

where the = initialization-value may be omitted. The declaration of an array takes the form:

variable-type variable-name[*element-count*] = {*initialization-value1, initialization-value2, initialization-value3,…}* ;

where the = {list of initialization values} may be omitted.

The variable-type expression can include a fundamental data type (e.g., int, char, double float) that may also, in some cases, be preceded by a qualifier (e.g., short, long, signed, unsigned) or followed by an asterisk (*)—in which case the variable (or array) being declared holds an address (or an array of addresses).

Variables may be declared in a number of places within a program. Variables declared outside of any function are referred to as global or static variables, and may be used by any function defined within the file after the declaration. The extern keyword may be used to allow such variables to be shared between files. Variables defined within a function are said to be local to that function. Their values can only be accessed by name within the function where they are declared. As soon as a function returns, the memory locations used by all of its local variables are discarded, and made available for reuse by other functions.

C has four fundamental data types: char, int, float and double. The char and int data types are integer data types, while double and float represent real numbers. The sizes and ranges of the integer data types are summarized below:

Type	***.NET Size (in bytes)***	***Minimum value***	***Maximum value***	***Comments***
char	1	-128	+127	Two byte characters are on the way, to support Unicode
unsigned char	1	0	+255	
short	2	-32768	+32767	Was the default size for **int** in pre-1995 MS compilers (v. 1.5 and below)
unsigned short	2	0	+65535	
int	4	-2147483648	+2147483647	**int** is currently the same as **long**
unsigned int	4	0	+4294967296	
long	4	-2147483648	+2147483647	8-byte **long** already used in C#. MS provides **_int64** 8-byte integer type.
unsigned long	4	0	+4294967296	

The sizes and ranges of the real number data types are summarized below:

Type	***.NET Size (in bytes)***	***Range***	***Significant digits***	***Comments***
float	4	+/- 3.4 * 10^{38}	~ 6-7	Not used that much any more
double	8	+/- 1.7 * 10^{308}	~ 15	**double** and **long double** are same in .NET. 10 byte **long double** were used in some earlier versions
long double	8	+/- 1.7 * 10^{308}	~ 15	

Although the char data type is really just a very small integer, it is mainly used for text purposes. When storing characters, letters are represented using the ASCII coding system. So that we do not have to remember the coding sequence, apostrophes around a character can be used to represent its ASCII value (e.g., writing 'a' is the same as writing 65 or 0x41). For text sequences, such as words or sentences, an array of char data elements is typically used. The C library makes extensive use of NUL terminated strings, which are arrays of characters with a NUL character (a byte with the value 0, or 0x00) used to mark the end of the text. Although we will assume 1-byte characters throughout this text, it is likely that applications will transition to 2-byte characters over the next decade, and that ASCII will be replaced by the more flexible Unicode representation scheme.

The syntax for defining a function is simply a declaration followed by a code block—which is a collection of one or more statements within enclosing braces {}. Such code blocks are critical in C/C++ syntax, because they can enclose multiple statements (semi-colon terminated expressions) anywhere a single statement can be used. Within functions, one or more **return** statements are normally present. Each return statement must be followed by an expression that matches the return type in the function declaration. The only exception to this rule is functions with a return type specified as **void**, which are functions that don't return any value. For such functions, a **return** statement does not need to be present. If one or more **return** statements are present, a semi-colon—instead of an expression—must follow them immediately.

By passing arguments into a function, we allow the same program code to be used in many places. There are two ways that arguments can be passed in: by value and by reference. When function arguments are passed in by value, local copies of the arguments are made that any changes to arguments made within the function affect only the copies—the original values passed into the function remain unchanged. The other way that arguments can be passed in is by reference. When passed in this way, the function creates an alias for its arguments instead of a copy. As a result, changes made to an argument within a function also affect the variable that was passed in. In C++, a reference argument can be declared by using an ampersand (&) in front of the argument name. C++ also provides two additional extensions to C: default values for function arguments can be specified when a function is declared, and allowing different functions to be defined with the same name—provided their arguments differ. This latter extension is called function overloading.

The basic process through which a running program is created is called the compose-compile-link-load cycle. Composing is the process of writing you own source code, in a language such as C or C++, and saving it in files. Compiling is the process of taking a file containing source code and translating it into a file containing object code—which is similar to machine language except that certain key pieces of data, such as memory addresses, are left unspecified. In addition to object files created by the compiling the programmer's source code, a typical program also uses library files (supplied with the compiler) that contain object code for standard functions. The process of merging object code files—both created by the compiler and libraries—is referred to as linking. This process is performed by a program known as a linker, and produces an executable (e.g., a .exe) file. A program called a loader, normally part of the operating system, can then be used to start the executable file running. In MS Windows, double-clicking an icon will normally invoke the loader on the associated program.

Visual Studio .NET is an integrated development environment, or IDE. This means the environment contains editors for composing source code, a built-in compiler and linker for creating executable code, and debugger that can be used to load and test running programs. The primary organizing tool in .NET is a solution (sometimes referred to as a workspace), into which one or more projects can be inserted. A project, in turn, is an organized collection of source code and other files developed by a programmer that can be combined (e.g., compiled and linked) to create a single executable target. In the present text, we will primarily be interested in projects, so that all our solutions will consist of a single project.

To create a project in Visual Studio .NET, you must first decide on the appropriate target type. In this book, we will always be creating console applications, the simplest form of program. Once a new project has been created, you add new or existing source files to it (.c, .cpp or .h) files. You can then build the project (which causes the source files to be compiled and then, if the compile is successful, linked together). When getting started in programming, errors are commonly encountered during the compile and link stages.

These need to be corrected before your program is ready to run. Once a build is successful, you can begin to debug the project.

The debugger that is built into .NET is one of its most powerful features. The term debugger is somewhat of a misnomer, however, since the tool does not remove bugs. Instead, it helps the programmer find bugs. There are four main capabilities the debugger provides that assist in finding bugs. Breakpoints allow the programmer to pause the program while it is running. Inspection and watch windows allow the programmer to examine the values of program variables while the program is paused. Stepping capabilities allow the programmer to execute code one step at a time, normally used in conjunction with inspection windows. Finally, the call stack allows the programmer to identify the sequence of function calls leading to a particular location in the program where it is paused, or where an error has been encountered. These debugging tools are invaluable in finding program errors. They also provide a window on program execution that can help novice programmers understand how their programs work.

1.10.2: Glossary

Abstract Data Type – A user defined data type that is derived from the primitive data types such as char, int, long, unsigned float, etc.
Array – A variable type that is of the same name and type but can hold multiple values.
ASCII – American Standard Code for Information Interchange a code for representing the English language using a standardized set of numbers in this case 0 to 127.
Binary – A numbering system having only two unique digits 0 and 1.
Breakpoint – A user defined pause in the source code of a program written in Visual Studio.NET.
Call Stack – A window in Visual Studio.NET that shows the processes that a computer program called in order to execute the source code.
Class – A programmer-defined description of an object.
Code Block – A collection of source code statements, delimited by braces {}, located within a function or other construct.
Debugger – A program that is designed to find errors in source code in order to make compilation into an executable program easier.
Decimal – A numbering system having ten unique digits 0 to 9 inclusive.
Declaration – A statement that specifies a data type and associates it with a variable name.
EBCDIC – Extended Binary Coded Decimal Interchange Code used by IBM as a way of representing characters on their large mainframe computers.
Executable Code – Code that can only be read by a computer can also be referred to as a computer program.
extern – A keyword used to allow variable declarations to be shared between files.
Function Declaration – A line of code that specifies the name, return type and argument types of a function. It may appear as part of a function definition or in a header file.
Function Definition – A declaration line and body used to establish the code to be invoked when a function is called.

Global Variable – A variable whose scope extends across all files in a program
Header File – A file used by C/C++ that allows for the declaration of functions, classes and other predefined variables necessary to programming an executable version of source code.
Hexadecimal – A numbering system having sixteen unique digits 0 to 9 and A to F.
Integrated Development Environment – An environment or computer program that allows for the easier transformation of source code to executable code by allowing for the complete linking, loading and debugging processes to be handled by one computer program.
Namespace – A name given to a collection of code in order to prevent inadvertent duplication of the functions and objects being defined within the code.
NUL Character – A character with a value of 0x00 that represents the end of a string of characters in an array.
Octal - A numbering system having eight unique digits 0 to 7.
Pass by reference – To pass a variable by reference means to specify it in such a way that it can be changed by the function into which it is passed.
Pass by value – To pass a variable by value means to make a copy of the variable when passing it into a function, saving the original from being altered.
Pointer – A variable type that holds a memory address.
Preprocessor – A component of the build process that performs processing operations on source code (such as inserting header files into the code) prior to compilation.
Prototype – An alternative name for a function declaration.
Real Number – A number that will allow for fractions to be represented.
Return Type – The data type of the value returned by a function.
return Statement – The construct that C++ uses to specify a return from a function, consisting of the return keyword followed by an expression that matches the return type.
Return Value – A value returned by a function or procedure after the source code has been completed and a result determined.
Scope – The portion of code in a program in which a variable can be accessed by name.
Skeletal Code – Code placed by the compiler in order to make the task of programming easier by showing an outline view of a program already defined.
Source Code –A program written in a manner that is readable to humans and that can be compiled by a compiler in order to transform the code into an executable format
Static Variable – A variable that exists throughout the execution of a computer program, frequently a global variable.
Unicode – A standard for representing characters that is 16 bits in length, allowing for up to 65,536 characters to be represented.

1.10.3: Questions

Questions 1.1-1.10: For each of the following errors that might be present in a program, identify where it is most likely to be detected:

A. By the compiler
B. By the linker
C. By the programmer

#	Error
1	A variable name is misspelled
2	The programmer forgot to define one of the functions called in a project
3	A variable is declared to be of type "integer" instead of "int"
4	A semicolon is missing at the end of a line
5	A function is missing a closing brace (}).
6	A required library (.lib) file was accidentally deleted
7	A function defied by the project (e.g., DisplayInteger) is called with the wrong number of arguments
8	The "using namespace std; " is omitted after an include statement that requires it
9	The programmer defined the same function twice, in separate files
10	An incorrect formula is used to compute the return value of a function

1.11. If you were to create a project with many .cpp source files, discuss the pros and cons of:

a. Creating a single .h file with every function declaration
b. Creating a separate .h file for every .cpp file you created

1.12. Explain why a definition implies a declaration, but a declaration does not necessarily imply a definition.

1.13. In a multifile project, does the compiler care what file contains a particular function definition. If so, why? If not, why should we care if it doesn't?

1.14. In a new single file project called *Doubler*, create a main function that:

- Prompts the user to enter an integer
- Reads an integer from the user
- Multiplies the integer by 2
- Displays "Your integer doubled is: " followed by the value
- Ends

1.15. In a new single file project called *Circle*, create a main function that:

- Prompts the user to enter an radius value
- Multiplies the radius by 2*pi (3.1416)
- Displays the circle's circumference
- Ends

Although this is very similar to *Doubler* (Question 14). It can't be done with integers. That means you'll need to use the real number data type (double) instead of the integer type (int). You'll also need top choose the right functions from SimpleIO (from functions listed Example 1.5).

1.16. In a new multifile project, *MDoubler*, create a program that does what Doubler (Question 14) does except that a function you create called Double() is used. The Double() function should be declared as follows:

```
int Double(int arg);
```

It should take its argument, multiply it by two, then return the result. (Note: this will be quite similar to the Plus() function in the chapter, only even simpler). Your project should have the same structure as the multifile project (Section 1.8), namely:

- A main() function defined in one .cpp file
- The Double() function defined in another .cpp file
- An include file, holding the declaration of Double().

1.17. In a new multifile project, *Circum*, create a program that does what Circle (Question 15) does except that a function you create called Circle()—which takes a radius value as an argument and returns a circumference—is used. Your project should have the same structure as the multifile project (Section 1.8), namely:

- A main() function defined in one .cpp file
- The Circle() function defined in another .cpp file. (You should decide how it should be prototyped.)
- An include file, holding the declaration of Circle().

1.18. Explain what is meant when we say that "the debugger doesn't fix bugs"?

1.19. What are the four principle debugger capabilities that we covered?

1.20. In Figure 1.11, identify the source of each of the three values in the variable window.

Chapter 2

C++ Operators and Programming Constructs

Executive Summary

Chapter 2 shows how operators and programming constructs are implemented in the C/C++ programming language. It proceeds on the assumption that the reader is familiar with the basics of program construction, and therefore focuses on issues that are particularly relevant to C++.

The chapter begins by examining C++ operators, noting that operators are really just another syntax for calling a function. Assignment operators, arithmetic operators, relational operators, logical operators and bitwise operators are then discussed, along with the concept of operator precedence. Finally, the notion of type conversions and type casts within operator invocation and function calls is introduced. The chapter then turns to the C++ branching constructs (if…else, switch…case) and looping constructs (while, for, do…while). It then provides a series of examples, emphasizing how commony used library functions can be implemented in C++.

Learning Objectives

Upon completing this chapter, you should be able to:

- Describe the similarities and differences between operators and functions
- Explain operator precedence
- Use the assignment operator, understanding the difference between assignment and initialization
- Use the various arithmetic operators
- Apply prefix and postfix operators, distinguishing between their effects
- Understand what is meant by Boolean values, and how these values are generated by relational and logical operators
- Understand bitwise operators, and use them to set and read bits within an integer
- Explain how conversions are used in C/C++ to work with different value types, and how to force conversions.
- Take any logical problem that involves branching and turn it into a C/C++ if construct or switch…case construct
- Take any logical problem that involves looping and turn it into a C/C++ while, for or do…while construct.
- Explain the purpose of a number of common C/C++ library functions, including strlen, strcmp and atoi, and explain how they could be implemented.

2.1: Introduction to Operators

Operators are special characters or keywords in C/C++ that can be applied to one, two or (in one case) three expressions. Their purpose is to invoke code that would otherwise require a function call. The use of operators in programs offers a number of benefits:

- Use of operators can make expressions look less convoluted than function calls. For example, W+X+Y+Z is easier to understand than its functional equivalent might be, e.g.,:

  ```
  Plus(W,Plus(X,Plus(Y,Z)))
  ```

- Use of operators provides some capabilities that are also available in complex C++ function declarations. For example:
  - Ability to access variables by reference
  - Limited overloading

The apparent naturalness of C++ operators disguises a lot of activities going on beneath the surface. Up to this point in the text, where we have used operators we have relied on this naturalness to write expressions such as:

```
nTotal=nArg1+nArg2;
```

(which involves two operators: + and =) and then implied "you should be able to figure out what this does". In this chapter, however, it makes sense to be a little bit more systematic and rigorous in our approach. Towards this end, we first consider the fundamental equivalence between operators and functions, then address the specific issue of operator precedence.

2.1.1: Operators as Functions

Operators in C/C++ provide a convenient mechanism for invoking code that would otherwise require a function call. Indeed, throughout this text, we think of operators as being equivalent to functions—having both arguments and return values. For example, when we use the addition operator (+) operator:

```
int val1=1,val2=3,val3;
val3=val1 + val2;
```

we are effectively calling a function—a function we will refer to as operator+ (for reasons that will become clear when we discuss C++ operator overloading)—that could be prototyped as follows:

```
int operator+(int a1,int a2);
```

So prototyped, our code could be rewritten:

```
int val1=1,val2=3,val3;
val3= operator+(val1,val2);
```

By now, you should be able to recognize that our "operator+" function does exactly the same thing as the Plus() function we have already talked about so many times. As it turns out, there was justification for using Plus() instead of operator+() as the name of that function: calling the function operator+ would not be legal, as you are not allowed to change the behavior of operators as applied to primitive data types. Otherwise, as we shall see in Chapter 15, you can call our "operator functions" as an alternative to using actual operators—although no obvious reason for doing so can be imagined.

Thus, our use of function names such as operator+(), operator<() and operator=() in this chapter will be for conceptual purposes only—to illustrate argument and return types. Why those particular names were chosen becomes apparent only when we reach the In Depth section on C++ operator overloads at the end of the chapter.

In C++, the correspondence between operators and functions is nearly perfect. What confuses most people in seeing the relationship between operators and functions is the placement of operators. Function names are always placed in front of their arguments, which are kept separate from other pieces of code using parentheses. Operators, however, can appear in front of, behind, and between the arguments. In fact, the five legal placements of operators in C/C++ are listed in Table 4.1:

Type	Operator Appearance	Functional Equivalence	Examples	Comments
Unary (prefix)	**op** arg1	**op**(arg1)	+12, ++i and **-**3	
Unary (postfix)	arg1 **op**	**op**(arg1)	i++	In C++, a second argument is added to operator overload, to distinguish from prefix versions
Containment	**op1** arg1 **op2**	**op**(arg1)	val**[**3**]**, **(**x+2**)**	Used to identify start and end of value, and to control execution order
Binary	arg1 **op** arg2	**op(**arg1,arg2)	X+3, i<100	
Ternary	arg1 **op1** arg2 **op2** arg3	**op**(arg1,arg2,arg3)	i<100 **?** X **:** Y	Conditional operator is only ternary operator

Table 4.1: Operators and functional equivalence

2.1.2: Operator Precedence

A final characteristic of operators that distinguishes them from functions is the concept of operator precedence. When calling a function, the use of parentheses unambiguously tells us the order in which the various expressions will be resolved. For example, if we call our function Plus() as follows:

```
Plus(3,4*5);
```

we know that 4 will be multiplied by 5 before the 3 is added to it. Written in operator form, however, it is less clear:

```
3+4*5
```

Should the answer be 23 (3+20) or 37 (7*5)? In algebra, we learned we should do multiplication before addition. Thus, we'd probably guess (correctly) that the answer is 23. But what about an equally simple expression such as:

```
20/5*2
```

Should the answer be 8 (4*2) or 2 (20/10)?

Programming languages solve the problem of operator ambiguity by using a concept called *operator precedence* (or *operator priority*). Conceptually, we can think of each operator having a numeric rank. In any expression, operators having the highest rank get performed first. For example, * has higher precedence than +. That means in 3+4*5, the functional equivalent becomes:

```
operator+(3,operator*(4,5))
```

If the plus operator had the higher precedence, the functional equivalent would have been:

```
operator*(operator+(3,4),5)
```

Where operators have the same precedence, operations are performed from left to right, In our example, then, the equivalent would have been:

```
operator*(operator+(3,4),5)
```

if + and * had the same precedence, since the plus sign appeared first.

Test your understanding 4.6: Write the functional equivalents of 20/5*2 assuming: a) * has higher precedence than /, b) / has higher precedence than *, c) * and / have the same precedence. Which of these orderings would you think is most likely?

Whatever an operator's precedence, parentheses—themselves a form of operator—can be used to change the order of execution. For example, (20/5)*2 is 8, while 20/(5*2) is 2.

Every C/C++ operator has an assigned precedence. While it is quite possible to create detailed tables of precedence, you will not be given one here. Memorizing such tables, and then relying on your memory of what gets done before what, is an invitation to hours of debugging pleasure. Even if your memory is so prodigious that you actually got the order right, nobody reading your code will be sure. Just put the parentheses in!

2.1.3: Assignment Operators

C/C++ provides an assignment operator (=), plus a collection of assignment operators combined with other operators (e.g.,+=, *=, -=, /=, etc.) to be used in setting variable values. The pure assignment operator takes an expression on the right hand side (RHS) of the operator and evaluates it, then places the value in the variable or variable expression (e.g., array coefficient) on the left hand side (LHS). Examples of its use include:

```
int x1,x2,x3;
int arVals[5]={1,2,3,5,8};  // Initialization is different from assignment
x1=1;
x3=x1+(x2=4);
// x1 is 1, x2 is 4, x3 is 5
arVals[2]=x3+8;
// And the third element of arVals (arVals[2]) is now 13
```

Structurally, it is applied as follows:

lvalue = expression

In functional form, the operator can be prototyped roughly as follows:

lvalue-type **operator=**(*lvalue-type* &lvalue,*lvalue-type* expression);

The functional form gives us a number of insights into the way the operator works. First, it shows us that the LHS argument is actually a reference—meaning its value can be changed during the assignment process (which, of course, is the purpose of the process). In fact, the LHS of an assignment operator is encountered so often in descriptions of C/C++ functions, it is sometimes referred to as an **lvalue** (or **l-value**) expression. Up to this point, we have seen two types of lvalue expressions:

- Variable names (e.g., x1=1)
- Array elements (e.g., arVals[2]=x3+8)

When we discuss pointers and structures (Chapter 3), we'll find a number of other types of lvalue expression also exist.

Assignment Return Values
The second piece of information the functional form provides us with is the fact that an assignment operation returns a value (the same data type as the lvalue expression). The actual value returned is whatever value the RHS evaluates to. Thus, the operator X=4 returns the value 4. This aspect of the assignment operator is demonstrated in the line:

```
x3=x1+(x2=4);
```

This line not only assigns a value to x3, it also assigns a value of 4 to x2. Upon completion of that assignment, the value 4 is returned, which is then added to the 1 that was placed in x1, leading to the value 5 being assigned to x3.

The most common use of the return value of the assignment operator is in *chaining* assignments. For example, the expression:

```
x1=x2=x3=4;
```

would be accomplished first by evaluating x3=4, using the 4 returned by that to complete the assignment of x2, using the 4 from that assignment to complete the assignment to x1. Functionally, this would be constructed:

```
operator=(x1,operator=(x2,operator=(x3,4)))
```

which would force the operator= calls to be evaluated from right to left (since function arguments must be evaluated before outer functions can be called).

Distinction Between Assignment and Initialization
The final point that needs to be made about the assignment operator is that its use in expressions is not the same as in declarations. For example, the appearance of an equal sign in the declaration:

```
int arVals[5]={1,2,3,5,8};
```

does not mean the assignment operator has been applied. Rather, the equal sign is used purely as a matter of C/C++ syntax to specify the initialization that is to take place.

This difference between assignment and initialization usually has limited practical impact in structured C++ programming—basically, it means you can't use a brace delimited value list to assign values to an array (or structure) except when you're declaring it. In OOP C++, on the other hand, assignment and initialization are accomplished through calling entirely different functions (the assignment operator overload and the copy constructor, respectively). As a result, the difference can be quite significant.

Composite Assignment Operators
C++ allows many operators to be combined with simple assignments, for example:

```
int x=3,y=2;
x+=y;
// Same as writing x=x+y
```

Their structure and functional form are identical to the basic assignment operator. Their main purpose, then, is to save a few characters worth of typing.

All the arithmetic operators we will be discussing can be combined with assignment in this fashion, leading to:

- += : Adds LHS to RHS and places value in LHS
- -= : Subtracts LHS from RHS and places value in LHS
- *= : Multiplies LHS by RHS and places value in LHS
- /= : Divides LHS by RHS and places value in LHS
- %= : Divides LHS by RHS and places the remainder in LHS

Assignment can also be combined with a variety of bitwise operators.

There was probably a time, in the early days of C, when the use of such composite operators could actually lead to more efficient machine code after a program was compiled. In the world of today's optimizing compilers, however, achieving any performance benefits by using these composite assignment operators is unlikely. As a result, their use will be minimized in this text.

2.1.4: Arithmetic Operators

C++ provides the standard binary arithmetic operators, although more complicated operations (e.g., exponentiation) are accomplished using library functions, such as pow(), found in the math.h standard library. The language also provides a number of unary operators, for specifying positive and negative signs, and special unary operators for incrementing and decrementing integers.

Binary Arithmetic Operators
C/C++ provides operators for the four standard arithmetic operations (+, -, * and /). It also provides a special operator, the modulus operator (%) that is used to return the remainder in a division operation. Some examples of the arithmetic operators in use include:

```
int x1=3,x2=7,x3=21,x4,x5;
double d1=3.00,d2=5.00,d3,d4,d5;
x4=x1+x2;
// x4 is now 10
```

```
d3=d1*d2;
// d3 is now 15
x5=x3 % x4;
// x5 is now 1
d4=x2*d1/x3;
// d4 is now 1
d5=x2/x3*d1;
// d5 is now 0
```

Structurally, the binary arithmetic operators are applied as follows:

numeric-value1 **op** numeric-value2

The exception here is the modulus operator, which is applied:

integer-type1 **%** integer-type2

Functionally, the operators call both LHS and RHS by value, and are functionally equivalent to each other (excepting %, which is limited to integers). Their function prototypes, shown only for +, are as follows:

promoted-type operator+(*numeric-type1* arg1,*numeric-type2* arg2);

The two major issues that programmers should be aware of when applying these operators deal with type promotion and various integer issues.

Type Promotion

One of the most significant hassles of assembly language programming is the need to convert between similar data types. Taking a real value and placing it in an integer register, for example, could easily take 5-10 lines of library function calls. C++ eliminates most of these hassles through a process called *type promotion*.

Type promotion occurs whenever arithmetic arguments of two different types are encountered in an operation. Type promotion converts the return value of the operation to the most "capable type"—the type capable of holding the largest range and number of values. The rules of promotion are relatively simple:

- Smaller integers are promoted to larger integers (e.g., **char** and **short** become **int** and **long** when both types are encountered in an expression).
- **float** real numbers are promoted to **double**.
- Integers are promoted to real numbers.

Type promotion would occur, for example, in the line:

```
d4=x2*d1/x3;
```

because the RHS mixes integers (x2, x3) with a double (d1). The actual order of execution, using the values in the code fragment, would involve:

- performing x2*d1 and promoting that result to a double (21.000),
- initializing a temporary double to the value of x3 (21.000)
- performing the division using the two doubles (21.000/21.000➔ 1.00)
- returning the result and placing it in d4.

Integer Issues
Programmers need to be aware of certain issues in working with integers. One of these is overflow. For example, look at the code fragment that follows:

```
char c1=20,c2=10,c3;
c3=c1*c2;
// c3 is now –56
```

When multiplying c1 and c2, we exceeded the limit of +127 for 1-byte integers. The result, -56, bears little resemblance to the expected value of 200.

> *Test your understanding 4.7:* Where did the value –56, that ended up in c3, come from?

Application of arithmetic operators, when using integers, can also be very sensitive to ordering of operations. In particular, the programmer needs to be continuously vigilant about writing code where integers are divided by other integers within the calculation. *Whenever an integer is divided by a larger integer, the result is 0.* An illustration of this was present in the code:

```
d4=x2*d1/x3;
// d4 is now 1
d5=x2/x3*d1;
// d5 is now 0
```

According to the rules of algebra, d4 and d5 should have the same values: d4 is (x1*d1)/x3, which is the same as d5, which is (x1/x3)*d1. Both should result in a value of 1. But, if you actually run the code, d5 ends up with a value of 0.

The explanation speaks volumes about the need to be careful with integers. In computing the value of x2/x3*d1, the expression is broken into two parts.

- First, x2/x3 is evaluated. Since both x2 and x3 are integers, integer division is performed. Since 7/21 is 0—according to the rules of integer division—the result returns 0.
- Next, 0*d1 is evaluated. Since d1 is double and 0 is an integer, 0 is promoted to 0.000 before the multiplication is performed. But 0.000*7.000 is still zero.

What made the first code (x2*d1/x3) different from (x2/x3*d1) was that the double was in the middle, meaning everything got promoted earlier in the process and no problematic integer division was performed.

In section 4.7, we examine the issue of conversions and a tool called type casting that can be used to avoid some of the problems associated with integers—provided we recognize that those problems exist.

Unary Arithmetic Operators
There are two common categories of unary operators that operate on numbers: those for assigning signs (negative or positive) and those that perform increment/decrement operations. The sign operators, + and -, are unary overloads of the addition and subtraction operators. They have a higher precedence than the various binary arithmetic operators, so the compiler can properly interpret expressions such as:

```
12 * -3 - +17
```

The sign operators cause relatively few problems. The – operator has the effect of multiplying the numeric value of the variable or value to the right by -1. The + unary operator serves no actual purpose—except, perhaps, to provide symmetry with the – operator.

Test your understanding 4.8: What is the value of the expression 12*-3-+17?

The increment and decrement operators (from which C++ got its name), need a bit more explanation than the sign operators. There are two such operators, ++ and --. Their purpose is to increment or decrement the *lvalue* they are applied to (see Section 2.1.1 for explanation of *lvalue* expressions).

Each comes in two versions, a prefix version and a postfix version. Examples of the ++ operator in use are as follows:

```
int x1=3,x2=3,x3=10,x4,x5,x6,x7;
// Uses that are okay
x4=x1++;
// Results: x4==3, x1==4
x5=++x2;
// Results: x5==4, x2==4
// And some really "ugly" uses
++x6=10;
// Results: x6==10
x7=++x3=x1;
// Results: x7==5, x3==5, x1==4
```

Structurally, the prefix versions look like:

++ *integer-lvalue*
-- *integer-lvalue*

Structurally, the postfix versions look like:

integer-lvalue ++
integer-lvalue --

The functional forms of the prefix and postfix are similar, but subtly different. The functional version of the postfix forms can be thought of as:

integer-type operator++(*integer-type* &*lvalue);*
integer-type operator--(*integer-type* &*lvalue);*

The prefix operator, in contrast, is more properly thought of as:

integer-type &operator++(*integer-type* &*lvalue);*
integer-type &operator--(*integer-type* &*lvalue);*

The prefix and postfix versions of the increment/decrement operators differ in three significant ways:

- *Timing of application:* Postfix versions are applied to the lvalue expression after the expression is used in other contexts, while prefix versions are applied before the lvalue is used. In our sample code, for example:

  ```
  x4=x1++;
  ```

  causes x4 to get the value in x1 before the increment takes place. In contrast,

  ```
  x5=++x2;
  ```

  causes x5 to get the value in x2 *after* the increment has taken place.

- *Use of reference in return of prefix:* The functional form of the prefix versions of the operator returns a reference, rather than a value. The practical meaning of this is that the return value of the prefix operator can be thought of as an lvalue expression—meaning that it can be placed on the left hand side of an assignment. Thus, the code:

  ```
  ++x6 = 10;
  ```

  is legal, while the code:

  ```
  x6++ = 10;
  ```

is not, since x6++ returns an integer type, not an lvalue.

- The prefix and postfix versions have different precedence—of particular concern when we get to pointer dereferencing and structures (Chapters 11 and 12).

If the above differences seem rather technical, good! The lesson to be learned here is not to get caught up in the subtleties of prefix and postfix differences. Incrementing and decrementing integers are common enough procedures so ++ and -- get used a lot. *But use them on a line by themselves!* When you do, the differences won't matter.

> *Test your understanding 4.9:* What would be the value of x10 after the following fragment of code is executed:
> ```
>     int x10=10;
>     ++x10=x10+5;
> ```
> What good reason would there be to write code in this fashion?

2.1.5: Relational, Logical and Bitwise Operators

The ability to test values, and modify program execution accordingly, is fundamental to writing any non-trivial program. In this section, we examine the large collection of operators that C/C++ provides for performing testing. We begin by examining what constitutes a test—in the context of the conditional (ternary) operator—then we examine the operators available for relationship testing, logical testing and bit testing.

The Conditional (Ternary) Operator and Boolean Values
Within a C/C++ program, most testing occurs with constructs—such as the branching (e.g., if) and looping (e.g., while) constructs introduced later in this chapter. There is, however, one C/C++ operator that employs tests directly, the conditional or ternary operator (?:). While this operator is not normally of major importance in writing C/C++ programs, it provides a convenient starting point for our discussion of tests.

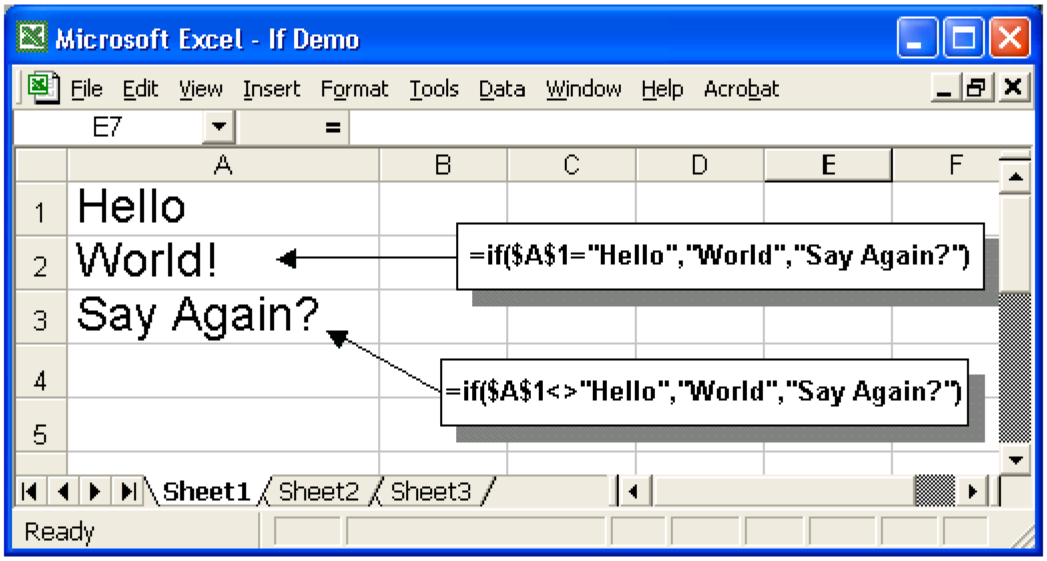

Microsoft Excel - If Demo
File Edit View Insert Format Tools Data Window Help Acrobat
E7
=
Hello
World!
Say Again?
=if(A1="Hello","World","Say Again?")
=if(A1<>"Hello","World","Say Again?")
Sheet1
Sheet2
Sheet3
Ready

Figure 2.1: =if() function in Excel

The ternary operator is the only three-part operator in C/C++. Its three parts are separated by two symbols, the ? (question mark) and the : (colon). Its purpose is identical to that of the =IF() function in spreadsheet tools such as MS Excel (see Figure 2.1). The first expression, to the left of the ?, is a test. The remaining two expressions, both to the right of the ?, are separated by a : symbol. If the test is true, the value to the left of the colon is returned. If the test is false, the value to the right of the colon is returned. Some samples of the conditional operator in action are provided below:

```
int x1=3,x2=3,x3,x4;
x3= (x1>4) ? 17 : 32;
// x3 is now 32
x4 = (x1<4) ? x2+4 : -1;
// x4 is now 7
```

Structurally, the conditional operator is applied as follows:

test **?** *expression1* **:** *expression2*

Functionally, the operator can be represented as follows:

promoted-type operator?:(*test-type* test,*type1* expr1,*type2* expr2);

The return type of the operator is determined according to the rules of type promotion, as described in Section 4.6.1. What we are really interested in, however, is the issue of what constitutes a *test-type*.

Boolean Data

Data that can have two values—interpreted as True/False, Yes/No, On/Off, etc.—is frequently referred to as *Boolean* data. Conceptually, such data is different from all the other data types. For example, there is no logical reason to assume an expression such as:

```
x1 > 3
```

is equivalent to an integer, a real number or a pointer. It's just a test whose result can either be true or false.

Many languages, such as Java and C#, offer a true Boolean data type—distinct from all other data types. The potential benefits of a pure Boolean type were recognized in the design of C++—but so was the need to maintain compatibility with C (which used integers for Boolean expressions). As a consequence, although the data type **bool** (whose legal values were **true** and **false**) is available in C++, it is not a true Boolean type because it can participate in integer promotion. In other words, if a **bool** data element is used in an integer expression, it would be promoted to integer—**false** being treated as 0, **true** being treated as 1. This is illustrated by the following code:

```
int x1=3,x2,x3;
bool b1,b2,b3,b4;
b1=(x1>3);
// b1 is false
b2=(x1<4);
// b2 is true
x2=b1+b2;
// x1 becomes 1--integer conversion used
x3=true+true;
// x3 becomes 2
b3=b1+b2;
// b3 becomes true, but warning is issued...
b4=4*3;
// b4 becomes true, but warning is issued...
```

At first glance, the use of **bool** doesn't seem to offer us a lot of benefits. For example, we can still add two bools together and not get a warning, as in the lines:

```
x2=b1+b2;
// x1 becomes 1--integer conversion used
x3=true+true;
// x3 becomes 2
```

There are some advantages of using **bool** in place of **int**, however. To being with, we can be confident that any expression that returns **bool** will either return 0 (**false**) or 1 (**true**). Second, although the compiler will allow us to promote **bool** values to integers (as above), it will warn us if we try to place integer values into bool variables. For example, consider the two lines below:

```
b3=b1+b2;
// b3 becomes true, but warning is issued...
b4=4*3;
// b4 becomes true, but warning is issued...
```

Since the RHS in both assignments are integer types, warnings are issued for both lines. Furthermore, once the assignment is made, we can be assured that the values in b3 and b4 will either be 0 or 1 (i.e., **true** or **false**).

Relational Operators

Walkthrough available in RelationalOps.avi

C/C++ provides the full complement of relational operators for testing equality (==), inequality (!=) and relative values (<, >, <=, >=). Some examples of their use (and misuse) are presented below:

```
int x1=3,x2,x3;
unsigned int u1=5;
double d1,d2;
bool b1,b2;
x2=(x1>3) ? 1 : 2;
// x2 now is 2
x3=(x1<u1) ? 10 : 20;
// x3 is 10, but warning is produced
d1=(5*4.5*7.33)/12.47;
//d1 is 13.225741780272655
d2=5*4.5*(7.33/12.47);
// d2 is 13.225741780272653
b1=(d1==d2) ? true : false;
// b1 is false
b2=(d1=d2) ? true : false;
// b2 is true
```

Structurally, the relational operators all fall into the form:

expression1 **op** *expression2*

Their functional form, shown for < operator, is as follows:

bool operator<(*type1 expression1*,*type2 expression2*);

If *type1* and *type2* are not the same, the less capable type is promoted to the more capable type before the comparison is performed. The return type is **bool** in C++ (0 for false and 1 for true in C).

Although the relational operators are usually fairly straightforward, there are three common issues associated with their use:

- Comparing signed and unsigned integers
- Problems in equality testing of floating point numbers
- Accidentally confusing equality testing and the assignment operator

Signed/Unsigned Mismatch
A common, but often not serious, error can be encountered using the relational operators when signed and unsigned numbers are tested, as illustrated by the following code:

```
int x1=-1;
unsigned int u1=0xffffffff;
bool b1;
b1=(x1==u1);
// b1 is true
```

While technically true (-1 does equal 0xffffffff in twos complement), the numbers aren't really the same. To alert the programmer such potential problems, the compiler always issues a warning when signed and unsigned values are compared. Often, these warnings can be removed with a type cast (see Section 4.7), once the programmer is 100% confident that the signed/unsigned difference is not significant.

Rounding Errors and Equality Testing
Any real number that results from an arithmetic expression is potentially subject to rounding error. In our sample code, this phenomena is illustrated by the lines:

```
d1=(5*4.5*7.33)/12.47;
//d1 is 13.225741780272655
d2=5*4.5*(7.33/12.47);
// d2 is 13.225741780272653
b1=(d1==d2) ? true : false;
// b1 is false
```

Algebraically, the expressions (5*4.5*7.33)/12.47 and 5*4.5*(7.33/12.47) are identical. In the computer, however, different orders of execution lead to different rounding errors. As a result, they are only equivalent to 16 digits—the seventeenth digit is different (see comments, which hold values observed in the debugger).

The practical impact of this rounding effect is that equality and inequality testing for real numbers should always involve a range (e.g., plus or minus some small percentage), rather than a test for precise equality. This is one of the main reasons that integers are so much preferred to real numbers whenever the choice is available.

Confusing Equality Testing and Assignment

Accidentally using an assignment operator in place of an equality test holds a place near the very top of any "Top Ten Ways That C++ Says Gotcha!" list. Three factors conspire to make this error so inevitable:

- In natural language, we tend to use the term *equals* to mean both "set equal to" and "is equal to", in spite of the fact that these are very different concepts.
- C/C++ uses very similar symbols for assignment and equality testing (single equal sign vs. double equal sign).
- Because the C/C++ assignment operator returns the value being assigned, and because any numeric value can be interpreted as a test result, assignments where tests were meant to be are often not detected by the compiler.

An example of the type of code that causes the problem was presented in our sample:

```
b1=(d1==d2) ? true : false;
// b1 is false
b2=(d1=d2) ? true : false;
// b2 is true
```

In this code, the result of the first test tells us that d1 doesn't equal d2. Yet the result of the second test suggests that it does. The problem, of course, is that the second "test" wasn't a test at all. It was an assignment. But, since the assignment returned a non-zero value, that fact was interpreted as a true value.

Unfortunately, decades of programming experience suggest that although the frequency of this error can be reduced, it can never be eliminated. Given that prevention is impossible, fast detection is the only cure.

Logical Operators

Walkthrough available in AndOrOps.avi

The logical operators are provided by C/C++ to work with Boolean values, such as those returned by the relational operators. A unary operator (! or **not**) reverses **true** and **false**. Two binary operators, **and** (&&) and **or** (||), combine logical conditions.

not Operator
The logical not (!) operator is a unary operator applied to a Boolean expression that reverses the expression value: **true** expressions become **false**, **false** expressions become **true**. In C or C++, the logical not operator is performed using the ! (exclamation point) symbol In C++, the **not** keyword may also be used instead (provided the file *iso646.h* is included, in Visual Studio .Net). Some examples of the logical not operator in action follow:

```
int x1=3;
bool b1,b2;
b1= ! (x1>3);
b2= not (x1>3);
// b1 and b2 both true, since statements are equivalent
```

Structurally, the operator is applied before the test expression it is to negate:

! *test-expression*

Functionally, it can be presented as follows:

bool operator!(**bool** *test*);

As noted in Section 4.7.1, virtually any numeric (or pointer) expression can be converted to a **bool** value in C/C++. Therefore, the practical net effect of the operator is to turn any non-zero value to zero, and any zero value to 1.

In general, it is good programming practice to avoid the logical not operator. While there are times when its use is appropriate (e.g., when you are testing using a function that returns a Boolean value opposite of what you are interested in), most of the time logical nots can be avoided by changing the condition being tested. For example !(x1>3) is the same as writing (x1<=3)—and the latter is much clearer.

Logical AND and OR
Logical **and** (&&) and **or** (||) operators are used to combine the results of two Boolean expressions into a single Boolean expression. The logical **and** takes the two expressions and returns **true** only if both are true. If either is **false**, the return value is **false**. In contrast, logical **or** returns **false** only if the two expressions are both **false**, otherwise it returns **true.** As was the case with **not**, keyword versions **and** and **or** may substitute for their symbol equivalents, && and ||, in C++ (provided the file *iso646.h* is included, in Visual Studio .Net).

Some examples of logical operators in action are presented in Example 2.1. Structurally, the operators are invoked like standard binary operators:

test1 **&&** *test2*
test1 **and** *test2*   // C++ only

test1 || *test2*
test1 **or** *test2*   // C++ only

The operators function equivalents, shown only for &&, can be prototyped as follows:

bool operator&&(**bool** *test1*,**bool** *test2*);

As was the case for **not,** any numeric value can be converted to **bool**, so the operator can be applied to almost any pair of numeric or pointer values—although the propriety of doing so is questionable.

Short-Circuiting
The **and** and **or** operators exhibit a behavior sometimes referred to as short-circuiting. To understand this behavior, you must first realize that it is fairly common programming practice to include expressions (e.g., function calls) on one or both sides of a logical operator. As we have discussed previously, the normal practice for a function is to evaluation all its arguments before entering the function body. **and** and **or** operators in C/C++ are different, however. They only evaluate enough arguments to determine whether their answer is true or false. In some cases, that will mean that they only need to evaluate a single argument. For example:

- If the first (LHS) argument of an **and** operation is **false**, we know that the result will be **false**—no need to check the second (RHS) argument.
- If the first (LHS) argument of an **or** operation is **true,** we know that the result will be **true—**again, no need to check the second (RHS) argument.

C/C++ takes advantage of this knowledge, evaluating only those arguments that *need* to be evaluated. As a result, the && and || operators can be used to control program execution. This is demonstrated in the code, from Example 2.1, that follows:

```
// ShortCircuit demonstration
b4=(RetTrue1() and RetFalse3());
// Returns false, both called
b5=(RetFalse3() and RetTrue2());
// Returns false—RetTrue2() never called
b6=(RetFalse3() or RetTrue1());
// Returns true, both called
b7=(RetTrue1() or RetFalse4());
// Returns true, RetFalse4() never called
```

In the demonstration, the functions RetTrue1(), RetTrue2(), RetFalse3(), and RetFalse4() were defined to return either **true** or **false** (which is self-evident from the function name), and to increment a global variable: nSE1, nSE2, nSE3 or nSE4 (which variable, once again, is self-evident from the function name), used to keep track of the calls in the debugger.

Because of short-circuiting, in the assignments of b5 and b7, only the first of the two functions were called. In the case of b5, RetFalse3() returned a value of **false**. Since the operator was **and**, that meant the return value was going to be **false**. Similarly, in the case of b7, RetTrue1() returned **true**—meaning that there was no need to call RetFalse4() since the operator was **or**.

Example 2.1: And/Or Operator Demonstration Code

```
// AndOrTest.cpp
#include "AndOrTest.h"
#include <ISO646.H>

int nSE1=0,nSE2=0,nSE3=0,nSE4=0;

void AndOrTest()
{
	bool b1,b2,b3,b4,b5,b6,b7;
	double d1,d2;
	double delta;
	d1=(5*4.5*7.33)/12.47;
	d2=5*4.5*(7.33/12.47);
	delta=d1*1E-15;
	// bring delta to 15 digits
	b1=(d1==d2);
	// b1 is false
	b2=(d1>d2-delta && d1<d2+delta);
	// equality test (to 15 places): true
	b3=(d1<d2-delta || d1>d2+delta);
	// Inequality test (to 15 places): false
	// ShortCircuit demonstration
	b4=(RetTrue1() and RetFalse3());
	// Returns false, both called
	b5=(RetFalse3() and RetTrue2());
	// Returns false--RetTrue2() never called
	b6=(RetFalse3() or RetTrue1());
	// Returns true, both called
	b7=(RetTrue1() or RetFalse4());
	// Returns true, RetFalse4() never called
	return;
}
bool RetTrue1() {
	nSE1++;
	return true;
}
bool RetTrue2() {
	nSE2++;
	return true;
}
bool RetFalse3() {
	nSE3++;
	return false;
}
bool RetFalse4() {
	nSE4++;
	return false;
}
```

Not all languages short-circuit logical testing in this fashion and some, such as Java, offer short-circuiting and non-short-circuiting operators. Thus, it is an important capability to be aware of, whether or not you decide to use it.

> *Test your understanding 4.11:* Assuming the code in Example 2.1, what will the values in nSE1, nSE2, nSE3 and nSE4 be when the program reaches the return statement in AndOrTest()?

Bitwise Operators

Walkthrough available in Bitwise.avi

Nothing underscores C++'s ancestry from a mid-level language designed for building operating systems more than the full complement of operators it provides for manipulating the individual bits of integers. Such capabilities were invaluable in the early days of programming, where they could be used for tasks such as performing multiplication and division in software, performing data translations and drastically reducing the space requirements of some types of data.

The capabilities of modern hardware and software have reduced the need for bit manipulations. Unfortunately, the need has not been entirely eliminated. In particular, any programmer planning to work in MS Windows needs to understand the use of these operators to pack and unpack data into integers that are then used as function arguments.

A second reason for being familiar with some of the bitwise operators—most notably bitwise-or (|) and bitwise-and (&)—is their unfortunate similarity to their logical operator cousins (|| and &&). Similar to the problem with the equality testing (==) and assignment(=), failing to "double up" on these operators when a logical test is desired, can lead to unexpected results.

The bitwise operators are always applied to integer arguments. To actually predict what will happen when the bitwise operators are applied to their integer arguments, the best procedure is to:

- Convert the argument or arguments to hexadecimal bytes (using twos complement, if necessary).
- Convert the hex bytes to binary digits
- Apply the operator to the binary digits individually
- Convert the result back to hex
- Convert the result back to decimal

Normally, performing this procedure once or twice will satisfy a lifetime's worth of curiosity.

Bitwise Negation
The bitwise negation operator ~ (tilde) is a unary operator that takes every bit in its integer argument and reverses it, returning the resulting integer. In the questions in Chapter 3, we described such a reversal of bits as the "ones complement" of a number, further noting that the twos complement of a number could then be achieved by adding one to the ones-complement value. This is demonstrated in the code that follows:

```
int val1=173,val2;
val2=~val1;
val2++;
// val2 is now –173
```

Structurally, the operator is applied as follows:

> *~ integer-expression*

The functional prototype of the operator is roughly as follows:

> *integer-type* operator~(*integer-type expr*)

As an example, suppose we wished to find the value of ~43. Our procedure would be as follows:

- Convert 43 to hex: 0x2B
- Convert 0x2B to bits: 0010 1011
- Reverse the bits: 1101 0100
- Convert back to hex: 0xD4
- Convert back to decimal:
  - 212 (unsigned)
  - –44 (signed)

Binary Bit Operators
C/C++ offers three binary operators used to manipulate bits: bitwise-or (|), bitwise-and (&) and bitwise-exclusive-or (^). Each operator takes two integer arguments and, bit-by-bit, returns a value computed from the original bits. The value is determined as follows:

- *Bitwise-or:* If either integer bit is on, the resulting bit is 1. Otherwise, the resulting bit is 0.
- *Bitwise-and*: If both integer bits are on, the resulting bit is 1. Otherwise, the resulting bit is 0.

- *Bitwise-exclusive-or*: If one, but not both, of the integer bits are on, the resulting bit is 1. If both or neither integer bit is on, the result is 0.

An example, illustrating the 43 & 31 computation for a 1 byte integer, is presented in Figure 2.2.

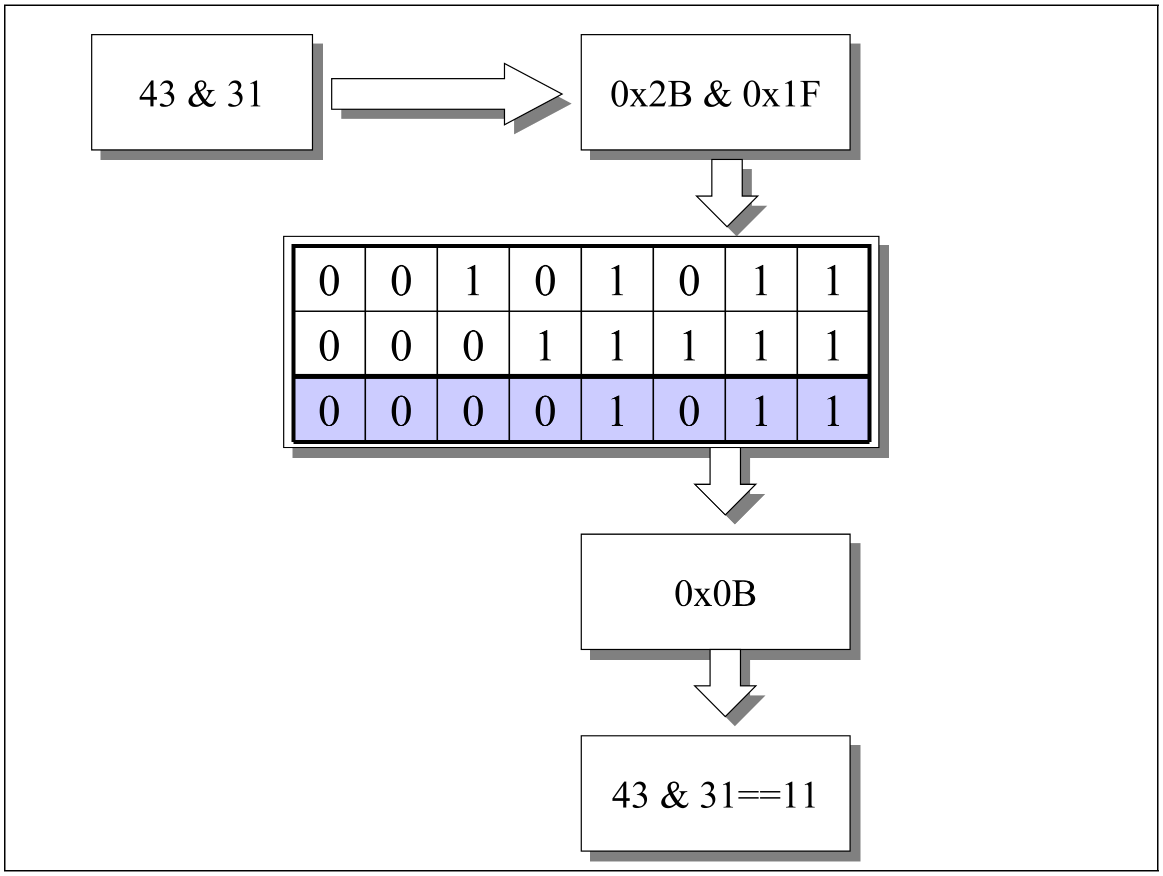


Figure 2.2: Illustration of bitwise-and computation

Test your understanding 4.12: Compute the results for the 43 | 31 and 43 ^ 31 operations.

The structure of the bitwise operators is:

integer-expr2 **op** *integer-expr2*

Functionally, the bitwise operators can be prototypes as follows (shown for | case only):

promoted-integer-type operator|(*integer-type1* expr1,*integer-type2* expr2);

The operators all apply the same basic rules for promotion that we have already discussed.

Example 2.2: Using Bitwise | and & to set and display bits

```
// BitOps.cpp: Using | and & operators to set and extract bits
#include "SimpleIO.h"

/* SetBits(): uses bitwise-or to set bit values in an unsigned char.
   returns the resulting unsigned char */
unsigned char SetBits(bool b1,bool b2,bool b3,bool b4,bool b5,
            bool b6,bool b7,bool b8)
{
      unsigned char nRet=0;
      nRet=nRet | ((b1) ? 1 : 0);
      nRet=nRet | ((b2) ? 2 : 0);
      nRet=nRet | ((b3) ? 4 : 0);
      nRet=nRet | ((b4) ? 8 : 0);
      nRet=nRet | ((b5) ? 16 : 0);
      nRet=nRet | ((b6) ? 32 : 0);
      nRet=nRet | ((b7) ? 64 : 0);
      nRet=nRet | ((b8) ? 128 : 0);
      return nRet;
}

/* DisplayBits(): displays the individual bit values in an unsigned
char */
void DisplayBits(unsigned char nVal)
{
      PrintString("Bits in ");
      PrintInteger(nVal);
      NewLine();
      PrintString((nVal & 1) ? "1. On\n" : "1. Off\n");
      PrintString((nVal & 2) ? "2. On\n" : "2. Off\n");
      PrintString((nVal & 4) ? "3. On\n" : "3. Off\n");
      PrintString((nVal & 8) ? "4. On\n" : "4. Off\n");
      PrintString((nVal & 16) ? "5. On\n" : "5. Off\n");
      PrintString((nVal & 32) ? "6. On\n" : "6. Off\n");
      PrintString((nVal & 64) ? "7. On\n" : "7. Off\n");
      PrintString((nVal & 128) ? "8. On\n" : "8. Off\n");
      NewLine();
}
```

The most common use for bitwise operations is to pack a series of true-false values into a single integer. While this may seem unnecessary, given today's RAM capacities, this capability is actually used quite a bit in functions, particularly in MS-Windows and to set flags in various C++ standard I/O objects. The problem this solves is one of too many arguments. A typical window or font might have literally dozens of settings. It is, however, cumbersome to define functions with so many arguments. You can begin to sense of the magnitude of the problem if you look at the function SetBits(), in Example 2.2, which has eight arguments (designed to represent each of the bits in an unsigned character). Imagine, now, a function with 30 arguments…

Using individual bits, 32 yes/no questions can be packed into a single int value. Thus, a function could have a single integer argument that contains the data that otherwise require 32 bool arguments. The trick then becomes how to set the bits, and how to retrieve them.

In SetBits(), we show how individual bits can be set using the | operator. We start with an integer value of 0, then we check each argument. Using the conditional operator, we bitwise-or the integer with the appropriate power of 2 (even powers of 2 always represent a single bit in binary). e.g.,

```
nRet=nRet | ((b5) ? 16 : 0);
```

The | leaves all bits already set in nRet untouched. Should the appropriate argument (e.g., b5) be true, the conditional operator cause the appropriate value (e.g., 16) to be turned on, in addition. If the argument is false, however, the conditional operator causes us to bitwise-or nRet with 0, which leaves it unchanged.

Just as | can be used to set bits, the bitwise-and (&) is used to see if they are set. In the function DisplayBits(), we & the argument (containing the bits that have been set) with an even power of 2. Since the even power of 2 only has one bit set, that becomes a test for that particular bit. If it is off in the argument, the result is 0 (a.k.a. false). If the bit in the argument is set. We then use the condition operator to determine what gets printed. For example:

```
PrintString((nVal & 16) ? "5. On\n" : "5. Off\n");
```

In this case, if the bit 00010000 (binary 16) is on in nVal, the conditional operator causes "5. On\n" to be sent to PrintString(). If the bit is not set in nVal, the string "5. Off\n" goes to PrintString().

Confusing Logical and Bitwise Operators

One of the biggest reasons to understand bitwise operators is to understand what happens when you accidentally confuse them with the logical operators && and ||. If you happen to be dealing with pure Boolean values, represented as 0 and 1, their behavior will be very similar (since you are "and"ing and "or"ing only the first bit). The one difference will be that bitwise & and | do not short-circuit. They can't—since doing a bitwise operation requires you to know the values of the integers on both sides of the operator.

A worse situation occurs when & is confused with &&, and we are not limited to 0 and 1 for our true/false values. Using the rules of C/C++, the value:

```
1 && 2
```

is **true**, since both sides are non-zero. One the other hand, the expression:

```
1 & 2
```

is 0 (or **false**), since the bits that are set on both sides miss each other, when they are "and"ed together. Thus, as was the case with the assignment and equality test operators, not keeping the two operators straight can lead to hours of debugging pleasure.

2.1.6: Precedence Ordering of Operator Categories

In introducing operator precedence, we proposed that it was a very bad idea to rely on precedence (and, especially, your memory of operator precedence) in writing expressions. Parentheses will make it easier for you, and for anyone who later reads your code, to understand what is going on.

Having made this disclaimer, it is usually reasonably safe to utilize precedence across major classes of operators. For the operators we have examined so far, these are as follows:

- Unary arithmetic sign operators (i.e., + and -) and increment/decrement operators (++, --) get applied before arithmetic operators
- Arithmetic operators (e.g., +,-,*,/) have a high precedence, with * and / being higher than + and -.
- Relational operators (e.g., <, >, ==) are the next level of precedence.
- Logical operators (e.g., && and ||) get done next . Since **and** and **or** have different precedence levels, use parentheses when forming complex logical expressions.
- Assignment operators (e.g., =, *=, +=) get done last.

You might also keep in mind that postfix and prefix versions of operators have precedences that are different.(e.g., the ++ operator in ++X and X++ are, effectively, very different operators)—which means that you should probably use parentheses when applying them. Or, better yet, apply them on separate lines.

2.1.7: Type Casting

In presenting the various arithmetic and relational operators, we already identified a number of situations where the default behavior of C/C++ was not precisely what we wanted. In some cases, the problem was that an arithmetic expression might not return the proper value. For example:

```
int n1=7,n2=21;
double r1=3.00,r2,r3;
r2=n1*r1/n2;
// r2 is 1.000
r3=n1/n2*r1;
// r3 is 0.000
```

The problem with the above code is that r2 and r3 should be equal, according to the normal rules of algebra, but r3 ends up being zero because the integer division (n1/n2) produces a 0 value. Another problem we noticed was compiler warnings that, might, in some cases, tell us about problems that do not exist. For example:

```
int i1=5,i2;
unsigned int u1=7;
double d1=12.00;
bool b1;
i2=d1;
// produces warning: assigning double to int
b1=(u1<i1);
// produces signed/unsigned mismatch warning
```

As noted in the comments, the above code fragment creates two warnings—both justified. The first warning occurs for the statement i2=d1. Since the range of a **double** far exceeds the range of an **int**, the compiler alerts us to the potential problem. The second warning occurs because we test (u1<i1)—a signed integer compared with an unsigned integer. We have already discussed the potential problems of mixing the two types of integers. Having applauded the compiler for doing its job, the fact remains that—in the code above—neither of these operations will lead to any actual problems. After all, we've made sure our values are within acceptable ranges. So the question becomes: how do we get the compiler to shut up?

Type Casting
C++ provides a mechanism, called *type casting*, that allows us to deal with many conversion issues and compiler warnings. The type cast operator—and yes, it is an operator—can be used to create a temporary value of a given type of data, or to force the compiler to change its interpretation of a given data type (used when pointers are type cast, discussed in Chapter 11). The most commonly used form of the operator is constructed by placing a type name in parentheses in front of the expression whose type is to be changed. Normally, what this will do is to force C++ to create a temporary variable of the desired type, initializing it using the value from the expression being cast. Some examples of type casts in action are presented below:

```
int n1=7,n2=21;
double r1=3.00,r2,r3,r4;
int i1=5,i2,i3;
unsigned int u1=7;
double d1=12.000;
bool b1,b2;
/***** Conversion problems *****/
r2=n1*r1/n2;
// r2 is 1.000
r3=n1/n2*r1;
// r3 is 0.000
r4=((double)n1/(double)n2)*r1;
//r4 is 1.000
/***** Compiler warnings *****/
i2=d1;
// produces warning
i3=(int)d1;
// no warning
b1=(u1<i1);
// produces warning
b2=((int)u1<i1);
// no warning;
```

Structurally, the type cast operator is applied as follows:

(*data-type*) *expression*

The functional form of the operator can be prototypes roughly as follows:

target-type operator *target-type*(*old-type expr1*);

Naturally, for a type cast to be possible, the *target-type* and *old-type* must be compatible. This means you can move between numeric types, and you can move between pointer types, but you *really* don't want to move between pointers and numeric types. (Don't want to and can't are not the same, however. In the early days of C, type casting between integers and addresses, in both directions, was fairly common. It is, however, just about the most hardware dependent code you could possibly write).

Forcing Proper Operations

The sample code illustrates ways in which sensible type casting can eliminate some of the problems we ran into. For example, in the line:

```
r4=((double)n1/(double)n2)*r1;
```

we type cast both n1 and n2 to **double** temporary variables—naturally, n1 and n2 themselves are not changed. Doing so forces the division to use floating-point arithmetic (7.00/21.000==0.33333), leading to the result of 1.000. That particular expression also illustrates the use of parentheses to ensure operator application in the proper order. Actually, the same expression could have been written:

```
r4=(double)n1/n2*r1;
```

since the type cast operator has higher precedence than division (and since automatic type promotion will cause the division to be done in floating point if either argument is **double**). Writing the expression in the more lengthy fashion, however, avoids any chance that the expression would be interpreted as (double)(n1/n2)*r1—which would give us our original problem of returning 0 since (n1/n2) would be done before the conversion to double.

Removing Compiler Warnings
The sample code also illustrates how type casts can be used to remove compiler warnings (that we have determined to be okay!). There were two examples of this:

```
i3=(int)d1;
b2=((int)u1<i1);
```

In the first line, the (int) type cast tells the compiler we are aware of the conversion that will be taking place between **double** and **int**. As a result, it does not need to warn us. In the second line, the (int) typecast tells the compiler to interpret u1 as signed for the purposes of the comparison. Once again, this eliminates the need for a warning.

2.1.8: Operator Overloads in C++

Operator overloading in C++ is a powerful capability that allows programmers to extend the capability of the C++ language, and write more natural code. In our first C++ program (Chapter 2, Section 2.3.1), we saw an example of such an overload in the line:

```
cout << "Hello, World "  << endl;
```

Although we referred to the << operator as the insertion or output operator, it is actually the little-used bitwise left shift operator in C (used to move the actual bits in an integer a specified number of positions to the left). In C++, however, the operator has been overloaded to allow it to serve the more common purpose of sending data to an output stream. Some of the common overloads in the iostream library might be prototyped as follows:

```
ostream &operator<<(ostream &out,int nVal);
ostream &operator<<(ostream &out,const char *str);
ostream &operator<<(ostream &out,double dVal);
```

etc.

The effect of an operator overload, like that of a function overload, is to cause C++ to call programmer-defined code where the operator is encountered (with matching arguments). cout happens to be an object of the *ostream* data type. As a result, any time the compiler sees:

cout << *expression*

it looks for an overloaded << operator where the first argument matches cout in type (ostream) and the *expression type* matches the second argument of one of the available overloads.

> *Test your understanding 4.13:* In the code *cout << "Hello, World*", which of the overload versions (presented above as prototypes) would be chosen?

Although operator overloading is a very powerful capability, its use is mainly limited to OOP situations. To avoid the creating of massive incompatibilities, C++ does not allow already defined operators on the basic data types to be overloaded. Thus, we can't overload the == operator to test the equivalence of two character strings, no matter how much we'd like to. For this reason, we do not return to operator overloading techniques until after we have introduced objects (in Chapter 15), where we consider their implications in much greater depth.

2.1 Section Questions

1. If it is possible to replace any operator with a function, why do we bother with operators at all?

2. Explain what we mean when we say the + operator is implicitly overloaded.

3. Why is it useful to think about operators in function form?

4. If a functional representation of an operator includes a reference as an argument, what does that imply about the operator?

5. What do we mean when we say the assignment operator returns its RHS value?

6. What is an *lvalue*?

7. Why doesn't it make sense to define a modulus (%) operator for real numbers?

8. What is type promotion, and why is it important to understand when applying operators?

9. What is the difference between ++ postfix and ++ prefix operators? It is a good idea to exploit this difference in your programs?

10. What's the difference between a true Boolean type and an integer Boolean type? Why does C++ have the latter?

11. What is the difference between relational and logical operators?

12. What is logical operator short-circuiting? Why can it have a major impact on the way a program runs?

13. What is the difference between the | and || operators?

14. What are bitwise operators most commonly used for?

15. Explain how type casting can be used to eliminate problems in that occur when integers and real numbers are mixed in expressions.

2.2 The if…else Construct

The if…else construct is used to implement two-way branching in C/C++, and is probably the most commonly used construct in a typical program. The basic format of the simple if statement is:

> **if (***test***)** *statement-or-codeblock*
> **else** *statement-or-codeblock*

2.2.1: The basic if construct

The basic if construct allows for a conditional test with or without an else block. To illustrate this, An example of a simple function that can be implemented using an if block is the ToUpper function, which takes its argument (a character) and—if the character is a lower case character—it returns its upper case equivalent (mimicking the C/C++ library toupper function). If the character is not an upper case character, it is returned unchanged. The function is prototyped as follows:

```
char ToUpper(char val);
```

The argument val is the character to be transformed into upper case, and the return value is the upper case equivalent of the character (or the character itself). The function is presented in Example 2.3.

Example 2.3: if construct without and with else block

```
char ToUpper(char val)
{
	if(val>='a'&&val<='z') {
		val=val-'a'+'A';
	}
	return val;
}

char ToUpper1(char val)
{
	char cRet;
	if(val>='a'&&val<='z') {
		cRet=val-'a'+'A';
	}
	else cRet=val;
	return cRet;
}
```

The case change in the ToUpper function is accomplished as follows:

1. It relies on the fact that the ASCII code first codes all the upper case characters together, in sequence, starting at 65, then codes all the lower case characters together, in sequence, starting at 97.
2. The expression val-'a'+'A' uses this fact to accomplish a case change. For example, if the character were 'd', 'd'-'a' is 3 (100-97, using their ASCII codes). Similarly, because the letters are coded in sequence, 'A'+3 is equivalent to 'D' (65, the ASCII code for 'A', +3 is 68, which is the ASCII code for 'D').

In the second version of the function, ToUpper1(), a temporary variable cRet is used, and an else block serves to initialize it to val if no case change takes place.

2.2.2: The if…else if…else Construct

If statements can also be used to implement a multi-way branch, such as:

> **if** (test0) statement-or-codeblock0
> **else if** (test1) statement-or-codeblock1
> **else if** (test2) statement-or-codeblock2
> // …
> **else if** (testN) statement-or-codeblockN
> **else** statement-or-codeblockN+1

This construct performs a series of tests and performs the codeblock associated with the first test to be true, and then skips the remaining tests.

Although the **else if** appears as if might be a separate construct, actually it is not—since the if…else construct itself is a form of statement. For example, the code:

> **if** (test0) statement-or-codeblock0
> **else if** (test1) statement-or-codeblock1
> **else if** (test2) statement-or-codeblock2
> **else** statement-or-codeblock3

is effectively the same as:

```
if (test0) {
        statement-or-codeblock0
}
else {
        if (test1) {
                statement-or-codeblock1
        }
        else {
                if (test2) {
                        statement-or-codeblock2
                }
                else {
                        statement-or-codeblock3
                }
        }
}
```

which is just a standard nesting of codeblocks.

An illustration of this, performing temperature conversion with error checking (since a temperature lower than –273 degrees Celsius is not possible) is presented in Example 2.4. The function takes two arguments, dTemp holding the original temperature and a flag variable nType, (where nType==0 signals Fahrenheit to Celsius, and nType==1 signals Celsius to Fahrenheit). The function returns the converted temperature, or –1000 if an illegal original temperature was passed in.

Example 2.4: Temperature Conversion Function

```
double ConvertTemp(double dTemp,int nType)
{
      if(nType==0) {
            dRet=(dTemp-32)*5/9;
            if(dRet<-273) dRet=-1000;
      }
      else if(dTemp<-1000)  dRet=-1000;
      else dRet=32+dTemp*9/5;
      return dRet;
}
```

2.2 Section Questions

1. What happens if three of the conditions in a long if…else if…else if…etc. construct happen to be true?

2. How do you construct an if. . . else if. . . else if. . . construct so that code blocks associated with all true tests get performed?

3. What constitutes a legal "test expression" in an if construct?

2.3: The switch…case Construct

The true multiway branch in C/C++ is implemented with the switch…case construct. The construct is defined as follows:

switch (*integer-value*)
{
    case *integer-constant1*:
        statement-or-statements-or-empty1
    case *integer-constant2*:
        statement-or-statements-or-empty2
    // etc.
    case *integer-constantN*:
        statement-or-statements-or-emptyN
    default:
        statement-or-statements-or-emptyN+1
}

Some comments:

- The integer-value can be any expression that evaluates to an integer, such as a variable, arithmetic expression or function call that returns an integer.
- The integer-constant associated with each case must be a value the compiler can determine, not an expression. Examples include:
    - numbers, such as 17
    - quoted constants, such as 'a' (same as writing 97) or '\t' (same as writing 9),
    - hexadecimal numbers, such as 0x20 (same as writing 32)
    - names created using a #define statement
    - enumerated types (discussed in Chapter 3, Section 3.4.2)
- The *statement-or-statements-or-empty* can either be:
    - individual or multiple semicolon terminated statements
    - a code block

    - nothing at all—which has the effect of allowing multiple case labels to apply to the same collection of statements
- The *default* label is optional. If present, any switch value encountered that does not match any of the specific cases will go to the default block.

Unless a break statement (or return statement, or other statement that causes a code jump) is encountered, cases will fall through. In the above case outline, for example, the case that matches integer-constant1 would do statement-or-statements-or-empty1, then statement-or-statements-or-empty2, then statement-or-statements-or-empty3, all the way through to statement-or-statements-or-emptyN+1.

2.3.1: The Structured Case Construct in C/C++

The C/C++ switch…case construct, used to its fullest level of flexibility, can produce pandemonium in code rivaling use of the vaunted goto statement. For this reason, a more structured approach to writing case statements is probably safer, and—in this book—we will always create case constructs according to the following framework:

```
switch (integer-value)
{
        case integer-constant1:
        {
                statement-or-statements-or-empty1
                break;
        }
        // for multiple cases doing the same thing
        case integer-constant2a:
        case integer-constant2b:
        case integer-constant2c:
        {
                statement-or-statements-or-empty2
                break;
        }
        // etc.
        case integer-constantN:
        {
                statement-or-statements-or-emptyN
                break;
        }
        default:
        {
                statement-or-statements-or-emptyN+1
                break;
        }
}
```

In this structure, the break statements prevent case fall through, and the explicit brace-delimited blocks after each case label help to reinforce the notion that each branch is distinct.

The construction of such a structured case construct is illustrated in Example 2.5, which illustrates a function that could be used to convert from dollars to other currencies (with the specific target currencies identified by defined constants.

Example 2.5: Currency Conversion

```
#define Pound 1
#define Yen 2
#define Euro 3

//Initialized elswhere
double DollarToPound;
double DollarToYen;
double DollarToEuro;

double Convert(double dVal,int nTarget)
{
      double dRet;
      switch()
      {
            case Pound:
            {
                  dRet=dVal*DollarToPound;
                  break;
            }
            case Yen:
            {
                  dRet=dVal*DollarToYen;
                  break;
            }
            case Euro:
            {
                  dRet=dVal*DollarToEuro;
                  break;
            }
            default:
            {
                  dRet=-1;
            }
      }
      return;
}
```

2.3.2: Problems with break

The flexibility provided by the C/C++ case construct implementation can lead to a number of common programming errors. Among the most common of these:

- Forgetting to put a break at the end of a structured case, causing unexpected case fall through.
- Using a break deeply nested within a case that is supposed to end a case and discovering it ends a loop nested within the case (or vice-versa).

The second of these problems can be avoided by limiting the use of breaks—in all but the most extreme circumstances—to case constructs. The former is pretty much unavoidable, even among experienced programmers. The three most common reactions to discovering you have just lost three hours of your life—hours you will never see again—as a result of a forgotten break are: 1) laughing at yourself, because you've managed to nail yourself yet another time, 2) directing loud profanity at the case construct, the C/C++ programming language, along with anyone who had a hand in developing the language, and 3) perpetrating actual physical violence on your hardware. Of these responses, the third is not recommended.

2.3 Section Questions

1. What is the only truly effective way to avoid leaving out break statements in C case constructs?

2. Can any expression that evaluates to an integer be used after **case** labels in a case statement?

3. What is the relationship of an enumeration to a case construct?

4. Is there anywhere else in C that labels, like the **case** label, are used?

2.4: Loops

Two versions of the pure while loop are implemented in C/C++: the while loop—that tests before looping—and the do…while loop—that tests after at least one pass has been performed. In addition, a for loop is provided that extends the while loop with built-in initialization and iterator blocks.

2.4.1: The While Loop

The while loop construct is both simple and powerful. In C/C++ its structure is:

while(*test*) *statement-or-codeblock*

The while loop tests first, then performs the statement or associated code block that follows. The process continues until the test is false, at which time the loop ends. In addition:

- A break statement can also be used to cause the loop to end.
- A continue statement causes the loop to immediate go to the test and, if the test is true, start the next pass.

As was noted in Chapter 5, Section 5.4.3, use of statements such as break and continue to control loop behavior violates a number of premises of structured programming, and is likely to be controversial in many organizations.

Example while Loop
A simple while loop, printing out the values, squares and cubes of 100 integers using a while loop, is presented in Example 2.6.

Example 2.6: Displaying Squares and Cubes with a While Loop

```
void Hundred()
{
      int x;
      x=1;
      while(x<=100) {
            cout << x << x*x << x*x*x << endl;
            x=x+1;
      }
      return;
}
```

Common Looping Errors
A couple of errors are particularly common when using loops in C/C++. These are:

- Prematurely terminating a loop with a semicolon, and
- Forgetting to increment a loop counter, leading to an infinite loop.

The first of these errors is similar to what has already been observed for the if construct. For example, the code:

```
      int x=1;
      while(x<10);
      {
            cout << x << endl;
            x++;
      }
```

would print a single integer—were it not for the fact that the statement:

```
       while(x<10);
```

is an infinite loop and so the code path will never get to the DisplayInteger() statement. The reason is that the semicolon after the while completely disconnects the while loop from the block underneath it.

The typical C/C++ programmer gets so sensitized to such errant semicolons that he or she avoids them even where a semicolon would be legal. Thus, the following block of code:

```
char str[80]="Hello, World!";
int i=0;
while(str[i++]!=0);
```

which counts the number of characters in "Hello, World!" (including the NUL terminator) is legal, but would normally be rewritten:

```
char str[80]="Hello, World!";
int i=0;
while(str[i++]!=0){}
```

with the empty braces after the while serving the sole purpose of reminding the programmer that this particular while loop does not need a body.

> *Test your knowledge:* Explain why the above block of code counts the characters in the string str (including the NUL terminator) and leaves their value in i. Would this be a recommended way of writing such a loop if clarity is desired?

The other common problem in loops is to forget to do something in the loop that moves it towards its conclusion, such as incrementing a counter. For example, the code:

```
int x=1;
while(x<10)
{
        DisplayInteger(x);
        NewLine();
}
```

which is similar to an earlier example, will keep printing 1's forever—since we haven't done anything to move X towards the >= 10 value that will cause the loop to end.

2.4.2: The do…while Loop

The C/C++ do…while construct takes the basic while construct and changes its position to the end of the loop. Its basic structure is:

> **do** *statement-or-codeblock*
> **while** (*test*);

Example do…while Loop

The while loop example of Example 2.6 is rewritten as a do…while loop in Example 2.7.

Example 2.7: Displaying Squares and Cubes with a do…while Loop

```
void HundredDo()
{
      int x=0;
      do {
            x=x+1;
            cout << x << x*x << x*x*x << endl;
      } while (x<=100);
      return;
}
```

Usage of do…while Loops
Although the do…while loop is relatively simple, the author has found there to be two specific reasons that it is used infrequently:

1. *Logical:* Most looping situations seem to benefit from having the test come first, rather than at the end of the first pass. Moreover, as you get used to writing loops that way, you tend to conceptualize your logic with the test first—even if the test coming afterwards is equally valid (as in the above example).
2. *Pragmatic:* Where code mixes while and do…while constructs, the fact that the while (test) components look the same, yet one almost never ends with a semicolon (while) and one must end with a semicolon (do…while) is a potential source of confusion. Indeed, many programmers (including the author) choose to place the while clause after the closing bracket of the do block (as shown in Example 2.7). This departure from normal indentation practices serves as a reminder that the while belongs to the end of a loop, and is not the start of a new loop.

2.4.3: The for Loop

The C/C++ for loop is basically a while loop with initialization and iteration blocks. Its basic structure is as follows:

for (*expression-list* ; *test* ; *expression-list*) *statement-or-codeblock*

The two *statement-list* items are what make the for-loop unique. They make use of the comma operator, which allows a series of expressions to be placed on a single line—separated by commas—each of which is evaluated in sequence. Such expressions can, in turn, include assignments, function calls and virtually anything that can be placed on the right hand side of an assignment statement (as discussed in Section 2.1.3). In C++, these lists can also include variable declarations, a subject to be discussed shortly. In other words, almost anything except program constructs and preprocessor instructions can be placed in expression lists.

Neither expression list need be present in a for loop. Thus, a for loop could be written in the form:

for(;*test*;) *statement-or-codeblock*

making it precisely the same as a while loop. Usually, if one or more of the two expression-list blocks is empty, it is preferable—stylistically—to use a while loop.

Example For Loop
Example 2.8 contains the Hundred example rewritten as a for loop.

Example 2.8: Displaying Squares and Cubes with a for Loop

```
void HundredFor()
{
      int x;
      for(x=1;x<=100;x=x+1) {
            cout << x << x*x << x*x*x << endl;
      }
      return;
}
```

break and continue Statements in for Loops
The break statement in the for loop works the same as it does in a while loop, causing the program to leave the loop. The presence of an iterator block, however, makes the behavior of the continue statement in for loops somewhat different:

- In the while loop, the continue statement leads directly to the test
- In the for loop, the continue statement leads to the iterator block (i.e., the X=X+1 in the Example 2.8 example) prior to the test.

Since structured programming style generally makes it preferable not to use either the break or continue statement regularly, this small difference between while and for loops is usually unimportant.

Declarations scope in for loop initializer blocks
Some care needs to be taken regarding declarations can be made when initializing a for loop—something that is commonly done. For example, the code:

for(**int** i=1;i<=10;i++) cout << i << endl;

The ability to define variables in for loops has led to widespread confusion with respect to variable scoping. There have always been two schools of thought regarding what the scope of a variable defined in a for loop initialization list should be:

- Conceptually, it would make sense to treat i as a variable whose existence was limited to the block within the for loop. That would mean its value could not be referenced after the loop was completed.
- Practically, it is often convenient to be able to look at the variable after the loop has ended. That would argue for a variable that continues to exist until the end of the codeblock containing the for loop. Under this scheme, code such as the following:

```
for(int i=1;i<=10;i++) cout << i << endl;
for(int i=11;i<=20;i++) cout << i << endl;
```

would lead to a compiler error in the second for loop—since i was declared above. (Under the first school of thought, it would have been fine since the scope of the first i ends with the loop).

So which approach was chosen? Both (unfortunately). In early versions of C++, the practical approach was preferred, and so variables remained in scope after the for loop. The C++ standard, however, deprecated that capability—and asserted that the conceptually cleaner limiting the local variable's scope to the for loop itself was the C++ standard (making it consistent with Java).

So which approach to for-loops is supported by Visual Studio? The answer is either, depending on what compiler switches you set. Such a situation is far from ideal for programmers, as it could require in numerous revisions to old code. For individuals just starting out learning programming, it is obviously safer to stick with the C++ standard.

2.4 Section Questions

1. Why might it be advantageous to write **while(test){}** instead of **while(test);** when writing an empty while-loop?

2. What is the one behavior in a for-loop that is hard to achieve in a while-loop?

3. Give some examples of situations where it might be acceptable (to Middle Ground and scruffier, at least) to use continue statements in a loop?

4. Is there any risk with break statements that are not present for continue statements?

5. What is the serious problem that can result from declaring variables in the initializer block of a for-loop?

2.5 Code Samples

In this section, we examine some commonly used C/C++ library functions and show how they could be implemented using the operators and constructs presented in this chapter.

2.5.1: Strlen

The strlen function, in C/C++, is used to determine how many characters are present in a NUL terminated string. The function we will develop mimics this function, and is prototyped as follows:

int Strlen(char str[]);

where the argument, str[] is an array containing a NUL terminated string, and the return value is the number of characters in the string.

The function is implemented in a very straightforward fashion, as illustrated in Example 2.9. We simply initialize a counter then test the array, character by character, until a NUL terminator (a byte with the value 0) is encountered. Then, we return the counter.

Example 2.9: Strlen implementation

```
int Strlen(char *str)
{
	int nCount;
	for(nCount=0;str[nCount]!=0;nCount++) {}
	return nCount;
}
```

2.5.2: Strcmp

In C++, we frequently find the need to check if two character strings are equal to each other, or to determine if one is alphabetically greater than the other. The equality test, for example, might be needed if we wanted to find out if a particular name was in a list of names. Similarly, a test for greater than/less than would be critical if we wanted to search a sorted array of names. Unfortunately, C++ does not allow us to use the standard operators (e.g., ==, !=, <, >) to compare NUL-terminated strings—the best we can do with them is to compare their first characters. As a result, we use a library function called strcmp() to make the comparison. Our function Strcmp performs the same task as the library function, and is prototyped as follows:

int Strcmp(char sz1[],char sz2[]);

where sz1 and sz2 are the two strings to be compared. The return value of Strcmp has the following significance (which is the same as the library function):

- 0 – means the two strings have the same characters (case sensitive) all the way to the NUL terminator.
- < 0 – means the first string is alphabetically less than the second (e.g., Strcmp("abc","xyz"))
- > 0 – means the first string is alphabetically greater than the second (e.g., Strcmp("xyz","abc"))

The Strcmp function is case sensitive, meaning that Strcmp("XYZ","abc") will return a value less than 0 (since capital 'X' is less than lower case 'a' in the ASCII coding scheme).

Example 2.10: Strcmp implementation

```
int Strcmp(char *sz1,char *sz2)
{
	int i;
	for(i=0;sz1[i]!=0&&sz1[i]==sz2[i];i++) {}
	return sz1[i]-sz2[i];
}
```

The Strcmp code, presented in Example 2.10, begins by declaring an integer local variable, i to keep track of our position in the string. The variable i is then initialized to zero in the initialization block of a for loop, which keeps looping until either:

- A NUL terminator in the first string is encountered (meaning the end of the first string), or
- The difference between the characters at position i of each string is not 0 (i.e., the characters are not equal).

When we exit the loop, we return the difference between the two characters we were looking at.

> *Test your understanding:* Explain why returning the difference between the two characters after we break out of the loop meets the requirements for the return value set forth in our function definition.

2.5.3: Atoi

The Atoi function mimics the C/C++ atoi() library function that takes a charter string and returns its integer equivalent. For example, the call Atoi("125") would return the integer 125. This functions turns out to be very useful, as we are often faced with situations where we are given character data (such as from the keyboard or a text file) that we need to transform into numeric data that we can use within our programs.

Our version of Atoi, like the C/C++ library version, is prototyped as follows:

```
int Atoi(char str[]);
```

where str contains a string of characters that are numeric digits (i.e., '0' through '9') that may be preceded by either a plus ('+') or minus('-') character. The return value is the integer equivalent of the string.

Just to have a little fun, we will also allow the user to put some spaces, tabs or other non-displaying characters before the number or between the sign and the number. In other words, the following strings would all be allowable:

```
"125"
"-125"
"   +125"
"-  125"
```

Skipping over such characters, often referred to as white characters (because they are white on the printed page—presuming you are using white paper!), requires us to know a little bit about how the ASCII system is organized. In particular, every character less than or equal to the space character (ASCII value of 32) is a white character. Thus to skip over white characters, we just need to check to see if a character's value is less than or equal to ' ' (or, equivalently, <= 32).

GetNonWhite
Since we need to do such skips twice in our Atoi function (once before and once after the sign, if a sign is present), it makes sense to define a separate function to do the skipping. So, before we write Atoi, we write GetNonWhite(), which is prototyped as follows:

```
int GetNonWhite(char str[],int nPos);
```

Where:

- str is the string where we want to skip over white characters
- nPos is the position in the string where we start looking (0 if we are at the start of the string)

The function returns an integer that tells us the position of the first non-white character encountered, or the position of the NUL terminator (if there were no more non-white characters in the string). Note: the return value will always be >= nPos, since the search starts at position nPos.

The GetNonWhite() function, illustrated in Example 2.11, is a simple while loop that continues as long as the character at str[nPos] is white (i.e., <= ' ') and is *not* the NUL terminator. After each test, we increment nPos. When we break out of the loop, either because we reached the end of the string or encountered a non-white character, we return nPos.

Example 2.11: GetNonWhite() Helper Function

```
int GetNonWhite(char *str,int nPos)
{
      while(str[nPos]>0&&str[nPos]<=' ') {
            nPos++;
      }
      return nPos;
}
```

Atoi

Having defined GetNonWhite, we can now proceed to defining Atoi. The basic operation of Atoi is as follows:

- Three local integer variables are declared and initialized: nRet to hold our return value, nPos to keep track of where we are in the string—both initialized to 0—and nSign (initialized to +1) which keeps track of whether the number is positive or negative.
- It calls GetNonWhite() to skip over leading blanks.
- It checks to see if the first character is a sign (+ or -). If it's a minus sign, it sets nSign equal to –1, then moves to the next character. If it's a plus sign, it just moves to the next character.
- GetNonWhite() is called again, to move us to the first digit.
- We enter a while loop that continues as long as we are still reading digits, which are any ASCII characters between '0' and '9' (which is a result of how ASCII is laid out).
- During each pass of the loop, we take our old nRet value, multiply it by 10, and add the character minus the '0' character—which turns the character into its integer equivalent. For example, the ASCII character '2' is 50, and the ASCII character '0' is 48. Therefore '2' – '0' == 50 – 48 == 2. (This works only because the character digits are coded in sequence, i.e., '0' is 48, '1' is 49, '2' is 50, and so forth).
- When any non-digit character is encountered, the function returns the accumulated value of nRet.

The least intuitive aspect of the Atoi function is how the digit-processing loop works. The easiest way to illustrate this is with a table of values for an example—say the string "125" (with a NUL terminator at the end)

nRet coming in	nPos	str[nPos]	str[nPos]-'0'	nRet coming out
0	0	'1'	1	1
1	1	'2'	2	12
12	2	'5'	5	125
125	3	'\0'	N/A	125

Table 2.1: Values in Atoi() loop as digits are processed

The Atoi function code is presented in Example 2.12.

Example 2.12: Atoi() Function

```
int Atoi(char *str)
{
      int nRet=0;
      int nPos=0;
      int nSign=1;
      nPos=GetNonWhite(str,nPos);
      if(str[nPos]=='-') {
            nSign=-1;
            nPos++;
      }
      else {
            if(str[nPos]=='+') {
                  nPos++;
            }
      }
      nPos=GetNonWhite(str,nPos);
      while(str[nPos]>='0'&&str[nPos]<='9') {
            nRet=nRet*10+str[nPos]-'0';
            nPos++;
      }
      return nRet*nSign;
}
```

2.5.4: Strcat

Walkthrough available in Strcat.avi

The Strcat function, modeled after the C/C++ library function *strcat*, takes a string—the second argument—and adds it to the end of another string—its first argument. It is prototyped as follows:

void Strcat(char *s1,const char *s2);

(the library version of strcat returns a pointer, however the return value is irrelevant to the function's normal use, so we won't bother).

From a user's point of view, the operation of the construct is demonstrated by the following code:

```
        char s1[20]="Hello,";
        char s2[20]="World!";
        Strcat(s1,s2);
        // If we look at s1, it will now contain "Hello,World!"
```

An example of how the function could be implemented is presented in Example 2.13.

Example 2.13: Strcat() Function

```
void Strcat(char *s1,const char *s2)
{
        int i;
        int j;
        i=Strlen(s1);
        for(j=0;s2[j]!=0;j++) {
                s1[i+j]=s2[j];
        }
        s1[j+i]=0;
        return;
}
```

Conceptually, we can break this function into three parts:

Part 1: Move to the end of s1

Although we could find the end of s1 with a loop, having already implemented our own Strlen function, it makes more sense just to call that function. Thus the line:

```
        i=Strlen(s1);
```

In our earlier example, the first string contained "Hello,", so i becomes 6 when we call Strlen(). The position s1[i] is then illustrated in Figure 2.3, precisely on the NUL terminator that ends the string in s1.

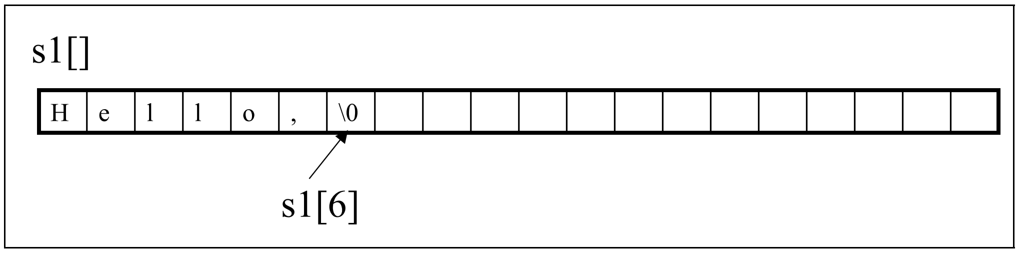


Figure 2.3: Position of s1[i], where i == Strlen(s1)

Part 2: Move the characters from the second string to the end of the first

This is the purpose of the loop. We use j to keep track of where we are in the second string, and i+j to identify our position in the first. Thus, the following loop will copy the characters over:

```
for(j=0;s2[j]!=0;i++,j++) {
        s1[i]=s2[j];
}
```

Continuing with our example, suppose we are on our third pass of the loop. At that point, j will equal 2 and i will continue to equal 6—since it is not changed by the loop. The resulting positions, and the character being moved are presented in Figure 2.4.

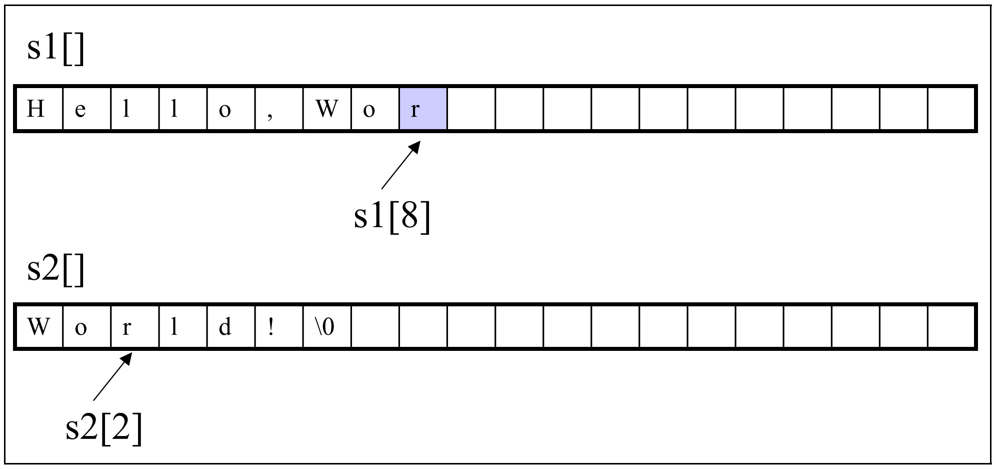


Figure 2.4: Moving character over when j==2 and i==6

Part 3: Finish off the string

Unfortunately, as we copied the characters over from s2 to s1, we stopped upon hitting the NUL terminator in s2, as illustrated in Figure 2.5. As a result, we need to place a NUL terminator at the end of s1 to finish off the string. This is done with the statement:

```
s1[i]=0;
```

In the Figure 2.5 illustration, this would have the affect of placing a '\0' character in the gray cell in s1.

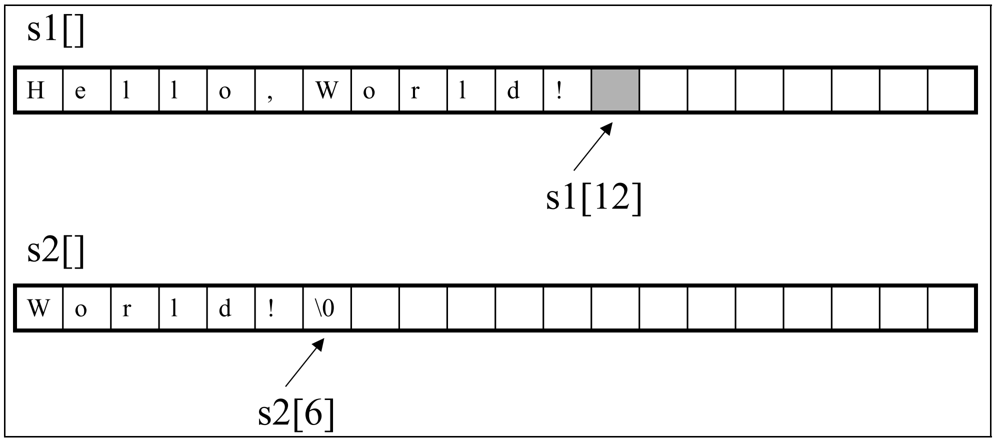


Figure 2.5: Situation when the loop breaks out

2.5.5: Strcpy

Walkthrough available in Strcpy.avi

The strcpy() function is used to copy a NUL terminated string from one array to another. The function is very commonly used in C (and even in C++) because there is no assignment operator for NUL terminated strings. The function is prototyped as follows:

```
char *strcpy(char dest[],const char src[]);
```

In using the function, it is very important to keep two things in mind:

- The src[] and dest[] arrays must not overlap. If they do, the function's behavior is "undefined" (never a good thing!).
- The dest[] array must be large enough to hold the number of bytes in the src[] string. Since C++ won’t check for array boundaries, if the string in src[] is too large, it will overflow the dest[] buffer and corrupt whatever memory follows.

A sample of how the strcpy function could be implemented is presented in Example 2.14.

Example 2.14: Sample implementation of strcpy() library function

```
char *Strcpy(char dest[],const char src[])
{
	int i;
	for(i=0;src[i]!=0;i++) {
		dest[i]=src[i];
	}
	dest[i]=0;
	return dest;
}
```

The function is about as simple as a C++ function can get. It has two basic parts:

1. A for loop copies individual characters from src to dest until the NUL terminator in src is encountered. The line src[i]!=0 could also be written src[i]!='\0' or src[i]!=0x00. On the other hand, src[i]!='0' would definitely *not* work!
2. A NUL terminator is placed at the end of the string in dest[], since we broke out of the for loop before that character was moved over.

   Test your understanding: In addition to the alternative tests for the NUL terminator listed in (1) above, the for construct could have been written:

   ```
   for(i=0;src[i];i++) {
   	dest[i]=src[i];
   }
   ```

Explain why this would work. Why would it probably be preferable to write the test in the original form (or using one of the two alternatives provided).

2.5.6: Mainmenu function

Walkthrough available in MainMenu.avi

The MainMenu function allows us to illustrate how we can implement a simple menu using a case construct embedded within a while loop. We will be using this type of menu frequently in the rest of the text, because it allows us to insert test code for many functions that we might want to test.

The MainMenu function takes no arguments and returns no value. (Indeed, most of the time when we implement a menu, we won't write a separate function, but instead will place the code described here in main).

The function does three basic things:

1. It prints out a menu of user options
2. It reads a line of text from the user, then uses the first letter of that line as the option code
3. It routes the code, using a case statement, using the option key.

Because the MainMenu provided here is intended to be a skeleton, rather than a real application, only the "Quit" (i.e., 'q') option is actually implemented. The framework for options 'A' and 'B' has been put in place, however, with comments showing where code would normally go. Additional options can easily be added by following this framework.

Example 2.15: MainMenu() function skeleton

```
void MainMenu()
{
      int nChar;
      int szIn[256];
      int nEnd=0;
      while(nEnd==0) {
            cout<<"A.ToDo"<<endl;
            cout<<"B.ToDo"<<endl;
            cout<<"Q.Quit"<<endl;
            cin >> szIn;
            nChar=szIn[0];
            switch()
            {
                  case 'A':
                  case 'a':
                  {
                        //Put code here;
                        break;
                  }
                  case 'B':
                  case 'b':
                  {
                        //Put code here;
                        break;
                  }
                  case 'Q':
                  case 'q':
                  {
                        nEnd=1;
                        break;
                  }
                  default:
                  {
                        cout<<"Illegal!"<<endl;
                  }
            }
      }
      return;
}
```

The code for the MainMenu function is presented in Example 2.15. The operation of the function proceeds as follows:

1. Before entering the main loop, a variable nEnd is initialized to zero.
2. We enter a while loop that will keep looping as long as nEnd remains equal to zero
3. Within the while loop:
    a. A menu of options is displayed, using a series of printf statements
    b. The user is prompted to enter an option, which is placed in the array szIn[]. The expression **cin >> szIn** works the opposite of cout << value, in

    that it retrieves input from standard input (the keyboard) and places it in the variable on the right.
    c. The first character of szIn (i.e., the first character in the user's option string) is assigned to nChar
    d. A case construct, switching on nChar, is used to route the option. Upper and lower case labels are used for each option
4. In the event the user chooses the option 'Q' (or 'q'), the case construct assigns nEnd to 1, which will cause the while loop to end
5. In the event the user chooses an option that does not exist, the default case is to print "Illegal" on the screen.

2.5.7: Int2String Function

Walkthrough available in Int2String.avi

The Int2String() function takes an integer variable and creates its character string representation. For example, it would take the integer 125 and place the characters '1', '2', '5', '\0' into an array (the last character being the NUL terminator that ends the string). The function is prototyped as follows:

```
void Int2String(char szNum[],int nVal);
```

where str[] is the string array that we will place our characters into and nVal is the integer that we will be converting.

Operation
The Int2String function must be passed an array with sufficient room to hold the resulting string. It could be called as follows:

```
char num[20];
Int2String(num,125);
// contents of num[] would now be "125" (with a NUL terminator at the end)
```

The function is essentially the reverse of the Atoi function already covered.

Implementation
The Int2String function implements the remainder method for converting an integer from one base to another. This involves taking the value to be converted and successively dividing by the base, storing the remainder after each division. The remainders are then reversed to create the new number. Some examples of the process are presented in

Example 2.16, with the last example showing how digits can be extracted using the process.

Example 2.16: Remainder Method Conversions

Convert 136_{10} to base 5 using the remainder method

136/5 = 27	Remainder: 1
27/5 = 5	Remainder: 2
5/5 = 1	Remainder: 0
1/5 = 0	Remainder: 1

Therefore, using the remainders and building from right to left, the new value is 1021_5

Convert 5887_{10} to base 16 using the remainder method

5887/16	= 367	Remainder: 15 = F
367/16	= 22	Remainder: 15 = F
22/16	= 1	Remainder: 6
1/16	= 0	Remainder: 1

Therefore, using the remainders and building from right to left, the new value is 0x16FF.

Convert 125_{10} to base 10 using the remainder method(allowing us to extract the digits)

125/10	= 12	Remainder: 5
12/10	= 1	Remainder: 2
1/10	= 0	Remainder: 1

Therefore, using the remainders and building from right to left, the value is125.

The Int2String() implementation is presented in Example 2.17. The function is essentially an implementation of the remainder method for switching between bases. There is, however, a problem with the remainder method that needs to be addressed: the method only works on positive numbers. For this reason, the function works as follows:

- Local variables are declared as follows:
    - char szN[80], to hold our raw (i.e., reversed) number.
    - int nDig, to keep track of the current digit we are processing
    - int nSign, which is initialized to 1 (indicating a positive integer).
- We determine if the argument nVal is negative using an if construct with no else block. If it is, we:

    - Set nSign to –1, flagging the fact we have a negative number
    - Multiply nVal by –1, making it positive. Thus nVal of –125 becomes +125.
- We implement the remainder method in the do…while loop—chosen because we want at least one digit stored, even if nVal is 0. Within the loop, we store each remainder—at position nDig in szN[]—as an ASCII digit by adding '0' (48) to it. (See the Atoi function, which works in the reverse direction, for the explanation). We then increment nDig to position us on the next digit. Some of the key variables in the loop, for nVal starting at 125, are summarized in Table 2.2.
- Once nVal==0, we break out of the loop because we have completed the remainder method.
- We then check nSign, appending a '-' (minus sign) to our string in szN[] and incrementing nDig. This will cause our number to be preceded by a minus sign after the reversal.
- We place a 0 (NUL terminator) at position nDig of szN[], to end the string.
- We call Reverse() to reverse the string, placing the reversed string in szNum, our original argument.

nVal Starting pass	*nVal % 10*	*nDig Starting pass*	*szN[nDig] before nDig++*	*szN[] at test*	*nVal at test*	*nDig at test*
125	5	0	'5'	"5"	12	1
12	2	1	'2'	"52"	1	2
1	1	2	'1'	"521"	0 (ends)	3

Table 2.2: Key values in Int2String loop

Example 2.17: Int2String() function implementation

```
void Int2String(char *szNum,int nVal)
{
      char szN[80];
      int nDig=0;
      int nSign=1;
      if(nVal<0) {
            nSign=-1;
            nVal=-1*nVal;
      }
      do {
            szN[nDig]=nVal%10+'0';
            nVal=nVal/10;
            nDig++;
      } while (nVal>0);
      if(nSign<0) {
            szN[nDig]='-';
            nDig++;
      }
      szN[nDig]=0;
      Reverse(szNum,szN);
      return;
}
```

As suggested by its name, Reverse() (Example 2.18) is used at the very end of Int2String to reverse the digits. For example:

```
	char szFrom[10]="521-";
	char szTo[10];
	Reverse(szTo,szFrom);
	// Will place "-125" in szTo[]
```

The function operates similarly to Strcat, in that we maintain different positions in different arrays. The function works as follows:

- The variable nLen is set to the length of the szFrom string that we are reversing.
- We enter a loop with a counter variable i. The loop proceeds as follows:
    - A character at position nLen-i-1 in szFrom is placed at position i in szTo. Notice that when i is 0, this places us at the character directly in front of the NUL terminator in szFrom (i.e., at position nLen-1, the last actual character), while it puts us at the beginning of szTo.
    - i is incremented. The effect of incrementing i is to move our position in szTo one character to the right (towards the end), while our position in szFrom is moved one character to the left (towards the beginning)
    - When i=nLen, we have reversed the characters, and break out of the loop.
- We place a 0 (NUL terminator) at the position i (which is equal to nLen when we break out of the loop) to terminate the string
- We return, because the function is complete.

Example 2.18: Reverse() function implementation

```
void Reverse(char *szTo,char *szFrom)
{
	int nLen;
	int i;
	nLen=Strlen(szFrom);
	for(i=0;i<nLen;i++) {
		szTo[i]=szFrom[nLen-i-1];
	}
	szTo[nLen]=0;
	return;
}
```

Test your understanding: What would happen if we changed the szFrom[nLen-i-1] in the function to szFrom[nLen-i], and changed the loop test to i<=nLen, so we could reverse all the characters, including the NUL terminator?:

2.6: Review and Questions

2.6.1: Review

The best way to think about operators is as a special type of functions. For example, the + operator used to add two integers can be thought of as having the following declaration:

```
    int operator+(int a1,int a2);
```

There are six categories of operators that we most commonly use in C. There are:

1) *Arithmetic operators*: These include: a) five key binary operators (+,-,*,/ and %). The first four are self-explanatory, and can be applied to both integers and real numbers. The last, the modulus (%) operator, returns the remainder resulting from an integer division, b) + and – unary prefix operators, used to specify a number's sign, and c) ++ and – operators, used to increment and decrement integers by 1. These operators can be applied either in prefix or postfix mode.

2) *Assignment operators:* These include the pure assignment operator (=) and a variety of composite operators (e.g., +=, -=, *= and /=). The pure assignment operator takes the value on its right-hand side (RHS) and places it in its left-hand side (LHS) argument, which may be a variable, a reference to an array element, or certain other types of expressions (to be introduced in Chapters 10 and 11). The types of expressions that can appear on the LHS of an assignment are often referred to as l-values or lvalues.

3) *Relational operators:* Operators used to determine relative size of arithmetic arguments, such as >, <, >=, <=, == and !=. These are binary operators that return a Boolean (true or false) result based on the expressions on their LHS and RHS (e.g., 3 > 4 returns false, 3<4 returns true). A particularly common error using relational operators is to confuse the equality test (==) with the assignment operator (=).

4) *Logical Operators:* Operators used to combine test results, such as those returned by relational operators. The most common of these are two binary operators: logical AND (&&), which returns true only if both sides are true, false otherwise, and logical OR (||), which returns true if either or both sides are true, false only if both sides are false. A unary operator, NOT (!), is also provided that changes a true expression to false, and a false expression to true.

5) *Bitwise operators:* Operators that apply logical-style truth tables to the individual bits of integer arguments. These include the binary operators bitwise AND (&), bitwise OR (|) and bitwise exclusive-OR (^). It is important to be aware of these operators for two reasons: i) they are often used to compress a lot of true-false information into a single integer argument (e.g., arguments to some MS-Windows functions), ii) they can easily be confused with their logical counterparts, with results that can be hard to detect. A bitwise

unary operator, bitwise NOT, is also available that reverses all the 0s and 1s in an integer, returning the 1s complement of that integer.

6) *Conditional operator:* The only ternary operator in C/C++, it is of the form:

test ? *value1* : *value2*

and functions very much like the =IF() function in MS-Excel, returning the first value if the test is true, the second value if the test is false.

When operators are applied, the order in which they are applied is determined by operator precedence. For example, in the expression x1+x2*x3, the expression x2*x3 is evaluated before the + operator is applied because multiplication has a higher precedence than addition. Operator order can be forced using parenthesis—e.g., (x1+x2)*x3—which is recommended in this text, as programmer-created precedence errors can waste hours of debugging time.

In addition, to these six categories, C++ also supports a number of IO-driven operators (such as >> and <<). In addition, C++ allows programmers to overload operators in much the same way that its functions can be overloaded. These capabilities cannot be exploited effectively except when OOP is being used. As a consequence, they are not discussed in detail in this book.

C++ also provides constructs for implementing functions, branching and looping. The most important of these are summarized in the table that follows:

C++ Construct Syntax

Function:

return-type **FunctionName(***arg-list***)**
{
    // code block
    return *expr-matching-return-type***;**
}

Notes:

- **return** statements may appear anywhere in the function
- ***arg-list*** is set of declarations separated by commas

If construct:

```
if (test-expr) if-code-block ;
else if (test-expr) else-if-code-block;
else else-code-block;
```

Notes:

- More than one **else if** test may be provided
- Only **if** portion of construct is required

Case construct:

```
switch (integer-expr)
{
        case int-const1: {
                // code block
                break;
        }
        case int-const2:
        case int-const3: {
                // code block
                break;
        }
        default: {
                // code block
        }
}
```

Notes:

- If **break** (or **return**) not included, case fall through will occur
- Multiple case labels may be associated with a single code block
- **default** block is optional

while loop:

```
while (test-expr ) {
        // code block
}
```

Notes:

- **break:** sends control to the bottom node, breaking out of the loop
- **continue:** jumps to top, beginning next pass
- code block must change some value related to *test-expr* or infinite loop will occur (unless **break** is present).

for loop:
 	
for(*initialize-code* **;** *test-expr* **;** *iterator-code*) {
    // code block
}
 	
Notes:
• *initialize-code* is a comma-separated list of statements that are executed once, before the loop begins.
• *iterator-code* is a comma-separated list of statements that gets executed after the code block in each iteration.
• **break:** sends control out of the loop, bypassing *test-expr* and *iterator-code*.
• **continue:** sends control to *iterator-code* in preparation for next iteration of the loop

while loop:
 	
do {
    // code block
} **while** (*test-expr*);
 	
Notes:
• **break:** breaks out of the loop without performing any further test
• **continue:** jumps to *test-expr* at the bottom, beginning next pass
• code block is always executed at least once
• code block must change some value related to *test-expr* or infinite loop will occur (unless **break** is present).

2.6.2: Glossary

Arithmetic operators – Operators defined to perform mathematical computations against any expression or statement in the source code.
Assignment operators – Operator that takes a variable from the right hand side (RHS) and places into the left hand side (LHS) these can be formed using composites such as +=, etc.
Bitwise operators – Operators that apply logical style rule tables to each bit in a byte to determine the final result.
break – A statement indicating the end of a loop or the end of a particular block in a multi-branch construct
By Reference – To pass a variable by reference means to change the values in both the function and the variables.
By Value – To pass a variable by value means to make a copy of the variable thereby saving the original from being altered.
case – A label, followed by an integer constant, used to identify where control should pass in switch block

Code Block – A collection of source code statements, delimited by braces {}, located within a function.
Conditional operators – Operators that take place when only a condition that forces the result to be true is executed.
continue – A statement indicating that the next iteration of a loop should begin immediately, skipping the remainder of a code block
Declaration – A formal way of expressing the manner in which a function is defined.
Definition – A statement expressing the manner in which a subroutine gets its parameters and return statements.
do – A keyword signifying the beginning of a do…while loop, where the loop test is at the end of the loop
else – A keyword signifying the start of a code block to be performed if no tests in an **if** construct are true
else if – A pair of keywords specifying an incremental test in an **if** construct
for – A keyword specifying the beginning of a test-first loop with an initializer block of statements performed before entering the loop and an iterator block, performed after each pass.
if – A keyword, followed by a parenthesized test, signifying the start of an **if** construct
return – A keyword specifying the completion of the function that contains it.   function. If a function return type is specified, an expression compatible with that type must follow the keyword.
lvalue expression – An expression that is evaluated on the left side of the assignment operator.
Logical operators – Used to determine combined test results and returns only a Boolean return type.
Nesting – The mechanism of placing constructs within other constructs in a logical manner.
Operator Overloading – The ability of using multiple references to the same operator to overload the operator and change its function to suit some custom means.
Operator Precedence – The manner in which a expression is evaluated by some mathematical means.
Relational operators – Operators used to determine the relative size of each when compared to one another.
Return Type – Expresses the relative size of the variable being evaluated so as to not conflict or lose data when the result is computed.
Return Statement – The format that C used to pass a variable back to the calling function.
switch – A keyword signifying the outer boundaries of a case construct, followed by a parenthesized integer expression
Type Cast – To change a variables pre-defined size by using another variable
while – A keyword followed by a test expression that controls whether of not a loop continues repeating. **while** loops test before repeating a code block, whereas **do…while** loops test after the repeated code block has been done once

2.6.3: Questions

Question 2.1. MainMenu: Create your own main menu function, modeled after the function described in Section 6.6.2 that can be used to test the functions you will be writing below. From this point on in the text, it is assumed that you will set up such a function, as needed, to conduct tests.

Question 2.2. FindChar: Write and test a function called FindChar(), prototyped as follows:

```
int FindChar(char str[],char cChar);
```

that looks for a character (cChar) in a string (str[]) and returns the position where it was found (or –1 if not found).

Question 2.3. SumDigits: Write and test a function called SumDigits that sums every numeric digit character (i.e., character '0' through '9') encountered in a string—no matter what its position.

Question 2.4. Int2OctalString: Modify and test the Int2String function presented in 6..6.3 so it would produce strings in octal (base 8). *Hint:* look back at the remainder method and you will find this is a relatively trivial matter!

Question 2.5. Int2HexString: Modify and test the Int2String function presented in 6..6.3 so it would produce strings in hexadecimal. *Hint:* This is a bit harder than the Int2OctalString function, but is still relatively simple—the key difference is making sure you actually create hex digits.

Question 2.6. Int2BaseString: Modify and test the Int2String function so it takes a third argument, the base of the number you want to display. The new prototype would be:

```
void Int2BaseString(char szNum[],int nVal,int nBase);
```

where nBase is a number between 2 and 36. *Hint:* Figuring out why nBase is limited to numbers less than or equal to 36 is a good starting point in solving this problem.

Question 2.7. Real2String: Create and test a version of the Int2String function that displays real numbers as decimal numbers. The prototype of this function should be:

```
void Real2String(char szNum[],double dVal,int nPrecision);
```

Where szNum is where the resulting number should be placed, dVal is the double to be converted, and nPrecision is the number of decimal places. You may assume that it only needs to work for values of dVal between –2 billion and +2 billion. *Hint:* This function can be excruciatingly hard to write, or very easy, depending on how you approach it. The easy way is to look at it as a problem of writing out two integers—one to the left of the decimal point, and one to the left of it:

- The part on the left of the decimal point is easy (all you need to do is assign dVal to an integer). Get it working first!
- The part to the right isn't that much harder, once you figure out how to use nPrecision
- Once you've got both parts, tack a decimal point on the the end of the left hand part, then concatenate them (e.g., using Strcat)

Once you've finished this function, you should have a pretty good idea how the formatting capabilities of printf work.

Question 2.8. Atof: The C++ library function atof() is very similar to atoi(), except it returns its result as a real number and accepts strings that can include a decimal point. Write your own version of the function. (Hint: use the Atoi() implementation as a starting point, then figure out how you will handle the decimal point).

Chapter 3

Pointers, Structures and Dynamic Memory

Executive Summary

Chapter 3 focuses on three topics that are particularly applicable to C++ as a language: pointers, structures and dynamic memory. Pointers provide C++ with the ability to manipulate addresses. Structures allow us to represent complex data, and is the core component of the "object" in object-oriented programming. Dynamic memory allows C++ to acquire the memory it needs when it needs it, and release it when it is done. Taken together, these features are the source of many of C++'s greatest efficiencies (and weaknesses). Their usage may also be unfamiliar to readers whose initial programming experience was in another language.

The chapter begins by examining how pointers are declared, how to place useful addresses into pointers, and how to get the data that a pointer "points" to. The subject of pointer arithmetic—what happens when you apply arithmetic operators such as addition and subtraction—is then explored. We then return to the subject of arrays, introduced the previous chapter, and extend the concept to multidimensional arrays and arrays of pointers. The subject of structures is then introduced, showing how related data elements can be aggregated into a single object. Using pointers to access structure data and various types of arrays of structures are also considered. Finally, the topic of dynamic memory is discussed, and contrasted with local and static memory. The new and delete operators are then explained and the importance of avoiding memory leaks is discussed.

Learning Objectives

Upon completing this chapter, you should be able to:

- Declare, initialize and access data with pointers
- Perform basic pointer arithmetic
- Define and use multidimensional arrays
- Distinguish between multi-dimensional arrays and arrays of pointers
- Define. declare and access data in structures
- Use arrays of structures and structure pointers to maintain data collections
- Explain the three types of memory used by a C++ program
- Allocate dynamic memory using the new operator
- Release dynamic memory using the delete and delete [] operators

3.1: Nature of a Pointer

When we introduced the concept of an array, we explained that an address in memory was the most natural way to refer to an array. Since all the elements in an array are *contiguous* in memory—meaning that they follow one another with no space in between—if you know: a) the address of the first array element, and b) the size of each array element (in bytes), you can easily determine the location of any other element in the array.

> *Test your understanding:* If the start of an array of one-byte characters buf[80] is located at address 0x1000 in memory, where would buf[10] be located (remember to convert the 10 to hex before specifying the address)? How would that change if buf[80] was an array of double real numbers, starting at the same address? What key fact would you need to know about a double to answer the question?

Addresses can be used for many other things besides telling us where an array starts, however. In this section we examine *pointers*, variables that we declare to hold a memory address, instead of an actual value.

C and C++ are relatively unusual among computer languages in that they allow programmers to manipulate the values of memory directly, using variables defined to hold addresses instead of actual values. Using these variables, the C/C++ programmer can write code of unparalleled efficiency. Compiled with a good compiler, well written C/C++ code is nearly as efficient as assembly language (and more so, as application size grows).

Pointers also have a down side, however. Because they provide a direct pathway to memory, and because memory contains practically everything that keeps the computer going—including the operating system, program code and application data—writing data to just one bad address can cause severe problems to practically any program that happens to be running when the error occurs. On the PC, the introduction of operating systems (such as MS Windows NT, 2000 and XP) that are designed to protect applications from each other, has made the situation markedly better than it used to be. Still, it is probably a good rule of thumb to save any data that you care about (such as your taxes, or that twenty-three page term paper on medieval footwear that you need to hand in tomorrow) before you start running untested code that uses addresses.

3.1.1: Pointers as Variables

A pointer is declared just like any other variable, except an asterisk is placed in front of the variable name to let the compiler know that the variable will hold an address, instead of a value. For example, in the code:

```
char *cpointer;
double val,*pval;
```

cpointer is declared as a pointer that will hold the address of a character. Similarly, val is declared as a double, while pval is declared as a variable that will hold the address of a double.

From the above illustration, it is apparent that whenever we declare an address, we must also declare the type of data that we expect to find at that address. There are two main benefits of doing so. The first is that it allows us to perform pointer arithmetic, to be discussed later in this chapter. The second advantage is that it gives the compiler the ability to ensure that we aren't doing anything obviously incorrect. For example, consider the strcpy library function that we discussed in the previous chapter. It is declared as follows:

```
char *strcpy(char *dest,const char *src);
```

Suppose that, in our code, we called the function as follows (using the variables cpointer and pval as defined above):

```
strcpy(cpointer,pval);
```

The compiler would look at the first argument and, since it was expecting a pointer to a character and that is how cpointer is defined, it would move on. Then it would examine the second argument. Although pval is an address, the way it is defined we would expect to find 8 byte double-precision real numbers at the address pval contains. The strcpy function, however, expects the address of a character string, so it doesn't seem to make sense that we're giving it the address of a double. Thus, the compiler will do its job, and issue a warning. As the programmer gains experience, such warnings come to be deeply appreciated. By telling us the exact line where our code doesn't make sense, the compiler saves us minutes, hours or days of searching to figure out why we are ending up with such weird characters in the string pointed to by cpointer after we call the function.

Despite the fact the C++ requires us to specify the type of each pointer we use, an address is an address, regardless of what it points to. Thus, in the above declarations, the actual variables cpointer and pval are going to be the same size. On a typical PC, Windows 95 or above, those pointers will both be four bytes long—even though a char is 1 byte and a double is 8 bytes on the same machine. The bottom line: the length of the address tells you nothing about the size of the house there.

3.1.2: Initializing the Value of a Pointer

When you declare a pointer, you are not setting aside any memory for the data it will point to.

Some statements are so important that they warrant constant repetition. So let's say it again, altogether now:

> ***When you declare a pointer, you are not setting aside any memory for the data it will point to!!!***

Let us consider what this means by looking at a series of declarations, and figuring out what they are actually doing:

```
short int i,parray[10],*pi;
char *mypointer,c1,carray[80];
unsigned char uc2;
```

If we look at the first line, we are declaring three variable names. The variable i refers to a single short integer, taking up 2 bytes of memory (in Visual C++). The variable parray, on the other hand, refers to an array of 10 short integers (say 20 bytes of memory). When parray is used by itself, as we have already said, it refers to the address where that 20 bytes of memory start.

What about pi? Well, based on what we already know (especially given the above statement that was repeated), we can deduce two things:

- Somewhere in memory we have set aside enough room to hold an address (probably 4 bytes)
- Whatever value is currently contained in those four bytes has nothing to do with anything useful. Most likely, it is the address of a random location in memory—an address that our system may not even have installed (e.g., if the address is hex 0xCCCCCCCC, a favorite of the Visual Studio debugger, it probably doesn't exist unless you have over 3 gig of RAM installed).

From the second of these facts, we can quickly deduce a third fact. It is extremely unlikely that anything good will come of using the pointer pi until we get the address of a short integer we want to access into it.

There are four common ways of getting an address into a pointer:

1. Applying the & operator to a variable name or array element (or, more generally, any lvalue)
2. Calling a function that returns an address or changes a pointer reference argument
3. Assigning the value from another pointer or array name

4. Assigning the value from a pointer arithmetic expression

The & operator
The & operator can be applied to any C++ expression referring to a value in memory, sometimes referred to as an lvalue, a term introduced in passing when we discussed the assignment operator (i.e., any value that it makes sense to put on the left hand side of an assignment statement).

To provide a simple example, the & operator could be used as follows:

```
char c1='c',carray[80]="Hello, World";
char *pc1,*pc2;
pc1=&c1;
pc2=&carray[6];
```

In this example, pc1 would contain the address of c1 by the time we reach the end of the code. If we went to that address, we'd find a 'c' (the integer 97, according to the ASCII coding scheme) there. Similarly, pc2 would contain the address of element 6 (zero-based, of course!) of the array named carray. If we went to that address, we'd find a space (ASCII value of 32).

> *Test your understanding.* If we changed the last line of our example to pc2=&carray[7], what value would we find if we went to the address contained in pc2 (character and integer versions)?

The & operator can also be applied to pointers. For example, the expression &pc1 is perfectly legal. Its interpretation is "the address in memory where the (4 byte) pointer pc1 is located". You should notice that this is very different from the *value* of pc1—which is "the address in memory where the (1 byte) character c1 is located". The relevance of this will become clearer when we turn our attention to complex arrays, later on in the chapter.

Calling a function
Many functions, particularly string functions, return an address as a value. You can therefore use such a function call to acquire the address to assign to your pointer. We illustrate this by considering the strstr() standard library function:

*char *strstr(const char *s1,const char *s2);*

The function searches the string pointed to by its first argument (i.e., s1) for the first occurrence of the second string (i.e., s2). If it finds the substring, it returns the address within s1 where the occurrence begins. If not, it returns the NULL pointer, signifying no value. For example, consider the following fragment of code:

```
char ar1[80]="The quick brown fox";
char *found;
found=strstr(ar1,"ow");
```

After the last line is executed, the pointer found will contain an address that points to the string "own fox".

Test your understanding: If the array ar1 begins at address 0x00001000, what would the address in the pointer *found* be after the fragment in the example was run?

In C++ it is also possible to assign a value to a pointer argument that is passed as a reference to a function, for example:

```
void ElementAddress(char *&pEle,const char *ar,int nPos)
    {
            pEle=&ar[nPos];
    }
```

Calling this function will result in the first argument taking on the address of the element of ar[] at position nPos. Notice that the & in the function header—which declares pEle to be a reference—serves an entirely different purpose from the & in the body—which is the operator that takes the address of element ar[nPos].

Assigning the value from another pointer or array name
Since pointers are variables, they can be assigned to each other. Since array names refer to the address where the array starts, these names can also be used to get values into pointers. Consider the fragment of code below:

```
char c1='c',carray[80]="Hello, World";
char *pc1,*pc2;
pc1=carray;
pc2=pc1;
```

The variables pc1 and pc2 would both end up referring to the same address—the address where carray starts.

The assignment of addresses between array names and pointers does not work in the opposite direction, however. The code:

```
    carray=pc1;             /* definitely *not* legal */
```

will lead to compiler errors. The problem is that carray is not a variable. Instead it has a value assigned to it by the compiler (and then adjusted by the linker and loader) based upon the translation of the program to machine language. To use an analogy, variables are like data on a chalkboard: we can constantly write new values and erase existing ones. Array names, in contrast, are like addresses carved into a stone facade. We can read them easily enough, but we can't change them. We will return to this point when we consider the relationship between arrays and pointers.

Initializing a string pointer
Because they are so useful in C/C++, NUL-terminated strings are given special treatment as far as initialization is concerned. We have already seen this in arrays, where we can write:

```
char howdy[10]="hello";
```

instead of writing:

```
char howdy[10]={'h','e','l','l','o','\0'};
```

which is how we initialize other arrays.

String notation, using double quotes, can also be used to initialize pointers. To understand how this differs from initializing an array, consider the following two statements:

```
char howdy[10]="hello";
char *hi="hello";
```

In the first of these statements, the compiler will set up code that creates a 10-character array, then perform a strncpy of "hello" into the array, initializing the last 4 characters in the array to the NUL terminator. In the second:

- the variable hi, a (4-byte) pointer is first created
- the compiler then counts the number of characters in "hello" and then, somewhere in memory, creates an array of that size (at least 6 bytes)
- "hello" is then used to initialize the newly created array
- finally, it takes the address of the array it just created and places that address in the pointer hi.

This more elaborate process of initializing a pointer has some significant practical repercussions. Specifically, most compilers assume a double-quoted string to be constant, meaning it should not be changed. As a consequence, the following code fragment, which attempts to change "hello" to "jello" creates an access violation:

```
char *hi="hello";
hi[0]='j';
```

In Visual Studio, this particular access violation results in an exception, causing the program to stop running.

The best way to avoid such a violation is either:

- To declare string variables as arrays, if you know you are going to change them
- To use the const qualifier when declaring pointers to strings that should not be changed (e.g., const char *hi="hello";).

Both of these avoid the problem. Which is better depends entirely on the situation.

> *Test your understanding:* What do you suppose would happen if you wrote the following code and then compiled it?
>
> ```
> const char *hi="hello";
> hi[0]='j';
> ```
>
> If you really wanted to turn "hello" to "jello", how would you need to declare the string?

3.1.3: Retrieving the value of a pointer

There would not be much purpose to placing addresses in pointers if we could not then retrieve the values at those addresses. In fact, for the most part, we usually don't care very much about the actual addresses that are stored in a pointer. These addresses can easily change every time we run our program, depending upon where it is loaded in memory. Thus, getting data values from the location specified by a pointer, also called *dereferencing* a pointer, is a very important task.

There are two operators that are typically used to get addresses from pointers, the * operator (sometimes referred to as the *indirection* operator, or the *dereferencing* operator) and the familiar brackets we have already learned to use with arrays.

Dereferencing a pointer with the * operator
If you place an * operator in front of a pointer (or any expression that returns an address), the resulting expression refers to the value contained at that address. The type of pointer being dereferenced will, of course, determine the type of value returned. For example, in the code fragment below:

```
char c1='c',carray[80]="Hello, World";
char *pc1,*pc2;
pc1=&c1;
pc2=&carray[3];
carray[2]=*pc1;
*pc1=*carray;
```

The line *carray[2]=*pc1* will change the first l in "Hello, World" to a ‘c’ (which is the value that pc1 points to). The line **pc1=*carray* is even more interesting. The right hand side (*carray) refers to the character at the address specified by carray—which is the first element of the array, or 'H'. The left hand side (*pc1), refers to the data that pc1 points to, which turns out to be the variable c1. So, this line has the same effect as writing:

```
c1='H';
```

In other words, executing the line of code **pc1=*carray* actually changes the value in c1. This ability to change variable values by using their addresses turns out to be one of the most important capabilities that pointers provide us with.

> *Test your understanding:* In the code fragment just discussed, what would the following expressions do: 1) *carray[8]=*pc2*, and 2) **pc2=carray[1]*? Would it matter what order we did them in?

When you initially encounter the asterisk used to perform dereferencing, it may seem like too much of a good thing. After all, that same asterisk is used to declare a variable as a pointer (as well as being used for multiplication). The choice of the asterisk as a dereferencing operator was intentional, however. To understand why, consider the following declaration:

```
int i1,*i2,i3;
```

As we know by now, this declaration specifies that i1 is an integer, i2 is a pointer to an integer and i3 is an integer. You could read the declaration in another way, however. You could say we are declaring i1 to be an integer, *i2 to be an integer, and i3 to be an integer. Notice the subtle difference here: while i2 is a pointer, *i2 would, in fact, refer to an integer. Thus, it makes some sense to use the same notation for declaring pointers and dereferencing them—particularly since there are a limited number of special characters available for the language to use!

Using Brackets
We already know how to use brackets to get elements from an array. Well, good news! We can use that exact same approach to retrieve data from a pointer. In fact, every time we used brackets inside a string function in previous chapters, that is exactly what we were doing. When we pass array names into functions, C++ creates a temporary local pointer to refer to the address of the array inside the function. In fact, C++ almost never passes a copy of an entire array into a function. Why copy all that data when the address will take us to where we want to go?

> *Test your understanding:* Given that we can use array notation instead of the * operator to dereference pointers, the code fragment presented earlier could be rewritten:

```
char c1='c',carray[80]="Hello, World";
char *pc1,*pc2;
pc1=&c1;
pc2=&carray[3];
carray[2]= pc1[0];
pc1[0]= carray[0];
```

Explain the changes. How would you rewrite the lines: 1) *carray[8]=*pc2*, and 2) **pc2=carray[1]*?

The equivalence of [] and * notation will be further explored when we consider pointer arithmetic.

3.1.4: The NULL pointer

There are times when we need to indicate that a pointer points to nothing. Perhaps we want to show that it has not been initialized. Or, perhaps, we want to signal a function call was unsuccessful (such as a call to strstr() that did not find a matching substring). The problem is that once you define a variable, any variable, it always has *some* value, whether it is a valid value or not. And invalid pointer values are prodigiously dangerous, as they usually point to memory locations where your program has no business going.

To get around this problem, a constant NULL (case sensitive) has been defined. Its value is usually the address 0x00000000, but is only guaranteed to be so in C++, not C. Whenever a pointer has been assigned the value of NULL, it is assumed not to point to anything. One common example of its use is testing function return values, such as that of strstr() as described in section 3.1.2.

3.1.5: Pointers as function arguments

Using pointers as arguments to functions solved a key limitation of C functions. When we first considered functions, we noted that C functions couldn't change the values of their arguments (the exception of references, in C++, being duly noted). The explanation given was that function arguments are copied into local variables before they are used within the function. Thus, the only *direct* way to get information from a function was through its single return value. And only getting a single value back from a function is very limiting.

Passing in a pointer to a function allows us to sidestep the single return value limitation. Instead of passing in the actual variable we want to change, we pass in the address of that variable—in other words, a pointer to that variable. Although the function cannot change the value of the pointer itself (i.e., it can't change the address where the data is located), it can dereference the pointer, allowing it to change the value of the variable we were interested in.

ToUpper vs. MakeUpper
As an example, consider the Toupper() and MakeUpper() functions below:

```
char Toupper(char in) {
        if (in>='a' && in<='z') in=in-'a'+'A';
```

```
		return in;
	}
	void MakeUpper(char *pin) {
		if (*pin>='a' && *pin<='z') *pin=*pin-'a'+'A';
		return;
	}
```

Suppose, these functions were called from a code fragment specified as follows:

```
	char c1='a',c2='b',c3;
	c3=Toupper(c1);
	MakeUpper(&c2);
```

After calling the code, c1 would have the value 'a' (unchanged), c3 would have the value 'A' (from the return value of toupper) and c2 would have the value 'B' (changed by MakeUpper).

Side Effects and the const Modifier
The changing of a value pointed to by an argument of a function is sometimes called a side effect. Code that makes extensive use of side effects is often much more difficult to understand and debug than code that relies purely on return values. The reason is that when you call a function, you don't necessarily expect it to make changes to its arguments. As a result, you may use the same values passed in as arguments later on in your code, without realizing they have been changed. This problem can be particularly vexing when you are using functions you did not write, such as library functions.

One way of addressing the problem of which arguments are modified and which are not is to use the **const** modifier in front of the pointer declaration (e.g., const char *arg1 instead of char *arg1). If the **const** modifier is present, a compiler error will be produced if the function attempts to change the dereferenced value of the pointer. In general, it is good programming practice to use the **const** modifier any time you do not expect to change the value that argument points to. **const** can also modify a reference argument.

Test your understanding: What would be a better way of specifying each of the following definitions for standard C/C++ library functions that we have covered:

- int strlen(char *str)
- char *strcpy(const char *dest,char *src);
- int strcmp(char *s1,char *s2);

3.1.6: The *void* pointer and pointer type casting

Throughout this section, we have emphasized the advantages of specifying a type for every pointer we declare. There are times, however, when it is convenient to declare a pointer that can point to anything. The most common examples of situations where such general pointers are useful are:

- *Generic memory functions*. C/C++ provides a number of functions for initializing and copying the contents of memory. These memory functions can be used on all data types (including C structures, to be defined later). For that reason, their arguments are generally pointers.
- *Generic collection objects*. Advanced programming techniques, some of which are touched on in the SwapSort lab exercise of this chapter, often require that complex collections of data be created, such as lists of data elements and lookup tables. Using void pointers, it is possible to create generic functions that can be used to create and maintain these collections.

void *

When such general purpose addressing is needed, a special pointer type, void *, is available. You can directly assign any type of pointer to a void pointer without a compiler warning. For example:

```
int myarray[10];
void *parray;
parray=myarray;
```

It is up to you, the programmer, however to keep track of the type of data that is assigned to the void pointer. There is no way the program can determine that information for itself.

If you have a void pointer, you can also assign it to a typed pointer—once again, assuming you know what type of data the void pointer actually points to. Since void pointers have no inherent type, you will normally need to assign them into a typed pointer before you can do anything useful with them. To illustrate this, consider the standard library's memset function, as shown in Example 3.1.

> *Test your understanding:* Does Memset code in Example 3.1 code initialize the block from start to end, or end to start? Would this code continue to work properly on a machine where the character size was 2 bytes?

Example 3.1: the memset function

The standard library function memset takes a block of memory and initializes every byte within it to a single value. Although the actual C declaration is slightly different, the function *could* be prototyped as follows:

```
void *memset(void *addr,unsigned char c,int bytes);
```

where *addr* is the starting address of the memory we are initializing, *c* is the character we are initializing it to and *bytes* is the number of bytes we are initializing. It returns *addr* (the same address being passed in), although the return value is rarely used.

The memset function provides a convenient means of initializing large objects, such as arrays. For example, in the code fragment that follows:

```
int bigarray[10000];
memset(bigarray,0,10000*sizeof(int));
```

the call to memset initializes every byte in bigarray to 0.

Our own version of the memset function (Memset) could be defined as follows:

```
void *Memset(void *addr,unsigned char c,int bytes)
{
        unsigned char *pblock=addr;
        // the above line will generate a C++ warning, to be explained later
        while(bytes-->0) pblock[bytes]=c;
        return addr;
}
```

By assigning the void pointer *addr* to *pblock*, we are able to access every byte in the block of interest.

Typecasting pointers
When moving between pointer types, a compiler warning will often be produced. In C, which is much more flexible in this regard than C++, assigning a const void pointer to any other type of pointer (including void) that is not const will generate such a message. In C++, assignments across types (including void pointers to other pointers) will generate warnings or errors.

Test your understanding: Why would the line of code:

```
unsigned char *pblock=addr;
```

found in the memset function (example 11.4) compile without warning in C, yet lead to an error in C++?

The reason for such warnings/errors is that the compiler wants to focus your attention on the possibly dangerous thing you are doing—taking an address of one type and putting it into a pointer of another type. Since pointer-to-pointer assignments are usually warnings, not errors, a number of "avoidance" behaviors are available to you, such as ignoring the warning or turning the compiler warning settings off. The problem with the first, ignoring the warning, is that it desensitizes you (like the boy who cried wolf), so you end up ignoring all warnings—including the ones that would save you hours or days in finding code problems. The second approach, turning the warning settings off, is even worse. It's like telling your doctor not to inform you of any areas of concern found during a routine physical, just because some of those concerns might turn out to be unfounded.

So, how do you get rid of a warning when you know the code is correct? Similar to what we did when errors surfaced during numeric conversions, we can typecast the pointers. Pointer typecasting involves placing the correct pointer type (within parentheses) directly in front of the pointer being converted. Typecasting to eliminate a warning is illustrated in a sample implementation of the library function *memcpy()*.This function takes a block of memory and copies it, byte by byte, to another location. Effectively, then, it is the same as strcpy, except you specify how many bytes are to be copies instead of copying until the NUL terminator is reached. Although the actual C/C++ declaration is slightly different, the function *could* be prototyped as follows:

```
void *memcpy(void *dest,const void *src,int bytes);
```

where *dest* is the starting address of the block we are copying to, *src* is the starting address of the block of memory we are copying from, and *bytes* is the number of bytes we are copying. It is critically important, for this particular function, that the *src* and *dest* blocks of memory do not overlap. The function returns *dest* (the same address being passed in), although the return value is rarely used.

The memcpy function provides a convenient means of moving data between large objects, such as arrays. For example, in the code fragment that follows:

```
double bigarray1[10000],bigarray2[10000];
memset(bigarray1,0,10000*sizeof(double));
memcpy(bigarray2,bigarray1,10000*sizeof(double));
```

the call to memset initializes every byte in bigarray1 to 0. The call to memcpy then copies all the bytes in bigarray1 into bigarray2—effectively doing the same thing.

Our own version of the memcpy function (Memcpy) could be defined as follows:

```
void *Memcpy(void *dest,const void *src,int bytes)
{
        unsigned char *p1,*p2;
        p1=dest;
        p2=(unsigned char *)src;
        while(bytes-->0) p1[bytes]=p2[bytes];
        return dest;
}
```

The type cast, in the lines *p2=(unsigned char *)src* is required because src is defined as const, whereas p2 is not.

3.1 Section Questions

1. What is the main potential drawback of using pointers in your code?
2. One advantage of declaring pointer types is the ability to perform pointer arithmetic. What is the other advantage?
3. Is taking addresses the only use of the & operator in C/C++? If not, how do you suppose C/C++ decides what operator to use when it encounters an &?
4. What does the term dereferencing a pointer mean?
5. Explain the possible motivations for declaring a **void** pointer.

3.2: Pointer Arithmetic

Once you accept the fact that bracket notation can be used in place of the * for dereferencing pointers, the whole concept of pointer arithmetic becomes much less mysterious. We will begin the topic by examining what happens when integers are added and subtracted from pointers, then we will move on to other arithmetic operations. Finally, we will show the precise correspondence between array notation and * notation.

3.2.1: Adding and Subtracting Integers from Pointers

When we observed, in 3.1.3, that we could substitute bracket notation for pointer notation, we concluded that the following expressions are equivalent for any variable p1 declared to be a pointer:

$*p1 \equiv p1[0]$

Given this equivalence, it is natural to wonder what, if any, equivalences exist for coefficient values other than 0, such as 1, 2, 3, etc. It turns out, there is a direct equivalence that is both straightforward and elegant.

For any pointer p1, and any integer i, the following expressions are precisely equivalent:

***(p1 + i) ≡ p1[i]**

This identity is probably *the single most important thing you need to remember if you want to master pointers*. Moreover, this equivalence turns out to be mandated by the design of typical C/C++ compilers. In these compilers, whenever an expression in bracket notation is encountered, it is translated to pointer (*) notation before the expression is actually compiled. In other words, pointer notation can be viewed as the "natural" notation for address dereferencing.

> *Test your understanding:* Given that pointer notation is the natural notation for dereferencing, would you expect the following fragment of code to compile?
>
> ```
> char x,y,arrc[]="Howdy";
> x=arrc[2];
> y=2[arrc];
> ```
>
> Explain why or why not. If it would compile, what would the values of the variables x and y be after running the fragment?

Given that *(p1+i) is the same as p1[i], we can then make some deductions about the expression p1+i. Specifically, we know:

1. Since we get a value by putting an asterisk in front of it, p1+i must be an *address*.
2. Since the value at the address p1+i points to is always p1[i], the address at p1+i must be the same as &p1[i]. In other words, p1+i must point to the ith element (zero-based) of the array.

These two deductions allow us to make some further deductions:

(1)	(p1+i) ≡ &p1[i]	➔ What we deduced above
(2)	*(p1+i) ≡ p1[i]	➔ Our initial identity
(3)	&*(p1+i) ≡ &p1[i]	➔ We apply the & operator to both sides

Given (1) through (3), we can come up with two important conclusions:

- The & and * are *inverse* operators. That means that if you apply one right after the other, they cancel each other out. We can see this by comparing (1) and (3).
- The address produced by adding the integer i to an address is the address of the ith element of the array.

This, then, is pointer arithmetic in a nutshell:

Pointer Arithmetic: *If you take an address p1, then add an integer i to it, the resulting address will be the value of p1 plus i * sizeof the type of object p1 points to.*

The easiest way to illustrate pointer arithmetic is by looking at a few examples.

Example 3.2: Adding Integers to Addresses

Assume the following declarations have been made somewhere in the program:

```
char *p1,*p2;
double *d1,*d2;
float *f1,*f2;
short int *s1,*s2;
int *i1,*i2;
```

Assume, as well, that at the time we reach the following block of code in the program, p1==0x00001000,d1==0x00002000, f1==0x00003000, s1==0x00004000 and i1==0x00005000.

```
p2=p1+2;
d2=d1+2;
f2=f1+2;
s1=s1+2;
i2=i1;
i1=i1+2;
i2++;
```

After the above block of code, and assuming the normal sizes for **char** and **int**, we would find the following tests would all be true:

```
p2==0x00001002
d2==0x00002010
f2==0x00003008
s2==0x00004004
i1==0x00005008
i2==0x00005004
```

The values are arrived at because 2*1 is 2 (p2), 2*8 is 0x10 (d2), 2*4 is 8 (f2 and i1) and 2*2 is 4 (s2). The value of i2 results from the fact that i2++ is the same as writing i2=i2+1, the following the rules of pointer arithmetic.

Having determined what happens to an address when an integer is added to it, subtraction is no different. We simply take the integer we are subtracting, multiply it by the size of the data being pointed to, and subtract the result from the initial address.

Test your understanding: Suppose that p1 is a pointer to a **char**, and contains the address 0x00001000. Suppose that d1 is a pointer to a **double**, and also happens to contain the address 0x00001000. For any integer k, which of the following two statements is true:

- p1+k*sizeof(d1) refers to the same address as d1+k
- p1+k*sizeof(double) refers to the same address as d1+k

Given the values of p1 and d1 above, and if k==1, what would the values of (a) p1+k*sizeof(d1), (b) p1+k*sizeof(double), and (c) d1+k be for code compiled with Visual Studio.Net?

3.2.2: Other Arithmetic Operations on Pointers

The effects of the different pointer arithmetic operations are summarized in the table that follows.

First Argument	*Operator*	*Second Argument*	*Effect*
Address	+/-	Integer	Takes the integer times the size of the object being pointed to and adds it to (or subtracts it from) the address
Integer	+/-	Address	
Address	-	Address	Takes the difference between the two addresses, in bytes, and divides them by the size of the objects being pointed to. Can only be used on pointers of the same type, and would normally be relevant only for pointers in the same array.
Integer	-	Address	Illegal
Address	+	Address	Illegal
Address	++/--	(N/A)	Returns the value of the address, then adds/subtracts the size of the object being pointed to/from the address. Cannot be used on array names (whose address cannot be changed).
(N/A)	++/--	Address	Adds/subtracts the size of the object being pointed to/from the address and returns the resulting address. Cannot be used on array names (whose address cannot be changed).

Table 3.1: Pointer arithmetic operations

3.2: Section Questions

1. If d1 is a pointer to a **double**, and contains the address 0x00001050, what address would the expression d1-4 evaluate to?
2. Pointer arithmetic is also called address arithmetic. Why is "address arithmetic" a more accurate term, given our definition of a pointer?
3. We think of the increment operator (++) as taking its argument and adding 1 to it. Why is what happens slightly more complicated when the argument being incremented is a pointer?
4. If d1 and d2 are both pointers to double, and d1==0x00001f30 and d2=0x00001ebc, what would the expression d1-d2 return? Can you tell if these values are from the same array?
5. If arr is declared as follows: ***int arr[10]*** and arr begins at address 0x00001000 in memory, what would happen if the expression arr++ was encountered in the code?

3.3: Complex Arrays

Up to this point, we have limited our discussions to *one-dimensional arrays*. Such arrays are very powerful, and many things can be done with them. C/C++ provides even more powerful representational tools, however, that include multidimensional arrays and, even more importantly, arrays of pointers. In this section, we examine these more complex array types, and consider what they can be used for.

3.3.1: Multidimensional Arrays

A multidimensional array is an array that is declared and accessed with more than one coefficient. In programming languages that do not allow programmers access to addresses, such as FORTRAN, COBOL and BASIC, the ability to define multidimensional arrays can be a very useful capability. Although C and C++ also support the definition of multidimensional arrays, their utility is dramatically reduced by the ability to use arrays of pointers in their place. (The reason arrays of pointers are usually better than multidimensional arrays will become clearer later in this section, and when we discuss dynamic memory, in Chapter 13.)

To understand the nature of a multidimensional array, it is helpful to think visually. If we think of a variable as a point in memory, then our standard array can be visualized as a line of these points in a row. Its single coefficient represents a point on the line, and just as we refer to a line as one-dimensional, we refer to an array with one coefficient as one-dimensional. A two-dimensional array has two coefficients, usually referred to as a row and a column, and can be visualized as a table (which is, of course, a two-dimensional object). In order to reach a particular piece of data, we need to supply a row number and a column number (both zero-based in C/C++, naturally). A three-dimensional array has three coefficients, and can be visualized as three dimensional cube or rectangle. To get a piece of data in such an array, you need three numbers, representing the X, Y and Z coordinates. An illustration of these different array types is provided in Figure 3.1. Multidimensional arrays in C/C++ are not, however, limited to three dimensions. The process of adding dimensions can go on indefinitely, with as many dimensions being added as needed (although beyond three is hard to visualize and harder to draw!).

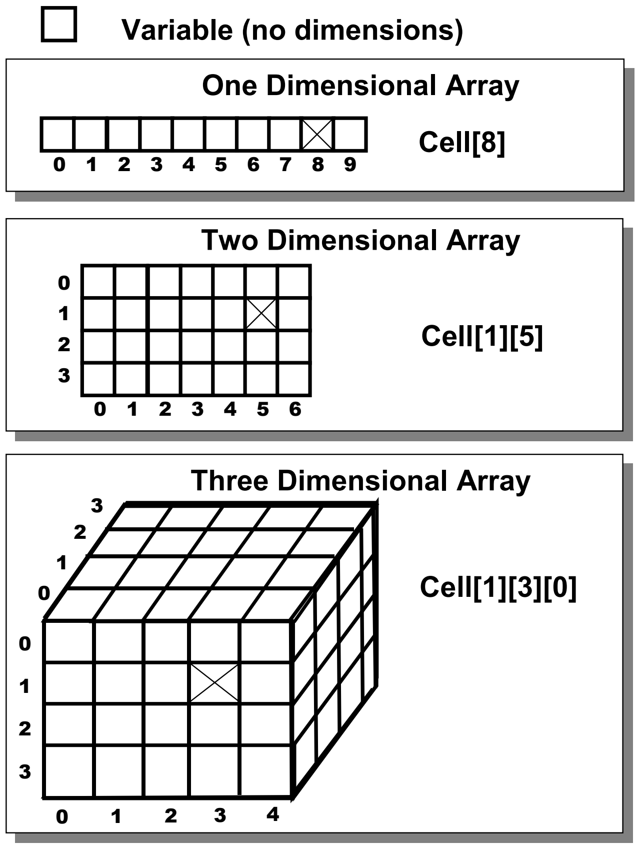


Figure 3.1: Illustration of Array Types

There are a number of characteristics of C/C++ multidimensional arrays that need to be understood before they can be used effectively:

- Arrays are declared, and coefficients accessed, using separate brackets for each coefficient (see Figure 3.1). This differs from many programming languages, where commas are often used to separate coefficients.
- Arrays are stored in row-major order. For example, if you view the first coefficient of a two dimensional array as the row, and the second as the column, then all the data for a specific row is stored before the data for the next row.
- If a reference is made to a multi-dimensional array with less than the declared number of coefficients, the reference itself refers to an array.

The first two of these characteristics are relatively straightforward. The third, however, takes a bit more explanation.

Partial coefficients in multidimensional arrays
Suppose we have a two-dimensional array defined as follows:

```
char screen[25][80];
```

Such an array might, for example, hold the contents of an old PC-based text screen (which had 25 rows and 80 columns). If we were to refer to element screen[5][17], we would be talking about the character at the 6th row, 18th column (assuming, like most humans, we start at 1 when we talk about text screen positions). But what if we just referred to screen[5]? Given that we stated that coefficients are stored in row-major order, screen[5] would seem to refer to the entire row. That, in fact, is how C/C++ interprets. And, since the row is an array of 80 characters, screen[5] is—effectively—an array name.

> *Test your understanding*: Explain why the following code is legal:
>
> ```
> char screen[25][80];
> char *pscreen;
> pscreen=screen[5];
> ```
>
> Would the assignment *pscreen=screen* also be legal? Why or why not?

Initializing multidimensional arrays
We have already seen that one-dimensional arrays can be initialized with a comma-separated list of values. For example:

```
int myarray[10]={1,4,6,7,8};
```

would initialize the first five elements of the array myarray to 1, 4, 6, 7, and 8 respectively, with the remaining 5 elements being initialized to 0.

For multidimensional arrays, the same initialization process applies, however nested brace-enclosed collections are used for the different dimensions. For example:

```
int array3[2][3][2]={{{1,2},{3,4},{5,6}},
        {{7,8},{9,10},{11,12}}};
```

Note that this order also reflects the order in which the array elements would be stored in memory.

> *Test your knowledge:* In the previous example, what would be the value of example, array3[1][2][0]? What would be the value of array3[0][1][1]? Given what you know about C/C++'s disregard of array boundaries, and how

multidimensional arrays are stored, what would you expect the value of array3[0][3][1] to be?

3.3.2: Arrays of Pointers

The array of pointers is one of the most powerful representational techniques in the C/C++ language family. Conceptually, it differs little from a normal, one-dimensional array—the difference being that the elements are pointers to data elements, instead of the actual data elements themselves. This is illustrated in Figure 3.2, where the actual array is the column of addresses on the left, while the data being referenced are the random strings on the right.

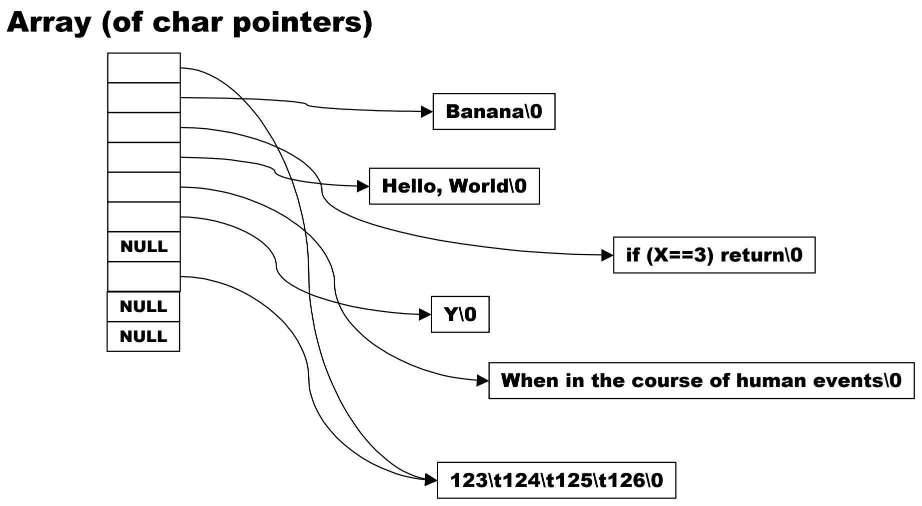


Figure 3.2: An array of character pointers

A particularly important aspect of this diagram is the fact that while the array itself is contiguous and in order, there is no reason that the memory addresses being referenced should be contiguous, or even in order. Thus, while Array[1] points to the string "Banana", and Array[2] points to the string "if (X==3) return", there is no basis for assuming anything about where the two strings ("Banana" and "if (X==3) return") are located in memory. They could be anywhere.

Declaring an array of pointers

Declaring an array of pointers, is very straightforward. You simply declare an array, then place an asterisk in front of the name. For example:

```
int *array1[10];
char *array2[4]={"Hello","World","It\'s","me"};
```

In the first example, we are declaring an array of 10 pointers to integers. In the second, we are declaring an array of 4 pointers to characters. In addition, in the second case, we are telling the compiler to create constant strings in memory, then assign each of their addresses to one of the pointers. Just as with any other array, had we left the 4 out of the declaration, the compiler would have counted the number of elements and used that count to size the array.

> *Test your understanding:* Suppose the second declaration above had been specified as follows:
>
> ```
>     char *array2[10]= {"Hello","World","It\'s","me"};
> ```
>
> What do you suppose would have happened, given your understanding of how arrays are initialized?

Passing an array of pointers as an argument to a function
As with any C/C++ data object, the ability to pass it as an argument to a function is critical. Since C and C++ do not normally make copies of array arguments, it makes sense that we will use a pointer. But how will that pointer be declared?

To understand the declaration of a pointer to an array of pointers, it is useful to return to how we handled an array. For example:

```
char array1[80];
char *parr1=array1;
```

If the same logic were to hold, it would then appear that:

```
char *array2[80];
char **parr2=array2;
```

This was accomplished simply by placing an extra asterisk in front of both of the original expression. It also turns out that this is precisely how we create a pointer to an array of pointers (or to a multidimensional array).

Example 3.3: Linear Search

Suppose we wanted to write a search routine that looked for a matching string in an array of pointers, say days of the week, and returned the array coefficient of the matching element (or –1 if there was no match). For example, our array might be declared as follows:

```
    char *days[]={"Monday","Tuesday","Wednesday","Thursday","Friday",
            "Saturday","Sunday"};
```

Our function could be declared as follows:

```
    int FindDay(const char *day,const char **dayarray);
```

We could then write the function as follows:

```
    int FindDay(const char *day,const char **dayarray)
    {
            int i;
            for(i=0;i<7 && strcmp(dayarray[i],day)!=0;i++) {};
            if (i==0) return –1;
            else return i;
    }
```

Test your understanding: In Example 3.3, what advantages are there to having a function such as FindDay() take the array to be searched as an argument, as opposed to making it a static global array? (Hint: think flexibility and designing code for the international environment). What modifications to this function would be required to make it suitable for searching any array of strings?

3.3.3 Higher order arrays of pointers

Adding even more flexibility to the C/C++ language is the ability to declare multilevel arrays of pointers—that is, arrays of pointers that point to other arrays of pointers. While such complexity takes a while to get used to, it does offer an extremely powerful tool for representing data.

Conceptually, arrays of arrays of pointers look something like the illustration in Figure 3.3. The initial array contains pointers that point to other arrays of pointers which, in turn, point to character strings. The declaration of the array in Figure 3.3 would be something like:

```
char **Plants[5];
```

where the 5 represents the number of elements in the first (leftmost) array. The remaining arrays and the strings would then need to be established somewhere else. (Normally such arrays would be created using dynamic memory, discussed in Chapter 12.)

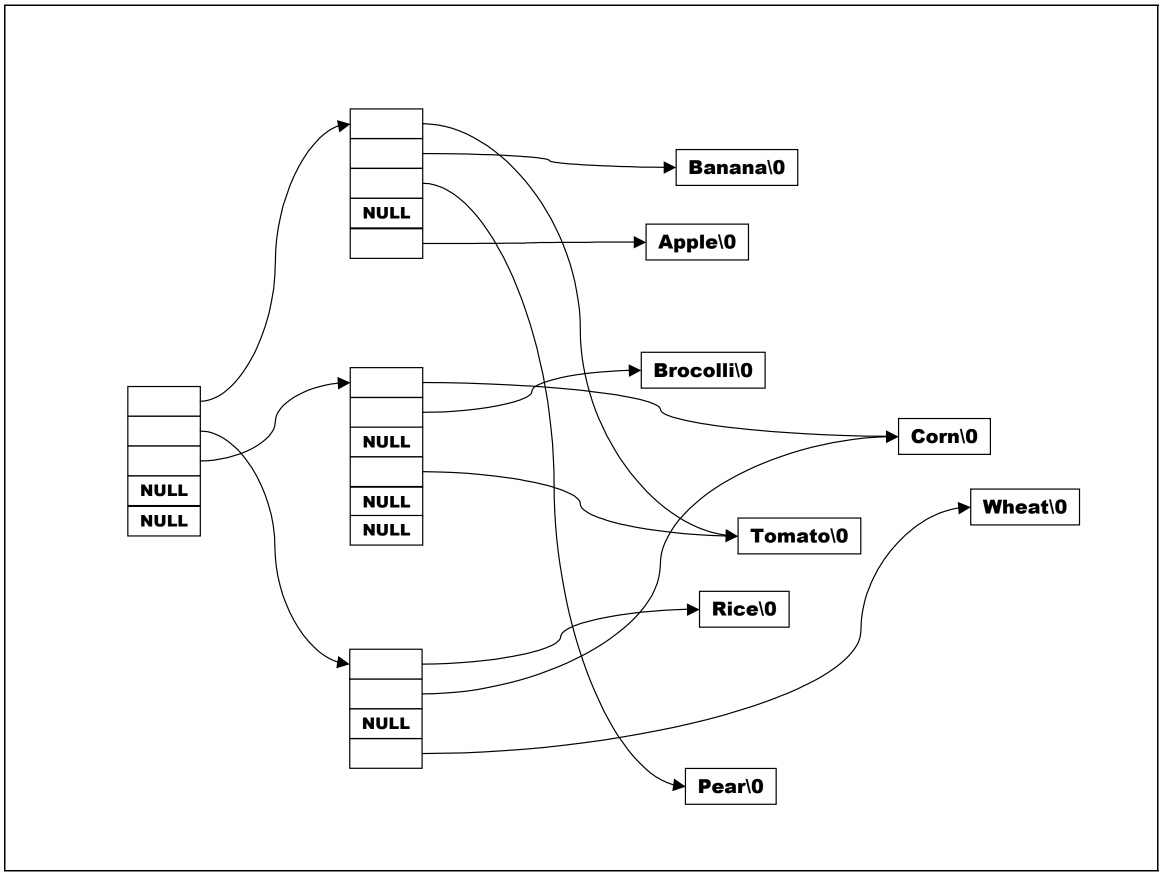


Figure 3.3: An array of arrays of pointers

Following the same line of logic used for single dimension arrays of pointers, if we needed to pass it into a function, or declare a pointer to it, we'd simply have to add an additional asterisk. For example, the following fragment of code creates a pointer to the Figure 3.3 array.

```
char **Plants[5];
char ***pplant=Plants;
```

There is no mechanical limit to the number of levels of indirection (e.g., pointers to pointers to pointers to pointers, etc.) that can be declared. From a practical standpoint, however, more than three asterisks tend to numb the mind of even experienced programmers. When greater depth of indirection is needed, other representational techniques (such as C structures and C++ classes) are usually used. (We see how arrays and other objects can be incorporated into structures in Chapter 12)

3.3: Section Questions

1. Given your understanding of multidimensional arrays and their initialization, would the following declaration work:

   ```
   char screen[25][80]={"Hello","world","I\'m","back."};
   ```

If it would work, what would it do?

2. When declaring a multidimensional array, and leaving the compiler to determine the size of the coefficient (e.g., similar to int test[]={1,2,3}; for single dimensional arrays), what coefficient would it make sense to leave empty, the left or the right?
3. Why might arrays of pointers usually be preferable to multidimensional arrays when dealing with "real world" data?

3.4: Introduction to Structures

A structure is a construct used for organizing data. Used effectively, structures can help the programmer manage the complexity of applications as they grow ever larger. In this section, we look at the nature of a structure from a conceptual perspective, contrasting it with an object. Subsequent sections detail the declaration and use of structures in C/C++.

3.4.1: Nature of a structure

Up to this point in the text, we have learned a lot of ways of representing information within our programs. Using the C/C++ fundamental data types, along with arrays and pointers, we can—with a little bit of work and creativity—come up with our own representations for most pieces of data we commonly encounter.

Although we may celebrate how far we have come, we should also feel a certain uneasiness, knowing that something is missing. Consider, for example, the most common method of accessing and storing information on computers: the data base table. All of us have worked with tables, such as the example in Table 3.2. But how would we represent such a table in a program?

Last Name	*First Name*	*Gross Pay*	*Fica*	*US W/H*	*State W/H*	*Insurance*
Washington	George	3000	180	450	150	175
Adams	Abigail	2500	150	375	125	175
Jefferson	Tom	4000	240	600	200	175
Madison	Dolly	2000	120	200	67	125
Madison	Jimmy	1500	90	150	50	125
etc..						

Table 3.2: Typical database table

Given the techniques we have already covered, our best approach would probably be to create a series of parallel arrays (or arrays of pointers), with one array for each column. For a simple 7-column table, that might be manageable. But what about the 97-column table we use to hold personnel data on an individual employee? Or the 43-column table we use to hold product information? Or the 143-column table that we use to save and categorize sales leads? Quickly, we see the limitations of parallel arrays. In real world situations they are a pain in the neck to use (and the author has heard even lower opinions expressed).

But, you might counter, why not just leave databases to applications such as SQL and Access? After all, why should we bring tables of data into our programs when we've got perfectly good tools to manage them? This is an excellent point. It does not, however, solve our problem. We're going to need to bring information from our database into our program if we want to use it—even if we limit ourselves to one row at a time. But here's the rub: even bringing the data from one row (record) of our above-mentioned sales-lead table would require that we declare and name 143 different data elements. And we can't use an array as a way to avoid assigning names, since the data comes in different types (e.g., integers, real numbers, strings, dates, times, Booleans). And think about trying to pass the information about a single sales lead to a function. How would you like to work with a function that had 143 arguments?

Conceptually, then, our problem is clear. Although arrays give us the ability to represent columns of a table nicely, we need some other tool to represent the rows of table. That is the role played by a structure.

> *Definition:* A structure is a construct for representing a collection of data elements all related to a common entity.

The practical effect of the above definition is that when we define a structure, we are defining a compound data type. Each of the individual *data elements* (sometimes called *data members* or *data fields*) is tied together by a common thread—they are all associated with some common theme—technically referred to as an *entity*. In the case of our database example, the theme or entity is what the row signifies (in the case of Table 3.2, for example, the entity might be the bi-weekly pay data for a given employee).

Two other terms are useful in referring to structures:

- A *structure definition* is a description of the data and organization used in a particular structure. In our database example, it is the equivalent of a table definition—specifying the name, type and order of the columns in our table.
- An *object* or *instance* of a given structure is an actual collection of data elements, organized according to our structure definition, that is related to a specific entity. In our database example, our object or instance would be the complete contents of an actual row in the table.

3.4.2: Structures vs. Objects

By using the term *object* to refer to a collection of data organized according to a structure definition, we seem to be implying that structures are somehow related to OOP. In fact, when we use structures within our C/C++ programs, we are engaging in a primitive form of OOP. Indeed, this text intentionally tries to apply structures in its examples in ways that will naturally lead to the development of the OOP philosophy.

So how are structure-objects and OOP-objects different? We go into this issue in greater detail in Chapter 14, when we introduce C++ objects. The biggest difference is relatively easy to understand, however:

> Structure objects are collections of data, OOP-objects are collections of data *and functions*.

What does it mean to have a function associated with an object? To get a basic idea, return to the example in Table 3.2. This table looks like a simplified version of the information used to create a paycheck. There is a piece of data missing however: the actual amount of the paycheck. One reason that information might not be stored is simple: it can be calculated with a simple formula:

> NetPay=GrossPay-FICA-US W/H-StateW/H-Insurance

Not only does using the formula save us some storage space, it also eliminates the possibility that someone might get into our data and modify the NetPay amount (i..e., the amount on the paycheck)—making it inconsistent with the other values.

Now suppose you are a programmer working on an application that uses a "Pay" structure such as that represented by the columns of Table 3.2.

- If you do not know how to compute net pay, and your Pay object does not have net pay explicitly included as a data element, then you're at an impasse.
- If the Pay structure were a true object, on the other hand, part of the object definition could include a function that would return the value of NetPay. When applied to an object, that function would cause the above formula to be computed using the data values contained within that object (e.g., the field values for a

specific row). Furthermore, you—as the user of the object—wouldn't need to know the nature of the formula being applied, or even if the result came from a formula or was just the return of existing data.

In other words, when using a true OOP object, the user of that object can often ignore the complexities of what data is being stored within the object and how it's being stored. When we are dealing with structures, however, we do not have that luxury.

3.4 Section Questions

1. Explain how a structure serves a purpose similar to an array
2. How does a true object (in the sense of OOP) extend the notion of a C structure?
3. Can data be associated with a C/C++ structure definition?

3.5: Declaring a Structure

In this section, we learn how to create a structure definition in C/C++. We also learn how one structure definition can include other structure elements, a process called composition. Finally we present a simple structure declaration sample.

3.5.1: The struct Construct: Defining a structure

Declaring a structure definition involves placing a series of standard data declarations within a struct construct. The basic framework is as follows:

```
struct structure-name {
        standard-data-declarations
};
```

It should be noted that the semicolon at the end of the definition is important—as we shall find out later.

For example, if we wanted to declare a structure to hold some basic information about individual employees in a firm, it might look like the following:

```
struct employee {
        unsigned int nId;
        char szLastName[30];
        char szFirstName[30];
        unsigned int nYearHired;
        double dSalary;
        bool bActive;
};
```

It is important to note that a construct in this form specifies the *structure definition*—it does not create any actual objects. Such definitions are normally included in header files, and perform the same basic purpose as two other types of constructs we've seen in such files:

- Function prototypes
- **extern** variable declarations

3.5.2: Declaring a structure object

Once a structure has been defined, there are two ways of creating actual objects of that structure type. One is associated with the definition itself, the other can be done anywhere.

Combining definition and object creation
One way to declare an object is to place an object name after the right brace, but before the semicolon, in the structure. For example:

```
struct employee {
        unsigned int nId;
        char szLastName[30];
        char szFirstName[30];
        unsigned int nYearHired;
        double dSalary;
        bool bActive;
} empSmith, empJones;
```

This code creates two objects, empSmith and empJones, both organized according to the employee structure. To describe them, we'd normally say that empSmith and empJones are both employee objects.

It is fairly unusual to declare objects in conjunction with the structure definition. The problem is that such definitions are normally placed in header files (as already noted) so that different source files in a project can share them. As soon as you share a file that creates an object, however, you'll get one of those pesky "already defined" linker errors.

In fact, the ability to declare objects before the semicolon in a structure definition mainly turns out to be an annoyance. It is one of the few places in C/C++ where you have to have a semicolon after a closing brace. On those occasions when you forget the semicolon—and it is nearly impossible not to forget it every once in a while—you'll be greeted by hundreds of compiler errors. These are caused by the fact that until a semicolon is found, the compiler assumes that everything following your structure is an object name—including function prototypes, variable declarations, preprocessor instructions, function definitions, other structure definitions, you name it…

Thus, if you get hundreds of unexpected errors compiling a simple program, always check your structure definitions for ending semicolons before wasting a lot of time on trying to interpret the errors.

Creating objects after the declaration
Fortunately, there is a much more convenient way to create objects. Once the compiler has processed the structure definition (e.g., by loading the .h file), you can define objects at any subsequent time just the way you define other variables. This is done by writing:

struct *structure-name object-name*;

For example, the same two structure objects created earlier could also be created as follows:

```
struct employee empSmith,empJones;
```

In C++, the **struct** keyword can be omitted, although we will continue to use it for consistency.

Once a structure object has been created, you can perform certain operations on it as if it were a normal data type. Specifically:

- You can assign it to another object of the same type (e.g., empSmith=empJones;)
- You can pass it into a function—in this case, a local copy is created just like any other C function argument (unless it is passed by reference in C++)
- You can return it from a function

On the other hand, there are certain things you can't do with structure objects that you can do with other primitive data types:

- You can't apply test operators to compare them (e.g., empSmith<empJones is illegal)
- You can't apply arithmetic operators to them—or any other operator except assignment (e.g., empSmith+empJones is illegal)

All of these rules can cease to apply when structure objects are extended to become OOP objects in C++. Indeed, the internal mechanisms for handling structures in C++ can be vastly different from those of C. As long as the programmer sticks with C syntax for structures when defining them in C++, however, the above rules remain valid.

Initializing Structure Objects
When a structure object is being created, it can be initialized using a process similar to that already discussed for arrays. Braces {} are used to surround the list, and the individual items in the list are used to initialize the elements in the order they are declared. Thus:

```
struct employee {
    unsigned int nId;
    char szLastName[30];
    char szFirstName[30];
    unsigned int nYearHired;
    double dSalary;
    bool bActive;
};

struct employee empSmith={101,"Smith","Joan",1999,100000,true};
```

would initialize the nId member to 101, the szLastName member to "Smith", etc.

In doing initialization, two important rules apply:

- The item on the initialization list must be appropriate for the data type of the member being initialized.
- If not all elements are initialized (e.g., there are few items on the list than there are in the structure), remaining elements of the object are initialized to the zero-equivalent (just as they are for array initializations).

It is important to note that you cannot initialize declarations within the structure definition itself. e.g., it is illegal to write:

```
struct bad_struct {
    char szName[80]="Smith"; // illegal
    double dPay=100000; // also illegal
};
```

The reason is that when you are defining a structure, there is no memory being set aside to initialize. It is only once an object has been created that you have a chunk of memory that you can initialize with values.

3.5.3: Composing structures

Composition is the ability to embed structures within other structures. To get the basic idea, suppose you were an automobile manufacturer trying to create a data structure to summarize the information about an automobile. You might discover—as you went around to the various departments—that a number of structures were already used in existing programs, such as less simplistic versions of the ones shown in Example 3.4.

Example 3.4: Example Automobile Structures

```
struct engine {
	unsigned int nCylinders;
	double dVolume;
	double dTorque;
	char szManufacturer[40];
};

struct body {
	char szColor[20];
	unsigned int nWheelbase;
	unsigned int nCubicCargo;
};

struct interior {
	bool bLeather;
	bool bDigitalConsole;
	bool bCDPlayer;
	char szColor[20];
};

struct epa_safety {
	double dMPG_city;
	double dMPG_highway;
	bool bSideAirbags;
	unsigned int nSafetyRating;
};
```

Naturally, you'd like your "car" structure to contain all the information in these various structures. So how do you accomplish this?

Composition

C/C++ provides a very simple mechanism for embedding one structure within another, called composition. To compose a structure within another structure, you just declare a structure object within your main definition. For example, we could create our car structure as follows:

```
struct car {
        unsigned int nSerialNo;
        struct engine e;
        struct body b;
        struct interior in;
        struct epa_safety safe;
};
```

In c defining the car structure in this manner, members for all the data from the other structures are created whenever we create a car object.

In order to compose a structure within another structure, the embedded structure:

- Must have been previously defined
- Must not be the same as the outer structure. This would be the memory equivalent of standing between two mirrors facing each other.

It is also possible to actually define structures within other structures, both with and without names, to create compositions. Since we will have little need for these capabilities, we will not present them.

Initializing Composed Structures
Initializing structures with embedded structures requires us to use initialization lists embedded within our main initialization list. For example:

```
struct car mycar= {10210, //nSerialNo
        {6,4.1},                                // engine initializers
        {"Hunter Green",81,183},                // body initilizers
        {false,false}                           // interior initializers
};                              // we stopped before initializing epa_safety
```

As was the case with other initializations, any elements not initialized are set to their zero-equivalents.

Alternatives to Composition
It is possible to use data from one structure in another without composition. The most obvious way is to declare a pointer with the structure. For example, the a revised version of the car structure could be declared as follows:

```
struct car_pointer {
        unsigned int nSerialNo;
        struct engine *pe;
        struct body *pb;
        struct interior *pin;
        struct epa_safety *psafe;
};
```

Declared in this way, we could create a car_pointer object as follows:

```
struct engine e1={6,4.1};
struct body b1={"Hunter Green",81,183};
struct interior in1={false,false};
struct epa_safety epa1={0.00};
struct car_pointer mycarptr={10210,&e1,&b1,&in1,&epa1};
```

The main differences between using composition and a pointer to another structure are as follows:

- When structures are composed, the life of the embedded structure is identical to that of the main structure
- When a pointer to a structure is used, changes made to the referenced structure (e.g., updates to values) are immediately available

If a particular object cannot exist outside of the main structure—such as a structure specifying something unique to the object being declared, such as a multi-part VIN number structure—it makes sense to compose it. If, on the other hand, the object has its own existence, such as a radio (that could be replaced with another radio at the customer's request), a pointer to a separate structure may make sense.

3.5 Section Questions

1. Why do we usually declare objects in a different place from where we define structures?

2. Why can't we compose a structure with itself?

3. Why can we assign (i.e., =), but not test (e.g., <, >) or perform arithmetic (e.g., +, -) on structure objects?

4. Should a structure be able to point to another structure of the same type. For example, should the following definition be allowable?

```
        struct test {
                int nVal;
                struct test *pTest;
```

```
            };
```

3.6: Accessing Data in a Structure

Now that we've discussed how to create objects and initialize them, we need to consider how to make them useful—and that means being able to get data into them and out of them.

3.6.1: Accessing data in a structure object

Once we have a structure object, it is relatively rare that we'd be interested in dealing with the entire object at once. Much more likely, we are going to be interested in accessing or setting individual data elements within our object. For this purpose, the period (.) operator is defined.

Accessing Top-Level Elements
The basic structure for accessing a data element within a scalar object is as follows:

object-name . member-name

For example, if empSmith is an employee object (as defined in Section 3.5.1), then:

```
empSmith.nId
```

refers to the specific value of the nId member of the empSmith object. Similarly, the expression:

```
empSmith.szLastName
```

refers to the starting address of the szLastName[] array, which is part of the structure. On the other hand:

```
empSmith.szLastName[0]
```

refers to the first character in that array.

The values of object elements that we access are lvalues, just like array elements. That means that they can be used on the right and left hand side of assignment statements. For example:

```
int nVal=empSmith.nId;                  // Places Smith's ID in nVal
empSmith.nId=105;                       // Sets Smith's ID to 105
strcpy(empSmith.szLastName,"Smyth.");   // Set's Smith's last name to "Smyth"
```

Accessing Composed Elements

Where structures are composed, we refer to the name of the composed object as a member, then use another dot to refer to the element within that embedded object. For example, using the definition of the car structure from Section 3.5.3:

```
struct car mycar= {10210,{6,4.1}, {"Hunter Green",81,183}, {false,false}};
int nWheel=mycar.b.nWheelbase;
// nWheel is now 81
```

To understand the reasoning behind the expression:

- We first look at the car structure and find that b refers to an embedded body object.
- This object is the third member element in the car structure, so it is initialized by the third element in the initialization list ➔ {"Hunter Green",81,183}
- Looking up the definition of the body structure (in Example 3.4), we find that the nWheelbase member of that structure is the second element within the body structure, meaning it was initialized by the second element in our initialization list—which is 81.

Test your understanding: Identify the values of the following expressions, given the code fragment above:

```
mycar.nSerialNo
strlen(mycar.b.szColor);
strlen(mycar.in.szColor);
mycar.e.dVolume;
```

3.6.2: Accessing data from a structure pointer

The procedure for accessing element data using a structure pointer is very similar to that used for accessing data from an object. The only difference is that we use the -> operator, instead of the dot.

Accessing Top-Level Elements

The basic form for accessing elements using a pointer to a structure is as follows:

structure-pointer -> *member-name*

For example:

```
struct employee empSmith={101,"Smith","Joan",1999,100000,true};
struct employee *pEmp=&empSmith;
int nYear=pEmp->nYearHired;
```

```
// nYear is now 1999
int nLen=strlen(pEmp->szFirstName);
// nLen is now 4
```

Notice, the fact that we use the -> operator with a pointer on the left has no effect on the value we are accessing. In fact, all four of the following expressions are *exactly* equivalent given the code above:

empSmith.nId ≡ pEmp->nId ≡ (&empSmith)->nId ≡ (*pEmp).nId

They all refer to the same precise value in memory. The only thing that determines whether or not you need a dot or an arrow is whether or not you have an object (dot) or a pointer (arrow) on the left.

And… some additional good news is that the compiler message you get when you use the wrong operator is pretty easy to understand.

Accessing Composed Elements

When you have a pointer to a structure, accessing composed elements is exactly the same as when you have an object itself, except your left-hand operator is the arrow (->). If the element you are referencing is a pointer to a structure itself (as it was in our definition of the car_pointer structure, in Section 3.5.3, you'll need to use the arrow operator to get element values from the object pointed to. These various combinations are presented in Example 3.5.

Example 3.5: Accessing elements in embedded objects and pointers

```
struct engine e1={6,4.1};
struct body b1={"Hunter Green",81,183};
struct interior in1={false,false};
struct epa_safety epa1={0.00};
struct car_pointer mycarptr={10210,&e1,&b1,&in1,&epa1};
struct car mycar= {10210,      //nSerialNo
      {6,4.1},                 // engine initializers
      {"Hunter Green",81,183}, // body initilizers
      {false,false},     // interior initializers
};
struct car *pCar=&mycar;
struct car_pointer *pCptr=&mycarptr;
// mycar is object, b is embedded object
int nWheel0=mycar.b.nWheelbase;
// pCar is pointer, b is embedded object
int nWheel1=pCar->b.nWheelbase;
// mycarptr is object, pb is embedded pointer to object
int nWheel2=mycarptr.pb->nWheelbase;
// pCptr is pointer, pb is embedded pointer to object
int nWheel3=pCptr->pb->nWheelbase;
```

Although it initially seems complicated, the same rule applies anywhere in an expression:

- If what's to the left of the operator is an actual object –or an expression equivalent to an object, such as (*object-pointer)—the dot operator is used
- If what's to the left evaluates to the address of an object, the arrow (->) is used.

In both cases, it is the value of the data element that is returned by the expression.

3.6.3: Employee Data Input and Output

Walkthrough available in SimpleEmp.avi

Our SimpleEmp demonstration uses the same employee structure defined in 3.5.2 (see Example 3.6). In addition, it defines three functions:

void EmployeeTest(): a function that initializes, displays, edits, then redisplays an employee object, to demonstrate the effect of the two functions for displaying and editing.

void DisplayEmp(struct employee emp): Displays the values of the employee structure passed as an argument to the console.

*void InputEmp(struct employee *pEmp)*: Prompts the user to change each element in the employee object pointed to by pEmp. A pointer is passed to allow changes to the actual object.

Example 3.6: employee Structure

```
#define MAXNAME 30

struct employee {
	unsigned int nId;
	char szLastName[MAXNAME];
	char szFirstName[MAXNAME];
	unsigned int nYearHired;
	double dSalary;
	bool bActive;
};
```

The three functions used in the application are presented in Example 3.7. The DisplayEmp() function is a straightforward example of how data values can be extracted from a structure object. Because the emp object is passed by value, the function operates using a temporary copy of the object. Any changes we were to make to the object inside the function would be lost when the function returns, and the object's memory is reclaimed by the stack.

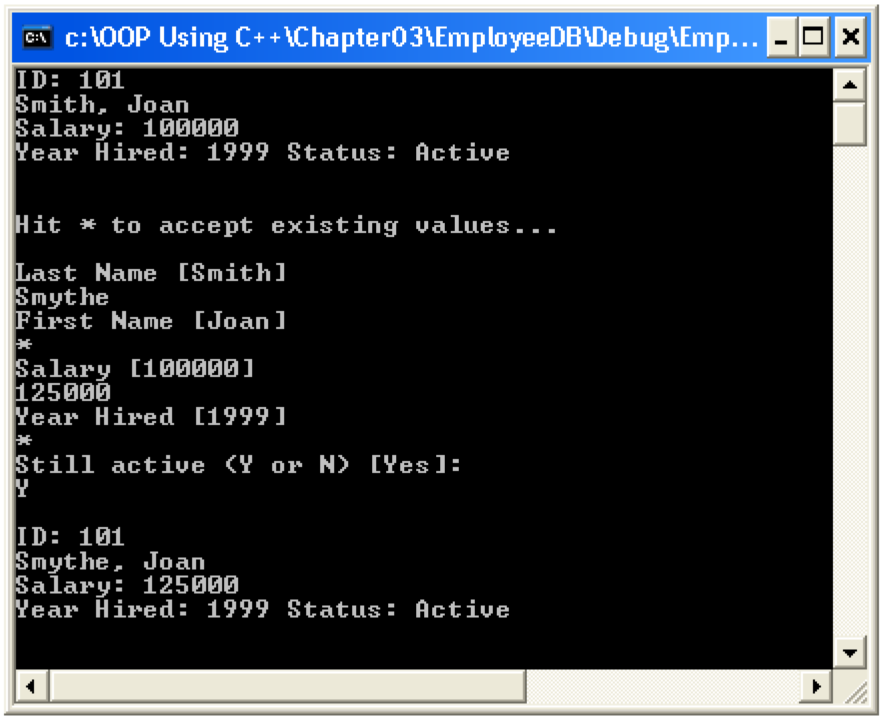
The InputEmp() is a bit more interesting. Among its key features:

- A pointer is passed to allow us to make changes that impact the empSmith object in the calling function.
- No matter what type of data is required for each element, the user input is taken as a string that goes into a temporary buffer (buf). There are two reasons for this design choice:
    - By checking if the string was empty, we can allow the original (default) value to be retained. This is as close to a "user friendly" interface as we can come in a console application.
    - Our two strings in the employee structure (szLastName and szFirstName) are both a lot shorter than the width of the screen. That causes a potential concern: the user might type in something longer than they could handle. Using a buffer, we can copy the values into szLastName and szFirstName limiting their size to MAXNAME. This way, no matter how much the user types, we'll still be safe.

Figure 3.4 shows the console display of a test session with EmployeeTest().

Example 3.7: Functions in the SimpleEmp application

```
void EmployeeTest()
{
      struct employee empSmith={101,"Smith","Joan",1999,100000,true};
      DisplayEmp(empSmith);
      cout << endl;
      InputEmp(&empSmith);
      cout << endl;
      DisplayEmp(empSmith);
}

void DisplayEmp(struct employee emp)
{
      cout << "ID: " << emp.nId << endl;
      cout << emp.szLastName << ", " << emp.szFirstName << endl;
      cout << "Salary: " << emp.dSalary << endl;
      cout << "Year Hired: " << emp.nYearHired << " Status: " <<
            ((emp.bActive) ? "Active" : "Former") << endl;
}

void InputEmp(struct employee *pEmp)
{
      char buf[1024];
      cout << "\nHit * to accept existing values...\n\n";
      cout << "Last Name [" << pEmp->szLastName << "]" << endl;
      cin >> buf;
      if (buf[0]!='*')strncpy(pEmp->szLastName,buf,MAXNAME);
      cout << "First Name [" << pEmp->szFirstName << "]" << endl;
      cin >> buf;
      if (buf[0]!='*')strncpy(pEmp->szFirstName,buf,MAXNAME);
      cout << "Salary [" << pEmp->dSalary << "]" << endl;
      cin >> buf;
      if (buf[0]!='*') pEmp->dSalary=atof(buf);
      cout << "Year Hired [" << pEmp->nYearHired << "]" << endl;
      cin >> buf;
      if (buf[0]!='*') pEmp->nYearHired=atoi(buf);
      cout << "Still active (Y or N) ["
            << ((pEmp->bActive) ? "Yes" : "No") << "]: " << endl;
      cin >> buf;
      if (buf[0]!='*')pEmp->bActive=(buf[0]=='y' || buf[0]=='Y');
}
```



```
ID: 101
Smith, Joan
Salary: 100000
Year Hired: 1999 Status: Active


Hit * to accept existing values...

Last Name [Smith]
Smythe
First Name [Joan]
*
Salary [100000]
125000
Year Hired [1999]
*
Still active (Y or N) [Yes]:
Y

ID: 101
Smythe, Joan
Salary: 125000
Year Hired: 1999 Status: Active
```

Figure 3.4: Output of EmployeeTest() function

3.6 Section Questions

1. Why is mixing up the . and -> operator to access data inside a structure usually not too great a problem?

2. If pEmp is a pointer to an employee structure, what are two alternative notations you could use to get the last name string out of it that would use the . (instead of the ->) operator?

3. Why do we pass an object into DisplayEmp() but a pointer into InputEmp() in Example 3.7. If we wanted to pass a pointer into DisplayEmp() could we do so? If yes, what would be the best way to prototype the function?

4. In Example 3.5, we see two very different ways of accessing wheelbase information:

   pCar->b.nWheelbase
   mycarptr.pb->nWheebase

   Explain the difference between the two accesses.

3.7: Arrays of Structures

While an individual structure would allow us to represent a row of a table, such as Table 3.2, we need to represent a collection of structures if we want to manage the whole table. Up to this point, the technique we have used for representing such collections is an array.

Fortunately, arrays of structures and pointers to structures are treated exactly the same as arrays of fundamental data types. Since structures present the additional issue of member access, however, it is useful to present some examples.

3.7.1: Arrays of structures

Suppose, instead of a single employee, you wanted to hold the data for 100 employees in your application. This could be accomplished by declaring an array of employee objects, such as:

```
struct employee arEmp[100];
```

Accessing Array Element Values
Once declared in this fashion, individual elements of individual objects can be accessed. For example, the ID of the 2nd element (3rd employee) would be accessed using:

```
arEmp[2].nId
```

Since element accesses are lvalues, this could appear on either the left or right hand side of an assignment operator.

Array names of arrays of structures are also treated the same as any other array name—they refer to the address where the array starts. That means the following two expressions refer to the identical value in memory:

```
arEmp->nId ≡ arEmp[0].nId
```

Also, like any other array name, you can't change the address where the array starts.

> *Test your understanding:* Why is the arrow (->) operator used in arEmp->nId while the dot (.) operator is used in arEmp[0].nId?

Initializing Array Element Values
All the rules for initializing arrays and structures that we have already applied can be used to initialize arrays of structures. Specifically, the array initialization is accomplished by a list of comma-separated elements within braces {}. Because the elements are

structures themselves, they are also initialized within nested braces. For example, we could initialize the first four elements of our array of employees as follows:

```
        struct employee arEmp[100]={
                {101,"Smith","Joan",1999,100000,true},
                {102,"Brown","Franklin",2001,60000,false},
                {103,"Green","Mary",1988,45000,true},
                {111,"Johnson","Jeremy",2002,75000,true},
        };
```

Naturally, if composed objects were present, additional nesting levels would be required within the structure braces.

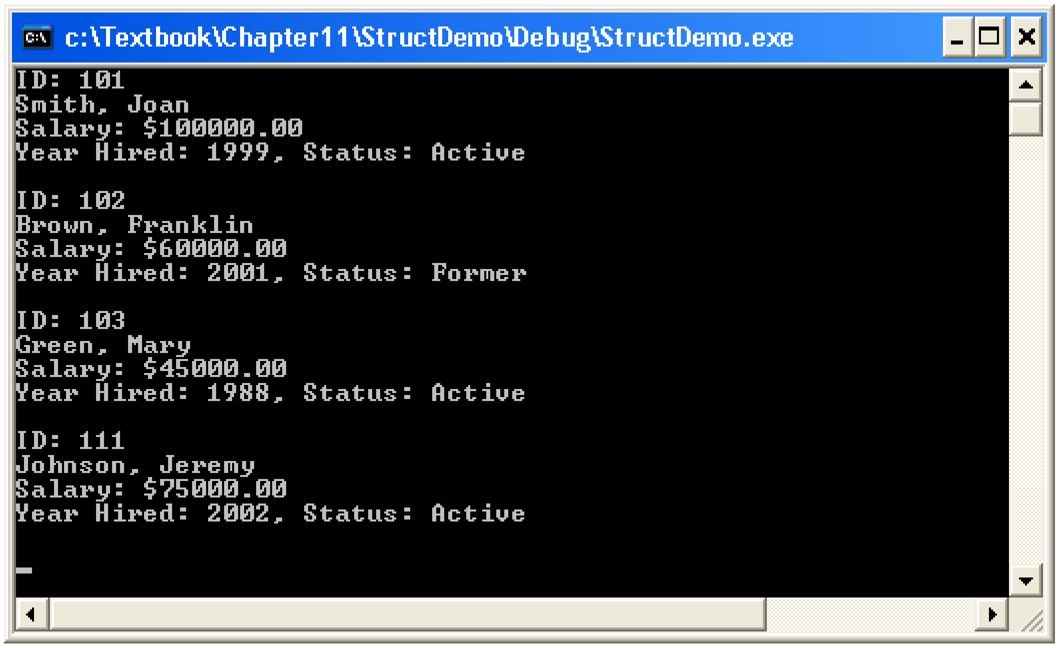
Structure Array Sample

A sample routine, demonstrating a structure array (and the use of the DisplayEmp()) function from Example 3.7), is presented in Example 3.8.

Example 3.8: EmployeeArrayTest()

```
void EmployeeArrayTest()
{
      int i;
      struct employee arEmp[100]={
            {101,"Smith","Joan",1999,100000,true},
            {102,"Brown","Franklin",2001,60000,false},
            {103,"Green","Mary",1988,45000,true},
            {111,"Johnson","Jeremy",2002,75000,true},
      };
      for(i=0;i<4;i++) {
            DisplayEmp(arEmp[i]);
            NewLine();
      }
}
```

The console output produced by running the function is presented in Figure 3.5.



```
c:\Textbook\Chapter11\StructDemo\Debug\StructDemo.exe
ID: 101
Smith, Joan
Salary: $100000.00
Year Hired: 1999, Status: Active

ID: 102
Brown, Franklin
Salary: $60000.00
Year Hired: 2001, Status: Former

ID: 103
Green, Mary
Salary: $45000.00
Year Hired: 1988, Status: Active

ID: 111
Johnson, Jeremy
Salary: $75000.00
Year Hired: 2002, Status: Active

_
```

Figure 3.5: Output from EmployeeArrayTest() function

3.7.2: Arrays of structure pointers

Our concept of an array of pointers also extends to arrays of structure pointers. For example, we could declare an array of 100 pointers to employee objects as follows:

    struct employee *arpEmp[100];

Because the elements of an array declared in this fashion are pointers, not structure objects, to access individual member values, the arrow (->) operator needs to be used. For example, to access the ID for the structures pointed to by the second pointer in the array (array element 1) , we would use the following expression:

    arpEmp[1]->nId

Since our array name, defined as above, is actually a pointer to a pointer, the following two expressions are equivalent:

    (*arpEmp)->nId ≡ arpEmp[0]->nId

This is precisely consistent with the rules we established for dereferencing pointers in Chapter 11.

As we also mentioned in Chapter 11, a big advantage of arrays of pointers over arrays of structures occurs when we want to sort them. Example 3.9 creates a 4-element array of pointers and demonstrates how it can be used to order the display of existing structures.

Example 3.9: EmployeePointerArrayTest() function

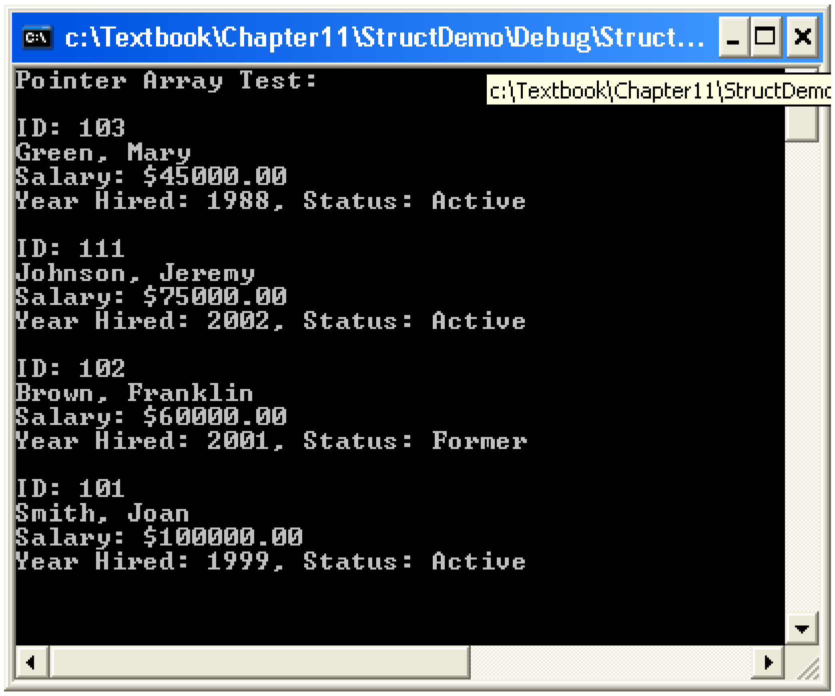
```
void EmployeePointerArrayTest()
{
      int i;
      struct employee arEmp[100]={
            {101,"Smith","Joan",1999,100000,true},
            {102,"Brown","Franklin",2001,60000,false},
            {103,"Green","Mary",1988,45000,true},
            {111,"Johnson","Jeremy",2002,75000,true},
      };
      struct employee *arPtrEmp[4]={arEmp+2,arEmp+3,arEmp+1,arEmp};
      cout << "Pointer Array Test:\n\n";
      for(i=0;i<4;i++) {
            DisplayEmp(*arPtrEmp[i]);
            cout << endl;
      }
}
```

The highlights of how the function works are as follows:

- We declare and initialize an array, arEmp, exactly the same as was done in Example 3.8.
- We create a 4-element array of pointers, arPtrEmp. We initialize it using structures in arEmp (pointer arithmetic is used to get their address), in a different order. Specifically, the pointers refer to:
    - arPtrEmp[0] points to arEmp[2]
    - arPtrEmp[1] points to arEmp[3]
    - arPtrEmp[2] points to arEmp[1]
    - arPtrEmp[3] points to arEmp[0]
- We loop through the four elements of the arPtrEmp array and display them. Because these elements are pointers, we need to dereference them (using the * operator) in order to pass them into DisplayEmp()—which expects a value, not a pointer.

The output of the function is presented in Figure 3.6.

> *Test your understanding:* Instead of using pointer arithmetic to initialize the values in the arPtrEmp array, how could we have used the & operator to get the addresses of the elements in arEmp?


c:\Textbook\Chapter11\StructDemo\Debug\Struct...
c:\Textbook\Chapter11\StructDemo

```
Pointer Array Test:

ID: 103
Green, Mary
Salary: $45000.00
Year Hired: 1988, Status: Active

ID: 111
Johnson, Jeremy
Salary: $75000.00
Year Hired: 2002, Status: Active

ID: 102
Brown, Franklin
Salary: $60000.00
Year Hired: 2001, Status: Former

ID: 101
Smith, Joan
Salary: $100000.00
Year Hired: 1999, Status: Active
```

Figure 3.6: Output from EmployeePointerArrayTest()

3.7 Section Questions

Suppose we had the following arrays defined:

```
struct employee ar1[10];
struct employee *ar2[10];
// Some initialization code…
```

1. Write two different expressions that would take element 2 of ar1 and place its address in element 3 of ar2.

2. Write two different expressions for assigning the object data pointed to by element 1 of ar2 int element 5 of ar1.

3. Write two different expressions that return the second (element 1) character in the last name (szLastName) of element three of ar1.

4. Write an expression that could be used to test if the first name (szFirstName) of the employee of element 8 of ar1 matches the first name of the employee pointed to by element 0 of ar2.

5. Write an expression that tests if element 7 of ar2 points to ar1 element 6.

6. Write an expression to tell if the data in element 7 of a2 is the same as that in ar1 element 6.

3.8: Dynamic Memory

Dynamic memory is memory that we acquire from the operating system while our program is running. The ability to utilize such memory frees us from the limitation of having to predefine the size of all our arrays. Such flexibility proves to be crucial as our programs become more complex.

3.8.1: Types of Memory

C++ provides us with three types of memory:

- static memory, allocated when the program starts running, and
- local memory, acquired and released as functions are called and returned.
- dynamic memory, acquired by the program from the operating system when needed, then released by the program when no longer needed.

To understand the third type of memory, *dynamic memory*, it is useful to develop a mental model of how a simple operating system might organize memory. Such an example is presented in Figure 3.7.

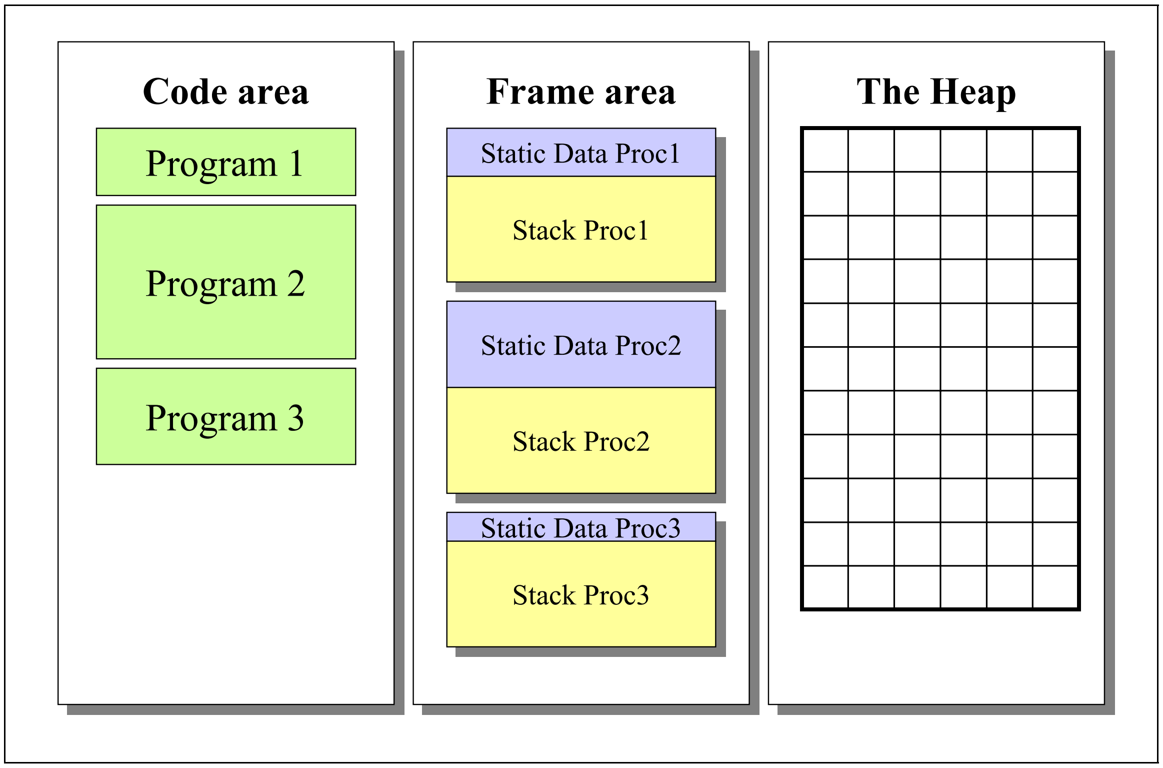


Figure 3.7: Simple Memory Organization Scheme

Figure 3.7 shows memory as being divided into three conceptual areas:

1. *Program area:* Contains the actual program code (i.e., machine language) for all the programs we have running.
2. *Frame area:* The second area, the frame area, contains data associated with each process that is running. Within each process block, we have two types of space: the space for the static data that is required by the process *and* the space set aside to hold the stack to be used by each executable program—which holds the active local variables declared by each function that is running.
3. *The Heap:* The remaining memory that is not being used by the operating system, programs and running processes.

If we are going to get more memory while our program is running, its source will have to be "the heap". Since the heap is managed by the operating system (or by intermediate libraries that interact with the operating system), to get that memory we're going to have to ask the OS to give us a chunk. Correspondingly, when we are done with the memory (e.g., the user closes an open document while the program is running), we also need a way to give the memory back.

3.8.2: Acquiring Memory in C++

The **new** operator provides a powerful, type-safe tool for acquiring memory from the operating system. There are three common forms of the operator:

> **new** *data-type* ;
> **new** *data-type* [*nCount*] ;
> **new** *data-type* (*initialization-list*) ;

The last of these forms is strictly for OOP usage. As a result, we'll only be considering the first two here. Examples of these two forms are:

```
char *pC,*pArr;
pC=new char;
pArr=new char[80];
```

In functional form, the two versions of the new operator can be prototyped roughly as follows:

> *data-type* *operator new(*data-type*);
> *data-type* *operator new(*data-type*,unsigned int nCount);

The first form is used to acquire a single data element. The second form, using array brackets, is used to allocate an array of nCount elements. Both versions return a pointer to the data type being allocated.

Checking for NULL (or 0, in C++) after **new** is applied is not normally necessary. A failure of **new** generates a memory exception[1], which has the effect of preventing the code that follows it from being executed. Thus a test for NULL will have no effect.

While not having to test for NULL may initially sound like good news, it presents a bit of a problem when using C++ purely for the purposes of structured programming. The proper way to handle an exception is inherently object-oriented (since exceptions throw an exception object). Unfortunately, not handling exceptions properly invariably leads to a program crash (often displaying that annoying window asking if you want to send a message to Microsoft). So, you'll have to live dangerously when your programs call **new** until you move to the next level, and start using OOP.

3.8.3: Releasing Memory in C++

The **delete** operator returns memory (acquired with **new**) to the operating system. There are actually two versions of the operator:

> **delete** *object-pointer* ;
> **delete** [] *array-pointer* ;

In use, it appears as follows:

```
char *pC,*pArr;
pC=new char;
pArr=new char[80];
delete pC;
delete [] pArr;
```

When you are deleting objects with **delete**, you need to keep track of whether or not the object was allocated as an array. Using **delete** on an array object, or using **delete []** on a non-array object leads to "undefined behavior". And this is not good.

[1] Unless the programmer explicitly suppresses the behavior, such as through the use of std::nothrow. e.g.,

```
char *pNew=new (std::nothrow) char[80];
```

In this case, a NULL return would need to be tested.

3.8 Section Questions

1. What occurs during C++ memory allocation and deletion in addition to acquiring and releasing a block of memory?

2. Are then any ways in which C++ memory allocation seems awkward to use?

3. What are the implications of new and delete being operators, instead of functions?

4. How would allocate an array of 100 char pointers using new?

3.9: Managing Dynamic Memory for Structures

The basic principle of managing dynamic memory for structures is identical to managing any other form of dynamic memory. Many structures incorporate pointers to other dynamic memory objects, however. Some care has to be taken in allocating and releasing the memory for such structures, if memory leaks are to be avoided. In this section, we examine both simple structures, and structures that manage their own memory.

3.9.1: Acquiring Memory for Structures

Memory for C++ structures is acquired using **new** just as it is for basic data types. e.g.,

```
struct employee *pEmp=new employee;
struct employee *pArr=new struct employee[10];
```

The first expression above allocates and constructs a single employee (pEmp). The second allocates and constructs an array of 10 employees (pArr).

Once a structure has been allocated, it should be initialized. When we reach the OOP section of the text, we'll see how constructor functions can be defined to perform this type of initialization automatically. For a pure structure, however, we could also define a function—such as that shown in Example 3.10—to perform the initialization.

Example 3.10: Initializing C++ structure element by element

```
#define MAXNAME 30

struct employee {
	unsigned int nId;
	char szLastName[MAXNAME];
	char szFirstName[MAXNAME];
	unsigned int nYearHired;
	double dSalary;
	bool bActive;
};

void DemoEmpInit(struct employee *pEmp)
{
	pEmp->nId=0;
	memset(pEmp->szLastName,0,MAXNAME*sizeof(char));
	memset(pEmp->szFirstName,0,MAXNAME*sizeof(char));
	pEmp->nYearHired=0;
	pEmp->dSalary=0.0;
	pEmp->bActive=false;
}

//  Normally, such initialization will be part of an OOP constructor function
```

Structures with embedded dynamic memory
It is quite common to incorporate pointers to other dynamic memory objects within a structure. For example, we might rewrite the employee structure above as:

```
struct employee_rev {
	unsigned int nId;
	char *pszLastName;
	char *pszFirstName;
	unsigned int nYearHired;
	double dSalary;
	bool bActive;
};
```

We could the use dynamic memory to hold the last and first names. The advantage of this is that:

1. We don't waste space for short names
2. We're not limited to 30 characters for very long names.

Where a structure manages dynamic memory, however, it is critical that the managed memory be released prior to releasing the structure itself. In OOP, this process will normally be performed within a destructor function that can be written by the programmer. In structured programming, our best bet would be to write a special function to do the job. Example 3.11 illustrates a C++ function that could be used to release the memory for an employee structure.

Example 3.11: Releasing managed memory in employee_rev structures

```
void DeleteEmpRev(struct employee_rev *pEmp)
{
        delete [] pEmp->pszLastName;
        delete [] pEmp->pszFirstName;
        delete pEmp;
}
```

Obviously, the problem with just deleting the structure is that doing so would cause us to lose our pointers to the two name strings.

Programming Tip for Working With Dynamic Structures
When working with dynamic structures in C++, you can save yourself a lot of debugging time if you always define two functions for each type of structure you create:

- A function that you always use when you allocate memory for a structure
- A function that you always use when you delete memory for a structure

When working with true objects in C++, the construction and destruction of structure objects will normally be accomplished using OOP constructor and destructor functions.

3.9 Section Questions

1. Why is it a good idea to allocate and free structures in special functions, instead of just calling new and delete when needed?

2. Why will we normally not write independent functions for allocating and freeing structures in C++?

3. What does it mean to say a structure manages dynamic memory? Why is it particularly important to be careful in your memory management if you're using structures that manage dynamic memory?

4. Why is it not a good idea to initialize C++ structures using memset?

5. If you're managing a dynamic array of pointers inside a structure, how is it likely to be declared?

3.10: Review and Questions

3.10.1: Review

Pointers are variables that hold addresses, instead of values. They are declared just like normal variables, except an * is placed in front of the variable name. For example:

```
int *i1;
```

would declare i1 to be a pointer to an integer—meaning it holds an address that is (presumed to be) the address of an integer. One fact is critical to remember when you declare a pointer:

When you declare a pointer, you are not setting aside any memory for the data it will point to

This is a critical fact to remember, because a common mistake made by programmers is to use a pointer that has not been initialized. Just like a variable, however, a pointer always has a value. In the case of a pointer, however, until it has been initialized you are pointing to a random location in memory. A constant, NULL, is often used to signify that a pointer does not contain an address. In C++, NULL is guaranteed to be 0. In C, it is nearly always so.

Pointers are always initialized according to the type of data they are expected to point to. This allows the compiler to check that appropriate pointer types are used (e.g., in function calls) and also permits pointer arithmetic. One type of pointer, void *, **is permitted to point to anything. This type of pointer is most often used for generic memory functions (e.g., memcpy, memset) and for generic collections.**

There are three common ways of assigning an address to a pointer:

1. Applying the & operator to a variable name or array element (or, more generally, any lvalue)
2. Calling a function that returns an address
3. Assigning the value from another pointer or array name
4. Assigning the value from a pointer arithmetic expression

There are also two common ways of getting the value that a pointer refers to, known as *dereferencing*. These are:

The * operator, placed in front of the pointer (e.g., *i1 gets the value i1 points to)

The [] operator, placed after the pointer (e.g., i1[0] gets the value i1 points to)

These two approaches are completely equivalent. In fact the following relationship is *always* true:

***(p1 + i) ≡ p1[i]**

where p1 is any address (e.g., pointer or array name) and i is any integer. This always holds, and is the basis of pointer arithmetic.

Pointer arithmetic is the algebra of adding pointers and integers. Because if the correspondence between [] and * notation, it is constrained to operate as follows:

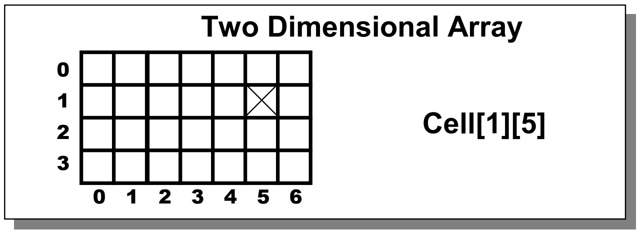
> Whenever a pointer and an integer are added together, the result is an address which is equal to the original address plus the integer*the size of the object pointed to.

The easiest way to visualize this is to thing of pointer arithmetic as jumping from element to element in an array. It is also possible to subtract two pointers:

> Whenever two pointers are subtracted, the result is the number of elements (i.e., objects) between the two pointers (equivalent to the difference in addresses divided by the size of the objects pointed to).

Pointer arithmetic only makes sense when applied to objects in the same array.

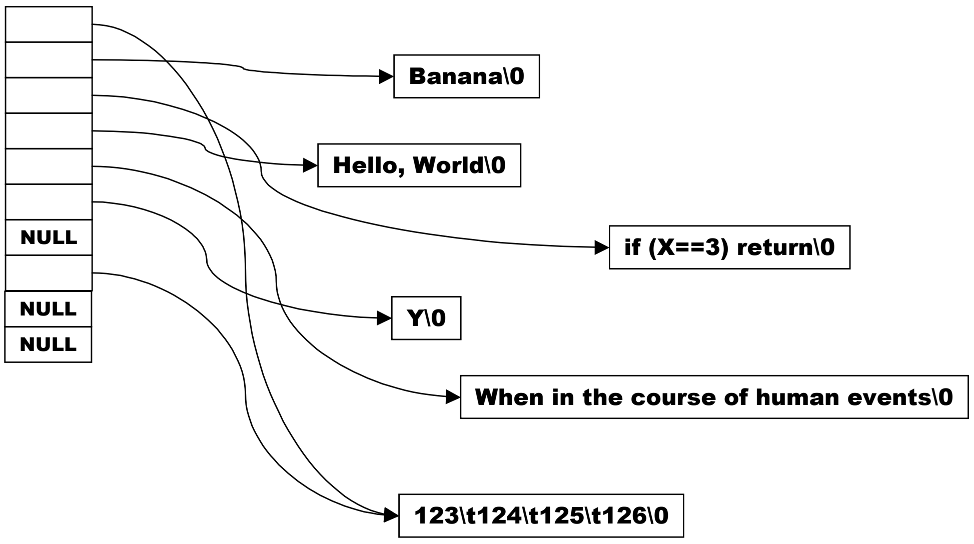
Another important tool provided by C/C++ is the ability to define complex arrays, which we define as multidimensional arrays or arrays of pointers. Conceptually, a single array can be visualized as a row of values, while multidimensional array can be visualized as a table, a 3-D table, or an even more complex table—depending on the number of dimensions.



Each row of a multidimensional array is, itself, treated as an array. For example, in the above example, if Cell were an array of integers:

Cell	– refers to the address of the start of the array (type int **)
Cell[1]	– refers the address of row one, an array of integers (type int*)
Cell[1][5]	– refers the value at the cell indicated with an X (type int)

Arrays of pointers tend to be more flexible than multidimensional arrays. Conceptually, they can be visualized as a table of 4-byte address values (under current compilers) which point to data that can be anywhere in memory, as illustrated below:



Multidimensional arrays of pointers can also be defined.

A structure is a technique for creating a data object that is a collection of different data elements. Conceptually, a structure is like the layout of a database table, which each element being a separate column. A data object, on the of the other hand, is like a row (record) in that table. The object contains the actual data.

Defining a structure in C/C++, which serves to identify how memory is to be laid out without actually creating any objects, us done using the following syntax:

> **struct** *structure-name* {
>   ***data-declarations…***
> };

where **struct** is a keyword, *structure-name* is the name the programmer chooses to give to the structure (which must be unique within a particular program) and ***data-declarations…*** are a collection of one or more declarations—identical as to how they would appear within a function, except that they cannot be initialized. The individual declarations that make up a structure are called elements or, sometimes, members.

To create an actual structure object, a syntax just like any other declaration is used (except the struct keyword must be present in C, although not in C++):

> **struct** *structure-name obj-name* ;

where *structure-name* is the name given to a previously defined structure, and *obj-name* is the name being given to the object, which can be any legal variable name. It is also possible to declare arrays of objects, pointers to objects and arrays of pointers to objects, etc., just as would be done with any other variable. It is also possible to declare structure objects by listing them after a structure definition, before the final semi-colon. This is not normally done, however, since structure definitions are normally placed in header (.h) files, and actually declaring objects in a header file usually leads to multiple-definition linker errors.

Within a structure definition, it is possible to declare another structure, in which case the process is called composition. The rules for composition are the same as for declaring a structure—the structure being composed within another structure must have been previously defined.

Structures can be initialized using braces, just like arrays. The order and type of the elements in the initializing list must match the order and type of the declarations within the structure. If a composed element of the structure or an array inside the structure is to be initialized, its initialization must be nested in braces.

To access data elements within a structure object, two operators are used, the . (period) and -> (arrow) operators. They are applied as follows:

> *structure-object* . *element-name* OR
> *structure-object-pointer* -> *element-name*

where *structure-object* can a variable defined as a structure, an array element or a dereferenced pointer. (It can also refer to composed object members of a structure). A *structure-object-pointer* refers to the address of a *structure-object*. In both cases, the result is the value of the specific element. Which operator is used (. or ->) is determined entirely by whether or not the expression to the left of the operator is an object or a pointer (address).

Three memory types can be used in a C++ program:

- *Static memory:* memory set aside when a program begins running and freed when it ends. Examples of this type of allocation include global variables.
- *Local or automatic memory:* Memory allocated on the stack (a.k.a. frame) that is available while a function (or block within a function) is being executed, but is removed once the function returns. An example is local variables.
- *Dynamic memory:* Memory provided from "the heap", a memory pool made available to all processes by the operating system that can be requested by a program and must also be returned.

Dynamic memory is different from the other two forms is several ways. First, it must be explicitly requested when needed, and returned when no longer needed. Failure to return dynamic memory can lead to a memory leak that can slow system performance dramatically. Second, it continues to exist until the program returns it to the O/S—even if the program has lost any way to access. Third, it is always acquired and returned using a pointer.

In C++, memory is acquired and released using operators. The **new** operator, which acquires memory, can be applied in three ways:

> new *type ;*
> new *type* [*count*] ;
> new *type* (*arguments…*) ;

The *type* above refers to any primitive (e.g., int) or user-defined (e.g., structure) type name. The last version of the new operator is relevant only to OOP, so we consider only the first two—which are distinguished by whether or not we are allocating memory for an array of data objects, or just an individual object.

To return memory in C++, we use the **delete** operator. It comes in two forms:

> delete *pointer* ;
> delete [] *pointer* ;

The second should always be used for arrays (even arrays of one element), the first for scalars. The version of **new** called to acquire memory determines the appropriate **delete** version, Using the wrong version can lead to an exception.

3.10.2: Glossary

Array notation – A method of accessing data associated with a pointer or array name using brackets, equivalent to pointer notation in its effect.
Array of structures – An array of structure objects that is conceptually similar to a table in a database, with each object representing a record
Automatic memory – Another name commonly used for local, or frame-based, memory.
const – A declaration modifier that, when used in front of a pointer, prevents that pointer from subsequently being used to change its associated value (i.e., being rereferenced and changed).
delete operator – Releases the memory for its argument to the operating system, thereby making the memory available for other programs.
delete [] operator – Releases the memory for an object allocated as an array.
Dereferencing – Acquiring the value associated with an address.
Dynamic Memory – Memory acquired from the operating system on demand, and freed when no longer needed.
Element – A specific item of data in an array (referenced by coefficient) or a specific data item within a structure (referenced by name)
Frame Area – The area in memory used for local data by each process that is running, organized according to the principles of a stack.
Heap – The space that the operating system has available for supplying memory to processes.
Initialize – To set a variable to its initial value, often while the variable is being declared.

Local memory – Memory that has a limited scope and is automatically released once execution of the particular function or block is finished.
Member – An alternative name for an element within a structure, more commonly used in OOP.
Memory leak – The failure to release dynamic memory once it is no longer required.
new operator – Used by C++ to acquire space for an object or array of objects
NULL pointer – A pointer that is used to signify that a pointer is not pointing anywhere—usually to the 0 position in memory (although this is guaranteed only for C++).
Object – A block of memory, organized according to a particular structure definition, that contains data elements. In C++, objects also include a set of functions that can be performed on those data elements.
Pointer Arithmetic – The rules for applying arithmetic (+ and -) operators on integers and addresses. It is defined in such a way that these arithmetic operations always lead to results consistent with array notation.
Pointer notation – A method of accessing data or addresses associated with a pointer using the * (dereferencing) operator.
Scope – The portion where the variable is available for use in a subroutine, defines the lifetime of the variable.
Static memory – Memory that exists for the duration of a program, such as globally defined data in C/C++..
struct – The keyword used to specify that the {} enclosed block of declarations that follow are to be used to determine how a structure is laid out. If no block follows, the names following the structure names are to be created as objects, object pointers or arrays.
Structure (or object) declaration – The process of specifying that an object, based on a particular structure definition, is to be created.
Structure definition – The process of defining how a structure will be laid out in memory, without creating any actual objects.
void pointer – A pointer (or memory address) that has is not specified to be pointing to any specific type of data.

3.10.3: Questions

Questions 3.1 – 3.5 are pointer problems

3.1. *Memcmp.* The C function memcmp() compares two blocks of memory—using their ASCII codes—and returns a result comparable to strcmp() (see Chapter 2). It is prototyped roughly as follows:

```
int memcmp(const void *addr1,const void *addr2,size_t nCount);
```

The arguments addr1, addr2 are the start addresses of the two blocks to be compared, and nCount is the number of bytes to be compared. (The main difference between memcmp and strncmp is that NUL terminators are ignored, and all bytes are always compared).

Create and test your own version of memcmp(), using the following prototype:

```
int Memcmp(const void *addr1,const void *addr2,int nCount);
```

(*Hint:* the technique for handling void pointers in Memset and Memcpy, Section 3.1.6, can be useful in this regard).

3.2. *Strchr.* The C library function strchr works like strstr (see Section 3.1.2) except that it looks for a character inside a string. It is prototyped as follows:

```
char *strchr(const char *str,char c);
```

For example, a call to the function:

```
strchr("Hello, World",',');
```

would return a pointer to the middle of the first string that would display as ", World". If the character is not present, it returns NULL.

Create and test your own version of the function, Strchr, using the same arguments.

3.3. *Strstr.* Implement your own version of strstr (see Section 3.1.2), prototyped as follows:

```
char *Strstr(const char *s1,const char *s2);
```

(*Hint:* If you have not already done so, implement Memcmp() from Question 1 or use the library version. If you use this function effectively, Strstr becomes very easy to write).

3.4. *Stristr*. Create and test a case-insensitive version of Strstr() called Stristr(). (*Hint:* consider implementing a case-insensitive version of Memcmp, called Memicmp, in which case the function becomes a trivial modification of Strstr).

3.5. *CountStrings.* Create and test a function, CountStrings(), that counts the number of independent (i.e., non-overlapping) occurrences of its second argument within its first argument. For example:

```
CountStrings("Yellow is the color of  mellow Hello","ll"); // Returns 3
CountStrings("xxxxxx","xxx"); // returns 2 (non-overlapping) occurrences
CountStrings("Hello World", "x"); // returns 0
```

(Hint: definitely use your own or the library version of strstr() to do this efficiently).

The following questions (3.6-3.13) need to be done in sequence and are designed to create a shopping list application.

3.6: *Designing a Structure:* Assume you are down with the flu, and are going to send a friend (or spouse) to the store for you to pick up some things you really need. Naturally, you'll need to write up a list. Create an empty project and begin by defining a structure that could be used to hold the information for each item on the list.

The above may be slightly trickier than it looks, because among the types of things that you certainly want to communicate are things such as: a) what type of item is it, b) where to find the item, c) does it have to be a specific brand, d) what's the maximum price you're willing to pay, etc.

3.7: *Creating I/O functions.* Create and test a pair of functions, modeled after DisplayEmployee() and InputEmployee() to allow display and edit of list items.

3.8: *Creating a Menu*. Create a function, modeled after the menu skeleton introduced in Chapter 2, that will be the interface of your application.

3.9: *Create a Collection Structure*: Create a structure called ShoppingData that contains two elements: 1) an array of shopping list structures, and 2) an integer that holds the number of active items. Create a global object of that type and populate it with some test data by initializing it with 10 or so item elements (using braces, as shown in Section 3.7.1). Create and test a ListItems function that displays all the items on your list.

3.10: *Write a FindItem function.* Write and test a function that finds the list item for a particular item type (e.g., "PaperTowel").

3.11: *Write an AddItem function.* Write and test a function that adds an item to the list. It should not add any item of a type that is already in the list. Note that this is somewhat different from the employee example, because ItemType will—presumably—be a string, whereas the employee ID was an integer.

3.12: *Write a RemoveItem function.* Write and test a function that removes an item from the list. You should design your function so it finds the list item based on item type, and has an argument that specifies whether the user should be prompted or not. For example, it might look like:

```
bool RemoveItem(struct ShoppingData *pList,const char *szType,bool nPrompt);
```

where it returns true (1) if successful, false (0) if it is not.

3.13: *Write an EditItem function.* Write and test a function that edits an item. One thing that you must think about here is to be sure that the item type doesn't change when it is edited or—if it does—that the new item type is not already in the list.

Questions 3.14-3.18 involves a structure that manages a dynamically allocated NUL terminated string.

3.14. *The String struct.* Create a structure that manages a string, defined as follows:

```
struct String {
        char *pStr;
};
```

Create and test three different functions for initializing the structure as follows:

```
struct String *CreateEmpty();  // Allocates memory for an empty, initialized String
struct String *Create(const char *str); // Creates a String structure with a copy of str
struct String *Copy(const struct String *s); // Creates a new String that's a copy of s
```

3.15. *Remove.* Create and test a function that cleans up the memory managed by a String structure, then deletes/frees the structure itself, prototyped as follows:

```
void Remove(struct String *str);
```

3.16. *Assign.* Create and test a function that assigns one String object to another, prototyped as follows:

```
const String *Assign(struct String *dest,const struct String *src);
```

Consistent with similar functions (e.g., strcpy), the function should assign its second argument to its first argument, then return a pointer to its second argument. On thing you should be sure to check for: make sure it doesn't crash in the event the programmer attempts to assign a String object to itself.

3.17. *String manipulators:* Implement and test the following functions:

struct String *Rest(const struct String *src,int nPos): Creates a new String object based on the portion of the original string starting at position nPos.
struct String *Left(const struct String *src,int nCount): Creates a new String object based on the leftmost nCount characters of the original string.
struct String *Right(const struct String *src,int nCount) : Creates a new String object based on the rightmost nCount characters of the original string.
struct String *Mid(const struct String *src,int nPos,int nCount): Creates a new String object based on the nCount characters of the original string starting at position nPos.

In each case, the function should use the src argument for the string to be manipulated, but should create a new String object, initialized with a NUL-terminated string created consistent with the operation. If the resulting string is too short, the truncated version should be returned (e.g., the leftmost 10 characters of the string "Hello" should return "Hello"). A one-byte string containing the NUL terminator should be returned if the operation is illegal (e.g., the Rest() function applied to a structure containing "Hello" with nPos equal to 10).

3.18 *String concatenation.* Implement and test a function Concat that adds its second argument to the end of its first argument (similar to what was done in 13.5). The function should be prototyped as follows:

```
void Concat(struct String *dest,const struct String *added);
```

Note that unlike the previous functions, this function changes its *dest* argument by adding the NUL-terminated string from *added* to whatever was initially in *dest*. It does not return a new String object.

Chapter 4

Introduction to File Stream I/O

Executive Summary

Chapter 4 examines how we can use files to load and save data in our applications. Without the ability to use files, computers are pretty much limited to computational activities—the information system activities that represent most of today's technology applications need to load and save data. The chapter introduces the idea of a generalized file stream, defined to accomplish I/O without knowing the precise characteristics of the device being used. It also explains how text and binary files differ, noting the compatibility and speed characteristics of the two approaches. The view of file I/O presented in this chapter is a very general one, with SimpleIO functions being used to hide the object-oriented nature of file I/O in C++. The specific I/O classes available from the C++ standard template library are presented in Chapter 11.

The chapter begins introducing the concept of a stream, noting that many I/O functions are defined in three forms: standard I/O form, file stream form and buffer form. It then introduces the notion of a text file, showing its basic equivalence to console I/O. The topic of binary files is then presented, demonstrating how such files allow rapid storage and retrieval of data. The use of such files as substitutes for in-RAM arrays is also illustrated.

Learning Objectives

Upon completing this chapter, you should be able to:

- Describe what is meant by a file stream
- Explain why standard I/O and text file I/O are actually equivalent
- Open and close streams
- Write programs that write text data to a file
- Write programs that read text data from a file
- Identify key differences between text and binary files
- Explain the types of functions used to access data and navigate around binary files
- Write applications that incorporate binary files

4.1: I/O Streams

The underlying mechanics of reading and writing information using files and the console tend to be very complex. Fortunately, in C++ these mechanics are completely invisible to the programmer. Instead, C++ employs the concept of I/O streams, which we examine in this section
.

4.1.1: Generalized Stream I/O

The idea behind generalized stream I/O is relatively simple. If you look at a typical computer, there are three types of places we can get information from, or send it to:

- Attached devices, or peripherals, such as the keyboard, monitor and printer.
- Disk files
- Memory buffers (i.e., blocks of memory containing data)

The principle used by C++ in designing its I/O libraries is that the functions used to input and output data should be as similar as possible, regardless of destination. As a consequence, most of the I/O capabilities in C++ are provided in three versions:

- A console or standard I/O version, designed to take input from the keyboard and/or send output to a text-based screen.
- A file version, designed to go to a designated disk file opened as a stream, identified with a special file object.
- A memory buffer version (for text I/O functions), where output is directed to an area of memory instead of a file or device.

Stream-oriented operating systems, such as UNIX and DOS, were specifically designed to support C++'s view of I/O. Indeed, special file names (such as PRN and COM1) are assigned to specific devices (e.g., attached printers) and ports (e.g., the serial port) so that input from and output to these devices can be accomplished using the same commands used for moving data to and from files.

Unfortunately, GUI-based operating systems, such as MS-Windows, are slightly more complex. Unlike console I/O—which closely resembles both a teletype and a text file that can be read line-by-line—graphic activities typically do not correspond very closely to any normal file activities. In addition, many devices (e.g., printers, modems) and ports (e.g., COM ports) are managed by the operating system in a manner not consistent with the way files are opened and closed in stream I/O. As a consequence, stream based I/O tends to be mainly limited to file activities on these operating systems. Console programs created under Windows, however, are specifically designed to mimic the behavior of earlier operating systems. Thus, the programs we write in this text follow the traditional UNIX/DOS model reasonably well..

4.1.2: Stream Objects

Before we can worry about moving data between file streams and our programs, we need to know how to associate a file with a stream. This actually involves two issues:

- How do we identify a stream in our program?
- How do we associate a stream with a specific file?

Every operating system has a specific object it uses to keep track of all the information necessary to manage a specific file or device attached to the stream. The name of the structure is unimportant (an _iobuf structure in Visual Studio.Net), and its specific elements even less so. The reason we don't care about the structure itself is that the language provides us with encapsulated objects (C++) that we can work with, without worrying about the mechanics—which are handled by the operating system.

In C++, a variety of different object types are defined to allow us to work with file streams. To fully understand these object types, however, requires fairly substantial knowledge of how C++ objects are constructed, and how they inherit characteristics from other objects. For this reason, we delay their full description to a later chapter (Chapter 11).

For the present purposes, we will be concerned with three types of file objects:

- **ostream**: Any stream that is opened for output. This may include the standard console output streams (**cout**), the standard error stream (**cerr**) or any file streams opened for output within the program.
- **istream**: Any stream that is opened for input. This may include the standard input stream (**cin**) or any file streams opened for input within the program.
- **fstream**: A file stream that is opened within the program. Such a stream can be opened for input, for output, or for input/output.

In the code that we will be developing through Chapter 11, we will only be opening and closing fstream objects—ostream objects (cout, cerr) and istream objects (cin) will be opened automatically when our console programs start running, and will close when they return. As implied in the descriptions, however, fstream objects can be used in functions that call for istream or ostream object references. In other words, if a function can take cout as an argument, it can also take an open file stream—in which case the data is written to the file instead of two the console.

If your program uses istream or ostream objects, you must include the iostream library as follows:

```
#include <iostream>
```

```
using namespace std;
```

If your program uses file objects, you will need to include the fstream library as follows:

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

Most commonly, code that performs file I/O will need both libraries, in which case you will have:

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

Both include directives then share the same name space inclusion.

Opening fstream objects

If you are opening fstream objects within your program, you will need to provide two pieces of information: the file name and how you want the file to be opened. The actual opening of the file is accomplished using what is referred to as a *member function*. A member functions is function that is applied to an object—in this case, an fstream object—in much the same way that we access data within structures (using the . or -> operators, as presented in Chapter 3). What this means is that to call a member function, we first need to declare an object, then call the function. For example, to open a new text file called "sample.txt" for writing, we could use the following code:

```
fstream strm;                    // declares fstream object
strm.open("sample.txt",ios_base::out | ios_base::trunc);  // opens file for writing
```

The tricky part of the statement is the second argument, which contains a series of constants (defined in the iostream header file) specifying the opening mode. The bitwise or operator (|) is then used to combine them. The allowable values for the contants are specified in Table 4.1.

Constant	Meaning
ios_base::in	File is to be opened for input
ios_base::out	File is to be opened for output
ios_base::trunc	File is to be truncated (erased) upon opening
ios_base::ate	Move to the end of the stream upon opening
ios_base::app	All insertions take place at the end of the stream
ios_base::binary	File is opened in binary mode, with no end of line translations (See Section 4.3.1 for general discussion of text vs. binary files).

Table 4.1: Opening mode constants

To check if the file opening has been successful, the is_open() member can be used, which returns a Boolean value of true if the file has been opened. Reasons an open operation might fail could include: incorrect file name, incorrect path name, incompatible device (e.g., opening a file for writing on a read-only CD), the file is already open somewhere else (especially if sharing is not set), as well as other types of problems. For this reason, you should always check to see if an open operation has succeeded before going on to use the file. This can be done as follows:

```
if (strm.is_open()) cerr << "Open was successful!" << endl;
else cerr << "Open failed!" << endl;
```

As will become clear when we cover member functions in detail, the () in the member function call to is_open are critical—just as they would be for any normal function call. The fail() member function can also be used to test if any stream operation was successful.

Closing fstream objects
An open file stream can be closed using the close() member function, e.g.,

```
strm.close();
```

There are three reasons you should always close file streams when you are done with them:

- Since open files consume resources, most operating systems have a limit on how many files are open at once. Keeping a file open unnecessarily may impact other programs if that limit is reached.
- If your program has a file open using one stream, chances are it will not be able to open it using another (this is an O/S protection that may be disabled under certain circumstances). The practical result is that if you open a file, then lose track of its stream variable without closing it, you may not be able to access the file again until you shut down your program (which automatically closes all open files).
- To avoid slowing down your program by waiting for leisurely file accesses (and to avoid making your disk sound like a hummingbird from all the activity), file buffers in memory often hold output waiting to be sent to a file. When a file is closed, those buffers are flushed to the file. Until they are flushed, it is possible that data your program has written to disk may not have actually reached the disk—meaning it could be lost in the event of a system crash or power failure.

If you declare an fstream object within a function (or any other block of code), the stream is automatically closed when the object goes out of scope (e.g., the function returns). As a result, it is not uncommon to see local fstream objects opened but not closed.

4.1.3: SimpleIO Library

Because C++ stream I/O is inherently object oriented, and uses many advanced OOP features (e.g., inheritance, templates, operator overloading), it is convenient to hide some of the complexities of I/O in the early stages of this book. For this reason, we will develop a library of functions—called the SimpleIO library—in this chapter. We will then use the functions we developed, wherever it proves to be convenient, until we have completed our systematic treatment of I/O in Chapter 11. The source code for the complete SimpleIO library is on the textbook CD in the files SimpleIO.h and SimpleIO.cpp.
Within SimpleIO.h we have defined three aliases for the stream types we introduced:

```
typedef istream INPUT;
typedef ostream OUTPUT;
typedef fstream STREAM;
```

The first SimpleIO functions that we define are Open() and Close(). The Open() function takes the opening mode as four separate Boolean variables, namely:

```
bool Open(STREAM &file,const char *szName,bool bRead,bool bWrite,
                             bool bTrunc,bool bText);
```

When we call Open(), we need to specify the following arguments:

- **file**: The STREAM variable we have previously declared
- **szName**: The file name (including a path, if desired) to be associated with the stream
- **bRead**: Is the file to be open for reading (input). If **true**, we will be able to read data from the file. If **false**, we can only write data to the file.
- **bWrite**: Is the file to be opened for writing (output). If **true**, we will be able to write data to the file. If **false**, we can only read data from the file. It is permissible for both the *bRead* and *bWrite* arguments to be true—in that case we can both read data from the file and write data to it. At least one (*bRead* or *bWrite*) must be true however.
- **bTrunc**: Is the file to be erased when it is opened? Many times, when you open a file (e.g., doing a "Save As…" for a document), it makes sense to erase anything that is already there. On the other hand, you would certainly not want *bTrunc* set to **true** if you were opening an existing document and you wanted to read the data from it.
- **bText**: It the file to be opened for text I/O (i.e., is it to be a text file or ASCII file) or is it going to hold raw—i.e., binary—data.

The function returns **true** if the opening operation is successful, **false** if it is not.

Text files
The final argument to Open() warrants a bit more explanation. Its origins go back to the early days of C, when I/O was frequently performed using teletype-style devices. To start a new line on these devices, two characters were required: a newline (\n), which moved the role up a line, and a return (\r), which moved the print head to the left-hand side of the paper. For this historical reason, many operating systems require that each line of a text file end with a "\r\n" pair of characters.

The problem with having two characters at the end of each text line is that when you read data from a text file (which is often done line-by-line, as we shall see), these characters can also be read as well—meaning that you end up with at least one extra white (non-printing) characters at the end of each line you read in. The purpose of opening a file in text mode (*bText* set to **true**) is to eliminate the extra character (the \r) when a line is read in. Specifically:

- When a file is designated as a text file, single '\n' characters are replaced by "\r\n" pairs in text files when outputting data. Conversely, "\r\n" pairs are replaced by '\n' characters when reading data from files.
- If a file is opened as a 'b' type, no translations are made.

Thus, while can read lines of text file that is opened as a binary file, if you do so, you'll often get an extra character at the end of each line you read. That, however, is the only major difference between files open in text and binary modes.

The SimpleIO Close() function is used to close an open file stream. Its prototype is simple:

    void Close(STREAM &file);

The file argument must point to a stream that has been opened using Open(). The SimpleIO source code for Open() and Close() is relatively straightforward, and is presented in Example 4.1. The main complexity is in the Open() function, which uses conditional operators to determine the proper Table 4.1 constants to use as the second argument to the file.open() member function call.

Example 4.1: SimpleIO Open() and Close() functions

```
bool Open(STREAM &file,const char *szName,
              bool bRead,bool bWrite,bool bTrunc,bool bText)
{
      ios_base::open_mode nMode=0;
      nMode= ((bRead ? ios_base::in : 0) |
            (bWrite ? ios_base::out : 0) |
            (bTrunc ? ios_base::trunc : 0) |
            (bText ? 0 : ios_base::binary));
      file.open(szName,nMode);
      if (file.fail()) return false;
      return file.is_open();
}


void Close(STREAM &file)
{
	file.close();
}
```

4.1 Section Questions

1. What is the purpose of the bTrunc argument to Open()?
2. Are text files produced under different operating systems necessarily byte-for-byte compatible with each other?
3. Why should you *always* check the return value of Open() after you call it?
4. What are the key problems of opening a file stream and not closing it when you are done?

4.2: Text File I/O

Up to this point in the text, all of our input and output has been in the form of collections of ASCII characters to and from the console. In this section, we extend this text-based I/O to files.

4.2.1: Text output

In this section, we'll examine how to write text to an open file stream. First we'll examine unformatted text, then we'll briefly consider formatted text.

Sending Unformatted Output to a Text Stream
The process for sending unformatted output to a text stream is identical to that introduced for sending text to cout in Chapter 2. The << (insertion) operator is overloaded for all primitive data types (integers, real numbers, Boolean values and NUL terminated strings) and the endl manipulator can be used to generate a new line. e.g.,

```
fstream strm;
strm.open("sample.txt",ios_base::out | ios_base::trunc);
if (strm.is_open()) strm << "Integer: " << 3 << "Real: " << 3.1416 <<
```

```
                "Boolean: " << true << endl;
```

Data sent to a text file is translated to its ASCII equivalent before being written to the stream. As a result, files produced in this manner can be opened in applications such as MS-Notepad and MS-Word. The default appearance of unformatted data (e.g., how the value *true* displays in the stream) is controllable through flags, discussed in Chapter 11.

SimpleIO Unformatted Text Output

The SimpleIO library provides two sets of functions for unformatted text output. The functions beginning with Display sent output to cout. Those beginning with Print send output to a specified output stream, which—as you may recall—may be one of two things:

- A STREAM variable that has been opened for writing in text mode, or
- A predefined object, such as **cout** or **cerr**.

The second of these is particularly important to note because it means that Print functions can be used for console, as well as file, output. For example:

```
PrintString(cout,"Hello, World");
```

is equivalent to both:

```
DisplayString("Hello, World");
```

and

```
    cout <<  "Hello, World";
```

The ability to treat the console as just another form of file is one of the principle benefits of defining streams.

In addition to treating the console as file stream, the operating system may provide other specialized file names. For example, in DOS any data directed to a file called PRN actually gets sent to the default printer. Thus, the code:

```
STREAM myPrn;
// Open printer stream for text output
if (!Open(myPrn,"PRN",false,true,false,true)) return;
PrintLine(myPrn,"Hello World");
```

would cause "Hello World" to be sent to the printer. (It's probably not a good idea to try printing this way in Windows, since it can lead to problems with the print queues. For Windows printing, the best approach is to print to a file, then open the file in a program such as MS-Word or MS-Notepad and print from there).

The SimpleIO unformatted output functions are listed in Table 4.2. Their definitions are presented in Examples 4.2 and 4.3.

Table 4.2: SimpleIO Text Output Functions

void DisplayString(const char str[]) Displays a NUL terminated character string to the console
void DisplayCharacter(char cVal) Displays a single character to the console
void DisplayInteger(int nVal) Displays an integer to the console
void DisplayReal(double dVal) Displays a real number to the console
void NewLine() Sends a newline to the console
void PrintString(OUTPUT &out,const char *szOut) Sends a string to the stream out
void PrintInteger(OUTPUT &out,int nVal) Sends an integer, as text, to the stream out
void PrintCharacter(OUTPUT &out,char cVal) Sends a character, as text, to the stream out
void PrintReal(OUTPUT &out,double dVal) Sends a real number, as text, to the stream out
void PrintLine(OUTPUT &out,const char *szOut) Sends a string, followed by a new line character, to the stream out
void EndLine(OUTPUT &out) Sends a newline to the stream out

Example 4.2: SimpleIO Display Functions

```
void DisplayString(const char str[])
{
      cout << str;
}
void DisplayCharacter(char cVal)
{
      cout << cVal;
}
void DisplayInteger(int nVal)
{
      cout << nVal;
}
void DisplayReal(double dVal)
{
      cout << dVal;
}
void NewLine()
{
      cout << endl;
}
```

Both the Display and Print versions of the functions have a trivial implementation, and are included primarily for completeness.

Example 4.3: SimpleIO Print Functions

```
void PrintString(OUTPUT &out,const char *szOut)
{
      out << szOut;
}
void PrintInteger(OUTPUT &out,int nVal)
{
      out << nVal;
}
void PrintCharacter(OUTPUT &out,char cVal)
{
      out << cVal;
}
void PrintReal(OUTPUT &out,double dVal)
{
      out << dVal;
}
void PrintLine(OUTPUT &out,const char *szOut)
{
      out << szOut << endl;
}
void EndLine(OUTPUT &out)
{
      out << endl;
}
```

Formatted Text Output

Walkthrough available in Sprintf.avi

C++ provides a large range of options for formatting text output, nearly all of which require a fairly heavy dose of OOP. Fortunately, there is a single function—provided as part of the C/C++ stdio.h library—that can be used to accomplish nearly all of the formatting that we need. That function is **sprintf()**.

sprintf() is an unusual function. It is prototyped as follows:

    int sprintf(char szBuf,const char szFmt,…);

What should immediately strike your eye about the prototype is the three dots (sometimes called an ellipsis). In a function prototype, this signals that the function can have a variable number of arguments. Although the means of defining variable argument functions is beyond the scope of this text, it is critical to be aware of one fact: whenever a variable argument C/C++ function is defined, there has to be a built-in way for the function to figure out: a) how many arguments are coming, and b) the type of data contained in each argument.

The purpose of the function is to take a formatting string (the second argument) along with any additional arguments and use them to create a text output string, which is placed in the first argument. The formatting string is interpreted as follows:

- If the format string contains no % signs, then no more arguments are coming. In that case, sprintf() just copies the format string into the first argument.
- If a % is encountered in the format string (followed by one or more special characters), sprintf() expects to see a corresponding argument passed in. It then translates the argument into a text representation and inserts it into the output string that will be placed in the first argument.
- If a double percent sign (%%) is encountered in the format string, sprintf() knows that what we really want it to do is display the percent sign, not specify an argument.

The organization of the special characters that can follow a % sign in the sprintf() formatting string is shown in Figure 4.1.

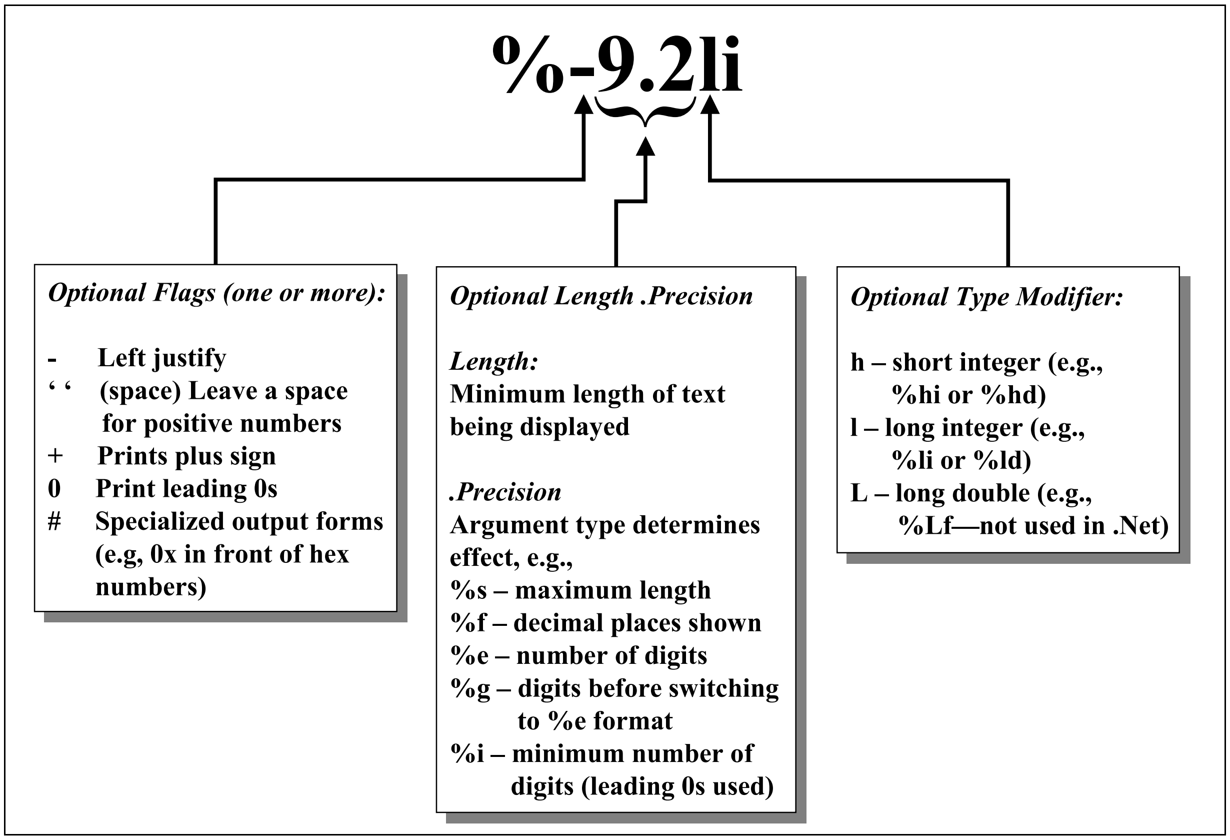

%-9.2li
Optional Flags (one or more):
- Left justify
' ' (space) Leave a space for positive numbers
+ Prints plus sign
0 Print leading 0s
# Specialized output forms (e.g, 0x in front of hex numbers)
Optional Length .Precision
Length:
Minimum length of text being displayed
.Precision
Argument type determines effect, e.g.,
%s – maximum length
%f – decimal places shown
%e – number of digits
%g – digits before switching to %e format
%i – minimum number of digits (leading 0s used)
Optional Type Modifier:
h – short integer (e.g., %hi or %hd)
l – long integer (e.g., %li or %ld)
L – long double (e.g., %Lf—not used in .Net)

Figure 4.1: Typical sprintf() full argument specification

Three elements of the specification—flags, length.precision and type modifier—are optional. The final element, the type specifier, must be present. Allowable type specifiers are summarized in Table 4.3.

Symbol	***Data type of argument***	***Comments***
d, i	int	Outputs a decimal integer
x, X	int	Outputs integer value in hexadecimal. Upper and lower case versions indicate how a-f will appear (e.g., f or F)
u	int	Prints unsigned integer value
c	int	Prints the ASCII character equivalent of the integer
f	double	Prints the floating point number, using standard notation
e, E	double	Prints the floating point number, using exponential notation. Upper and lower case versions indicate whether it is printed as 1.774234e+03 or 1.774234E+03
g, G	double	Uses exponential numbers for very small numbers (E-4 or less), standard notation otherwise
s	char *	Prints the string starting at the address (e.g., array name or pointer)
p	void *	Prints the value of the address passed in (e.g., array name or pointer)

Table 4.3: DisplayFormatted (printf) Type character codes

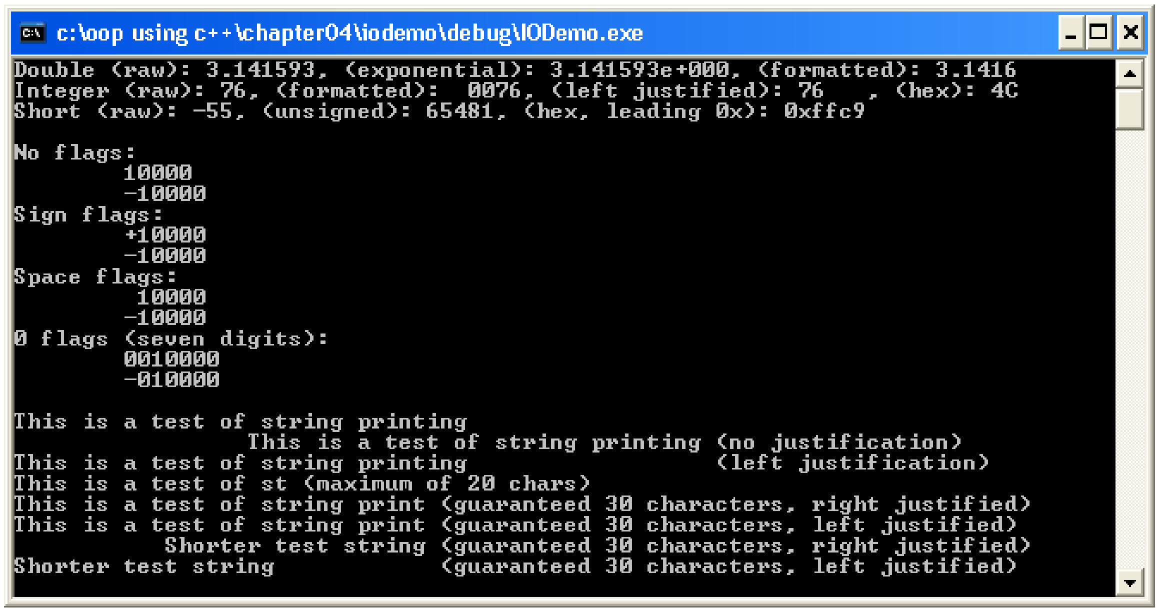
The interpretation of the example specification in Figure 4.1, *%-9.2li*, is as follows (moving from right to left):

- The i at the end specifies an integer type (as noted in Table 4.3).
- The letter l in front of the i specifies that it is a **long int**. Currently, the **long** data type is the same as **int**, so the type modifier would have no effect. It could, however, when the program was ported to a new OS, with 64-bit **long** data, or an old system (e.g., Windows 3.1), with 16 bit **int** data.
- The 9.2 is interpreted as follows:
    - 9 is the minimum number of characters to be printed
    - 2 is the minimum number of digits. Should the integer be 7, for example, it would add a second digit to make 07. This particular setting, known as precision, varies according to data type. For f and e specifiers, it means number of decimal places. For the s (string) specifier, it is the maximum number of characters to be displayed (useful in aligning columns). For the c specifier, it has no effect.
- The – flag specifies that the 9 character integer is to be left justified, meaning that any trailing blanks will be placed at the end of the integer (as opposed to right justifying it, which is the standard procedure).

The easiest way to learn the basics of using the sprintf function is to try it out. Example 4.4 contains a simple program that demonstrates a wide range of useful sprintf formatting capabilities. Figure 4.2 shows how they appear on the console screen.

Example 4.4: sprintf Formatting Demonstration

```
void SprintfDemo()
{
        char buf[1024];
        int i1=76,i2=10000,i3=-10000;
        double d1=3.1415926535;
        short s1=-55;
        char szTest[]="This is a test of string printing";
        char szShort[]="Shorter test string";
        // Simple number formatting
        sprintf(buf,"Double (raw): %f, (exponential): %e, (formatted): %6.4f\n",d1,d1,d1);
        cout << buf;
        sprintf(buf,"Integer (raw): %i, (formatted): %5.4i, (left justified): %-5i, (hex):
%X\n",
                i1,i1,i1,i1);
        cout << buf;
        sprintf(buf,"Short (raw): %hi, (unsigned): %hu, (hex, leading 0x): %#hx\n\n",
                s1,s1,s1);
        cout << buf;
        // test of flags in column (preceded by tab)
        sprintf(buf,"No flags:\n");
        cout << buf;
        sprintf(buf,"\t%5i\n",i2);
        cout << buf;
        sprintf(buf,"\t%5i\n",i3);
        cout << buf;
        sprintf(buf,"Sign flags:\n");
        cout << buf;
        sprintf(buf,"\t%+5i\n",i2);
        cout << buf;
        sprintf(buf,"\t%+5i\n",i3);
        cout << buf;
        sprintf(buf,"Space flags:\n");
        cout << buf;
        sprintf(buf,"\t% 5i\n",i2);
        cout << buf;
        sprintf(buf,"\t% 5i\n",i3);
        cout << buf;
        sprintf(buf,"0 flags (seven digits):\n");
        cout << buf;
        sprintf(buf,"\t%07i\n",i2);
        cout << buf;
        sprintf(buf,"\t%07i\n\n",i3);
        cout << buf;
        // Tests of string printing
        // No arguments
        sprintf(buf,szTest);
        sprintf(buf,"\n");
        // %s format tests
        sprintf(buf,"%s\n",szTest);
        cout << buf;
        sprintf(buf,"%50s (no justification)\n",szTest);
        cout << buf;
        sprintf(buf,"%-50s (left justification)\n",szTest);
        cout << buf;
        sprintf(buf,"%.20s (maximum of 20 chars)\n",szTest);
        cout << buf;
        sprintf(buf,"%30.30s (guaranteed 30 characters, right justified)\n",szTest);
        cout << buf;
        sprintf(buf,"%-30.30s (guaranteed 30 characters, left justified)\n",szTest);
        cout << buf;
        sprintf(buf,"%30.30s (guaranteed 30 characters, right justified)\n",szShort);
        cout << buf;
        sprintf(buf,"%-30.30s (guaranteed 30 characters, left justified)\n\n",szShort);
        cout << buf;
}
```



```
Double (raw): 3.141593, (exponential): 3.141593e+000, (formatted): 3.1416
Integer (raw): 76, (formatted):  0076, (left justified): 76    , (hex): 4C
Short (raw): -55, (unsigned): 65481, (hex, leading 0x): 0xffc9

No flags:
        10000
        -10000
Sign flags:
        +10000
        -10000
Space flags:
         10000
        -10000
0 flags (seven digits):
        0010000
        -010000

This is a test of string printing
                 This is a test of string printing (no justification)
This is a test of string printing                  (left justification)
This is a test of st (maximum of 20 chars)
This is a test of string print (guaranteed 30 characters, right justified)
This is a test of string print (guaranteed 30 characters, left justified)
           Shorter test string (guaranteed 30 characters, right justified)
Shorter test string            (guaranteed 30 characters, left justified)
```

Example 4.2: Output from SprintfDemo() function

The only real drawback of the sprintf() function (as opposed to the C++ formatting that will be presented in Chapter 11) is that it can be the source of runtime errors. These errors occur when the arguments specified in the format string do not match those passed into the function. Problems could include two many arguments, two few arguments or arguments of the wrong type. The compiler will not normally catch such errors, since the format string itself may not be known until runtime. Visual Studio .NET Version 2003 has been enhanced to catch some of them, however.

For convenience, the SimpleIO library contains two functions modeled after sprintf(), DisplayFormatted() and PrintFormatted(). They are prototyped as follows:

```
void DisplayFormatted(const char fmt[],…);
void PrintFormatted(OUTPUT &out,const char fmt[],…);
```

In both cases, the formatting string and arguments are identical to those of sprintf. They differ in that DisplayFormatted() calls a version of sprintf on its arguments, then sends the resulting formatted output to cout. PrintFormatted() does the same thing, except that it directs the formatted output to a file. Thus, DisplayFormatted(args…) and PrintFormatted(cout,args…) are completely equivalent. (Note: The implementation of these two functions involves variable argument passing and is therefore not presented here. The interested reader can examine the code by looking at SimpleIO.cpp).

4.2.2: Text Input

Text input from a stream can be accomplished by reading individual data elements, or by processing the stream line by line. The ability to read in input a line at a time is particularly useful owing to the ambiguity of input. In particular, when reading text input, the standard C++ library exhibits three key behaviors:

1. It does not start translating its inputs until it receives the end of the input line. That means that even if it is only reading a character, it pauses until the user hits <enter>. In the case of a text file, it brings in the entire file line before translation begins.
2. When it expects one type of input (e.g., an integer) and gets another (e.g., a string), it will keep reading input until it gets the expected type, or exhausts all input.
3. When reading a string, it assumes that the string ends as soon as a space is encountered. That means that you cannot read a string with spaces in it directly.

These behaviors—particularly the last two—mean that direct reads using the >> operator (as demonstrated in Chapter 2) can be quite perilous if a user is involved. For example, consider the code below:

```
char szAddress[80];
cout << "Enter your address: ";
cin >> szAddress;
```

If the user types in a typical street address, szAddress will end up containing only the street number. The rest of the address will remain in the input stream until the next >> operator use. We might then try to fix the code as follows:

```
char szAddress[80];
int nStreetNo;
cout << "Enter your address: ";
cin >> nStreetNo;
cin >> szAddress;
```

Unfortunately, in the revised code, if the street does not begin with a number then the input will be exhausted before we ever get to the address.

C++ Line Input

To avoid the problems of direct input, it is often better to read each line of input into a string, then perform any necessary translations on the result. To accomplish this, we need another member function, getline(). The getline() function an be called on any input stream (e.g., cin or an fstream opened for input). It takes two arguments: the destination character array and the maximum number of bytes to be read (which will normally be no more than the size of the destination array). For example, to read address from the user as a single line, we could use the following code:

```
char szAddress[80];
cout << "Enter your address: ";
cin.getline(szAddress,80);
```

After the getline() function call, szAddress will contain a NUL terminated string of no more than 80 characters that will contain all the test typed in by the user—including any spaces. This will often be the desired outcome when you are reading a string from the user.

When reading input from a text file, the situation is markedly different. Very often, such files will be formatted in a manner that allows the programmer to predict precisely the form of the incoming input. Thus, using the >> operator to extract individual data elements from an fstream object is less likely to result in errors. The problem with spaces causing separations in string variables remains, however.

One very common pattern of line input involves reading through the entire contents of a text file. To accomplish this, it is convenient to use the fstream member function eof(), which returns true when the end of a file has been reached. The TypeFile() function (Example 4.5), for example, reads the entire contents of a file and displays it to the screen. It works by opening the file, then reading it line by line until the eof() condition is reached.

Example 4.5: TypeFile() Function

```
void TypeFile(const char *szFileName)
{
      fstream strm;
      strm.open(szFileName,ios_base::in);
      if (strm.is_open())
      {
            while(!strm.eof()) {
                  char buf[1024];
                  strm.getline(buf,1024);
                  cout << buf <<endl;
            }
      }
}
```

Text Input in SimpleIO

To support text input, the SimpleIO library contains two sets of functions. One set, beginning with Input, takes input from standard input (cin) and assumes data will be coming in one line at a time. The second set, beginning with Get, reads data directly from a user specified file stream (which could be cin), using the >> operator. The functions are listed in Table 4.4, with their definitions presented in Example 4.6 and 4.7.

Table 4.4: SimpleIO Text Input Functions

void InputString(char str[]) Reads a line from the console and places it in the str[] array, followed by a NUL terminator.
char InputCharacter() Reads a line from the console and returns the first non-white character.
int InputInteger() Reads a line from the console and translates it int an integer, returning the value
double InputReal() Reads a line from the console and translates it int an double, returning the value
void GetString(INPUT &in,char *szIn) Reads a string from the in stream, placing the result in szIn. Stops at the first space.
int GetInteger(INPUT &in) Reads an integer from the in stream, returning the value.
char GetCharacter(INPUT &in) Reads a single non-white character from the in stream, returning the value.
double GetReal(INPUT &in) Reads a real number from the is stream, returning the value as a double.
bool GetLine(INPUT &in,char *szIn,int nMax) Reads a line of no more than nMax characters from the in stream, placing the value in the array pointed to by szIn.

The SimpleIO Input functions (Example 4.6) all begin with a call to cin.getline(). The two numeric functions then call either atoi() (returning the integer equivalent of a string) or atof() (returning the double equivalent of a string). The InputCharacter() function searches the line for the first non-white character, returning NUL (0) if none is found.

Example 4.6: Input Family of SimpleIO Functions

```
void InputString(char str[])
{
	cin.getline(str,MAXLINE);
}
int InputInteger()
{
	char buf[MAXLINE];
	cin.getline(buf,MAXLINE);
	return atoi(buf);
}
double InputReal()
{
	char buf[MAXLINE];
	cin.getline(buf,MAXLINE);
	return atof(buf);
}
char InputCharacter()
{
	char buf[MAXLINE];
	cin.getline(buf,MAXLINE);
	int i;
	for(i=0;buf[i]>'\0' && buf[i]<=' ' && i<MAXLINE;i++) {};
	if (buf[i]<= ' ') return (char)0;
	else return buf[i];
}
```

The SimpleIO Get functions (Example 4.7) are even simpler, using the >> operator to acquire their value. The >> **ws** preceding each read skips over any non-printable characters in the stream. Most of the time, it is unnecessary. As we have previously noted, the use of the INPUT (istream) argument makes the functions flexible, in that the Get family can be used to read data from cin. Unlike out output functions, however, the two input families aren't identical, as the Input family assumes one data input per line, whereas the Get family can process multiple inputs from a single line.

Line input from a file into a character array can be performed using the GetLine() SimpleIO function. The function returns **true** unless the end of the file has been reached (or some error occurs, such as passing in a stream not opened for reading) in which case it returns **false**. This fact can be used in reading lines from a text file (comparable to Example 4.5). For example, the following code fragment will read and display lines from "testfile.txt" until the file is empty:

```
STREAM infile;
char buf[255];
// Open existing text file for reading
if (!Open(infile,"testfile.txt",true,false,false,true)) return;
while (GetLine(infile,buf,255)) {
    cout << buf << endl;
}
```

Example 4.7: Get Family of SimpleIO Functions

```
void GetString(INPUT &in,char *szIn)
{
      in >> ws >> szIn;
}
int GetInteger(INPUT &in)
{
      int nVal;
      in >> ws >> nVal;
      return nVal;
}
char GetCharacter(INPUT &in)
{
      char cVal;
      in >> ws >> cVal;
      return cVal;
}
double GetReal(INPUT &in)
{
      double dVal;
      in >> ws >> dVal;
      return dVal;
}
bool GetLine(INPUT &in,char *szIn,int nMax)
{
      in.getline(szIn,nMax);
      if (in.fail()) return false;
      return true;
}
```

4.2.3: Example of SimpleIO Text I/O

Walkthrough available in TextIO.avi

In this section we'll look at a simple text I/O demonstration program, focusing on element-oriented I/O. (In 4.4.1, line-oriented file processing will be considered).

The SaveText() function, presented in Example 4.8, takes a set of arrays (set up as global variables, for convenience) and writes them to a text file. Its sole argument is the name of the file to which the data is to be saved. It returns **true** is successful, **false** otherwise.

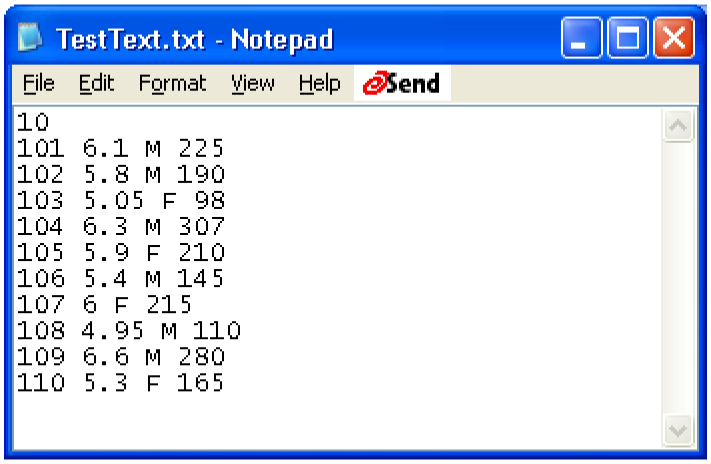
Example 4.8: SaveText() function

```
#include "SimpleIO.h"

int arID[10]={101,102,103,104,105,106,107,108,109,110};
double arHeight[10]={6.1,5.8,5.05,6.3,5.9,5.4,6.0,4.95,6.6,5.3};
char arSex[10]={'M','M','F','M','F','M','F','M','M','F'};
int arWeight[10]={225,190,98,307,210,145,215,110,280,165};

bool SaveText(const char *szFile)
{
	STREAM out;
	int nCount,i;
	if (!Open(out,szFile,false,true,true,true)) return false;
	DisplayString("How many values do you want to save? (1 to 10): 
");
	nCount=InputInteger();
	if (nCount<1 || nCount>10) {
		DisplayString("Illegal value: must be between 1 and 10");
		return false;
	}
	PrintInteger(out,nCount);
	EndLine(out);
	for(i=0;i<nCount;i++) {
		PrintInteger(out,arID[i]);
		WhiteSpace(out);
		PrintReal(out,arHeight[i]);
		WhiteSpace(out);
		PrintCharacter(out,arSex[i]);
		WhiteSpace(out);
		PrintInteger(out,arWeight[i]);
		EndLine(out);
	}
	Close(out);
	return true;
}
```

The SaveText() function operates as follows:

- The file named szFile is opened for writing as a text file. The bTrunc argument of Open() is set to **true**, so that any existing data will be erased. If the file fails to open, the function returns false.
- The user is prompted to identify how many values are to be saved, which is verified to be between 1 and 10. This is then written to the text file (where it will be very useful when it comes time to read the data), followed by an EndLine().
- A loop is performed whereby the ID, height, sex and weight data are all saved, followed by an EndLine(). This causes each set of data to be given its own line in the file.
- Upon completion of the loop, the stream is closed and the function return true, indicating success.

A data file produced by SaveText() function is displayed, in MS Notepad, in Figure 4.3.



```
10
101 6.1 M 225
102 5.8 M 190
103 5.05 F 98
104 6.3 M 307
105 5.9 F 210
106 5.4 M 145
107 6 F 215
108 4.95 M 110
109 6.6 M 280
110 5.3 F 165
```

Figure 4.3: Contents of file created by SaveText()

The function DisplayText() takes a file created by SaveText() and displays it to the console. This function is presented in Example 4.9

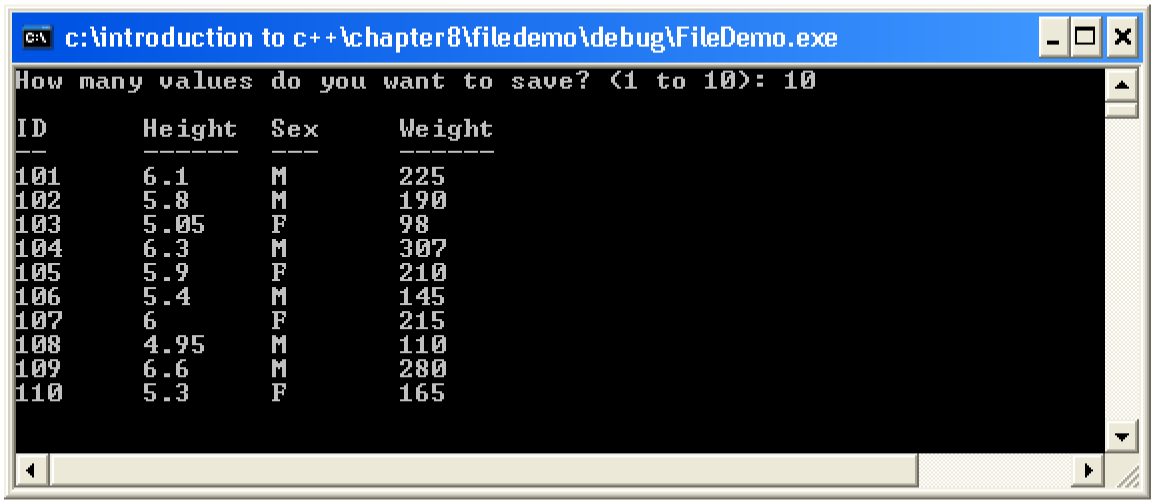
Example 4.9: DisplayText() function

```
bool DisplayText(const char *szFile)
{
	STREAM in;
	int nCount,i;
	if (!Open(in,szFile,true,false,false,true)) return false;
	nCount=GetInteger(in);
	NewLine();
	DisplayString("ID\tHeight\tSex\tWeight\n--\t------\t---\t------
\n");
	for(i=0;i<nCount;i++) {
		char szID[20];
		double dHeight;
		char cSex;
		int nWeight;
		// Reading data
		GetString(in,szID); // Note: ID read as string for demo
purposes
		dHeight=GetReal(in);
		cSex=GetCharacter(in);
		nWeight=GetInteger(in);
		// Displaying data
		DisplayString(szID);
		DisplayCharacter('\t');
		DisplayReal(dHeight);
		DisplayCharacter('\t');
		DisplayCharacter(cSex);
		DisplayCharacter('\t');
		DisplayInteger(nWeight);
		NewLine();
	}
	Close(in);
	return true;
}
```

The DisplayText() function operates as follows:

- The a file stream is opened for the szFile argument. The call to Open() specifies a readable text file, not truncated. If the open fails, the function returns **false**.
- The integers specifying the number of data items stored is read from the text file. This is used to control the file reading operations.
- Column headers, separated by tabs, are displayed to the console.
- A loop is performed whereby the ID, height, sex and weight elements are read from the file, then displayed (separated by tabs) to the console. It is interesting to note that although the ID was saved as an integer, it is loaded as a string. When dealing with text I/O, this type of switch presents no serious obstacles, as anything in a text file can be read as if it were text.
- Upon completion of the loop, the input stream is closed and the function returns **true**.

The console output from running SaveText() then DisplayText() is shown in Figure 4.4.



```
How many values do you want to save? (1 to 10): 10

ID      Height  Sex     Weight
--      ------  ---     ------
101     6.1     M       225
102     5.8     M       190
103     5.05    F       98
104     6.3     M       307
105     5.9     F       210
106     5.4     M       145
107     6       F       215
108     4.95    M       110
109     6.6     M       280
110     5.3     F       165
```

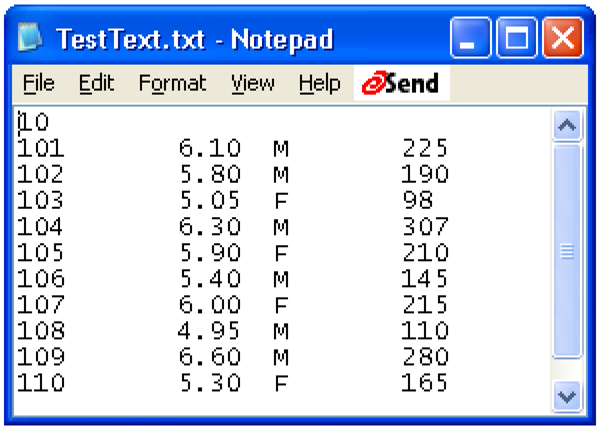
Figure 4.4: Console display of SaveText() followed by DisplayText()

It is worth noting, once again, that text files can be created using many techniques. Example 4.10 contains a revised version of the text saving function, called SaveFormattedText(), that uses a single call to PrintFormatted() instead of saving the data elements individually.

Example 4.10: SaveFormattedText()

```
bool SaveFormattedText(const char *szFile)
{
        STREAM out;
        int nCount,i;
        if (!Open(out,szFile,false,true,true,true)) return false;
        DisplayString("How many values do you want to save? (1 to 10):
");
        nCount=InputInteger();
        if (nCount<1 || nCount>10) {
                DisplayString("Illegal value: must be between 1 and 10");
                return false;
        }
        PrintInteger(out,nCount);
        EndLine(out);
        for(i=0;i<nCount;i++) {
                PrintFormatted(out,"%i\t%6.2f\t%c\t%i\n",
                        arID[i],arHeight[i],arSex[i],arWeight[i]);
        }
        Close(out);
        return true;
}
```

This function produces a text file that is nearly identical to the one produced by SaveText(), excepting that the real numbers for saved for height measurements are all displayed to 2 decimal places (as a result of the %6.2f formatting used). The output file produced by the function, shown in MS Notepad, is presented in Figure 4.5.



```
10
101        6.10  M      225
102        5.80  M      190
103        5.05  F      98
104        6.30  M      307
105        5.90  F      210
106        5.40  M      145
107        6.00  F      215
108        4.95  M      110
109        6.60  M      280
110        5.30  F      165
```

Figure 4.5: File generated by PrintFormattedText()

4.2 Section Questions

1. Why is using GetLine(cin,…) safer than using InputString()?
2. How can the contents of a string interfere with our ability to read it?
3. Why is GetString() not a mirror image of PrintString()?
4. What is the basic procedure for reading all the lines in a text file?
5. If you're not sure about the format of a text file, how do you examine its format?

4.3: Binary I/O

Although text files are used extensively for transferring data between systems and between applications, binary files are much more commonly used to store internal application data. Among the reasons:

- Binary file data can usually be processed much faster than text data
- Many data types, such as graphics, have no natural text representation

In this section we explore what we mean by a binary file. We then present some SimpleIO functions for working with such files.

4.3.1: Text vs. Binary Files

What is the difference between a text and a binary file? It's actually pretty simple. A text file is a file created with two important characteristics:

- All non-text data contained in the file represented using text equivalents (e.g., 125 would be represented by the three characters '1', '2', and '5').
- The separation between lines is indicated by specific characters—which may vary by operating system. Normally, the ends of line are indicated either by a single '\n' character, or by a "\r\n" combination. (The PC normally uses the two-character combination).

A binary file is any file that is not a text file.

The reason that binary files tend to be more efficient than text files has to do with translation. When writing to a text file, all non-character data has to be translated into text form (e.g., using the sprintf() function). When reading from a text file, the reverse translation needs to be made (e.g., the atoi() and atof() functions). Binary files normally handle data entirely differently. Instead of translating, they write data directly from RAM to disk, and read it from disk to RAM. It's as if you were doing a memcpy() directly to disk and back.

4.3.2: Reading and Writing Binary Files

What is really different about the binary file I/O functions is the use of block I/O. Whereas text I/O functions translate between data and its ASCII representation, binary I/O leaves the data unchanged. For this reason, C++ binary I/O can be accomplished using just two functions: read() and write(). The arguments to the functions are: 1) the address of the data in memory and 2) the number of bytes being copied. The functions are prototyped as follows:

```
istream& read(char *szBuf,int nCount);
ostream& write(const char *szBuf,int nCount);
```

The difference between the two prototypes is subtle. In the case of the write() function, we'll be copying from memory to the stream—so the location of our data in memory is declared to be const, since we aren't changing its contents, just copying them. In the case of read(), we are bring data from the stream into the block of memory pointed to by szBuf—so that particular pointer cannot, by definition, be declared const.

read() And write() Arguments
When first trying to apply the read() and write() functions, two problems quickly become obvious. First, the functions appear to be applicable only to character strings (which is how the first argument is declared). Second, it is not altogether obvious how to figure out how many bytes you're reading and writing (the second argument).

With respect to the first problem, a typecast can be used to force the compiler to allow us to read or write from non-text locations. For example, if we wanted to read an 8-byte double from an open input stream myin, we could use the following code:

```
double dDest;
myin.read((double *)&dDest,8);
```

To write the same double to an open output stream myout:

```
myout.write((double *)&dDest,8);
```

With respect to the second problem, **sizeof** operator is a special type of operator that is evaluated by the compiler and returns the number of bytes required for its argument. The operator may be applied to variables, expressions and data types. In each case, it returns the following result:

sizeof *scalar-variable-name*	➔	returns size of variable's data type
sizeof *array-name*	➔	returns number of bytes to hold entire array
sizeof (*data-type*)	➔	returns size of the data type
sizeof *other-expression*	➔	returns size of the return type

Parentheses are only required when the operator is applied to a data type, e.g., **sizeof**(int), **sizeof**(struct employee) or **sizeof**(STREAM).

Suppose, then, we wanted to save an array of 10 integers to an open file stream bintest. This could be accomplished in two ways (that are completely equivalent):

```
int arTest[10]={1,3,5,7,9,2,4,6,8,0};
// Method 1: Writes entire array arTest to file stream bintest
bintest.write((char *)arTest,sizeof arTest);
// Method 2: Writes entire array arTest to file stream bintest (a second time)
bintest.write((char *)arTest,10 * sizeof(int));
```

Reading is accomplished in the same manner. In fact, the write() and read() functions are specifically designed so that writing and reading looks the same, e.g.:

```
int arTest[10];
// Method 1: Reads bytes from disk into arTest
bintest.read((char *)arTest,sizeof arTest);
```

```
// Method 2: Reads bytes from disk into arTest a second time
bintest.read((char *)arTest,10 * sizeof(int));
```

In using **sizeof**, it is important to recognize where it is and is not appropriate. Despite its appearance, **sizeof** is an operator, not a function. That is why it can be applied without parentheses (except when a data type is specified). More importantly, the actual value returned by **sizeof** is computed by the compiler—it is not computed while the program is running, the way a function call is. Thus, when **sizeof** is applied to an array of characters, it returns the size of the entire array—not the length of any string that happens to be in it. Similarly, when applied to a character pointer that contains the address of a string, it returns 4—or whatever the size of a pointer is. It does not return the length of the string! Do not make the common mistake of confusing **sizeof** and strlen() in your functions.

Binary I/O in SimpleIO

SimpleIO provides a number of functions for reading from and writing to a binary file. These are listed in Table 4.5.

Table 4.5: SimpleIO functions for binary file I/O
bool WriteInteger(STREAM &file,int nVal); Writes a single integer to the open binary file stream. It returns **true** if the operation is successful, **false** otherwise.
bool WriteReal(STREAM &file,double dVal); Writes a single double real number to the open file stream. It returns **true** if the operation is successful, **false** otherwise.
bool WriteFloat(STREAM &file,float dVal); Writes a single float real number to the open file stream. It returns **true** if the operation is successful, **false** otherwise.
bool WriteCharacter(STREAM &file,char cVal); Writes a single character to the open file stream. It returns **true** if the operation is successful, **false** otherwise.
bool WriteShort(STREAM &file,short nVal); Writes a short (2-byte) integer to the open file stream. It returns **true** if the operation is successful, **false** otherwise.
bool WriteBlock(STREAM &file,const void *buf,int nBytes); Writes an array of any type of data, starting at position *buf*, to the open binary file stream. The amount of data to be written is *nBytes*. It returns **true** if the operation is successful, **false** otherwise.
int ReadInteger(STREAM &file); Reads an integer from the open binary file stream, returning its value.
double ReadReal(STREAM &file); Reads a double from the open binary file stream, returning its value.
float ReadFloat(STREAM &file); Reads a float from the open binary file stream, returning its value.
char ReadCharacter(STREAM &file); Reads a character from the open binary file stream, returning its value.
short ReadShort(STREAM &file); Reads a short (2-byte) integer from the open binary file stream, returning its value
bool ReadBlock(STREAM &file,void *buf,int nBytes); Reads an array from the open binary file stream, copying *nBytes* of data from the file to the memory location beginning at position *buf*. It returns **true** if the operation is successful, **false** otherwise.

The first three functions for writing data (i.e., WriteInteger, WriteReal and WriteCharacter) behave exactly the way their SimpleIO text counterparts (i.e., PrintInteger, PrintReal and PrintCharacter) do. The only difference is that raw data is written, instead of character strings. The same is also true of the first three functions for reading data (i.e., ReadInteger, ReadReal and ReadCharacter), which correspond to the text input functions (i.e., GetInteger, GetReal and GetCharacter). An additional group of functions (WriteFloat, ReadFloat, WriteShort and ReadShort) is also provided. Such functions were unnecessary in text I/O, since there is little need to distinguish between float (4-byte) and double (8-byte) reals, or short (2-byte) and regular (4-byte) integers when going back and forth to text. In a binary file, however, it is critical that any data be read exactly as it was saved. We will see an example of the use of these functions in the binary file example presented in Section 4.4.2.

The definitions of the SimpleIO output functions are presented in Example 4.11. Each function is very simple, calling the stream write() member either directly or indirectly (through the WriteBlock() function). WriteBlock() itself was also simple, returning true if the write operation was successful, false otherwise.

Example 4.11: SimpleIO Functions for Binary I/O Output

```
bool WriteBlock(STREAM &file,const void *buf,int nBytes)
{
      file.write((char *)buf,nBytes);
      return !file.fail();
}
bool WriteInteger(STREAM &file,int nVal)
{
      return WriteBlock(file,&nVal,sizeof(int));
}
bool WriteReal(STREAM &file,double dVal)
{
      return WriteBlock(file,&dVal,sizeof(double));
}
bool WriteFloat(STREAM &file,float fVal)
{
      return WriteBlock(file,&fVal,sizeof(float));
}
bool WriteCharacter(STREAM &file,char cVal)
{
      return WriteBlock(file,&cVal,sizeof(char));
}
bool WriteShort(STREAM &file,short nVal)
{
      return WriteBlock(file,&nVal,sizeof(short));
}
```

The SimpleIO binary input functions (Example 4.12) are also simple. The majority of functions work by declaring a local variable (e.g., int, double, char, short) , calling ReadBlock() to acquire the data from the stream, then returning the value. The ReadBlock() function—which is also used for reading text data—issues a call to read() then returns true if the read was successful, false otherwise.

Example 4.12: SimpleIO Functions for Binary I/O Input

```
bool ReadBlock(STREAM &file,void *buf,int nBytes)
{
      file.read((char *)buf,nBytes);
      return !file.fail();
}
int ReadInteger(STREAM &file)
{
      int nVal;
      if (!ReadBlock(file,&nVal,sizeof(int))) nVal=0;
      return nVal;
}
double ReadReal(STREAM &file)
{
      double dVal;
      if (!ReadBlock(file,&dVal,sizeof(double))) dVal=0.0;
      return dVal;
}
float ReadFloat(STREAM &file)
{
      float fVal;
      if (!ReadBlock(file,&fVal,sizeof(float))) fVal=0.0;
      return fVal;
}
char ReadCharacter(STREAM &file)
{
      char cVal;
      if (!ReadBlock(file,&cVal,sizeof(char))) cVal=0;
      return cVal;
}
short ReadShort(STREAM &file)
{
      short nVal;
      if (!ReadBlock(file,&nVal,sizeof(short))) nVal=0;
      return nVal;
}
```

4.3.3: Moving around in a binary file

Text files tend to be processed sequentially—with each line being read in then discarded. Binary files, in contrast, are often read randomly—meaning that we move around in the file to find specific data. By doing so, we can use our binary files as a substitute for RAM—any data we need we can read from the file, it just takes longer. Virtually all real-world databases are constructed on this principle: keep data on disk until it is specifically needed.

Concept of File Position
Before discussing the functions SimpleIO provides for navigating within a file, it is useful to consider the key similarities and differences between RAM and file storage. The two most important similarities are:

- Both files and RAM store data using digital representation. Thus, there are no problems moving data back and forth between files and RAM (as we saw using the ReadBlock() and WriteBlock() functions).
- Like RAM, every byte in a file has an address. The addressing scheme for files is simple: the first byte is 0, and the remaining bytes are numbered accordingly.

There is one important difference between files and RAM, however. Files always keep track of their current "position"—a concept that is not meaningful in RAM. Yet virtually every file operation takes place relative to its current position (sometimes called the file cursor, file window or file pointer). Thus, we need to explore the concept more fully.

The reason files keep track of a position is fairly easy to understand. In the early days of computing, card readers and tape drives were the most common devices used for mass storage. These devices are as different from today's hard drives as a cassette recorder is from a CD. You can't just jump from point A to point B in a cassette drive—it takes time because you've got to through all the intervening tape. (Or, in the case of a card reader, go through the stack of cards). For such devices, there is a huge advantage to keeping track of where you are. If, for example, you are at position 10,000 and need to go to 10,001, it would be very inefficient to rewind the tape to 0, then count 10,001 bytes from the beginning.

When a file is opened, it normally starts at position 0 unless the file is opened for appending—in which case the starting position is the end of the file, whatever that happens to be. From that point on, every read or write operation updates the position (just the way that *listening* to a song or *recording* a song changes the position on a cassette tape). Thus, after the following code:

```
STREAM in;
char buf[256];
if (Open(in,"Example.bin",true,false,false,false)) {
        ReadBlock(in,buf,250);
}
```

our new position in the file would be 250 (assuming the file opened properly). Our next read would take place at that position.

Setting File Position
In C++, the file position can be set with two separate member functions. The setp() function sets the position in a file opened for writing (think of p for print). The setg() function sets the position in file opened for reading (think g for get). Although there are a number of versions of the functions (described in Chapter 11), for the time being we can

make due with a single version—taking the file position as an argument. Thus, if mystrm were a file stream open for writing, to set the file pointer to the beginning of the file we would call:

```
mystrm.setp(0);
```

SimpleIO also provides two functions for setting file position:

```
bool SetWritePosition(STREAM &file,int nPos);
bool SetReadPosition(STREAM &file,int nPos);
```

Either can be used to position the file pointer in a read/write file.

4.3.4: Binary File Demonstration

Walkthrough available in BinaryIO.avi

For purposes of comparison, we can compare the functions used to create text and binary files. In Example 4.13, the SaveBinary() function is presented, designed to save exactly the same data saved in the SaveText() function presented earlier (Example 4.8).

Example 4.13: SaveBinary() function

```
bool SaveBinary(const char *szFile)
{
	STREAM out;
	int nCount,i;
	if (!Open(out,szFile,false,true,true,false)) return false;
	DisplayString("How many values do you want to save? (1 to 10): 
");
	nCount=InputInteger();
	if (nCount<1 || nCount>10) {
		DisplayString("Illegal value: must be between 1 and 10");
		return false;
	}
	WriteInteger(out,nCount);
	for(i=0;i<nCount;i++) {
		WriteInteger(out,arID[i]);
		WriteReal(out,arHeight[i]);
		WriteCharacter(out,arSex[i]);
		WriteInteger(out,arWeight[i]);
	}
	Close(out);
	return true;
}
```

The function is very similar to its text counterpart, first opening the file (this time, in binary mode), then prompting the user for the number of values to be saved. Among the key differences that need to be noted are the following:

- The binary SimpleIO functions (i.e., Write…) are used to save data to the file, instead of the text functions (i.e., Print…).
- No lines or spacing characters are inserted into the file. Such characters are unnecessary, since the data size of each element being saved is fixed (i.e., integers are 4 bytes, doubles are 8 bytes, characters are 1 bytes). That means that separators to tell us where each data element starts and ends are unnecessary.

The DisplayBinary() function, presented in Example 4.14, is also very similar to its text counterpart, DisplayText() (Example 4.9). In fact, the function calls to display text to the console are identical. What is different, however, is how data is read.

Example 4.14: DisplayBinary() function

```
bool DisplayBinary(const char *szFile)
{
	STREAM in;
	int nCount,i;
	if (!Open(in,szFile,true,false,false,false)) return false;
	nCount=ReadInteger(in);
	NewLine();
	DisplayString("ID\tHeight\tSex\tWeight\n--\t------\t---\t------
\n");
	for(i=0;i<nCount;i++) {
		int nID;
		double dHeight;
		char cSex;
		int nWeight;
		// Reading data
		nID=ReadInteger(in);
		dHeight=ReadReal(in);
		cSex=ReadCharacter(in);
		nWeight=ReadInteger(in);
		// Displaying data to console
		DisplayInteger(nID);
		DisplayCharacter('\t');
		DisplayReal(dHeight);
		DisplayCharacter('\t');
		DisplayCharacter(cSex);
		DisplayCharacter('\t');
		DisplayInteger(nWeight);
		NewLine();
	}
	Close(in);
	return true;
}
```

The key differences in the data reading are as follows:

- Binary SimpleIO input functions (i.e., Read…) are used in place of text input functions (i.e., Get…).
- ID must be read as an integer. Whereas when we read the text file, we could choose to interpret the data as text or numeric, that choice is not available in a binary file. Since the value was saved as a 4-byte integer, it must be read that way.

The difference text and binary files can be understood by looking at the actual bytes in binary and text files. Visual Studio .Net makes this easy to do, as it allows any file to be opened in a binary display format. This is accomplished by:

- selecting File|Open
- clicking "Open With…" (as shown in Figure 4.6), and
- selecting "Binary Editor" from the resulting list of options.

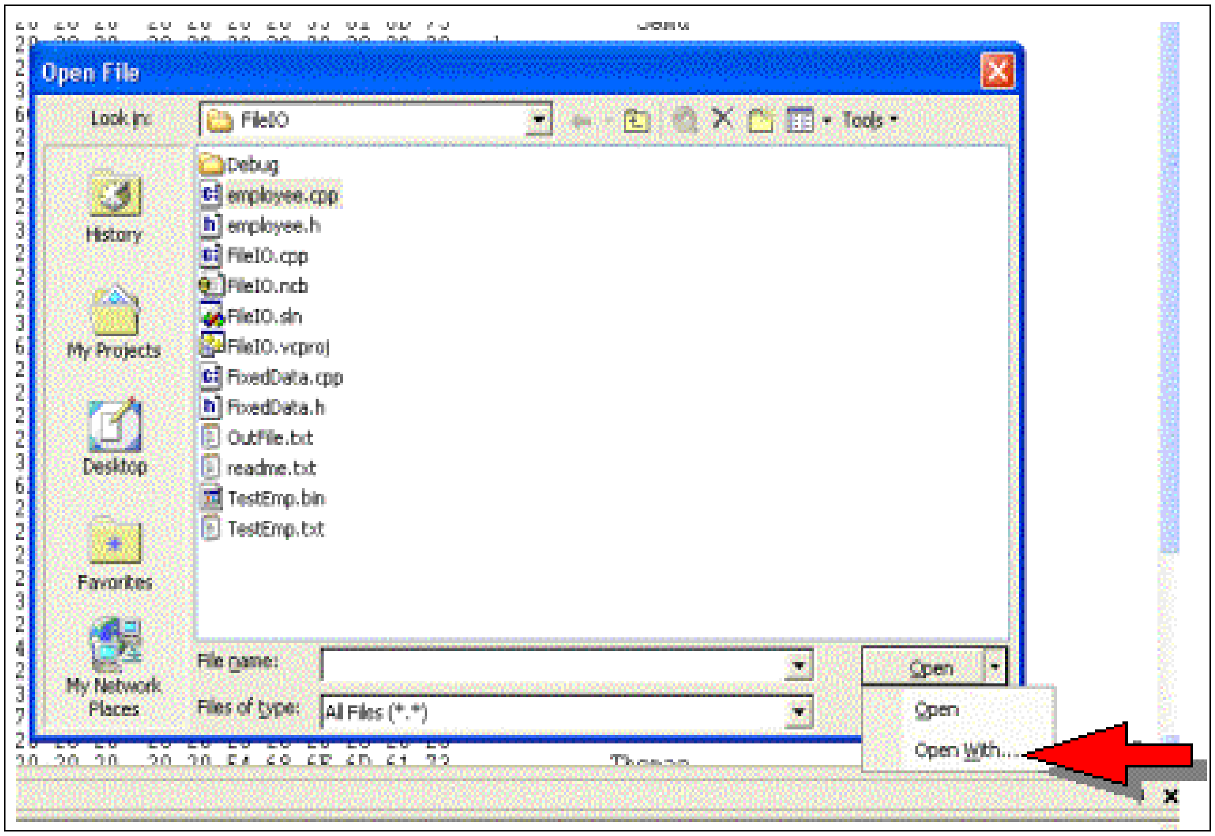
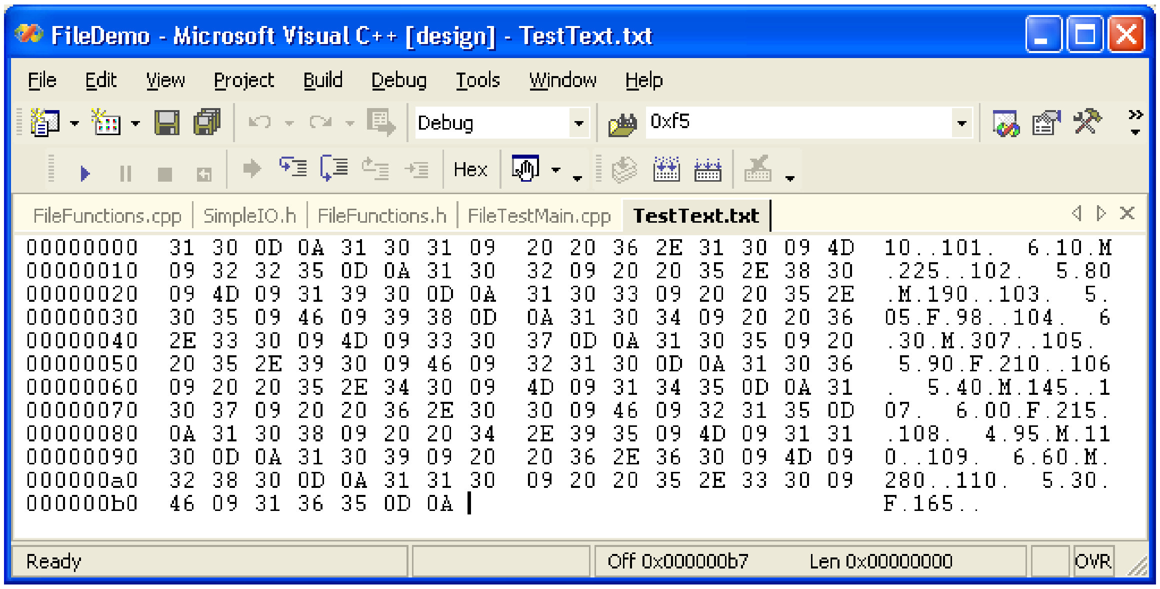


Figure 4.6: Opening file using "Open With" option

The text file TestText.txt (generated by running SaveFormattedText(), in Example 4.10) is presented in a binary file viewer in Figure 4.7. A binary file containing the same data (generated by calling SaveBinary(), in Example 4.13) is shown in Figure 4.8.



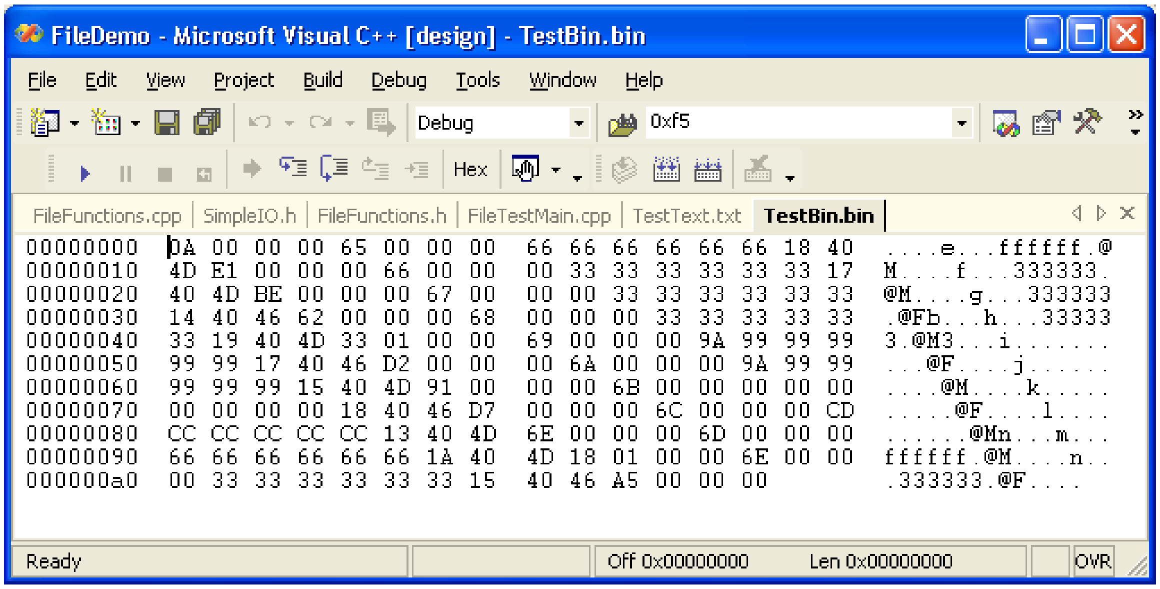
```
00000000  31 30 0D 0A 31 30 31 09  20 20 36 2E 31 30 09 4D  10..101.  6.10.M
00000010  09 32 32 35 0D 0A 31 30  32 09 20 20 35 2E 38 30  .225..102.  5.80
00000020  09 4D 09 31 39 30 0D 0A  31 30 33 09 20 20 35 2E  .M.190..103.  5.
00000030  30 35 09 46 09 39 38 0D  0A 31 30 34 09 20 20 36  05.F.98..104.  6
00000040  2E 33 30 09 4D 09 33 30  37 0D 0A 31 30 35 09 20  .30.M.307..105.
00000050  20 35 2E 39 30 09 46 09  32 31 30 0D 0A 31 30 36   5.90.F.210..106
00000060  09 20 20 35 2E 34 30 09  4D 09 31 34 35 0D 0A 31  .  5.40.M.145..1
00000070  30 37 09 20 20 36 2E 30  30 09 46 09 32 31 35 0D  07.  6.00.F.215.
00000080  0A 31 30 38 09 20 20 34  2E 39 35 09 4D 09 31 31  .108.  4.95.M.11
00000090  30 0D 0A 31 30 39 09 20  20 36 2E 36 30 09 4D 09  0..109.  6.60.M.
000000a0  32 38 30 0D 0A 31 31 30  09 20 20 35 2E 33 30 09  280..110.  5.30.
000000b0  46 09 31 36 35 0D 0A                              F.165..
```

Figure 4.7: Binary view of text file TestText.txt

The differences between the two files are immediately apparent. In the text file, the first piece of information is the number of data items followed by a newline pair, presented as the bytes:

0x31 ('1'), 0x30 ('0'), 0x0D ('\r') , 0x0A ('\n')

(ASCII equivalents of printable characters are printed in the right 16 columns, with periods used to signify a character is not printable). We then see the data elements separated by tab (0x09) characters. Because the file contains text, most of the ASCII equivalent area on the right hand side of the viewer contains printable characters.


FileDemo - Microsoft Visual C++ [design] - TestBin.bin
File Edit View Project Build Debug Tools Window Help
Debug
0xf5
Hex
FileFunctions.cpp
SimpleIO.h
FileFunctions.h
FileTestMain.cpp
TestText.txt
TestBin.bin
Ready
OVR

```
00000000  0A 00 00 00 65 00 00 00  66 66 66 66 66 66 18 40  ....e...ffffff.@
00000010  4D E1 00 00 00 66 00 00  00 33 33 33 33 33 33 17  M....f...333333.
00000020  40 4D BE 00 00 00 67 00  00 00 33 33 33 33 33 33  @M....g...333333
00000030  14 40 46 62 00 00 00 68  00 00 00 33 33 33 33 33  .@Fb...h...33333
00000040  33 19 40 4D 33 01 00 00  69 00 00 00 9A 99 99 99  3.@M3...i.......
00000050  99 99 17 40 46 D2 00 00  00 6A 00 00 00 9A 99 99  ...@F....j......
00000060  99 99 99 15 40 4D 91 00  00 00 6B 00 00 00 00 00  ....@M....k.....
00000070  00 00 00 00 18 40 46 D7  00 00 00 6C 00 00 00 CD  .....@F....l....
00000080  CC CC CC CC CC 13 40 4D  6E 00 00 00 6D 00 00 00  ......@Mn...m...
00000090  66 66 66 66 66 66 1A 40  4D 18 01 00 00 6E 00 00  ffffff.@M....n..
000000a0  00 33 33 33 33 33 33 15  40 46 A5 00 00 00        .333333.@F....
```

Off 0x00000000 Len 0x00000000

Figure 4.8: Binary view of TestBin.bin

In the binary file, on the other hand, the first four bytes are:

0x0A 0x00 0x00 0x00

This is a 4-byte integer saved in reverse order (which is how such integers are saved in a PC), meaning it is equivalent to writing the hex number:

0x0000000A

This is the same as 10 in decimal. This is, of course, the number of data element saved—just as it was for the text file. Following that:

- The next 4 bytes are the integer 0x00000065, which is 101 in decimal—the ID of the first element.
- The next 8 bytes are a double—which is nearly impossible to interpret.
- The next byte (first byte of the second row) is the character data identifying sex, which happens to be 0x4D, or 'M'.

- This is followed by the 4-byte weight integer 0x000000E1, which is 14*16+1 = 225 in decimal.

That is followed by the information for the next data item. You can get a sense of the layout of the overall layout file by looking the text side. Each item that we saved required 17 bytes (4 for ID, 8 for height, 1 for sex, 4 for weight). Since each row in the display shows 16 bytes, the diagonal pattern in the left hand side reflects the position of the sex character within each item, one byte further over in each row. Most other characters on the right hand side are purely coincidental, values contained in integers or real numbers that happen to correspond to ASCII characters.

Test your understanding: What is the significance of the 'e' that appears in the first row of the Figure 4.8 display?

4.3 Section Questions

1. Explain why reading and writing binary data is similar to the C function memcpy.
2. Why are binary files usually faster to work with than text files?
3. Why are the functions for positioning a file pointer nearly always reserved for use in files opened in binary mode?
4. Why does a file opened in binary model look like a memory grid from Chapters 10 and 11?
5. What's the difference between saving a structure as a block and saving it element by element?
6. Under what circumstances might it make sense to save a structure as a block?
7. What is the near-universal approach for saving collections of data (e.g., arrays)
8. Why might changes in integral type sizes (e.g., changing **long int** from 4 to 8 bytes) be of greater concern for applications using binary files than for those utilizing text files?

4.4 Fixed Length Text File Walkthrough

Walkthrough available in FixedFilm.avi

In this section we present a walkthrough that shows how information can be extracted from a text file where the data is laid out in columns of fixed width. In a business context, this type of data file is fairly common, as a fixed width layout can be produced directly by many applications (e.g., MS Access) and is also sometimes created by printing data from an application to a text printer whose output is directed to a file—something that can easily be set up in MS Windows.

4.4.1: Processing a fixed width text file

The particular file we will be working with contains a collection of some of the films released in 1999. It is a pure text file, with data laid out in columns. A section of the file is presented in Example 4.15.

Example 4.15: Sample of Films1999.txt

```
1    1999 200 Cigarettes                            0 1 0 1 0 0 0 0 0 0 0 0 0
2    1999 2001 Yonggary                             0 0 0 0 1 0 0 1 0 0 0 0 0
3    1999 8MM                                       0 0 0 0 0 0 1 0 0 0 0 0 0
4    1999 Magical Legend of the Leprechauns, The  0 0 0 0 1 0 0 0 0 0 0 0 0
5    1999 Storm of the Century                      0 0 0 0 0 0 0 0 0 0 0 0 1
6    1999 10 Things I Hate About You                0 1 0 0 0 0 0 0 0 0 0 1 0
7    1999 13th Warrior, The                         1 0 0 0 1 0 0 0 0 0 0 0 0
8    1999 Alice in Wonderland                       0 0 0 0 1 1 0 0 0 0 0 0 0
9    1999 All the King's Men                        0 0 0 0 0 0 0 0 1 0 0 0 0
10   1999 American Beauty                           0 0 0 1 0 0 0 0 0 0 0 0 0
11   1999 American Pie                              0 1 0 0 0 0 0 0 0 0 0 0 0
12   1999 Analyze This                              0 1 0 0 0 0 0 0 0 0 0 0 0
13   1999 Angela's Ashes                            0 0 0 1 0 0 0 0 0 0 0 0 0
14   1999 Animal Farm                               0 0 0 0 0 1 0 0 0 0 0 0 0
15   1999 Anna and the King                         0 0 0 1 0 0 0 0 0 0 0 1 0
16   1999 Any Given Sunday                          0 0 0 1 0 0 0 0 0 0 0 0 0
17   1999 Anywhere But Here                         0 0 0 1 0 0 0 0 0 0 0 0 0
18   1999 Apartment Complex, The                    0 0 0 0 0 0 1 0 0 0 0 0 0
```

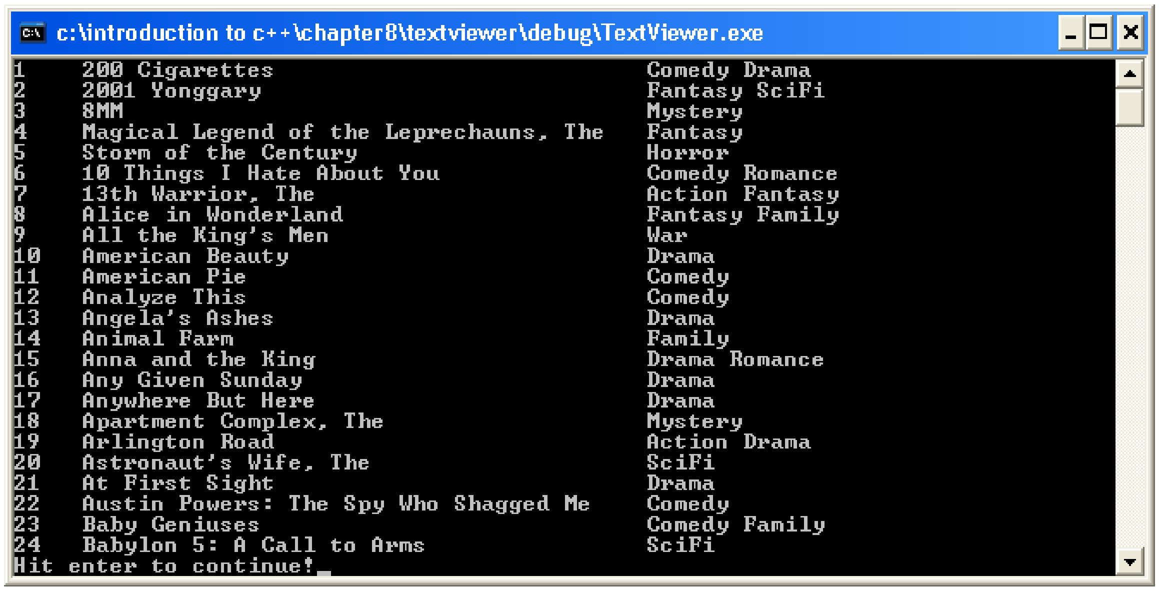
The definitions of the data elements in the file layout is presented in Table 4.6, which identifies the 0-based start column for each data element, its length and a description. A table such as this can be easily prepared using a text editor, such as MS-Notepad or the editor in MS Visual Studio .Net, which displays the column position of the cursor.

Table 4.6: Films1999.txt Layout

Start Column	Data Length	Description
0	5	ID code for the file
5	4	Year film was released (always 1999 in this file)
10	40	Film title
50	1	Action (1 if TRUE, 0 if FALSE)
52	1	Comedy (1 if TRUE, 0 if FALSE)
54	1	Cartoon (1 if TRUE, 0 if FALSE)
56	1	Drama (1 if TRUE, 0 if FALSE)
58	1	Fantasy (1 if TRUE, 0 if FALSE)
60	1	Family (1 if TRUE, 0 if FALSE)
62	1	Mystery (1 if TRUE, 0 if FALSE)
64	1	SciFi (1 if TRUE, 0 if FALSE)
66	1	War (1 if TRUE, 0 if FALSE)
68	1	Western (1 if TRUE, 0 if FALSE)
70	1	Musical (1 if TRUE, 0 if FALSE)
72	1	Romance (1 if TRUE, 0 if FALSE)
74	1	Horror (1 if TRUE, 0 if FALSE)

Application Description

The application being presented takes the text file and displays the film title, along with the categories it has been classified under. It pauses every 24 lines (i.e., when the screen fills up) and prompts the user for a key. Sample output is displayed in Figure 4.9.


c:\introduction to c++\chapter8\textviewer\debug\TextViewer.exe

```
1     200 Cigarettes                              Comedy Drama
2     2001 Yonggary                               Fantasy SciFi
3     8MM                                         Mystery
4     Magical Legend of the Leprechauns, The      Fantasy
5     Storm of the Century                        Horror
6     10 Things I Hate About You                  Comedy Romance
7     13th Warrior, The                           Action Fantasy
8     Alice in Wonderland                         Fantasy Family
9     All the King's Men                          War
10    American Beauty                             Drama
11    American Pie                                Comedy
12    Analyze This                                Comedy
13    Angela's Ashes                              Drama
14    Animal Farm                                 Family
15    Anna and the King                           Drama Romance
16    Any Given Sunday                            Drama
17    Anywhere But Here                           Drama
18    Apartment Complex, The                      Mystery
19    Arlington Road                              Action Drama
20    Astronaut's Wife, The                       SciFi
21    At First Sight                              Drama
22    Austin Powers: The Spy Who Shagged Me       Comedy
23    Baby Geniuses                               Comedy Family
24    Babylon 5: A Call to Arms                   SciFi
Hit enter to continue!_
```

Figure 4.9: Sample Output from TextViewer application

The TextViewer application consists of 3 functions:

- *ListFilms:* Reads each line of the text file into a buffer, the calls DisplayFilm() to cause it to be displayed the console.
- *DisplayFilm:* Takes a text string containing a single line from the file and displays the desired data elements to the console.
- *GetField:* Retrieves the data for a specific element (e.g., title) from a string containing all the data for the file.

These functions are now presented.

TextViewer Functions
The driving function for the TextViewer application is the ListFilms() function, presented in Example 4.16. The function is prototyped as follows:

    void ListFilms(const char szFile[]);

Its argument is the name of the file containing the film data.

Example 4.16: ListFilms() function

```
void ListFilms(const char szFile[])
{
      STREAM in;
      char szIn[4000];
      int nCount;
      // Open existing file for text reading
      if (!Open(in,szFile,true,false,false,true)) {
            DisplayString("File could not be opened!");
            NewLine();
            return;
      }
      for(nCount=1;GetLine(in,szIn,4000);nCount++) {
            DisplayFilm(szIn);
            if (nCount%24==0) {
                  DisplayString("Hit enter to continue!");
                  InputCharacter();
            }
      }
      Close(in);
}
```

The function illustrates a fairly common approach to reading a text file. It works as follows:

- It opens the file by calling the SimpleIO function Open(), specifying the file is to be read, is not to be truncated, and is a text file.
- It enters a loop that reads each line into a buffer by calling GetLine(). The maximum line length was arbitrarily set to 4000—although 80 would have been more than enough for the file in question.

    - The loop takes each line and sends to DisplayFilm()—which performs the actual display activity.
    - Every time the line number is an even multiple of 24 (computed using the remainder operator) it prompts the user to hit enter. This keeps the list from scrolling off the screen before it can be seen.
    - When GetLine() fails—which will occur when the end of the file is reached—the loop ends.
- The file is closed.

Before discussion the DisplayFilm() function, it is helpful to skip to the GetField() function, presented in Example 4.17, which extracts a single data element from the line read from the file. It is prototyped as follows:

```
void GetField(char szTarget[],const char szBuf[],int nStart,int nCount);
```

The *szTarget[]* argument will be used to hold the extracted data (e.g., similar to the first argument of the strcpy() library function). The second argument, *szBuf[]*, contains the line loaded from the file (in the ListFiles() function, that was passed into the DisplayFilm() function). The third argument, *nStart*, contains the starting position of the data, the fourth argument, *nCount*, the number of bytes of data to be copied. Appropriate values for these two arguments can be found in Table 4.6. For example, if szIn[] contained a line of text read from the file and szField[] was our target string, then the call:

```
GetField(szField,szIn,10,40);
```

would cause the 40 bytes of "Film title" information to be copied from szIn into szField.

Example 4.17: GetField() function

```
void GetField(char szTarget[],const char szBuf[],int nStart,int nCount)
{
	int i;
	for(i=0;i<nCount;i++) {
		szTarget[i]=szBuf[i+nStart];
	}
	szTarget[i]='\0';
}
```

The function’s operation is very simple, and reminiscent of many of the string functions introduced in Chapter 3. It works as follows:

- Starting at nStart in the szBuf[] array, it copies over nCount characters to szTarget[].
- It places a NUL terminator at the end of szTarget, to make it a NUL-terminated string.

The final function to be discussed is DisplayFilm(), shown in Example 4.18. This function is prototyped as follows:

void DisplayFilm(const char szBuf[])

Its argument is a single line read from the film file in ListFilms().

Example 4.18: DisplayFilm() function

```
void DisplayFilm(const char szBuf[])
{
	char szTarget[256];
	// ID
	GetField(szTarget,szBuf,0,5);
	DisplayString(szTarget);
	// Title
	GetField(szTarget,szBuf,10,40);
	DisplayString(szTarget);
	// Film types
	GetField(szTarget,szBuf,50,1);
	if (szTarget[0]=='1') DisplayString(" Action");
	GetField(szTarget,szBuf,52,1);
	if (szTarget[0]=='1') DisplayString(" Comedy");
	GetField(szTarget,szBuf,54,1);
	if (szTarget[0]=='1') DisplayString(" Cartoon");
	GetField(szTarget,szBuf,56,1);
	if (szTarget[0]=='1') DisplayString(" Drama");
	GetField(szTarget,szBuf,58,1);
	if (szTarget[0]=='1') DisplayString(" Fantasy");
	GetField(szTarget,szBuf,60,1);
	if (szTarget[0]=='1') DisplayString(" Family");
	GetField(szTarget,szBuf,62,1);
	if (szTarget[0]=='1') DisplayString(" Mystery");
	GetField(szTarget,szBuf,64,1);
	if (szTarget[0]=='1') DisplayString(" SciFi");
	GetField(szTarget,szBuf,66,1);
	if (szTarget[0]=='1') DisplayString(" War");
	GetField(szTarget,szBuf,68,1);
	if (szTarget[0]=='1') DisplayString(" Western");
	GetField(szTarget,szBuf,70,1);
	if (szTarget[0]=='1') DisplayString(" Musical");
	GetField(szTarget,szBuf,72,1);
	if (szTarget[0]=='1') DisplayString(" Romance");
	GetField(szTarget,szBuf,74,1);
	if (szTarget[0]=='1') DisplayString(" Horror");
	NewLine();
}
```

The function works by calling GetField() to extract the data for each field that needs to be displayed. ID and Title are extracted first, as these are displayed for every film. After that, each of the film type specifiers is checked. If a '1' is present, a type string is added to the display line.

4.4.2: Lab Exercise: Displaying a List of Films by Category

Another useful way to display the information in the Films1999.txt file is to organize it by category. In this lab exercise, you will create three functions that can be used for this purpose.

Objective
The objective of this exercise is to create a function that reads the film input file (i.e., Films1999.txt) an writes the data to an output file, organized by category. An extract of some of the output file, taken from the middle, is presented in Example 4.19. It should be noted that films may appear in more than one category.

Example 4.19: Extract from output file, organized by category

```
War
      9    All the King's Men
      44   Bravo Two Zero
      46   Bridge of Dragons
      77   End of the Affair, The
      101  Hornblower: The Frogs and the Lobsters
      104  Hunley, The
      133  Messenger: The Story of Joan of Arc, The
      157  One Man's Hero
      202  Tea with Mussolini
      206  Three Kings
      223  Wing Commander

Western
      112  Jack Bull, The
      171  Ravenous
      174  Ride with the Devil[vI]
      222  Wild Wild West

Musical
      82   Fantasia/2000
      187  South Park: Bigger Longer & Uncut
      212  Topsy-Turvy

Romance
      6    10 Things I Hate About You
      15   Anna and the King
```
etc…

Functions
The functions needed to implement the exercise are presented in Table 4.7.

Table 4.7: Functions to Implemented "Films by Category" Lab Exercise

void CreateCategoryFile(const char szInFile[],const char szOutFile[])

The driver function for the application, it takes the name of the input file (i.e., Films1999.txt) and the name of the file to which output is to be directed. The function should:

- Open the output file as a truncated text file for writing.
- For each category:
    - Print the name of the category to the output file (e.g., the "War", "Western", etc. in Example 4.10)
    - Call CreateCategoryList(), passing it the column where the category type is located (this can be found in either Table 4.2 or Example 4.9)
- Close the output file prior to returning

void CreateCategoryList(STREAM &out,const char szInFile[],int nStart)

Prints all the films matching a particular category. Its arguments are the open output file stream (out), the name of the input data file, and the column position of the type indicator for the particular category. The function should:

- Open the input file as a nun-truncated, readable text file.
- Read each line of the input file (in a loop similar to that of the ListFilms() function in Example 4.7)
    - Determine if the data at the specified column is '0' or '1'
    - If it is '1', call PrintFilm() to add it to the list in the output file
- Close the input file prior to returning

void PrintFilm(STREAM &out,const char szBuf[])

Prints the film to the output file. The function takes the open output file and a text string containing the data for a given film as arguments. It should:

- Print a tab ("\t") to the output file.
- Extract the ID from szBuf using a call to GetField() and send it to the output file.
- Extract the title from szBuf using a call to GetField() and send it to the output file.
- Send a line end to the output file

Procedure

The easiest way to implement the lab exercise is as follows:

- Set up the TextViewer project
- Add a new .cpp file to the project, with a name such as CategoryFile.cpp
- Write the PrintFilm() function, using DisplayFilm() (Example 4.9) as a model. The main changes are that the category display statements can be eliminated and the Display… statements become Print… statements directed to the output stream passed in as an argument.
- Write the CreateCategoryList() function, which will be very similar to the ListFilms() function (Example 4.7) except that it will only call PrintFilm() for files where the category column passed in as an argument is '1'. (Hint: the value of

the category column can be determined by calling GetField() or it can be looked at directly).
- Write a text version of CreateCategoryFile() that opens the output stream, and calls CreateCategoryList() for a single column (e.g., 50, the "Action" column). Test this function by calling it from your main() function.
- When the single column version of CreateCategoryList() works, expand it by making calls to the remaining categories.

4.5 In Depth: Binary .DBF File Walkthrough

Walkthrough available in DBFFilm.avi

In this section we present a walkthrough that shows how information can be extracted from a binary file organized in a known way. This will demonstrate the reading of binary data and navigating within a binary file.

4.5.1: The .DBF File Format

Before packages like MS Access, MS SQL Server and Oracle were on the scene, the most common PC database file format was based on Ashton Tate's dBase. The key building block of dBase was the .DBF file, which was used to hold data for a single table.

In many ways, actual data is stored in a .DBF file in a manner similar to that found in the fixed length text format presented in Section 4.4. There is, however, a key difference between DBF and text files: DBF files contain information about how the file is laid out in a section of the file known as the header section. Specifically, the header contains information such as:

- The date when the table was last modified
- The number of records (rows) in the table
- The length of each record
- The size and type of each data element

This header information is stored in binary format, organized into 32-byte chunks of information.

The overall organization of a DBF file, used in dBase III, is presented in Figure 4.10.

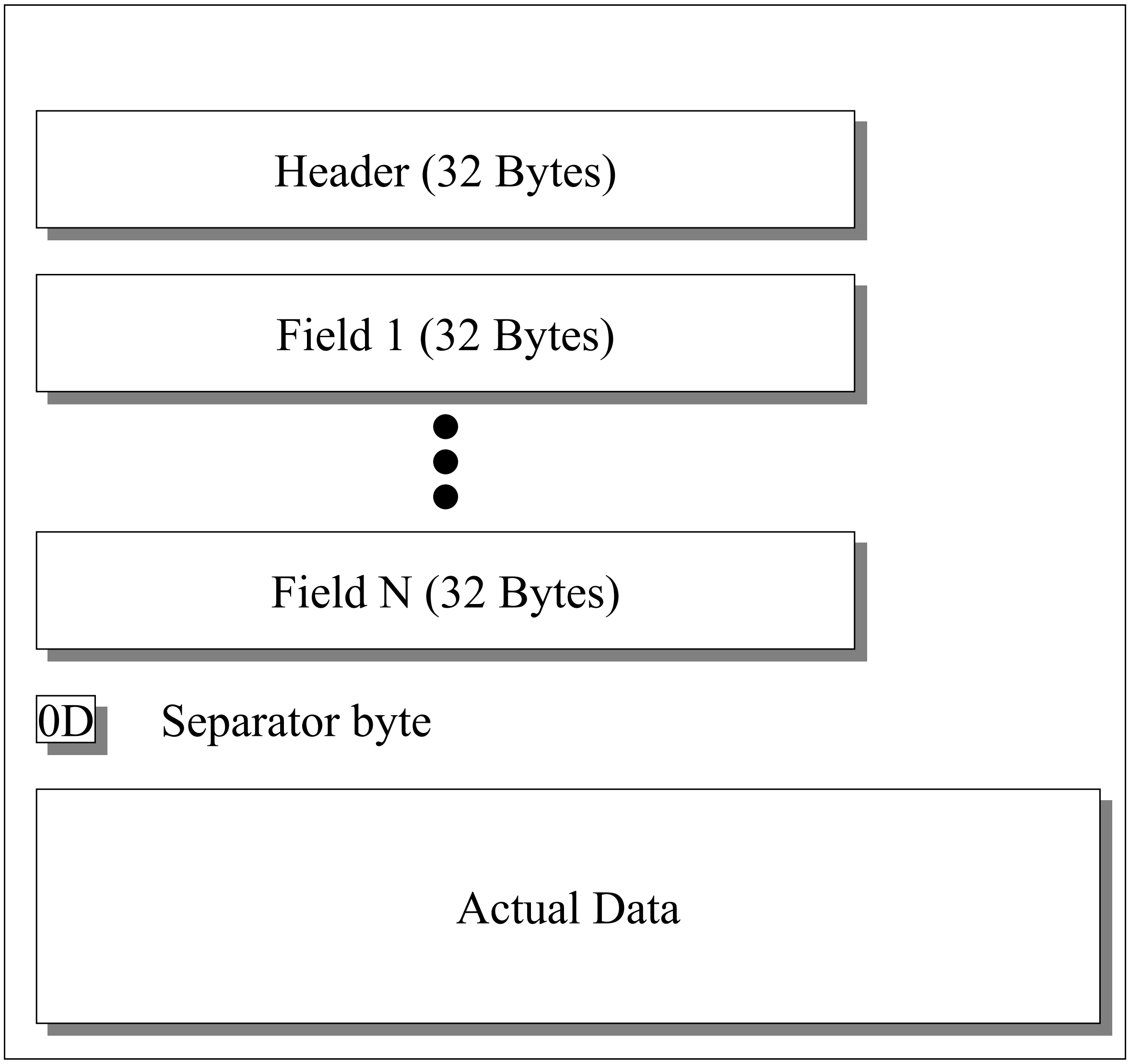
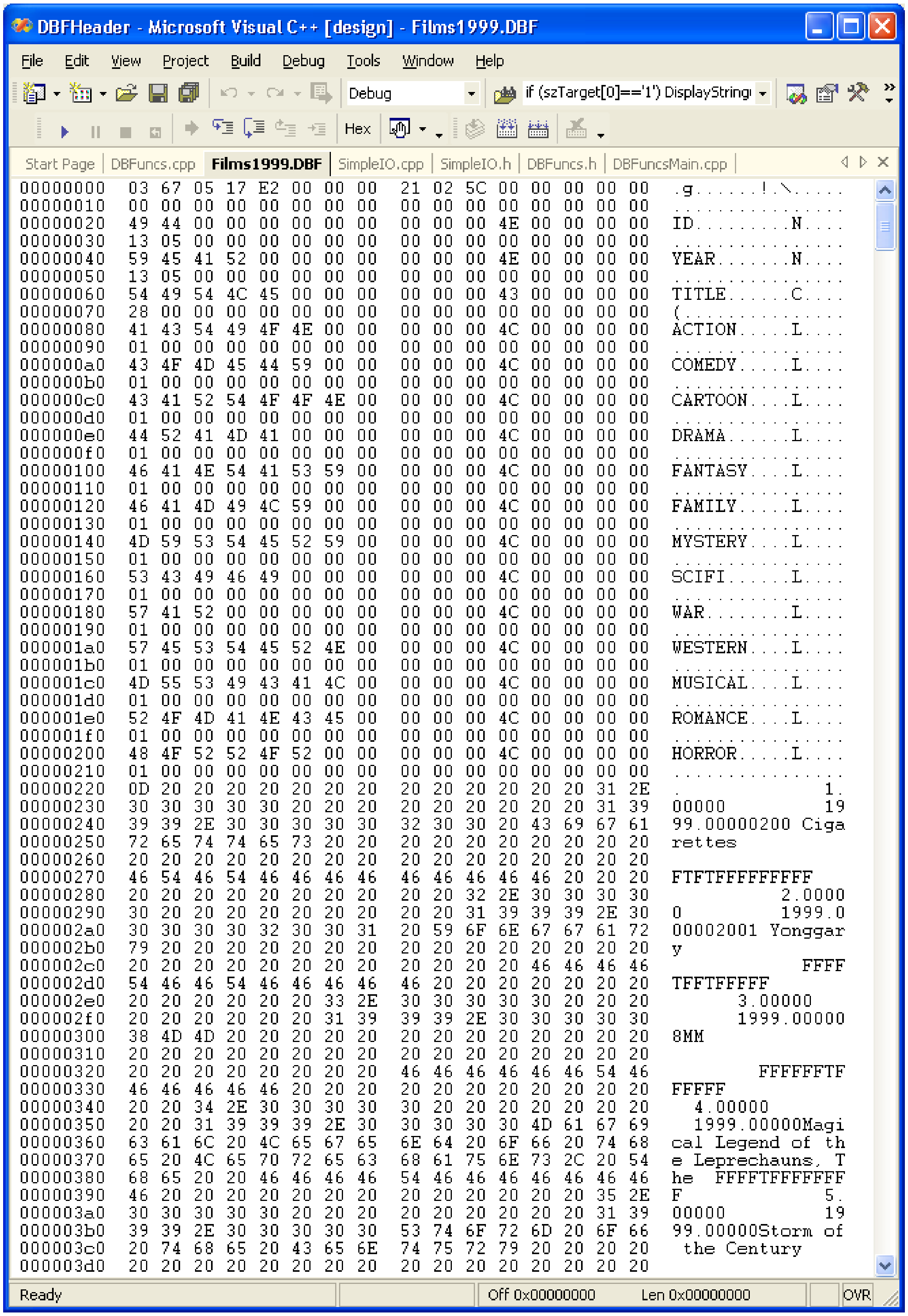


Figure 4.10: DBF file organization

The overall organization is as follows:

- The file begins with 32 bytes of information related to the organization of the entire file, including date modified, record length and number of records.
- Definitions of individual fields (up to 128) then follow. Each definition is 32 bytes and contains information about the field name, data type and size.
- A single carriage return, '\n' follows the field definitions.
- The actual data for the table then begins.

A binary display of the contents of an actual DBF file (Films1999.DBF) is presented in Figure 4.11. This file contains the same data as the Films1999.txt file discussed in Section 4.4.


DBFHeader - Microsoft Visual C++ [design] - Films1999.DBF
File Edit View Project Build Debug Tools Window Help
Debug
if (szTarget[0]=='1') DisplayString
Hex
Start Page | DBFuncs.cpp | Films1999.DBF | SimpleIO.cpp | SimpleIO.h | DBFuncs.h | DBFuncsMain.cpp
Ready
Off 0x00000000
Len 0x00000000
OVR

```
00000000  03 67 05 17 E2 00 00 00  21 02 5C 00 00 00 00 00  .g......!.\.....
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000020  49 44 00 00 00 00 00 00  00 00 00 4E 00 00 00 00  ID.........N....
00000030  13 05 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000040  59 45 41 52 00 00 00 00  00 00 00 4E 00 00 00 00  YEAR.......N....
00000050  13 05 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000060  54 49 54 4C 45 00 00 00  00 00 00 43 00 00 00 00  TITLE......C....
00000070  28 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  (...............
00000080  41 43 54 49 4F 4E 00 00  00 00 00 4C 00 00 00 00  ACTION.....L....
00000090  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000a0  43 4F 4D 45 44 59 00 00  00 00 00 4C 00 00 00 00  COMEDY.....L....
000000b0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000c0  43 41 52 54 4F 4F 4E 00  00 00 00 4C 00 00 00 00  CARTOON....L....
000000d0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000e0  44 52 41 4D 41 00 00 00  00 00 00 4C 00 00 00 00  DRAMA......L....
000000f0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000100  46 41 4E 54 41 53 59 00  00 00 00 4C 00 00 00 00  FANTASY....L....
00000110  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000120  46 41 4D 49 4C 59 00 00  00 00 00 4C 00 00 00 00  FAMILY.....L....
00000130  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000140  4D 59 53 54 45 52 59 00  00 00 00 4C 00 00 00 00  MYSTERY....L....
00000150  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000160  53 43 49 46 49 00 00 00  00 00 00 4C 00 00 00 00  SCIFI......L....
00000170  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000180  57 41 52 00 00 00 00 00  00 00 00 4C 00 00 00 00  WAR........L....
00000190  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000001a0  57 45 53 54 45 52 4E 00  00 00 00 4C 00 00 00 00  WESTERN....L....
000001b0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000001c0  4D 55 53 49 43 41 4C 00  00 00 00 4C 00 00 00 00  MUSICAL....L....
000001d0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000001e0  52 4F 4D 41 4E 43 45 00  00 00 00 4C 00 00 00 00  ROMANCE....L....
000001f0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000200  48 4F 52 52 4F 52 00 00  00 00 00 4C 00 00 00 00  HORROR.....L....
00000210  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000220  0D 20 20 20 20 20 20 20  20 20 20 20 20 20 31 2E  .             1.
00000230  30 30 30 30 30 20 20 20  20 20 20 20 20 20 31 39  00000         19
00000240  39 39 2E 30 30 30 30 30  32 30 30 20 43 69 67 61  99.00000200 Ciga
00000250  72 65 74 74 65 73 20 20  20 20 20 20 20 20 20 20  rettes
00000260  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
00000270  46 54 46 54 46 46 46 46  46 46 46 46 46 20 20 20  FTFTFFFFFFFFF
00000280  20 20 20 20 20 20 20 20  20 20 32 2E 30 30 30 30            2.0000
00000290  30 20 20 20 20 20 20 20  20 20 31 39 39 39 2E 30  0         1999.0
000002a0  30 30 30 30 32 30 30 31  20 59 6F 6E 67 67 61 72  00002001 Yonggar
000002b0  79 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  y
000002c0  20 20 20 20 20 20 20 20  20 20 20 20 46 46 46 46              FFFF
000002d0  54 46 46 54 46 46 46 46  46 20 20 20 20 20 20 20  TFFTFFFFF
000002e0  20 20 20 20 20 20 33 2E  30 30 30 30 30 20 20 20        3.00000
000002f0  20 20 20 20 20 20 31 39  39 39 2E 30 30 30 30 30        1999.00000
00000300  38 4D 4D 20 20 20 20 20  20 20 20 20 20 20 20 20  8MM
00000310  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
00000320  20 20 20 20 20 20 20 20  46 46 46 46 46 46 54 46          FFFFFFTF
00000330  46 46 46 46 46 20 20 20  20 20 20 20 20 20 20 20  FFFFF
00000340  20 20 34 2E 30 30 30 30  30 20 20 20 20 20 20 20    4.00000
00000350  20 20 31 39 39 39 2E 30  30 30 30 30 4D 61 67 69    1999.00000Magi
00000360  63 61 6C 20 4C 65 67 65  6E 64 20 6F 66 20 74 68  cal Legend of th
00000370  65 20 4C 65 70 72 65 63  68 61 75 6E 73 2C 20 54  e Leprechauns, T
00000380  68 65 20 20 46 46 46 46  54 46 46 46 46 46 46 46  he  FFFFTFFFFFFF
00000390  46 20 20 20 20 20 20 20  20 20 20 20 20 20 35 2E  F             5.
000003a0  30 30 30 30 30 20 20 20  20 20 20 20 20 20 31 39  00000         19
000003b0  39 39 2E 30 30 30 30 30  53 74 6F 72 6D 20 6F 66  99.00000Storm of
000003c0  20 74 68 65 20 43 65 6E  74 75 72 79 20 20 20 20   the Century
000003d0  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
```

Figure 4.11: Binary Display of Films1999.DBF file

Because the MS Visual Studio binary viewer displays 16 bytes per row, it is relatively easy to locate data in the file. The first two rows correspond to the header block of information. The key elements of the header are as follows:

- **Byte 0:** Should be 0x03 for dBase III files. (0x83 is also legal, but it implies a separate memo file, which will not be relevant here).
- **Byte 1:** Year the file was last modified (1900 is 0). In Figure 4.11, the value 0x67 corresponds to 103 decimal, or 1900+103=2003.
- **Byte 2:** Month the file was last modified. In Figure 4.11, the value 0x05 signifies May.
- **Byte 3:** Day the file was last modified. In Figure 4.11, the value 0x17 signifies 23. Thus, the file was last modified on 5/23/2003.
- **Bytes 4-7:** A 4-byte integer indicating the number of records in the file. Reversing the bytes (as discussed in Section 4.3.4), this becomes 0x000000E2, which is 226.
- **Bytes 8-9:** A 2-byte integer specifying where the actual data in the file begins. In this case (reversing the bytes), the position is 0x0221. Looking at the addresses on the left, you can verify that this is—in fact—the actual location where the data appears to be located (immediately after the 0x0D separator byte shown in Figure 4.10).
- **Bytes 10-11:** A 2-byte integer specifying the record length. In dBase III, the maximum record length was 4000—so this number should always be less than or equal to that value. In Figure 4.11, the value 0x005C specifies that each record in 92 bytes long.

The remaining bytes in the 32-byte header are ignored. Following the header block comes the field definitions. The number of fields defined for the table can be computed from the information in the header block. Specifically, you take the location where the data begins (0x221) and subtract the bytes used for the header (0x20 bytes) and the separator byte (0x01 byte). This leaves 0x200 bytes for all the field definitions (512 bytes, in decimal). We then divide this by the size of each field definition (0x20 bytes, or 32) and the result is 16 fields (512/32=16).

Each 32-byte field definition is laid out as follows:

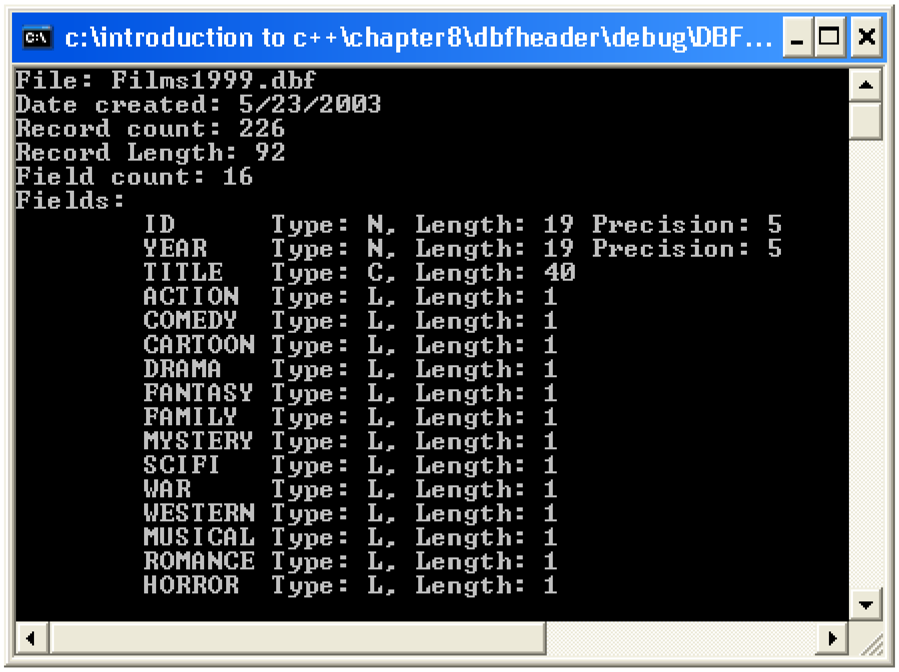
- **Bytes 0-10:** A NUL terminated string containing the field name. In dBase III, field names were limited to 10 characters (with the additional byte allocated for the NUL terminator). For example, the field beginning at 0x00000020 is named "ID", the field beginning at 0x00000060 is named 'TITLE", etc.
- **Byte 11:** A single character indicating the field type. Allowable types included 'N' (numeric), 'C' (character, or text), 'D' (date) and 'L' (logical, or TRUE/FALSE). For example, the ID field has a value of 0x4E (ASCII for 'N'), meaning its numeric. The TITLE field has a value of 0x43 (ASCII for 'C'), meaning it’s a text field.

- **Byte 16:** A single character specifying the field length (individual dBase III fields were limited to 255 bytes). For example, the ID field is 0x13 (19) characters long. The TITLE field is 0x28 (40) characters long.
- **Byte 17:** Relevant only for numeric fields, it's the number of places to the right of the decimal place—roughly equivalent to the precision of a numeric field. For example, the ID field has a precision of 0x05 (5), meaning it is written will 5 decimal places. (This can be verified by looking at the actual data).

The remaining bytes in each field definition are ignored.

4.5.2: The .DBF File Header Viewer

Creating an application that displays the definitions (header and fields) for a DBF file is relatively simple. The DBFHeader application, which can be used to display the structure of any dBase III file, does this using two functions: DisplayHeader() and DisplayFieldDef(). Its output is presented in Figure 4.12.



```
File: Films1999.dbf
Date created: 5/23/2003
Record count: 226
Record Length: 92
Field count: 16
Fields:
        ID      Type: N, Length: 19 Precision: 5
        YEAR    Type: N, Length: 19 Precision: 5
        TITLE   Type: C, Length: 40
        ACTION  Type: L, Length: 1
        COMEDY  Type: L, Length: 1
        CARTOON Type: L, Length: 1
        DRAMA   Type: L, Length: 1
        FANTASY Type: L, Length: 1
        FAMILY  Type: L, Length: 1
        MYSTERY Type: L, Length: 1
        SCIFI   Type: L, Length: 1
        WAR     Type: L, Length: 1
        WESTERN Type: L, Length: 1
        MUSICAL Type: L, Length: 1
        ROMANCE Type: L, Length: 1
        HORROR  Type: L, Length: 1
```

Figure 4.12: Output from DBFHeader application

Functions
Only two functions were required to implement the application. The first, DisplayHeader(), displays information from the top 32-byte block, then calls the second, DisplayFieldDef() to display each field definition. It is prototyped as follows:

```
void DisplayHeader(const char szFile[]);
```

It takes a single argument: file name for which the header will be displayed.

The function is presented in Example 4.20.

Example 4.20: DisplayHeader() function

```
void DisplayHeader(const char szFile[])
{
	STREAM in;
	int nRecords,nReclen,nFieldCount,nStartPos;
	int nYear,nMonth,nDay;
	int i;
	if (!Open(in,szFile,true,false,false,false)) {
		DisplayString("DBF File could not be opened!");
		NewLine();
		return;
	}
	// Check that first byte is 0x03
	if (ReadCharacter(in)!=3) {
		DisplayString("Illegal DBF file format");
		NewLine();
		return;
	}
	// Next data is date created
	nYear=ReadCharacter(in)+1900;
	nMonth=ReadCharacter(in);
	nDay=ReadCharacter(in);
	DisplayFormatted("File: %s\n",szFile);
	DisplayFormatted("Date created: %i/%i/%i\n",nMonth,nDay,nYear);
	// Next data is number of records
	nRecords=ReadInteger(in);
	DisplayFormatted("Record count: %i\n",nRecords);
	// Next data is position where data starts
	nStartPos=ReadShort(in);
	// To computer number of fields,subtract 33 bytes for header &
separator
	nFieldCount=(nStartPos-33)/32;
	if (nFieldCount<1 || nFieldCount>128) {
		DisplayString("Illegal DBF file format");
		NewLine();
		return;
	}
	// Next data is record length
	nReclen=ReadShort(in);
	DisplayFormatted("Record Length: %i\n",nReclen);
	DisplayFormatted("Field count: %i\nFields:\n",nFieldCount);
	for(i=0;i<nFieldCount;i++) {
		DisplayFieldDef(in,(i+1)*32);
	}
	Close(in);
}
```

The function works as follows:

- It opens the file as a binary file for reading (not truncated) using the SimpleIO Open() function.
- It reads the first byte to ensure it matches the expected value of 3.
- It reads the year, month and day bytes. After adding 1900 to the year, it displays the file name and date.
- It reads the number of records as a 4-byte integer, then displays it.
- It reads the start position of the data as a 2-byte integer, then uses the formula presented earlier (nStartPos-33)/32 to compute the number for fields. It then verifies that a legal number of fields is specified.
- It reads the record length data as a 2-byte integer, then displays it.
- It loops though the fields one-by-one, calling DisplayFieldDef() specifying the starting position of each field definition.

The second function, DisplayFieldDef(), is designed to output the definition for a single field. It is prototyped as follows:

void DisplayFieldDef(STREAM &in,int nStart);

Its argument is the open DBF file stream and the position in the file where the field definition starts. It is presented in Example 4.21.

Example 4.21: DisplayFieldDef() function

```
void DisplayFieldDef(STREAM &in,int nStart)
{
      char szName[20];
      char cType;
      int nLen,nPrec;
   SetReadPosition(in,nStart);
      ReadBlock(in,szName,11);
      cType=ReadCharacter(in);
      SetReadPosition(in,nStart+16);
      nLen=ReadCharacter(in);
      nPrec=ReadCharacter(in);
      DisplayFormatted("\t%s\tType: %c, Length: %i",szName,cType,nLen);
      if (cType=='N') DisplayFormatted(" Precision: %i\n",nPrec);
      else NewLine();
}
```

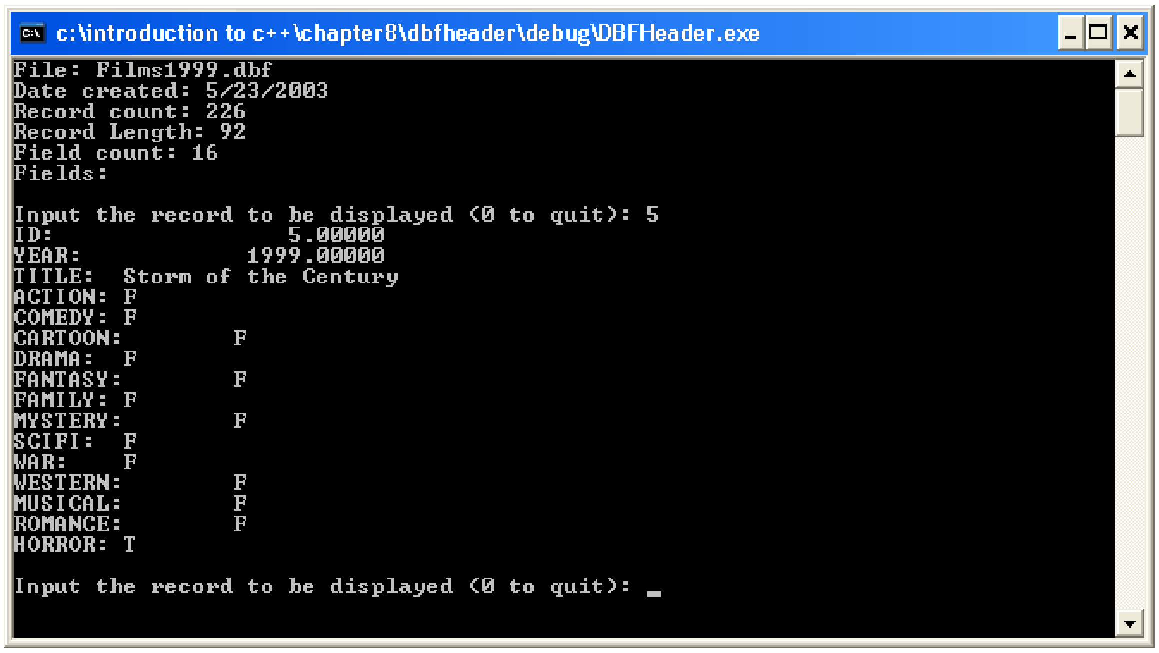
The function works as follows:

- The file read position is set to the offset specified by nStart. (This offset was computed in the loop contained in the DisplayHeader() function).
- It reads the field name as an 11-byte block into a local array, szName.
- It reads the field type as a character, assigned to cType.
- It repositions the pointer to offset 16 (where the field length is stored).

- It reads the field length byte and places the value in nLen
- It reads the precision byte and places it in nPrec.
- It displays the name, type and length of the field.
- If the field is of type 'N' (numeric), it also displays the precision.

4.5.3: Lab Exercise: The .DBF Record Viewer

In this exercise, we create a simple file viewer that displays the data from dBase III records. The interface is a simple loop that prompts the user for a record number, then displays the field names and associated data for that record. This is illustrated in Figure 4.13.


c:\introduction to c++\chapter8\dbfheader\debug\DBFHeader.exe

```
File: Films1999.dbf
Date created: 5/23/2003
Record count: 226
Record Length: 92
Field count: 16
Fields:

Input the record to be displayed (0 to quit): 5
ID:                     5.00000
YEAR:                1999.00000
TITLE:  Storm of the Century
ACTION: F
COMEDY: F
CARTOON:        F
DRAMA:  F
FANTASY:        F
FAMILY: F
MYSTERY:        F
SCIFI:  F
WAR:    F
WESTERN:        F
MUSICAL:        F
ROMANCE:        F
HORROR: T

Input the record to be displayed (0 to quit): _
```

Figure 4.13: DBF Record Viewer in operation

Functions
The DBF Record Viewer can be implemented using two functions. Their prototypes and descriptions are presented in Table 4.8.

Table 4.8: Functions to Implemented "DBF Record Viewer" Lab Exercise

void DisplayLoop(const char szFile[])

The driver function for the application, it takes the name of the DBF file (i.e., Films1999.dbf) as its only argument. The function should:

- Open the DBF file for reading, as a binary file
- Load much of the same header information as was loaded in DisplayHeader(). Of particular importance is the data starting position, number of records, number of fields and record length.
- Enter a loop that prompts the user for a record number. Within that loop:
  - If the user enters an invalid record number, it should display an error message.
  - If the user enters 0, it should break out of the loop.
  - If the user enters a valid record number, it should:
    1. Move to the appropriate position in the file (this can be computed using the data start position and the record length)
    2. Load the record data into a buffer, which can be done with ReadBlock() and the record length. If you size the buffer at 4001 characters (the maximum dBase record size), you should never have to worry about too many bytes being read.
    3. Call the DisplayData() function to cause the actual record data to be displayed.
- Close the file stream, once the user ends the loop

void DisplayData(STREAM &db,const char szRecord[],int nFieldCount)

Displays all the field data for a given record. Its arguments are the open file stream for the DBF file (*db*), a character string containing the currently active record (*szRecord*) and a count of the number of fields. The function will consist of a single loop, which iterates through the individual field values. Within the loop, the function will:

- Load the name and field length of each field definition (the DisplayFieldDef() function provides a good model for getting this information)
- Extract the actual data from the szRecord[] array. The GetField() function, presented in Section 4.4.1, Example 4.8, could be used to extract the data.
- Display the field name, followed by the associated data.

The one aspect of these two functions that warrants some additional explanation is the determination of each field starts and ends in DisplayData(). Each dBase III record is stored entirely in text form, as is evident from Figure 4.9. Within the text for each record, which is loaded in DisplayLoop(), the first byte is reserved—having one of two values:

- ' ' (space): Indicates the record is active.
- '*' (asterisk): Indicates the record is marked for deletion

That means that the actual field data begins a position 1 in the record, not the usual position 0. The first field, ID, therefore occupies positions 1-19 (since its length is 19 bytes), the second field, YEAR, occupies bytes 20-38 (again, 19 byte field length), TITLE occupies bytes 39-78 (40 byte length), and so forth. As you iterate through the field definitions, then, you need to:

- Start at position 1
- Add the length of the current field to the starting position to determine where the next field starts
- If you use GetField() to extract the data, the arguments will be the starting position (*nStart)* and the field length (*nCount*).

Procedure
To build the DBF Record Viewer, the following procedure is recommended:

- Make a copy of the DisplayHeader() (Example 4.20) function and name it DisplayLoop(). This provides an excellent starting point for the first function.
- Remove the loop from the original function that displays field definitions and replace it with a loop that:
  - prompts the user for a record number, verifies the record number is valid (or 0, in which case you can break of if the loop),
  - loads the record data into a buffer (which should be defined to be at least 4000 characters long, for safety's sake),
  - Calls DisplayData()
- Write the DisplayData() function. The DisplayFieldDef() (Example 4.12) provides a good starting point for extracting the two key pieces of information required (field name and field length). The loop that displays field definitions in DisplayHeader() (a loop removed in the previous step) also is useful in figuring out how to identify the starting point of each field definition in the file.

4.5.4 DBF Structures Lab Exercise

Walkthrough available in DBFStruct.avi

This exercise involves modifying the .DBF binary file viewer (Section 4.5.3) so that it uses structures. The specifications for the lab exercise include the data structures, functions and interface to be implemented to complete the exercise. Ideally, Section 4.5.3, will have been implemented.

Data definitions
Two structures will need to be defined as part of the exercise:

- *field*: A structure that holds a single field definition. One member (e.g., nPos) should be used to hold the starting position of the field within the record buffer.
- *dbf*: A structure to hold the entire database definition, including:
  - values such as record length, date modified, number of fields, etc.

- A STREAM object that will be opened to the .DBF file
- an array of field definitions (*Hint:* since dBase III files were limited to 128 fields, you can use that as the size of the array.

Functions

The functions to be implemented are listed in Table 4.9:

Table 4.9: Functions to Implemented "DBF Structures" Lab Exercise

void DisplayLoop(const char szFile[])

The driver function for the application, it takes the name of the DBF file (i.e., Films1999.dbf) as its only argument. The function should:

- Declare a local dbf structure object.
- Open the DBF file using the OpenDBF() function..
- Enter a loop that prompts the user for a record number. Within that loop:
  - If the user enters an invalid record number, it should display an error message.
  - If the user enters 0, it should break out of the loop.
  - If the user enters a valid record number, it should:
    1. Move to the appropriate position in the file (this can be computed using the data start position and the record length)
    2. Load the record data into a buffer, which can be done with ReadBlock() and the record length. If you size the buffer at 4001 characters (the maximum dBase record size), you should never have to worry about too many bytes being read.
    3. Call the DisplayRecord() function to cause the actual record data to be displayed.
- Closes the database by calling CloseDBF().

void DisplayRecord(const struct dbf &db,const char szRecord[])

Displays all the field data for a given record. Its arguments are the dbf structure (*db*) and character string containing the currently active record (*szRecord*). The function will consist of a single loop, which iterates through the individual field values. Within the loop, the function will:

- Extract the actual data from the szRecord[] array. The GetField() function, presented in Chapter 8, Section 8.4.1, Example 8.8, could be used to extract the data.
- Display the field name, followed by the associated data.

bool OpenDBF(const char szFile[],struct dbf &db)

Opens the file named szFile as a binary read stream using the STREAM object embedded in the db structure. The function should then:

- Load relevant information (e.g., record length, number of records, number of fields, date modified) into the dbf structure
- Load definitions for all fields into the field object array embedded within the dbf structure.

If successful, the function should return **true**, otherwise it should return **false**.

void CloseDBF(struct dbf &db)

Closes the open file stream embedded in the dbf structure, then sets all data items (e.g., record length, number of records, number of fields) to 0.

Interface

The interface for the lab exercise is to be identical to that described in Section 4.5.3.

Procedure

The following procedure is recommended to implement the lab exercise:

- Create the field and dbf structures, placing their definitions in an include file. The filed structure should be defined first in that file, to avoid error messages (since it is to be composed within the dbf structure). You will also need to include SimpleIO.h within the header file, since an embedded STREAM object is present in the dbf structure definition.
- Write the OpenDBF() function. Test it within a simple main() function, using the debugger to make sure the proper information is being loaded.
- Create your DisplayLoop() function, omitting or commenting out the call to DisplayRecord(), using the debugger to make sure that it is loading the proper data is loaded when the user specifies a record number.
- Write the DisplayRecord() function, then test the application.

4.6: Review and Questions

4.6.1: Review

Without the capability of file reading and writing, it would be almost impossible to construct today's information systems, and computers would be little more than fancy calculators. To facilitate using files, the C++ offers an extensive class library, covered in detail in Chapter 11. In this chapter, we focus on understanding the essential nature of file IO, hiding the specific implementation through the use of the SimpleIO functions provided with the text.

The key to file I/O is the creation of a stream, which is a place to which information can be sent, or from which information can be retrieved. To attach a file to a specific stream, we use the Open() function, prototyped as follows:

```
bool Open(STREAM &file,const char *szName,
                    bool bRead,bool bWrite,bool bTrunc,bool bText);
```

The arguments to Open() are as follows:

- *file:* A previously declared stream variable
- *szName:* The name of the file (including path, if desired, of the file to be attached to the stream)
- *bRead:* Will we be able to read from the file?
- *bWrite:* Will we be able to write to the file?
- *bTrunc:* Should we erase (truncate) the file when we open it?
- *bText:* Should the file be opened for text processing, as opposed to binary processing.

The function returns **true** if successful, **false** otherwise. Once a file is no longer needed by the program, its stream can be closed with the Close() SimpleIO function, prototyped as follows:

```
    void Close(STREAM &file);
```

Text files contain data stored translated into ASCII character format. They are also line oriented, meaning that they tend to be organized into variable-length lines that are ended with a sequence of newline and carriage return characters (in MS Windows, "\r\n" pairs signal the end of each line, but this can vary according to operating system).

SimpleIO Functions commonly used in text files include:

SimpleIO Text Stream Functions	
Output Functions	***void PrintString(OUTPUT &out,const char *szOut)*:** Write a text string to the text stream *out*.
	***void PrintInteger(OUTPUT &out,int nVal)*:** Translates an interger to a string, then write the integer to the text stream *out*.
	***void PrintCharacter(OUTPUT &out,char cVal)*:** Writes an ASCII character to the text stream *out*.
	***void PrintReal(OUTPUT &out,double dVal)*:** Writes an unformatted real number to a file.
	void PrintLine(OUTPUT &out,const char *szOut): Sends a string (*szOut*), followed by a newline, to the text stream *out*.
	void PrintFormatted(OUTPUT &out,const char *szFmt,...): Writes formatted text data to the text stream *out*, using a printf style formatting string (See Chapter 7, Section 7.3 for in depth discussion of formatting).
	void EndLine(OUTPUT &out): Sends an end of line to the text stream *out.* This is basically equivalent to PrintCharacter(out,'\n').
	void WhiteSpace(OUTPUT &out): Sends a space to the text stream *out.* This is basically equivalent to PrintCharacter(out,' ').
Input Functions	***void GetString(INPUT &in,char szIn[]):*** Reads a string from the text stream *in* and places it in the array szIn, which must be large enough to accommodate it.
	int GetInteger(INPUT &in): Reads an integer from the text stream *in* and returns its value.
	double GetReal(INPUT &in): Reads a real number from the text stream *in* and returns its value.
	char GetCharacter(INPUT &in): Reads a character from the text stream *in* and returns its value.
	bool GetLine(INPUT &in,char szIn[],int nMax): Reads a line from the text stream *in* and places it in *szIn*, followed by a NUL terminator. No more than *nMax* characters are read.

In the above table:

OUTPUT is either a STREAM variable open for writing, or **cout**.
INPUT is either a STREAM variable open for reading, or **cin**.

In general, text IO is performed using line-oriented functions (e.g., the GetLine() and PrintLine() functions). The reason for this is that white characters, such as space and tab, are treated as end of string markers when reading strings. That means data saved as a single string, e.g., "Hello World" using PrintString(), would be read as two separate strings ("Hello" and "World") using GetString(). Using line-oriented IO permits more sophisticated processing of text files (e.g., see fixed file processing example in Section 4.4).

Binary files differ from text files in that the data placed in them is usually a direct copy of data in memory (instead of being translated to text form). As a result, they tend to be much more efficient to work with than text files. Implicit in any binary file is a pointer, identifying the position from which the next read or write will start. Functions commonly used with binary files include:

SimpleIO functions for binary file I/O	
Output Functions	***bool WriteInteger(STREAM &file,int nVal);*** Writes a single integer to the open binary file stream. It returns **true** if the operation is successful, **false** otherwise.
	bool WriteReal(STREAM &file,double dVal); Writes a single double real number to the open file stream. It returns **true** if the operation is successful, **false** otherwise.
	bool WriteFloat(STREAM &file,float fVal); Writes a single float real number to the open file stream. It returns **true** if the operation is successful, **false** otherwise.
	bool WriteCharacter(STREAM &file,char cVal); Writes a single character to the open file stream. It returns **true** if the operation is successful, **false** otherwise.
	bool WriteShort(STREAM &file,short nVal); Writes a short (2-byte) integer to the open file stream. It returns **true** if the operation is successful, **false** otherwise.
	bool WriteBlock(STREAM &file,const void *buf,int nBytes); Writes an array of any type of data, starting at position *buf*, to the open binary file stream. The amount of data to be written is *nBytes*. It returns **true** if the operation is successful, **false** otherwise.
	bool SetWritePosition(STREAM &file,int nPos): Sets the 0-based file position to be used for the next write operation.
Input Functions	***int ReadInteger(STREAM &file);*** Reads an integer from the open binary file stream, returning its value.
	double ReadReal(STREAM &file); Reads a double from the open binary file stream, returning its value.
	float ReadFloat(STREAM &file); Reads a float from the open binary file stream, returning its value.
	char ReadCharacter(STREAM &file); Reads a character from the open binary file stream, returning its value.
	short ReadShort(STREAM &file); Reads a short (2-byte) integer from the open binary file stream, returning its value
	bool ReadBlock(STREAM &file,void *buf,int nBytes); Reads an array from the open binary file stream, copying *nBytes* of data from the file to the memory location beginning at position *buf*. It returns **true** if the operation is successful, **false** otherwise.
	bool SetReadPosition(STREAM &file,int nPos): Sets the 0-based file position to be used for the next read operation.

Using binary files effectively typically involves some general principles, which include:

- Unlike text files, the organization of binary files cannot necessarily be determined by inspection. When working with files that you did not create, some roadmap is normally needed.
- Binary data needs to be retrieved in exactly the same format that it was saved in. If data is saved as a 2-byte integer, for example, it cannot be read back as a 4-byte integer.
- Whereas text files are usually processed sequentially, a file pointer is often used in binary files to allow random access of data contained within the file.

4.6.2: Glossary

ASCII file – A file containing data that has been translated into text characters, using the ASCII coding scheme.
Binary file – A file containing data in its original (byte) form, as opposed to being saved as ASCII characters.
cin – The default input stream for functions such as InputString()
Close a stream – Release the association between a file and a given STREAM, so the file is not locked to prevent access by other programs and to reduce operating system resources (Close() function)
cout – The default output stream functions such as DisplayString().
File Position – The logical location (an integer address), within an I/O stream, for the next read or write operation.
Formatted output – Text output that is presented in a manner that is specified by the programmer (e.g., a fixed number of decimal places for real numbers, a fixed length for strings), rather than relying of default presentation.
I/O Stream – A general method of characterizing an input-output area, such as a display, hard-drive or RAM.
Line-oriented I/O – Input/output done on a line-by-line basis, rather than on a data element by data element basis.
Member function – A function that is applied to an object in much the same way that data is accessed within a structure.
Open a stream – Associate a file with a given STREAM, so it can be used for subsequent I/O (done with Open() function)
Open mode – How a stream is opened (e.g., for reading, writing, or read/write), the type of stream being opened (e.g., binary or text) and whether or not the file is to be erased upon opening (i.e., truncated).
STREAM – A SimpleIO typedef used to identify a file stream variable.
Text file – An alternate name for an ASCII file.

4.6.3: Questions

4.1: Line Counter. Write and test a function that counts all the lines in a text file. The function should be prototyped as follows:

```
int LineCount(const char *szFileName);
```

where szFileName is the name of the file, and the function returns the number of lines.

4.2: Word Counter. Write and test a function that counts all the words in a text file. The function should be prototyped as follows:

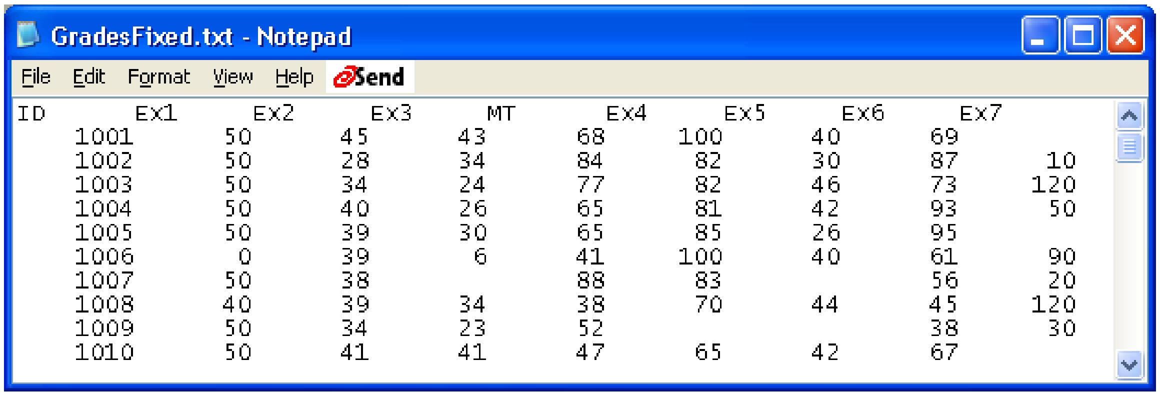
```
int WordCount(const char *szFileName);
```

where szFileName is the name of the file, and the function returns the number of words in the file. For the purpose of this function, any cluster of non-white characters separated by one or more white characters from any other cluster is considered a word. e.g.,

"Hello World 123 it's me!"

would be considered to have 5 words. (*Hint:* write a second function that counts the words in a line and the entire process becomes much easier).

The next two questions deal with the file GradesFixed.txt, in the Chapter 4\Exercises folder. The file contains grades from a course, where each row represents the grades for a particular student, whose ID is the first item on the row. If a grade is missing, it is left blank. A segment of the file is displayed in Notepad below:



```
ID        Ex1       Ex2       Ex3       MT        Ex4       Ex5       Ex6       Ex7
      1001        50        45        43        68       100        40        69
      1002        50        28        34        84        82        30        87        10
      1003        50        34        24        77        82        46        73       120
      1004        50        40        26        65        81        42        93        50
      1005        50        39        30        65        85        26        95
      1006         0        39         6        41       100        40        61        90
      1007        50        38                  88        83                  56        20
      1008        40        39        34        38        70        44        45       120
      1009        50        34        23        52                            38        30
      1010        50        41        41        47        65        42        67
```

4.3: Summing Rows of a Fixed Text File. Write two functions, TotalGrade() and DisplayTotals(), prototyped as follows:

```
bool TotalGrade(STREAM strm);
bool DisplayTotals(const char szFileName[]);
```

The TotalGrade() function should read a line from the file, then print out the student ID (the first column from the file) and the total of the remaining columns, treating any empty column as 0. It should return false if the line cannot be read, of it there is no ID. The DisplayTotals() function should open a file, skip the first line (containing the headers), then call TotalGrade() on each line until it returns false. Use the SimpleIO functions and the GetField() function (Example 4.8) to help you. You will need to open the file in a text editor to determine where each grade starts and ends. You will also need to write a small main() function to test your application.

4.4: Averaging Columns of a Fixed Text File. Write two functions, TotalCols() and DisplayAverages(), prototyped as follows:

```
bool TotalCols(STREAM strm,int nStart,int nCount,double dTotals[],int nVals[]);
bool DisplayAverages(const char szFileName[]);
```

The TotalCols() function should read a line from the file, then take the values of nCount columns, starting at column nStart, and add them to the values in dTotals[], incrementing the corresponding value in nVals[] if the column is non-blank. For example:

```
double dTotals[8]={0.0};
int nVals[8]={0};
STREAM strm;
// Missing code to open file, etc.
TotalCols(strm,1,8,dTotals,nVals);
```

The call to TotalCols() above would cause a line from the file to be read, the first column (column 0) to be ignored, and the values of subsequent columns to be added to dTotals (column 1 value is added to dTotals[0], column 2 added to dTotals[1], etc.). For any column that is not all spaces, nVals[Column number-1] will be incremented—this prevents blank cells to be used to compute the average.

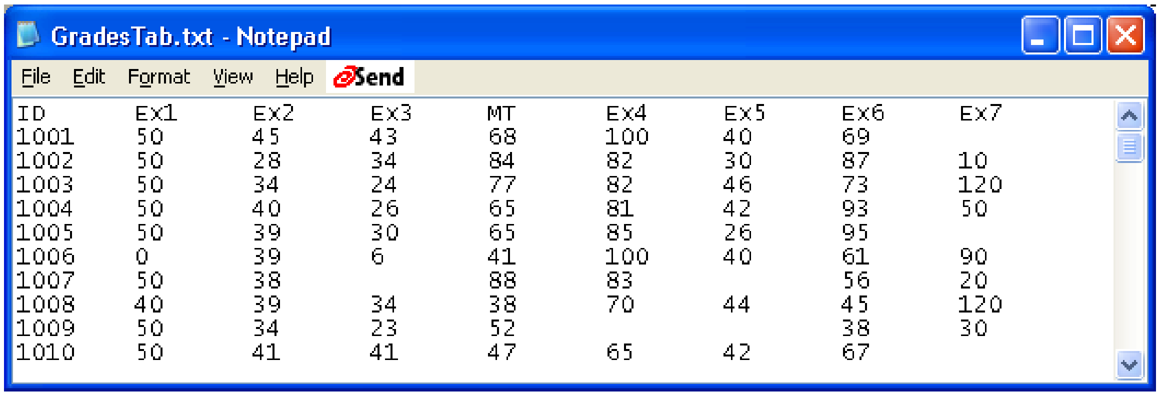
The DisplayAverages() function is like DisplayTotals() function of Question 4.3 except that after TotalCols() has been called on each line, it then computes the average by taking the total in dTotals[] and dividing it by the corresponding count in nVals[]. A simple main() driver function will also need to be written to test the application.

4.5 Creating a Fixed Text File. Write a function, ComputeTotals(), that computes the total grade for each student then writes the student ID and associated grade to a fixed text format file, one line per student. The function should be prototyped as follows:

```
bool ComputeTotals(const char szInputFile[],const char szOutputFile[]);
```

The arguments szInputFile[] contains the name of the input data file (i.e., "GradesFixed.txt") and the name you chose for the output file. A simple main() driver function will also need to be written to test the application. You should also feel free to use any functions written in Questions 4.3 and 4.4.

The next two questions deal with the file GradesTab.txt, in the Chapter 4\Exercises folder. The file contains the same data used in GradesFixed.txt except that date elements are separated by tab (0x09) characters. A segment of the file is displayed in Notepad below:



```
ID      Ex1     Ex2     Ex3     MT      Ex4     Ex5     Ex6     Ex7
1001    50      45      43      68      100     40      69
1002    50      28      34      84      82      30      87      10
1003    50      34      24      77      82      46      73      120
1004    50      40      26      65      81      42      93      50
1005    50      39      30      65      85      26      95
1006    0       39      6       41      100     40      61      90
1007    50      38              88      83              56      20
1008    40      39      34      38      70      44      45      120
1009    50      34      23      52                      38      30
1010    50      41      41      47      65      42      67
```

4.6: Summing Rows of a Tab Delimited File. Write three functions, TotalGradeTab () and DisplayTotalsTab(), prototyped as follows:

```
bool TotalGradeTab(STREAM strm);
bool DisplayTotalsTab(const char szFileName[]);
void GetFieldTab(char szTarget[],const char szBuf[],int nField)
```

The TotalGradeTab() function and DisplayTotalsTab() function should perform the same functions as those defined in Question 4.3. GetFieldTab() is the only different function. It will replace GetField() and—instead of copying a range of characters from szBuf—it will count tabs until it reaches nField (e.g., nField of 0 starts at szBuf[0], nField of 1 starts immediately after the first tab, nField of 2 starts immediately after the second tab, etc.), then copy all the characters until the next tab (or end of the line) is reached. You will also need to write a small main() function to test your application.

4.7: Averaging Columns of a Fixed Text File. Write two functions, TotalCols() and DisplayAverages(), prototyped as follows:

```
        bool TotalColsTab(STREAM strm,int nStart,int nCount,double dTotals[],int
nVals[]);
        bool DisplayAveragesTab(const char szFileName[]);
```

These functions should do precisely what their counterparts din in Question 4.4.

4.8 Creating a Tab-delimited File. Write a function, ComputeTotalsTab(), that computes the total grade for each student then writes the student ID and associated grade to a tab delimited format file, one line per student. The function should be prototyped as follows:

```
bool ComputeTotalsTab(const char szInputFile[],const char szOutputFile[]);
```

This should do precisely what the function defined in Question 4.5 did, except the output file fields should be separated by tabs.

In Depth Questions:

4.9: Creating a Binary File. Write a function, TranslateToBinary(), that reads the GradesFixed.txt file and saves each element as a binary file. The function should be prototyped as follows:

```
bool TranslateToBinary(const char szTextIn[],const char szBinOut[],int nCols);
```

where szTextIn is the name of the input file (i.e., "GradesFixed.txt"), szBinOut is the name of the output file you create and nCols is the number of columns of grade data to be saved (this would allow you flexibility in using the file for data on more or fewer items). The function should return true unless it cannot open one of the files. The binary file should include the following data:

- Number of columns
- Number of student records
- Name of each column (stored as a string)
- Data for each student (with –1 used to indicate missing values) and student ID saved as a string (rather than as an integer).

You should also write a simple main() driver function for test purposes.

4.10: Loading a Binary File. Write a function, CreateReport(), that reads the binary file created in 4.9 and displays summary statistics. The function should be prototyped as follows:

```
void CreateReport(const char szBin);
```

where szBin is the name of the file created in 4.9. The function should return true unless it cannot open the file. The report generated by the function should be displayed to the screen, and (at a minimum) should contain:

- A list of average grades for each assignment
- A list of total grades for each student
- An average across all students

In defining arrays, you may assume that no more that 256 assignments will ever be present. You should also write a simple main() driver function for test purposes.

Chapter 5

Debugging and Testing

Executive Summary

Chapter 5 introduces the important topics of debugging and testing. By some estimates, these activities can consume 50% of the resources required for a complex development project, yet they are often completely ignored in introductory programming courses. The chapter provides a mixture of conceptual overview and practical technique related to this important subject.

The chapter begins by introducing the debugging and testing process, distinguishing debugging (the detection and removal of defects) from testing (ensuring an application meets its specifications). Various forms of testing (e.g., walkthroughs, automated testing, end-user testing) are described, along with phases of testing (functional, component, integration and use). An extensive walkthrough of the process of debugging some defect-plagued code (provided by the text) is then performed. First compiler errors are explained and eliminated. Next, linker errors are detected and removed. Various runtime errors and logic defects are then detected and repaired, using the debugging techniques of breakpoints, stepping and inspection. Automated testing, implemented using input redirection, is then illustrated. Finally, the specifications for a second application are presented, along with some very buggy code. The reader is then encouraged to repeat the debugging and testing process.

Learning Objectives

Upon completing this chapter, you should be able to:

- Explain why debugging and testing are so important in program development
- Identify and describe different forms of testing
- Discuss the various stages of testing
- Understand the nature of a number of common compiler errors, and how to fix them
- Understand the nature of the two categories of linker errors, and how to fix them
- Use the debugger capabilities of breakpoints, stepping and inspection to help identify and fix logic errors
- Use the call stack to identify the source of runtime errors, such as unhandled exceptions and assertion failures
- Use input and output redirection to accomplish some basic automated testing
- Apply the knowledge acquired in the chapter to debug actual applications, both provided in the text and developed by the user.

5.1: Overview of Debugging and Testing

Debugging is closely related to the broader area software testing. In this section, we explore the relationship between the two areas, then examine the specifications of the application we will be debugging in the remainder of the chapter.

5.1.1: Introduction to Software Testing

Software testing is the process of ensuring an application—or other form of software—performs in accordance with its specifications. Defined in this way, we can state two things about the testing process:

- Testing an application does not necessarily imply that we are simultaneously fixing that same application. It is perfectly possible, indeed common, for testing processes to be conducted quite independently of development processes. Whether or not this is a good practice is a different issue.
- You can't test an application without a clear understanding of how the application is supposed to perform—that is the role of specifications.

Debugging is a particular form of testing, normally practiced by application developers, in which the process of testing *is* necessarily accompanied by the process of removing defects. As a result, the debugging process is normally associated with programming and programmers.

There is a wide range of techniques that can be employed in testing. Four of the most common techniques are summarized in Table 5.1.

Technique	*Description*
Code Inspections	Many forms of testing revolve around examining the code used in an application without actually running it. Procedures such as *walkthroughs*, for example, involve developers explaining their code—line by line—to other expert developers. Many logical errors can be caught using inspection-oriented techniques, including errors that are rarely encountered in normal usage. Sometimes inspections are also conducted while stepping through code in the debugger.
Developer Testing	Where one or more developers tests an application by running it—usually creating their own specialized routines and test cases in the process—the process is sometimes called *developer testing.* What distinguishes this form of testing from other manual testing forms—most notably end-user testing—is that the participants in developer testing are: a) intimately aware of the nature of the code being tested, and b) often concurrently performing debugging activities.
Automated Testing	A form of testing in which that actual testing of an application is performed automatically. A variety of technologies can be employed in the context of automated testing, ranging from simple I/O redirection to the use of highly sophisticated testing packages and hardware simulators. Some tools even feature the automated creation of test data sets to be used in testing. By its nature, automated testing processes are distinct from debugging—although they can be incorporated into an overall debugging strategy.
End-User Testing	The testing of applications performed by end-users. What makes this form of testing unique is that its participants—the users—normally have no knowledge (or desire to know about) the underlying implementation/code associated with an application. Thus, their testing focuses on performance of the specific tasks for which they will be using the software.

Table 5.1: Common forms of testing

Although it is easy to ignore the subject of testing when introducing programming, there is a serious risk in doing so. By some estimates, testing may account for as much as 50% of the cost of large-scale development projects—substantially more than actual programming cost. But the costs of testing are intimately tied to the quality of programs being developed. If programmers do not recognize this fact as they design and write their code, testing costs can skyrocket.

5.1.2: Stages of software testing

In large scale development processes, testing tends to occur in a series of stages. While these stages are not perfectly distinct, a simple five-stage model generally captures the spirit of the process. Such a model is presented in Table 5.2.

Stage	***Description***
Design Testing	Before an application has even been programmed, it is possible to systematically "test" its design in search of flaws. It is not unusual, for example, to discover a particular algorithm does not seem as simple as it appeared when you start flow-charting it. Relationships between data items can become much less straightforward when one starts designing tables. Performance issues can be raised as all the components appear together in a single drawing. One particularly common source of design errors is the creation of design specifications by analysts with no software development skills. That is one of the reasons that a strong foundation in programming is useful for all I/S professionals.
Debugging	The process of testing an application while it is still being constructed. It is during this phase that developer-testing activities are most active. It is also this phase that we will focus on in this text. When the debugging stage is complete, the completed code is sometimes "thrown over the wall" to a testing group—a somewhat disparaging reference to the transfer of responsibilities between independent development and testing organizations.
Component Testing	Independent testing of individual pieces, or components, of a system or application. A variety of testing techniques, particularly inspections and automated testing, occur during this stage. Where components being tested interact with other components not being tested, various software-testing tools have been devised to simulate the interaction.
Integration Testing	A continuation of the processes used in component testing in which all of a system's components are concurrently tested. Such testing can uncover defects that are not visible when interactions between components are simulated, and can also be used to address infrastructure performance issues (e.g., for software operating over a network).
Alpha and Beta Testing	The final stages of testing, dominated by end-user testing. The distinction between the alpha and beta stages is roughly as follows: • In alpha testing, normally only a subset of the specification is fully implemented and testing is performed with the understanding that defects will be present. Outcomes of alpha testing can lead to design changes. • In beta testing, substantially all of the specified features are present and further design changes are frozen. While application defects may exist, such defects are not known at the outset of testing, nor is it expected that a large number of defects will be present. In recent years, the fast pace of the high-tech industry has caused some organizations to pay (willingly) for software known to be in the beta testing stage in order to get access to an application earlier than the general user community (i.e., their potential competitors).

Table 5.2: Stages of software testing

Why, as a programmer, should you care about these stages? The main reason is economics. A rule-of-thumb sometimes used in predicting costs is that it will cost an order of magnitude less to fix a defect in one stage than in the stage that follows it. In other words, if you can find an error during the design process it will be ten times less expensive to fix than if you find it in the programming process. Defects detected in debugging are ten times cheaper to fix than the same errors uncovered by a testing group during component testing. And so forth.

While it is doubtful that the rule of thumb is an exact predictor, it does give a sense of the consequences of defects that escape the early stages of detection. It cost Intel several hundred million dollars, for example, when a defect in the floating-point division micro-code routines of its original Pentium™ processor was not detected until the processors had reached consumers. That same defect could have been repaired for a tiny fraction of that amount had it been caught during design.

Because debugging occurs in a relatively early phase of testing, good debugging can dramatically reduce the overall cost of deploying an application. Where programmers perceive that testing is someone else's responsibility, on the other hand, costs of repairing defects can become astronomical.

5.2: The Debugging Demonstration

Walkthrough available in Customer.wmv

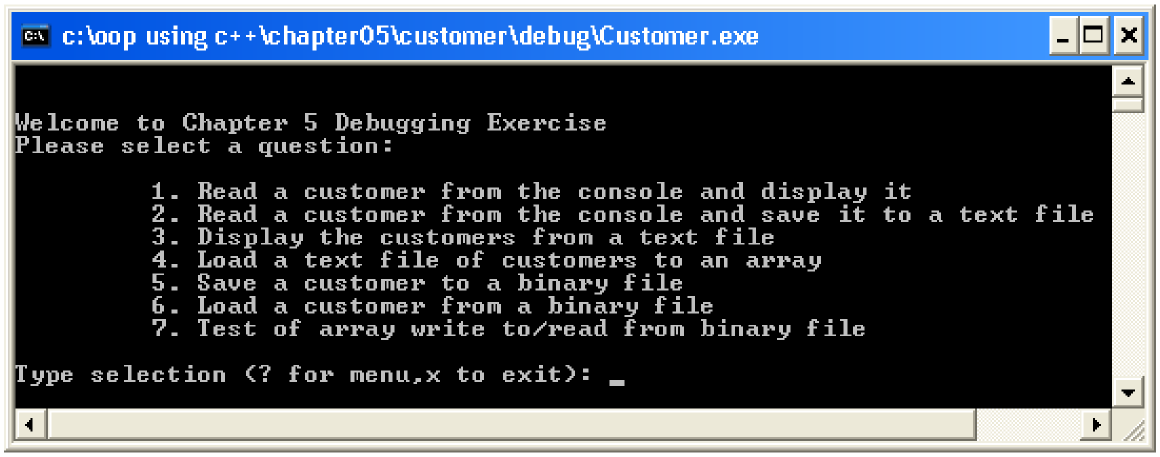
In this section, we'll look at the application that we are going to be debugging and describe the setup

5.2.1 Description

The Customer application is a simple application that demonstrates many of the C++ features presented in Chapters 1-4. Among the most important of these are:

- A variety of looping and branching constructs
- Formatted input and output (using the SimpleIO DisplayFormatted() and PrintFormatted() functions that are modeled after sprintf)
- Use of structures
- Use of pointers and arrays
- Dynamic memory allocation
- Text file I/O
- Binary file I/O

The main() function of the application implements a menu similar to the MainMenu() function presented at the end of Chapter 2. When running, it appears on the screen as is shown in Figure 5.1.



```
c:\oop using c++\chapter05\customer\debug\Customer.exe

Welcome to Chapter 5 Debugging Exercise
Please select a question:

        1. Read a customer from the console and display it
        2. Read a customer from the console and save it to a text file
        3. Display the customers from a text file
        4. Load a text file of customers to an array
        5. Save a customer to a binary file
        6. Load a customer from a binary file
        7. Test of array write to/read from binary file

Type selection (? for menu,x to exit): _
```

Figure 5.1: Customer application main menu

The focus of the application is the customer, each of which has a name, a 3 character ID, a credit limit and a date of last visit. The objective of menu option is listed in Table 5.3.

Option	Description
1	**Read a customer from the console and display it** Prompts user to enter the customer's name, ID, credit limit and date of last visit, then redisplays the data to the console.
2	**Read a customer from the console and save it to a text file** Prompts user to enter the customer's name, ID, credit limit and date of last visit, then appends the customer's data as a line in a text file.
3	***Display the customers from a text file*** Reads each line of a text file containing customers, then displays each customer's data to the console.
4	**Load a text file of customers to an array** Reads each line of a text file containing customers, placing each of them in an array of customer structures.
5	**Save a customer to a binary file** Prompts user to enter the customer's name, ID, credit limit and date of last visit, then saves the result to a binary file.
6	**Load a customer from a binary file** Reads a single customer from a binary file, then displays that customer's data to the console.
7	**Test of array write to/read from binary file** Reads a collection of customers from a text file (e.g., created through successive applications of Option 2) and loads int an array. Then saves the collection to a binary file, then reads the collection back from the binary file and displays it.

Table 5.3: Menu option descriptions

5.2.2: Implementation Description

The key structure used in the Customer application is the Customer structure, presented in Example 5.1. The structure consists of 4 elements:

- **szName**: A pointer to the customer's name. Since the memory to hold the name is not included in the structure, it must be acquired and released dynamically, using **new** and **delete []**.
- **szID**: A 4 byte array used to hold the customer's 3 byte ID code plus a NUL terminator.
- **dCreditLine**: A double intended to hold a number reflecting the customer's credit line
- **nLastVisit**: A long integer holding a representation of the data of the customer's most recent visit. The value of the integer should be in a YYYYMMDD form (e.g., 20041127 would represent 11/27/2004).

Example 5.1: Customer structure

```
struct Customer
{
	char *szName;
	char szCustID[4];
	double dCreditLine;
	long nLastVisit;
};
```

The application consists of 18 functions. The main() function is included in CustomerMain.cpp and implements the menu. The remaining 17 are in CustomerFuncs.cpp and can be described as follows:

long DateToLong(int nYear,int nMonth,int nDay)
DateToLong takes its arguments, integers representing the year, month and day of a date, and returns a long integer in the form yyyymmdd (e.g., 20030817 for 8/17/2003) if the date is valid. If the date is not valid, it returns 0.

long StringToDate(const char *src)
StringToDate takes its argument, a string that contains a date in MM/DD/YYYY format, and converts it to a date integer. It returns 0 if an illegal date is supplied. The key lines of the function are:

```
nArgs=sscanf(src,"%d/%d/%d",&nMonth,&nDay,&nYear);
if (nArgs!=3) return 0;
```

The first line is a call to the C/C++ library function sscanf (found in the <stdio.h> library), which works in the opposite direction of the sprintf() function (described in Chapter 2). The arguments to the sscanf() function are:

- a string to be scanned
- a formatting string, similar to that used for sprintf. Common format specifiers include:
    - %d – decimal integer
    - %x – hexadecimal integer
    - %i – integer (sscanf tries to figure out the integer format being used by looking for leading characters, such as 0x)
    - %s – NUL-terminated string. sscanf assumes a string ends at the first white character, such as a ' ' or tab.
    - %[*count*]c – One or more characters, specified by the optional count parameter (e.g., %20c). This is often used to read strings of known length that contain white characters.
    - %f – double precision real numbers
    - Non-format specifier codes are treated as literal characters. If the string being scanned does not contain characters matching the literals in the correct position, no further arguments are scanned.
- a set of pointer arguments (whose number depends on number of % argument specifiers in the formatting string) whose values are set when the first argument string is scanned.

The function returns the number of arguments successfully scanned.

Because sscanf takes the specified inputs and placed them into its arguments, pointers to those arguments (rather than the arguments themselves) must be passed—shich is different from the sprintf() function. The following code illustrates the difference:

```
char buf[80];
char str[80]="Hello!"
int i1=21,i2=14,i3=55;
sprintf(buf,"%d %d  %d %s",i1,i2,i3,str);
i1=i2=i3=0;
strcpy(str,"Goodbye!");
sscanf(buf,"%d %d %d %s",&i1,&2,&i3,str);
// restores i1, i2 and i3 to their original values and replaces "Hello!" in str
```

To display the values of i1, i2 and i3, the variables themselves are passed as arguments. To restore those values by scanning buf, their addresses need to be passed. Since str is an array, it is already passed in as an address to sprintf(), and is passed the same way to sscanf().

int InputCustomer(struct Customer *pCust)
InputCustomer reads in the data required to fill an Customer structure from the keyboard, then sets the values of the members associated with its argument. It returns the date of last visit as an integer, or 0 if an illegal date is supplied

void DisplayCustomer(const struct Customer *pCust)
DisplayCustomer displays the data from an Customer structure (pCust) to the screen.

int SaveCustomerToText(const char *filename,const struct Customer *pCust)
SaveCustomerToText opens a text file (named filename) and appends the data from an Customer structure (pCust) to it as a single line.It returns 1 if successful, 0 if it fails.

int LoadCustomerFromString(const char *arg1,struct Customer *pCust)
LoadCustomerFromString processes a formatted Customer record line from a text string (arg1) than was (presumably) read from a text file. The data from the string is then scanned and placed in the Customer structure argument (pCust). It returns 0 if the read fails.

int LoadCustomerFromText(STREAM &custfile,struct Customer *pCust)
LoadCustomerFromText reads a line from a text file containing Customer records, processes it, and places it in an Customer structure. If the operation fails it returns 0. Otherwise, it returns 1.

int DisplayCustomerTextFile(const char *arg1)
DisplayCustomerTextFile takes a text file containing Customer records and writes them to the screen. arg1 contains the file name. It returns 1 if successful, 0 otherwise.

**int ImportCustomersToArray(const char *filename,struct Customer *pCust,
int nMaxCount)**
ImportCustomersToArray takes a text file containing Customer records (named filename) and loads them into an array of Customer structures (beginning at address pCust). It will load no more than nMaxCount Customers. It returns a count of Customers loaded if successful, 0 otherwise.

bool SaveString(STREAM &binfile,const char *pstring)
SaveString takes an open binary file and writes a text string to it, first writing the length, then writing the characters (not including the NUL terminator. It returns true if successful, false otherwise.

bool LoadString(STREAM &binfile,char *pstring)
LoadString takes an open binary file and reads a text string from it, first reading the length, then reading the characters (not including the NUL terminator. It returns true if successful, false otherwise.

bool SaveCustomer(STREAM &binfile,const struct Customer *pCust)
SaveCustomer takes an open binary file and writes a the data from a Customer structure to it. It returns true if successful, false otherwise.

bool LoadCustomer(STREAM &binfile,struct Customer *pCust)
LoadCustomer takes an open binary file and reads a Customer structure from it. It does so by first reading the structure itself, then allocating memory for the szName string. It returns true if successful, false otherwise.

int DisplayCustomerArray(const struct Customer *pCust,int nCount)
DisplayCustomerArray takes an array of Customer objects containing nCount elements and displays each to the console. It returns 0 if nCount is 0, 1 otherwise.

void CleanupCustomerArray(struct Customer *pCust,int nCount)
CleanupCustomerArray takes an array of Customer objects containing nCount elements and deletes the szName string for each customer. This is designed to prevent memory leaks when the array goes out of scope or is reused.

bool WriteCustomerArray(const char *filename,const struct Customer *pCust, int nCount)
WriteCustomerArray takes an array of Customer objects containing nCount elements and writes them to a binary file, first opening the file, then saving the count, then saving each individual customer element and, finally, closing the file. It returns true if the operation succeeds, false otherwise.

int ReadCustomerArray(const char *filename,struct Customer *pCust,int nMaxCount)
ReadCustomerArray takes an array of Customer objects and populates it by reading them from a binary file, first opening the file, then reading the count, then reading each individual customer object and, finally, closing the file. No more than nMaxCount Customers are read. It returns the number of elements read.

5.2.3: Instructions

Prior to beginning the debugging exercise, you should familiarize yourself with the demonstration version of Customer.exe provided on the CD. When you run this, it will show you how the functions should run when you have finished debugging them. Make sure you understand what each function is supposed to do before you begin.

1. Create a new empty project (a Win32 C/C++ Console application) called Customer under whatever directory you have been using for projects.

2. Go into the CustomerDemo folder (as appropriate) under the Chapter05 folder in the CD. Copy the source files CustomerMain.cpp, Customer.h, and CustomerFunc.cpp into the newly created project directory. Copy the SimpleIO.cpp and SimpleIO.h files as well.

3. Add the ource files to the project

4. Build the project.

You should get a long list of compiler errors. It is time to do battle with the beast.

5.1 Section Questions

1. How large a percentage of total testing time for a project typically consists of debugging activities?

2. Why do you suppose that the later a defect is detected, the more expensive it is to correct?

3. Which types of errors tend to be hardest to deal with, syntax errors or logic errors?

4. What are the advantages of doing a code walkthroughs in the debugger?

5.3: Common Compiler Errors

Our first step in the debugging process is to examine the types of errors that can be found by the compiler. First we discuss these in a general sense, then we turn to the walkthrough, and look at specific errors in our code.

5.3.1: Working with the Compiler

In this section, we briefly highlight the activities of the compiler: the types of errors it catches, the types of errors it misses, and what it does.

What the Compiler Can Catch
Among the types of errors that compiling will find are included (to name a few):

- *Undefined variables:* Use of variables whose names are not recognized. C++ , which is a “strongly typed” language, differs from some earlier languages in this regard. For example, FORTRAN and some versions of BASIC will automatically create variables when they encounter names they don't recognize.
- *Unrecognized functions:* Calls to functions whose names are not recognized. C++ recognizes function names if: a) they are declared in an included header (.h) file, b) they are explicitly prototyped before the call in the file being compiled, or c) their definition is above the call to the function in the file being compiled. In C++ these unrecognized names generate an error, while in C they may generate a warning (as C is willing to make assumptions about functions it doesn't recognize).
- *Improper calls:* Calls to functions with mismatched arguments.
- *Syntax errors:* Illegal syntax, such as braces, semicolons, and operators that are missing or in the wrong place. Syntax errors of this type have a strong tendency to "confuse" the compiler—meaning a single syntax error can often lead to hundreds of errors all of which (except for the first) are meaningless. These phantom errors will disappear once the original syntax error is fixed.
- *Improper use of keywords:* Keywords—words that have a specific meaning and usage in the language being compiled (such as **if**, **for** and **switch**)—that are improperly used.

What the Compiler Can't Catch
Because compilation is done independently on each file that we compile, there are also some errors that compilation cannot catch. For example:

- *Declared but undefined variables:* Use of variables that we've declared, but never actually defined (i.e., set aside memory for.) As an example, this could be the result of an:

```
extern int X;
```

in the header (.h) file, specifying that X is an int, without a matching:

```
int X;
```

in one—and only one—of the source (.cpp) files, which actually tells the compiler to set aside the memory to hold the value of X.

- *Multiply defined variables.* Presence of variables that we've set aside memory for in more than one place. For example, if we had the extern declaration of X in the header file, then had the "int X" in more than one source file, we'd essentially have two copies of X. The compiler can't catch that unless we try to set aside memory for X more than once *in the same file*.
- *Undefined functions.* Calls to functions that we have described (usually in an included file) but never actually defined. If the definition of a function is not in the current file, the compiler simply assumes that any function we've described must be defined in some other file or library of functions we've included.
- *Multiply defined functions.* Calls to functions that we've defined in more than one place. Since the compiler only looks at one file at a time, duplicate function definitions can only be detected if the same function is included in the same file more than once.
- *Careless errors.* Errors where the code we write is legal, its just not the code we intended to write (e.g., calling strcmp() instead of stricmp() to do a case-insensitive comparison).
- *Logic errors.* Code that is the result of improper thought patterns in designing or implementing our program

What the Compiler Produces

The end product of a compile is a file containing object code (usually with the same name as the source file, with a .o or .obj extension). Object code is similar to machine language, except that a lot of the addresses need to be filled in. For example:

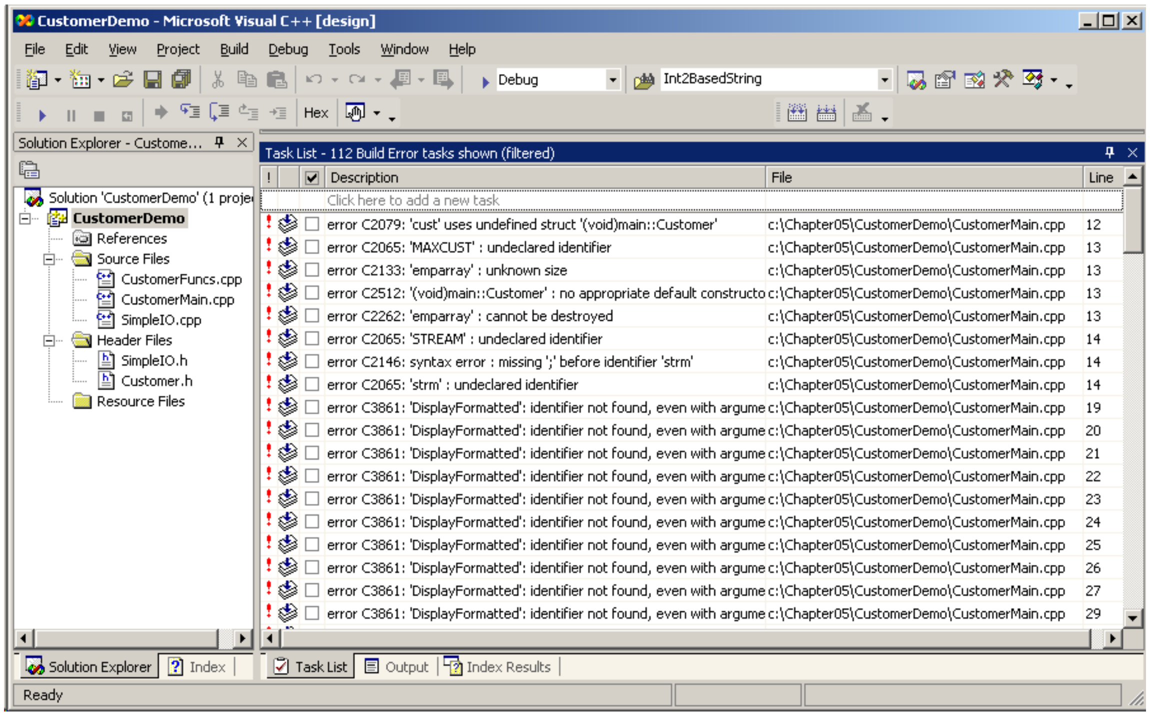
- Since all the object code files will need to be combined before the program can be run, actual addresses of the program instructions cannot be known until the files are combined
- The memory addresses of any variables that are described, but not actually created, are unknown
- The memory addresses for any functions that are called—but not defined—in the file are unknown.

These issues need to be resolved during the linking stage.

5.3.2: Compiling the sample code

Once you have set up your project, in accordance with the instructions in Section 5.2.3, you can perform your first build of the demo code.

The results are not pretty, as shown in Figure 5.2—which shows the Visual Studio .NET Task List. In your own build, the errors may appear in a different order, depending on how you added the files. Their substance should be the same, however.



Task List - 112 Build Error tasks shown (filtered)

Description	File	Line
Click here to add a new task		
error C2079: 'cust' uses undefined struct '(void)main::Customer'	c:\Chapter05\CustomerDemo\CustomerMain.cpp	12
error C2065: 'MAXCUST' : undeclared identifier	c:\Chapter05\CustomerDemo\CustomerMain.cpp	13
error C2133: 'emparray' : unknown size	c:\Chapter05\CustomerDemo\CustomerMain.cpp	13
error C2512: '(void)main::Customer' : no appropriate default constructo	c:\Chapter05\CustomerDemo\CustomerMain.cpp	13
error C2262: 'emparray' : cannot be destroyed	c:\Chapter05\CustomerDemo\CustomerMain.cpp	13
error C2065: 'STREAM' : undeclared identifier	c:\Chapter05\CustomerDemo\CustomerMain.cpp	14
error C2146: syntax error : missing ';' before identifier 'strm'	c:\Chapter05\CustomerDemo\CustomerMain.cpp	14
error C2065: 'strm' : undeclared identifier	c:\Chapter05\CustomerDemo\CustomerMain.cpp	14
error C3861: 'DisplayFormatted': identifier not found, even with argume	c:\Chapter05\CustomerDemo\CustomerMain.cpp	19
error C3861: 'DisplayFormatted': identifier not found, even with argume	c:\Chapter05\CustomerDemo\CustomerMain.cpp	20
error C3861: 'DisplayFormatted': identifier not found, even with argume	c:\Chapter05\CustomerDemo\CustomerMain.cpp	21
error C3861: 'DisplayFormatted': identifier not found, even with argume	c:\Chapter05\CustomerDemo\CustomerMain.cpp	22
error C3861: 'DisplayFormatted': identifier not found, even with argume	c:\Chapter05\CustomerDemo\CustomerMain.cpp	23
error C3861: 'DisplayFormatted': identifier not found, even with argume	c:\Chapter05\CustomerDemo\CustomerMain.cpp	24
error C3861: 'DisplayFormatted': identifier not found, even with argume	c:\Chapter05\CustomerDemo\CustomerMain.cpp	25
error C3861: 'DisplayFormatted': identifier not found, even with argume	c:\Chapter05\CustomerDemo\CustomerMain.cpp	26
error C3861: 'DisplayFormatted': identifier not found, even with argume	c:\Chapter05\CustomerDemo\CustomerMain.cpp	27
error C3861: 'DisplayFormatted': identifier not found, even with argume	c:\Chapter05\CustomerDemo\CustomerMain.cpp	29

Figure 5.2: Initial build results for CustomerDemo

Since we failed to achieve a successful compile, we cannot proceed to any further debugging activity until we fix the compiler errors, which we now do one by one.

CustomerMain.cpp

One way to keep errors from becoming overwhelming is to focus on one source file at a time. To accomplish this, we select the .cpp file we want to focus on, then use compile (Ctrl-F7, or use Debug|Compile) on the file, rather than doing a full build. This reduces the volume of errors we must pay attention to—although not by much.

Looking at the errors in Figure 5.2, we might notice a common theme. It doesn't appear to know what anything is. For example:

C2079: doesn't recognize the Customer structure

C2065: doesn't recognize the MAXCUST constant

C3861: doesn’t recognize DisplayFormatted(), which is part of SimpleIO

etc.

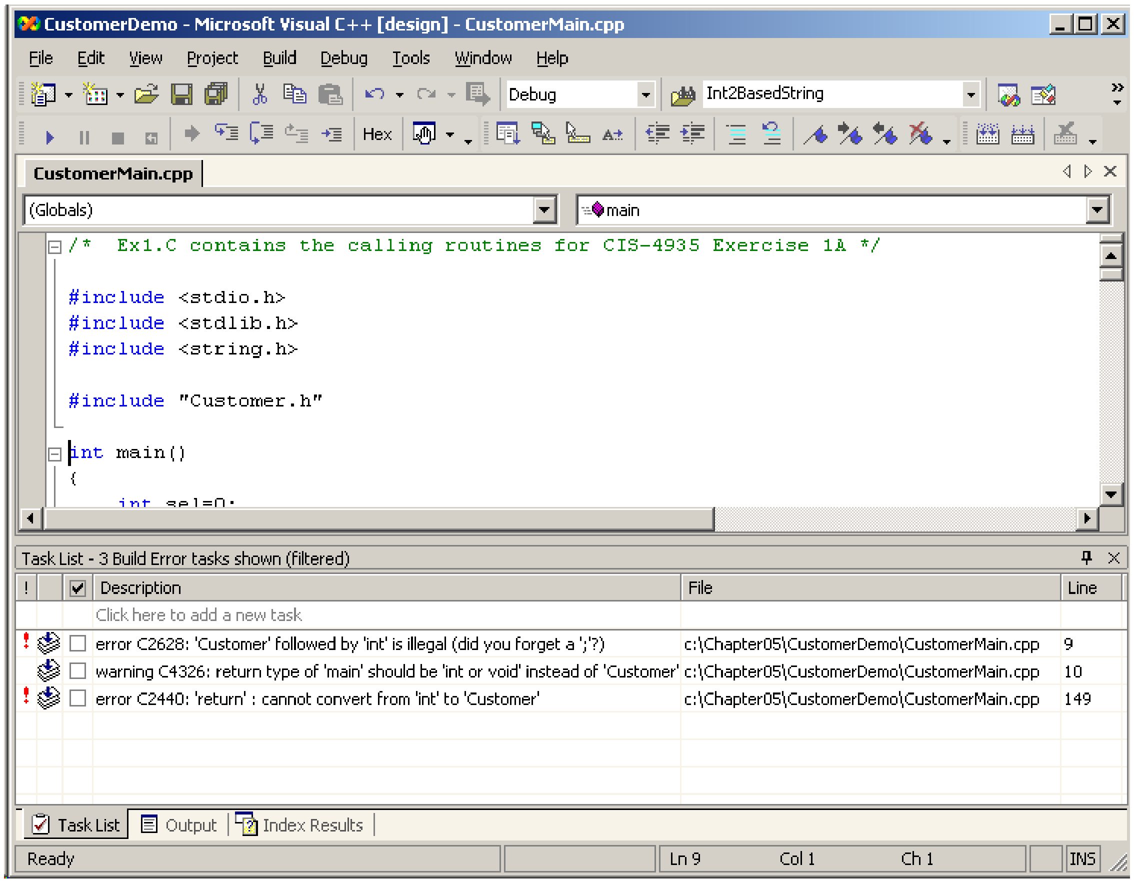
Whenever you get a collection of this type of errors, there's a good change there's some problem with the include files—since structure definitions and function prototypes are all placed in these files. Looking at the top of CustomerMain.cpp we then notice something: neither of the two header files in our project (Customer.h or SimpleIO.h) are included. Furthermore, if we were to look inside Customer.h, we'd notice that there is an:

```
#include "SimpleIO.h"
```

near the top—needed because STREAM is an argument to many of out functions. As a result, we can conclude that what we really need to do is place:

```
#include "Customer.h"
```

at the top of CustomerMain.cpp. Upon doing this, our number of errors drops down to 3 (assuming we compile CustomerMain.cpp by itself). These errors are presented in Figure 5.3.


CustomerDemo - Microsoft Visual C++ [design] - CustomerMain.cpp
File Edit View Project Build Debug Tools Window Help
Debug
Int2BasedString
Hex
CustomerMain.cpp
(Globals)
main
Task List - 3 Build Error tasks shown (filtered)
Task List
Output
Index Results
Ready
Ln 9
Col 1
Ch 1
INS

```
/*  Ex1.C contains the calling routines for CIS-4935 Exercise 1A */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "Customer.h"

int main()
{
    int sel=0;
```

!	✓	Description	File	Line
		Click here to add a new task		
!	☐	error C2628: 'Customer' followed by 'int' is illegal (did you forget a ';'?)	c:\Chapter05\CustomerDemo\CustomerMain.cpp	9
	☐	warning C4326: return type of 'main' should be 'int or void' instead of 'Customer'	c:\Chapter05\CustomerDemo\CustomerMain.cpp	10
!	☐	error C2440: 'return' : cannot convert from 'int' to 'Customer'	c:\Chapter05\CustomerDemo\CustomerMain.cpp	149

Figure 5.3: CustomerMain.cpp build errors after Customer.h included

These errors are rather perplexing. The first error, C2628, is located on the **int** return type for main. The second error, C4326, tells us the main() is attempting to return a Customer object, which is illegal. The third error, C2440, is on the return 0 at the end of main(), and tells us that it can't convert 0 into a Customer object. The bottom line of all this is that somehow the compiler has gotten the declaration of main() messed up.

Since there is nothing above our declaration of main() except #include statements, we can safely assume that the problem isn't in CustomerMain() at all, but is actually in one of the #include files. Furthermore, since the only one that we've written is Customer.h, it’s a safe bet that we should start looking there.

The bottom of Customer.h (which is where we should start, since you always work your way up when trying to find the source of an error) contains the code shown in Example 5.2. Superficially, it looks identical to the structure definition previously shown in Example 5.1. If you look very closely however, you'll notice one key difference—there is no semicolon at the end of the Example 5.2 definition.

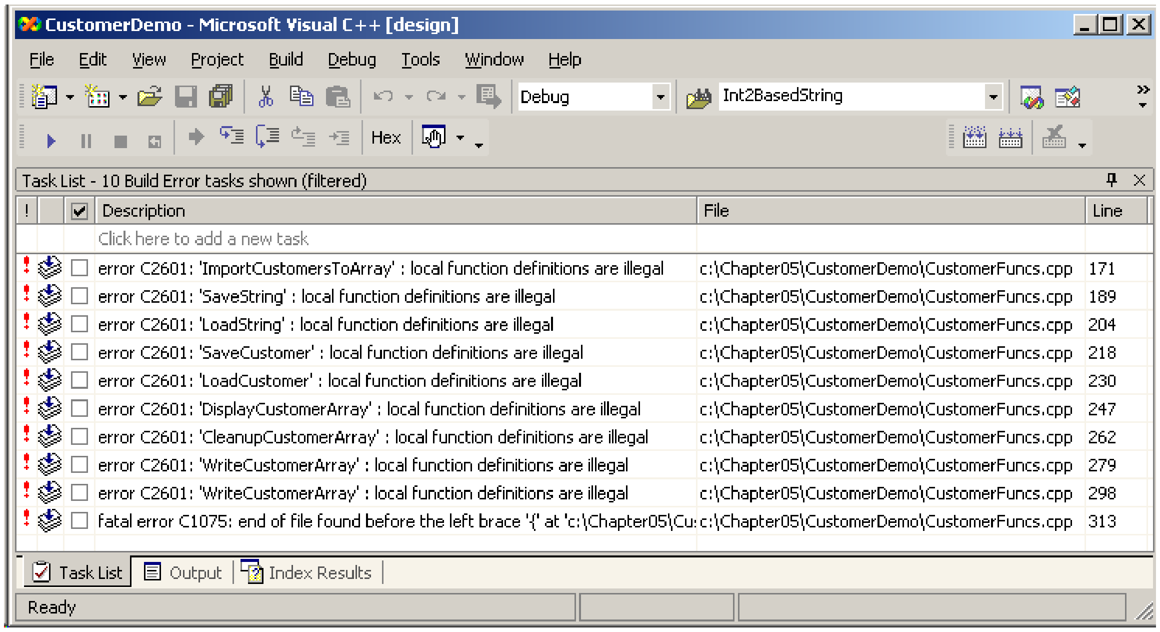
Example 5.2: Bottom of Customer.h file

```
struct Customer
{
	char *szName;
	char szCustID[4];
	double dCreditLine;
	long nLastVisit;
}
```

Failing to end a structure definition with a semicolon turns out to be something that really confuses the compiler. It takes this as meaning that whatever follows the definition is a variable that we are declaring. In this case, it what followed happened to be the int in the declaration line of main(). Moreover, the compiler also guessed that the structure definition that preceded main() was, in fact, our way of specifying main()'s return type. We can eliminate the confusion by putting the proper semicolon at the end of the structure definition in Customer.h. When we do so, we get a clean compile of CustomerMain.cpp. It is worth remembering that problems in .h files can lead to errors being listed in our .cpp files.

CustomerFuncs.cpp

Performing our compile of CustomerFuncs.cpp, the situation—once again—seems a bit bleak. The messages, presented in Figure 5.4, express this common theme regarding local functions definitions. If you were to check out Chapters 1-4, you'd find that we didn't mention local functions even once. Not surprising since—as the C2601 messages repeatedly state—such functions are apparently illegal.


CustomerDemo - Microsoft Visual C++ [design]
File Edit View Project Build Debug Tools Window Help
Debug
Int2BasedString
Hex
Task List
Output
Index Results
Ready

Task List - 10 Build Error tasks shown (filtered)

!		✓	Description	File	Line
			Click here to add a new task		
!		☐	error C2601: 'ImportCustomersToArray' : local function definitions are illegal	c:\Chapter05\CustomerDemo\CustomerFuncs.cpp	171
!		☐	error C2601: 'SaveString' : local function definitions are illegal	c:\Chapter05\CustomerDemo\CustomerFuncs.cpp	189
!		☐	error C2601: 'LoadString' : local function definitions are illegal	c:\Chapter05\CustomerDemo\CustomerFuncs.cpp	204
!		☐	error C2601: 'SaveCustomer' : local function definitions are illegal	c:\Chapter05\CustomerDemo\CustomerFuncs.cpp	218
!		☐	error C2601: 'LoadCustomer' : local function definitions are illegal	c:\Chapter05\CustomerDemo\CustomerFuncs.cpp	230
!		☐	error C2601: 'DisplayCustomerArray' : local function definitions are illegal	c:\Chapter05\CustomerDemo\CustomerFuncs.cpp	247
!		☐	error C2601: 'CleanupCustomerArray' : local function definitions are illegal	c:\Chapter05\CustomerDemo\CustomerFuncs.cpp	262
!		☐	error C2601: 'WriteCustomerArray' : local function definitions are illegal	c:\Chapter05\CustomerDemo\CustomerFuncs.cpp	279
!		☐	error C2601: 'WriteCustomerArray' : local function definitions are illegal	c:\Chapter05\CustomerDemo\CustomerFuncs.cpp	298
!		☐	fatal error C1075: end of file found before the left brace '{' at 'c:\Chapter05\Cu	c:\Chapter05\CustomerDemo\CustomerFuncs.cpp	313

Figure 5.4: Initial Compile of CustomerFuncs.cpp

Interestingly enough, the C2601 message (which can be quite common, unfortunately), is not as cryptic as it first seems. You already know that any variable declared within a block is a local variable. It stands to reason, then, that a function definition inside a block would be a "local function". Since we didn't intentionally define our function within a block, a good guess would be that there is a brace matching problem. This guess is further supported by the unexpected end of file message, C1075, which concludes the Figure 5.4 list.

As we suggested for the previous error, it is a good idea to work upwards when strange errors start appearing. The declaration of InportCustomersToArray() and the lines above it are presented in Example 5.3.

Example 5.3: Code above first local function definition error in CusomerFuncs.cpp

```
int DisplayCustomerTextFile(const char *arg1)
{
        fstream custfile;
        struct Customer cust;
        custfile.open(arg1,ios_base::in);
        if (!custfile.is_open()) return 0;
        while (!custfile.eof())
        {
                if (LoadCustomerFromText(custfile,&cust))
                {
                        DisplayCustomer(&cust);
                        if (cust.szName!=0) {
                                delete [] cust.szName;
```

```
                            cust.szName=NULL;
              }
       }
       return 1;
}

/* ImportCustomersToArray takes a text file containing Customer records (named
   filename) and loads them into an array of Customer structures
   (beginning at address pCust). It will load no more than nMaxCount
   Customers. It returns a count of Customers loaded if successful,
   0 otherwise. */

int ImportCustomersToArray(const char *filename,
                           struct Customer *pCust,int nMaxCount)
{
       STREAM custfile;
       int nCtr=0;
       custfile.open(filename,ios_base::in);
```

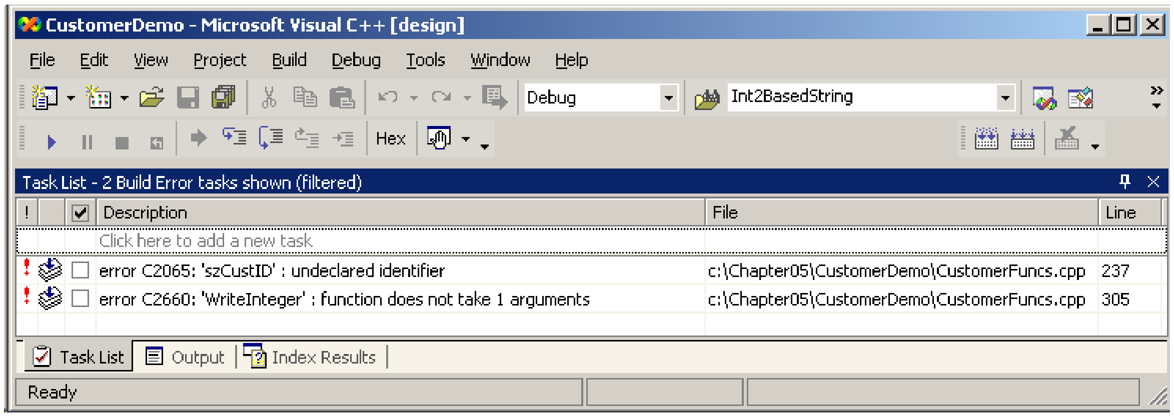
If you look carefully at the function above ImportCustomersToArray(), you'll notice something a bit strange—the if block beginning if (cust.szName!=0) doesn't seem to end properly, since the brace below it clearly seems to be associated the if block beginning:

```
    if (LoadCustomerFromText(custfile,&cust))
```

Maintaining consistent indentation is critical if you want to avoid such problems. Adding a closing brace to the inner test solves this particular problem, i.e.,

```
    if (cust.szName!=0) {
        delete [] cust.szName;
        cust.szName=NULL;
    } // add brace here
```

When recompiling CustomerFuncs.cpp after fixing the braces, two new errors surface—shown in Figure 5.5. Problems with syntax, such as improper braces and missing semicolons frequently hide subsequent errors in C++ code. Therefore, you should always try to fix errors in the order that they appear in the task list. You should also recompile every time you fix an error, since a single fix can often impact many errors.


CustomerDemo - Microsoft Visual C++ [design]
File Edit View Project Build Debug Tools Window Help
Debug
Int2BasedString
Hex

Task List - 2 Build Error tasks shown (filtered)

!		✓	Description	File	Line
			Click here to add a new task		
!		☐	error C2065: 'szCustID' : undeclared identifier	c:\Chapter05\CustomerDemo\CustomerFuncs.cpp	237
!		☐	error C2660: 'WriteInteger' : function does not take 1 arguments	c:\Chapter05\CustomerDemo\CustomerFuncs.cpp	305

Task List
Output
Index Results
Ready

Figure 5.5: Compile errors in CustomerFuncs.cpp after brace problem is fixed

The C2065 (undeclared identifier) error is in the memset() line of the function LoadCustomer(), shown in Example 5.4. The error is telling us that the variable szCustID has not been declared, so it doesn't know what to do with it.

Example 5.4: LoadCustomer() function in CustomerFunc.cpp

```
bool LoadCustomer(STREAM &binfile,struct Customer *pCust)
{
      if (!ReadBlock(binfile,pCust->szCustID,sizeof(pCust->szCustID)))
            return false;
      pCust->nLastVisit=ReadInteger(binfile);
      pCust->dCreditLine=ReadReal(binfile);
      char szCustname[256];
      memset(szCustID,0,256);
      if (!LoadString(binfile,szCustname)) return false;
      pCust->szName=new char[strlen(szCustname)+1];
      strcpy(pCust->szName,szCustname);
      return true;
}
```

There are two ways we could go about trying to fix the error. Once would involve tracing through the function and attempting to solve the logic. The second, "quick and dirty" approach would be to notice (from the first line) that szCustID is actually an element of the Customer structure, and simply change the line to pCust->szCustID. We will take the second approach for now, which will eliminate the compile error. Unfortunately, we will have cause to regret this approach later.

Example 5.5: Function containing WriteInteger() problem

```
/* WriteCustomerArray takes an array of Customer objects and populates it
by reading them from a binary file, first opening the file, then reading
the count, then reading each individual customer element and, finally, closing
the file. It returns the number of elements read. */
int WriteCustomerArray(const char *filename,struct Customer *pCust,
                           int nMaxCount)
{
	STREAM fbin;
	int i=0,nCount=0;
	if (!Open(fbin,filename,true,false,false,false))
		return false;
	WriteInteger(nCount);
	for(i=0;i<nCount;i++) {
		if (!LoadCustomer(fbin,pCust+i)) {
			if (i>0) CleanupCustomerArray(pCust,i);
			break;
		}
	}
	Close(fbin);
	return nCount;
}
```

The final remaining error, C2660, tells us that WriteInteger() does not take a single arguments—which SimpleIO tells us to be true since we need to specify both the file stream and the integer we want to write to it. The code for the function is presented in Example 5.5. In this code, the statement WriteInteger(nCount) is clearly missing its stream argument, so we replace it with WriteInteger(fbin,nCount). That clears up the compiler errors (but, once again, we will find that it hides a problem we should have addressed when we encountered it).

Upon making that final change, we get a clean compile. Unfortunately, when we build the entire project, we encounter our next set of errors.

5.3 Section Questions

A list of common compiler errors is specified below:

Fatal Error C1083: Cannot open include file
Error C2015: too many characters in constant
Error C2040: 'char *(const char *)' differs in levels of indirection from 'int (const char *)'
Error C2065: undeclared identifier
Error C2143: syntax error : missing ';' before 'type'
Error C2166: l-value specifies const object
Error C2601: 'function-name' : local function definitions are illegal
Error C2664: 'function-name' : cannot convert parameter 1 from 'const char *' to 'char *'
Warning C4127: conditional expression is constant
Warning C4700: local variable used without having been initialized

Identify the error associated with each of the following code defects:

1. You placed single quotes around a string instead of double quotes
2. You forgot to put an #include directive at the top of your source file
3. You wrote a test str == val, where str is a string and val is an integer
4. You are using a variable InterestRate, but actually declared it as Interestrate
5. You write a test if (MAXLINE<80), where MAXLINE is actually a constant you defined for the maximum size of a line
6. You define a value "int nVal" then use it next in the expression "if (nVal>3) return;"
7. Your function prototype is "void func1(char *b1,const char b2[], int b3)" and inside the function the line of code "b2[0]=b3" appears
8. You misspell the name of an include file
9. You're missing a brace in one of your functions
10. Your code contains strcpy("Hello",buf)

5.4: Common Linker Errors

The linker is much simpler than the compiler, and typically returns only two types of errors: errors about things we haven't defined, and errors about things we've defined too many times. In this section, we'll overview the linker's function, then proceed to fixing the linker errors in the CustomerDemo project.

5.4.1: Working with the linker

The linker's job is to take all the code that our program uses and pack it together into a single, self-contained unit—the .exe file. Static linking, the type we perform when we are creating our program, normally gets the object code it will join together from two sources:

- .obj files produced when we successfully compile each C/C++ source file
- .lib files, containing routines that have already been compiled, outside of our particular application. The .lib file format is a convenient way to hold a large collection of .obj files. For example, the standard C++ libraries are normally stored in .lib files—the only source code we need is the header files, which contain the function prototypes and other definitions.

The heart of the linking process involves resolving references to functions and variables made within the various .lib and .obj files. A function call made within one .obj to a function that is defined somewhere else is sometimes called an *external reference*. The same term also applies to global variables, which may be used in one file yet actually declared in another. These external references do not bother the compiler. Checking them is not its job. The linker, however, must make sure that every function is defined somewhere, and that every variable is declared somewhere.

When we discussed the compiler, we noted there were six types of errors the compiler could not catch. Consistent with the linker's purpose, it is able to detect all but two of these errors:

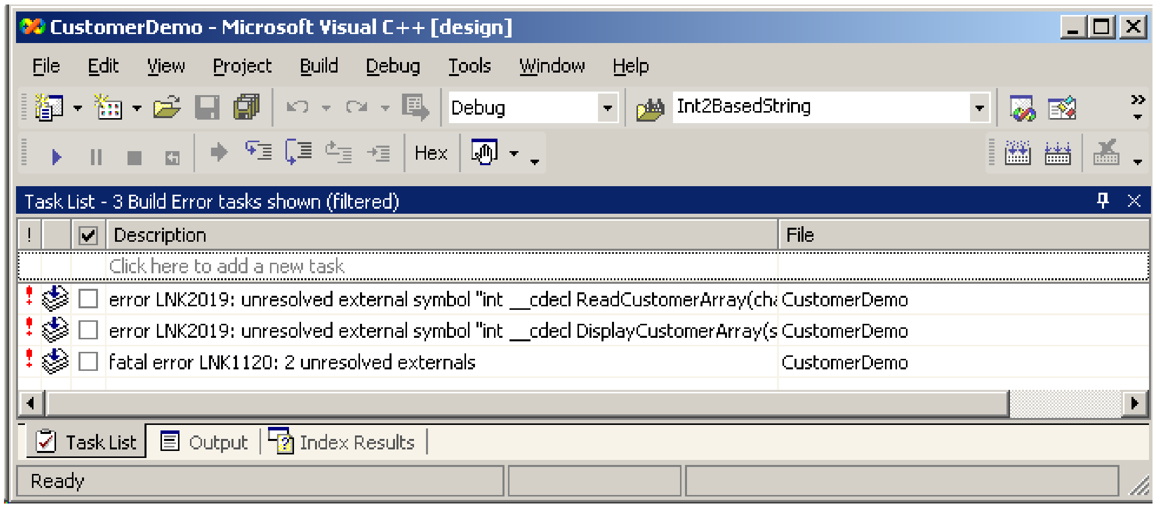
- Declared but undefined variables
- Multiply defined variables
- Undefined functions
- Multiply defined functions

Sadly, careless errors and logic errors elude the linker, just as they eluded the compiler.

Where undefined variables and undefined functions are detected, the linker usually has words to the effect of "unresolved external" in its error message. Where multiple definition or declaration is present, the message usually includes "already defined".

5.4.2: Linking the sample code

When you build your project in Visual Studio .Net, a link step automatically follows a successful compile of all source files. The results of the link for our sample project are presented in Figure 5.6.


CustomerDemo - Microsoft Visual C++ [design]
File
Edit
View
Project
Build
Debug
Tools
Window
Help
Debug
Int2BasedString
Hex
Task List - 3 Build Error tasks shown (filtered)
Click here to add a new task
Task List
Output
Index Results
Ready

!		✓	Description	File
!		☐	error LNK2019: unresolved external symbol "int __cdecl ReadCustomerArray(ch	CustomerDemo
!		☐	error LNK2019: unresolved external symbol "int __cdecl DisplayCustomerArray(s	CustomerDemo
!		☐	fatal error LNK1120: 2 unresolved externals	CustomerDemo

Figure 5.6: Linker errors in CustomerDemo project

The two errors we see in this project are LNK2019 errors, the "unresolved external" error. These errors tell us that a function we have promised to define in our header file cannot be found in out project. The two functions mentioned are:

ReadCustomerArray(), and

DisplayCustomerArray()

The presence of these errors typically means one of three things: 1) we forgot to define the function, 2) we made a misspelling when we defined the function, or 3) our function definition doesn't exactly match the declaration in the header.

Unfortunately, unlike compiler errors, you can’t just jump to a linker error to find it. This makes sense, since linker errors typically tell us something is missing—and how do you jump to something that is not there. So, a good way to start looking for the cause of a linker error (assuming you didn't forget to write the function) is to do a search for it. When we search for ReadCustomerArray in CustomerFuncs.cpp, however, the editor doesn't find it. In inspecting the file manually, however, we do find two copies of the function WriteCustomerArray(), the second of which was presented in Example 5.5. Looking more closely at the example, we might also notice that the comments suggest the function is supposed to be reading data, not writing it. So, perhaps, we just got the name wrong. Changing the name of the function to ReadCustomerArray(), the problem does, in fact, go away.

This type of scenario is not as implausible as it first seems. Very often, when you are creating parallel functions to do file I/O, you'll create a version of the function that saves (writes) first, the copy it to be sure that the function that reads the data performs every action in the same order. It's not that unusual to forget to change a name or two when altering the copy.

The second problem is more mysterious. When we search for DisplayCustomerArray in CustomerFuncs.cpp, we discover that the function is there, as shown in Example 5.6.

Example 5.6: DisplayCustomerArray() in CustomerFuncs.cpp

```
/* DisplayCustomerArray takes an array of Customer objects containing nCount
elements and displays each to the console. It returns 0 if nCount is 0, 1
otherwise */

int DisplayCustomerArray(struct Customer *pCust,int nCount)
{
	int i;
	if (nCount==0) return 0;
	for(i=0;i<nCount;i++)
	{
		DisplayCustomer(pCust+i);
	}
	return 1;
}
```

When the linker tells you a function isn't there, yet you see that it is, it’s a good time to check the declaration in the header file. In this case, the prototype is:

```
int DisplayCustomerArray(const struct Customer *pCust,int nCount);
```

Although it looks very similar to the declaration line in the function definition, they are not identical. Specifically, the function definition is missing the **const** that precedes the structure argument in the header file. Unfortunately, that small difference is sufficient to convince the linker that the function in Example 5.6 is not the one that we promised to declare in the header file. So it gives us an error.

The const qualifier can be very useful in surfacing errors. As a result, you should always use if on arguments that you are not planning to change. In this case, the purpose of the function is to display the contents of the array that is passed in. Since displaying something typically does not involve changing it, it seems reasonable to try changing the function definition to make the argument const, the way it is in the header file. The result is a clean compile and link. (Had we been changing the Customer array within our function, we'd have gotten a whole new set of compiler errors complaining about the change).

5.4 Section Questions

1. What is an "already defined" linker error?
2. What is an "unresolved external" linker error?
3. What other common types of linker errors are there?
4. Why didn't we get an "already defined" linker error when we had two copies of WriteCustomerArray()? (Hint: the answer can be found in the second linker error that we corrected).

5.5: Finding Logic Errors

In this section, we address the process of finding logic errors. First, we talk a bit about some of the philosophies involved, then we proceed to working with the debugger to find some of the errors that have been seeded into the CustomerDemo project.

5.5.1: Philosophies for finding logic errors

There is no magic formula for finding logic errors in a program. Like programming, it is a skill that is acquired with time. Also like programming, as you get good at it, you tend to have a hard time explaining what it is you're doing—sometimes referred to as the "Paradox of Expertise"[2]

Some Rules for Testing
As you start trying to find logic errors by running your program, it is probably useful to keep some general rules in mind:

- Every time you write a function, create another function to test it (or add a temporary test routine to main). The anxiety of having 10 untested functions is more than 10 times as great as the anxiety of having one untested function. Or at least it should be.
- When testing a function, make sure you test every case it was designed to perform. For example, if you were testing the Strmove function presented at the end of Chapter 7, you'd want to test insertion, deletion and the no-move case before pronouncing it operational.
- Always make sure you test the boundaries of your functions. For example, you should always explicitly check an insertion or deletion routine for insertions/deletions at the beginning and end of an array.
- Check for unusual cases. As the number of inputs to a function or program grows, it becomes increasingly tempting to focus on common cases. One of the benefits of automated testing is that it allows unusual cases to be generated and tested more easily. If you're working on an application that gets distributed to a large user community, the unusual cases tend to occur with frightening regularity. Thus, you should always be thinking about how you could incorporate some form of automated testing into your system.
- Check pathological cases. Check cases that you know should never happen—such as negative numbers being entered where only positive numbers are expected. Here what you want to be assured of is that your function degrades gracefully—providing an error message if appropriate, but never crashing your program. Even

[2] Waterman, D.A., *Expert Systems*. Addison Wesley: 1986.

users who are not intending to feed a program bad data can do so with an accidentally typed key or mouse click.

- Think of your program as the adversary. When you are developing an application, it is psychologically healthy to develop "affectionate" feelings towards it—though it’s also okay to call it names occasionally (just to keep it in its place). But once testing begins, you've got to try to bring it to its knees. Experts in the testing field encourage you to "think deviously" as you put a function or program through its paces. Devise scenarios you just "know" it can't handle.

None of these "rules" guarantee that you will find all the defects in an application, or that the defects you do find will surface easily. You should also be aware of an important fact: once you've got your code so that is basically works, it's almost always easier to fix a subtle defect than it is to find it. As a consequence, you need to fully understand an error before you write the code that tries to fix it. Otherwise, if the problem goes away, chances are that you haven't fixed it, you've just made it subtler—increasing the likelihood that some hapless user will be the lucky one to experience it first.

5.5.2: Running the code to find errors

By now, your program should be “runnable”. Unfortunately, being runnable and actually running properly are two very different issues. As your programs get larger and you become more experienced with a language, you’ll find about 2% of your debugging time is devoted to fixing compiler/linker errors, and 98% is devoted to dealing with runtime errors. (The author apologizes, in advance, to those offended by the extraordinary level of scruffiness implied by that last observation).

We will now go through finding and correcting some of these errors, making extensive use of the debugger in the process.

Runtime error: Program will not exit properly

When debugging a program, it is usually a good idea to start running the program and then just exit. (In a Windows-based program, the activity of just starting up and exiting actually tests quite a few features). When we do this with the DebugDemo program, we discover a problem very quickly—it won’t exit.

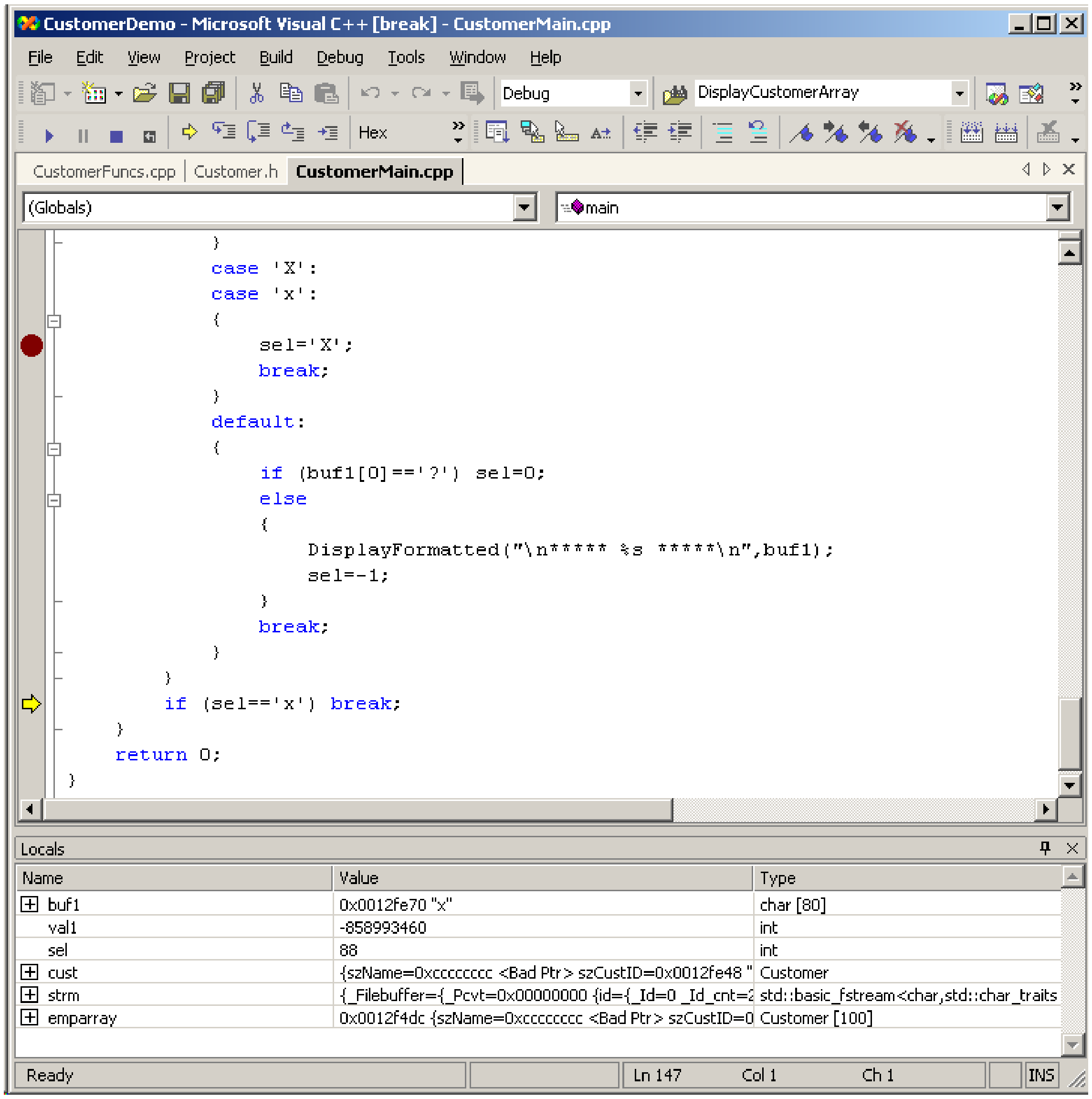
Since we know that the basic menu system is implemented in manner similar to the mainmenu function (presented in Chapter 2, Section 2.5.6), we know that each menu option has its own case, so it makes sense to put a breakpoint at the 'x' case, then run to it to see what happens. Upon reaching the line with our breakpoint, which is:

```
sel = 'X';
```

we step over it and drop out of the case construct at the bottom of the loop, as shown in Figure 5.7, to the line below:

```
if (sel=='x') break;
```

At this point our problem becomes pretty obvious—we set the variable sel to 'X' (capital x), but tested for 'x' (lower case x) to break out of the loop. To deal with this inconsistency, we change our test to a test for 'X', then recompile and retest. The gratifying result is that our program now exits.


CustomerDemo - Microsoft Visual C++ [break] - CustomerMain.cpp
File Edit View Project Build Debug Tools Window Help
Debug
DisplayCustomerArray
Hex
CustomerFuncs.cpp Customer.h CustomerMain.cpp
(Globals)
main

```
                }
                case 'X':
                case 'x':
                {
                    sel='X';
                    break;
                }
                default:
                {
                    if (buf1[0]=='?') sel=0;
                    else
                    {
                        DisplayFormatted("\n***** %s *****\n",buf1);
                        sel=-1;
                    }
                    break;
                }
            }
            if (sel=='x') break;
        }
        return 0;
    }
```

Locals

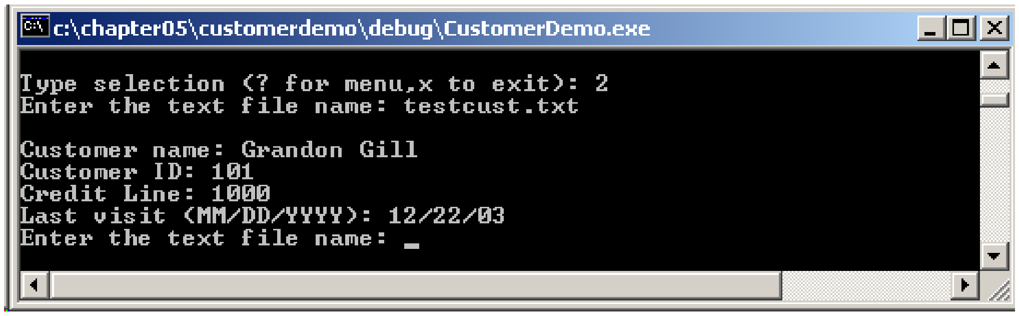
Name	Value	Type
buf1	0x0012fe70 "x"	char [80]
val1	-858993460	int
sel	88	int
cust	{szName=0xcccccccc <Bad Ptr> szCustID=0x0012fe48 "	Customer
strm	{_Filebuffer={_Pcvt=0x00000000 {id={_Id=0 _Id_cnt=2	std::basic_fstream<char,std::char_traits
emparray	0x0012f4dc {szName=0xcccccccc <Bad Ptr> szCustID=0	Customer [100]

Ready Ln 147 Col 1 Ch 1 INS

Figure 5.7: Stepping through the exit process

Runtime error: Double prompt on Option 2

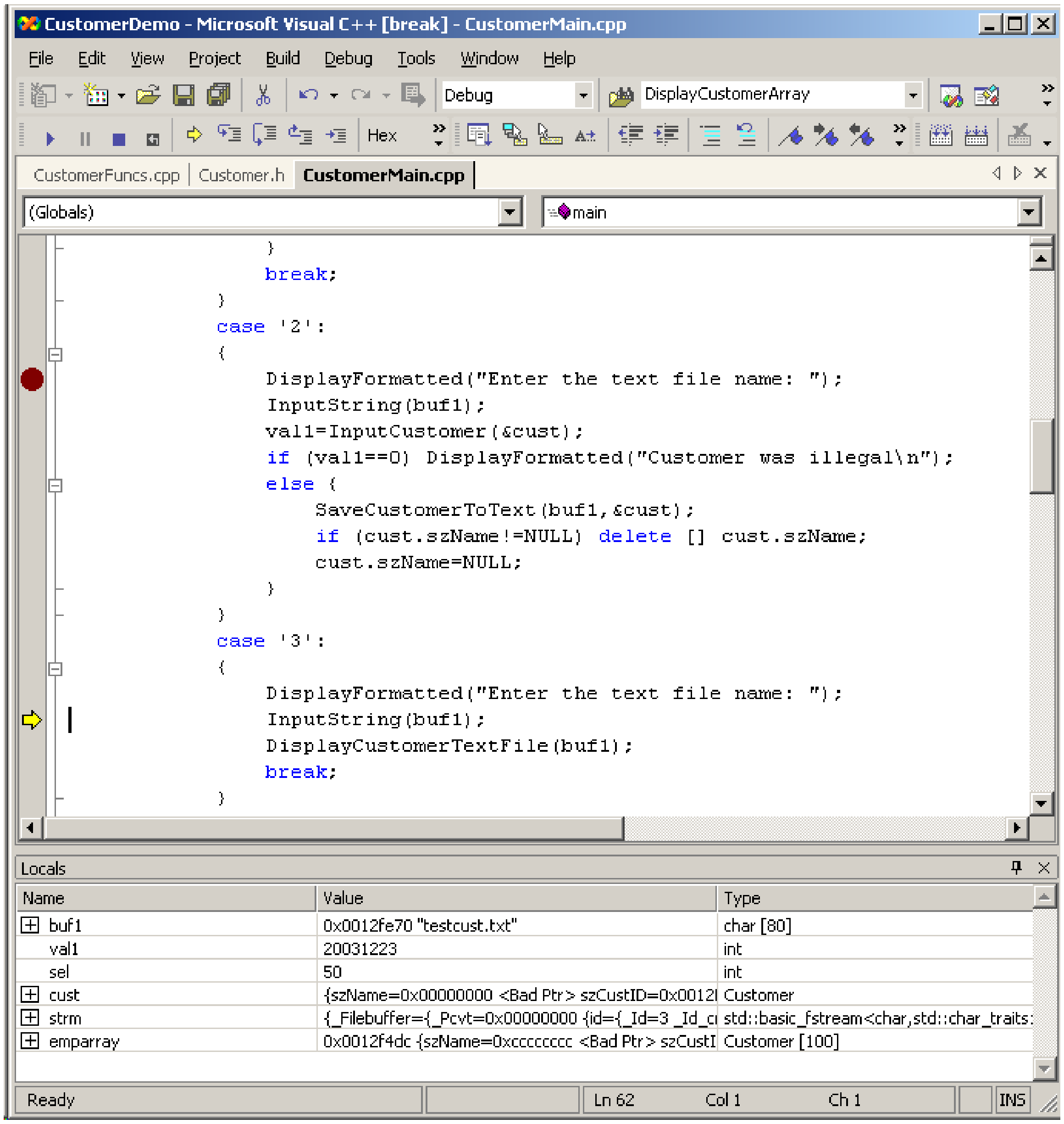
Having verified that the exit problem is fixed, we can now go on to test other functions. Option 1, inputting a customer and then redisplaying it, seems to work okay. When we select Option 2, however, a curious thing happens. It prompts us for a file name twice, as shown in Figure 5.8.



```
Type selection (? for menu,x to exit): 2
Enter the text file name: testcust.txt

Customer name: Grandon Gill
Customer ID: 101
Credit Line: 1000
Last visit (MM/DD/YYYY): 12/22/03
Enter the text file name: _
```

Figure 5.8: Getting prompted twice in Option 2

Because this double prompting was not part of the program's design, a good idea is to put a breakpoint in the code and step through it. Placing a breakpoint at the start of Option 2 in the case construct in main(), we find that everything proceeds according to plan until we finish saving to the file—at which time we find ourselves starting Option 3, as shown in Figure 5.9.


CustomerDemo - Microsoft Visual C++ [break] - CustomerMain.cpp
File Edit View Project Build Debug Tools Window Help
Debug
DisplayCustomerArray
Hex
CustomerFuncs.cpp Customer.h CustomerMain.cpp
(Globals)
main

```
                }
                break;
            }
            case '2':
            {
                DisplayFormatted("Enter the text file name: ");
                InputString(buf1);
                val1=InputCustomer(&cust);
                if (val1==0) DisplayFormatted("Customer was illegal\n");
                else {
                    SaveCustomerToText(buf1,&cust);
                    if (cust.szName!=NULL) delete [] cust.szName;
                    cust.szName=NULL;
                }
            }
            case '3':
            {
                DisplayFormatted("Enter the text file name: ");
                InputString(buf1);
                DisplayCustomerTextFile(buf1);
                break;
            }
```

Locals

Name	Value	Type
⊞ buf1	0x0012fe70 "testcust.txt"	char [80]
val1	20031223	int
sel	50	int
⊞ cust	{szName=0x00000000 <Bad Ptr> szCustID=0x0012l	Customer
⊞ strm	{_Filebuffer={_Pcvt=0x00000000 {id={_Id=3 _Id_c	std::basic_fstream<char,std::char_traits:
⊞ emparray	0x0012f4dc {szName=0xcccccccc <Bad Ptr> szCustI	Customer [100]

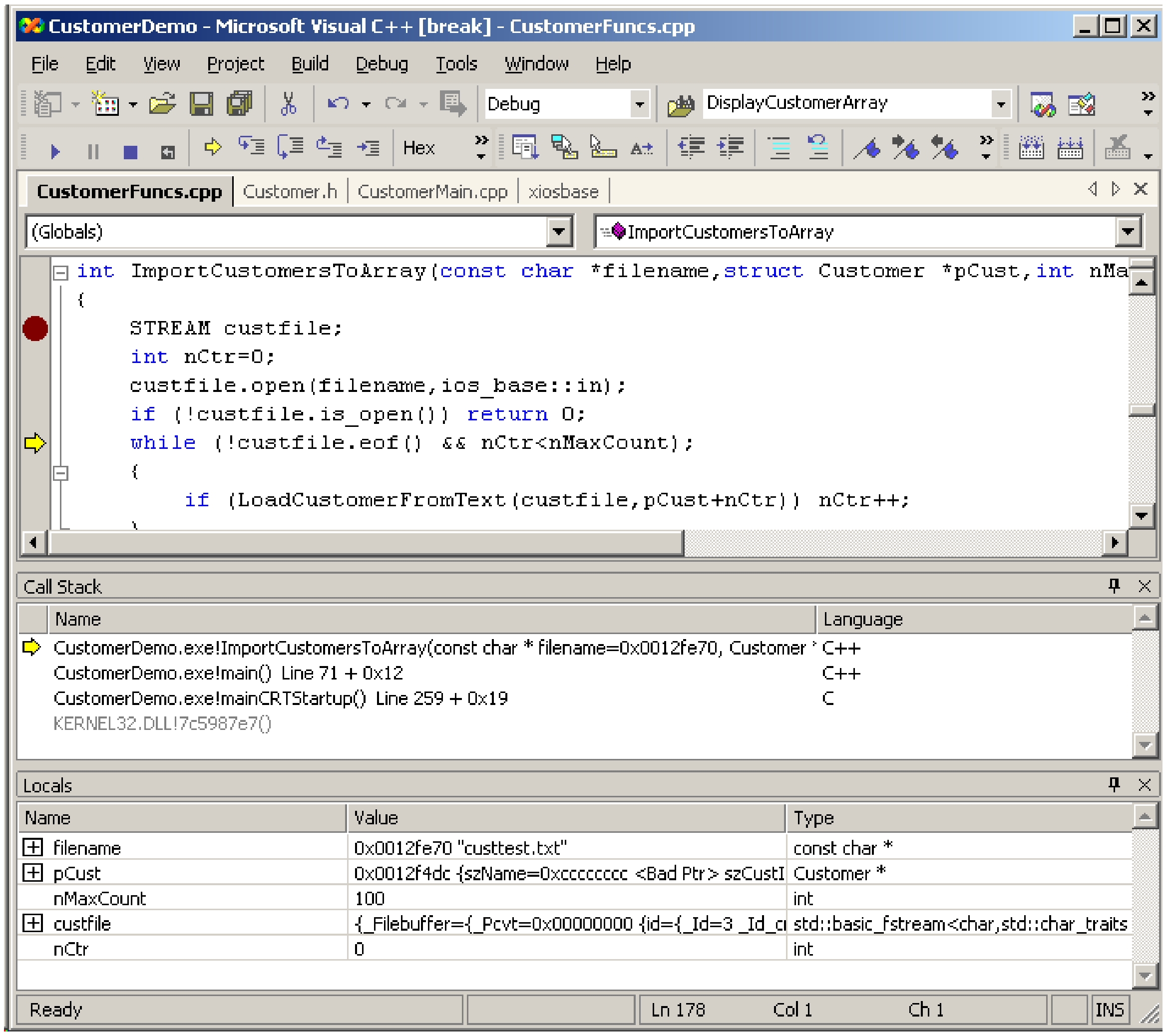
Ready | Ln 62 | Col 1 | Ch 1 | INS

Figure 5.9: Ending up in option 3 after completing Option 2

This behavior is absolutely typical of what happens when you forget to put a break in a case construct. Sure enough, the Option 2 block does not have a break, so we have fallen through. By putting a break at the end of the case '2' block, we solve the problem. We then test Option 3 and verify that it does, indeed, read in the customers that we saved and displays them.

Runtime error: Lockup on Option 4

Option 4 is supposed to read in customers from a text file, place them in an array. When we run it, however, the program just hangs (it might also generate an exception dialog, on some systems). To diagnose this problem, it makes sense to step into the function to find out what is going on. We can do this by placing a breakpoint at the top of the ImportCustomersToArray() function, then stepping until the program pauses. When we do this, we find the program stops at the first line of the while() loop, as shown in Figure 5.10.



```
int ImportCustomersToArray(const char *filename,struct Customer *pCust,int nMa
{
    STREAM custfile;
    int nCtr=0;
    custfile.open(filename,ios_base::in);
    if (!custfile.is_open()) return 0;
    while (!custfile.eof() && nCtr<nMaxCount);
    {
        if (LoadCustomerFromText(custfile,pCust+nCtr)) nCtr++;
```

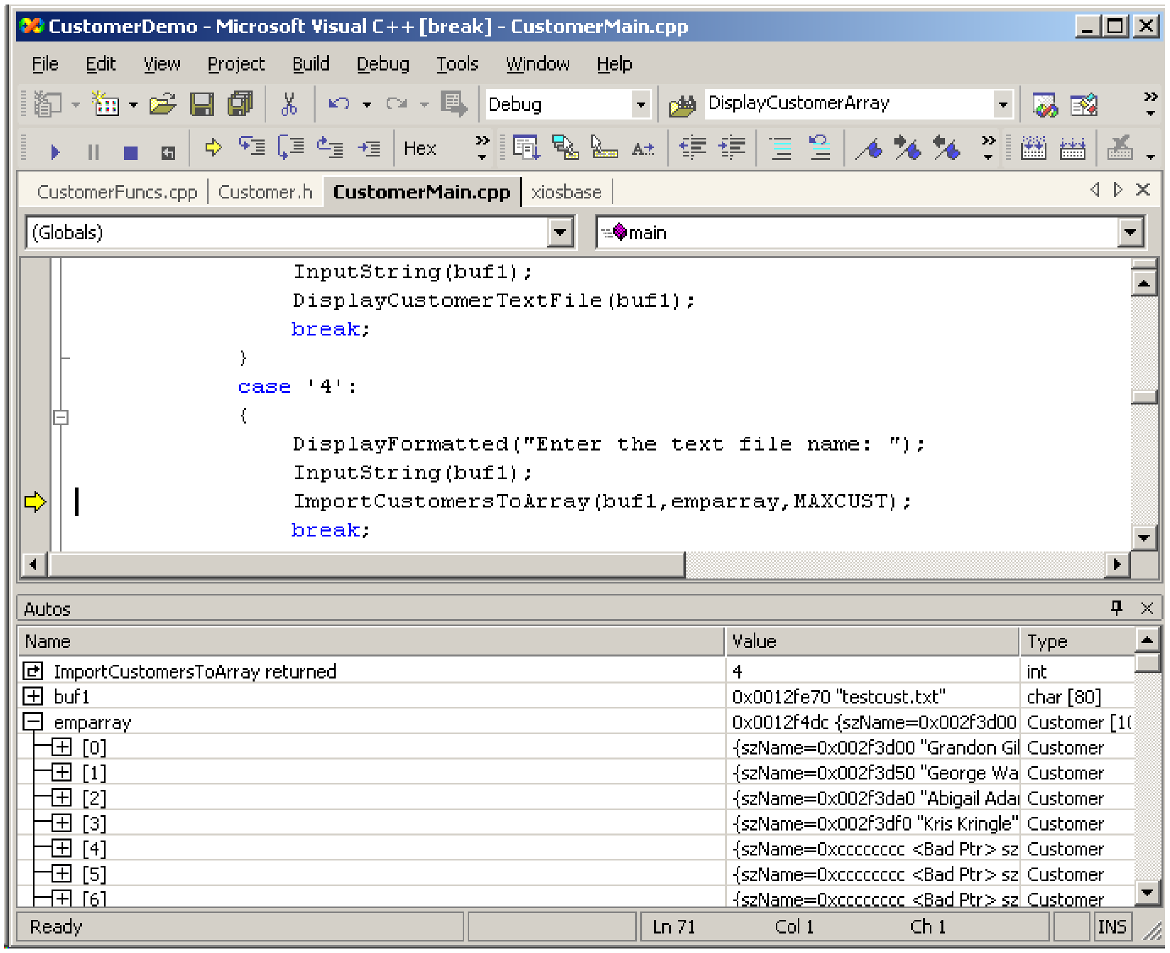
Name	Language
CustomerDemo.exe!ImportCustomersToArray(const char * filename=0x0012fe70, Customer *	C++
CustomerDemo.exe!main() Line 71 + 0x12	C++
CustomerDemo.exe!mainCRTStartup() Line 259 + 0x19	C
KERNEL32.DLL!7c5987e7()	

Name	Value	Type
⊞ filename	0x0012fe70 "custtest.txt"	const char *
⊞ pCust	0x0012f4dc {szName=0xcccccccc <Bad Ptr> szCustI	Customer *
nMaxCount	100	int
⊞ custfile	{_Filebuffer={_Pcvt=0x00000000 {id={_Id=3 _Id_c	std::basic_fstream<char,std::char_traits
nCtr	0	int

Figure 5.10: Position where program locks up (line pointed to by arrow)

The problem here is as simple as a semicolon. The while() loop is supposed to keep reading lines until an end-of-file condition is reached. Unfortunately, the semicolon following the while construct causes it to ignore the code block below the statement. It just keeps checking for the end-of-file over and over again, without doing anything to change its file position. Removing the semicolon solves the problem.

On the other hand, as a result of a design problem (not writing a function to display the contents of the array), we don't really know if the function works. The debugger can be useful in this regard. If we step out of the function, we can then use the "Auto" window to examine the return value and contents of the array, as shown in Figure 5.11, which also shows the call stack in its middle pane. Because the array is populated with the data that was entered, and the return value reflects the number of elements in the array (as it is supposed to), it is reasonable to assume that the function is working. Using the debugger in this manner often reduces or eliminates the need for lots of test code to display intermediate values.

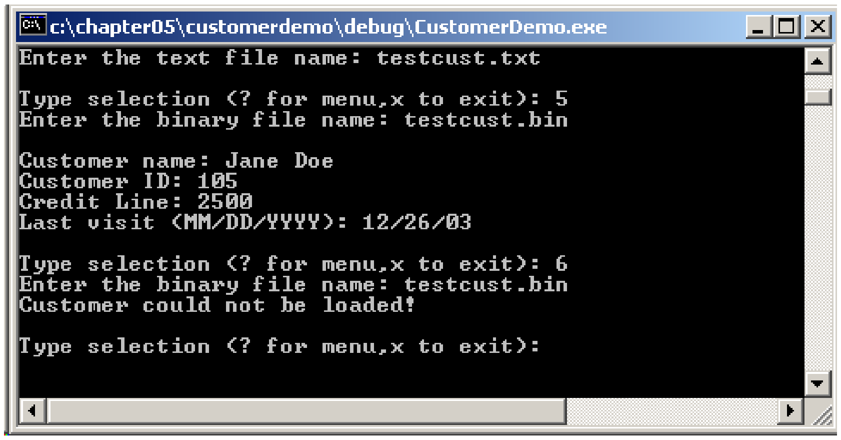


```
                    InputString(buf1);
                    DisplayCustomerTextFile(buf1);
                    break;
                }
                case '4':
                {
                    DisplayFormatted("Enter the text file name: ");
                    InputString(buf1);
                    ImportCustomersToArray(buf1,emparray,MAXCUST);
                    break;
```

Name	Value	Type
ImportCustomersToArray returned	4	int
buf1	0x0012fe70 "testcust.txt"	char [80]
emparray	0x0012f4dc {szName=0x002f3d00	Customer [1(
[0]	{szName=0x002f3d00 "Grandon Gil	Customer
[1]	{szName=0x002f3d50 "George Wa	Customer
[2]	{szName=0x002f3da0 "Abigail Adai	Customer
[3]	{szName=0x002f3df0 "Kris Kringle"	Customer
[4]	{szName=0xcccccccc <Bad Ptr> sz	Customer
[5]	{szName=0xcccccccc <Bad Ptr> sz	Customer
[6]	{szName=0xcccccccc <Bad Ptr> sz	Customer

Figure 5.11: Stepping out of the ImportCustomersToArray() function

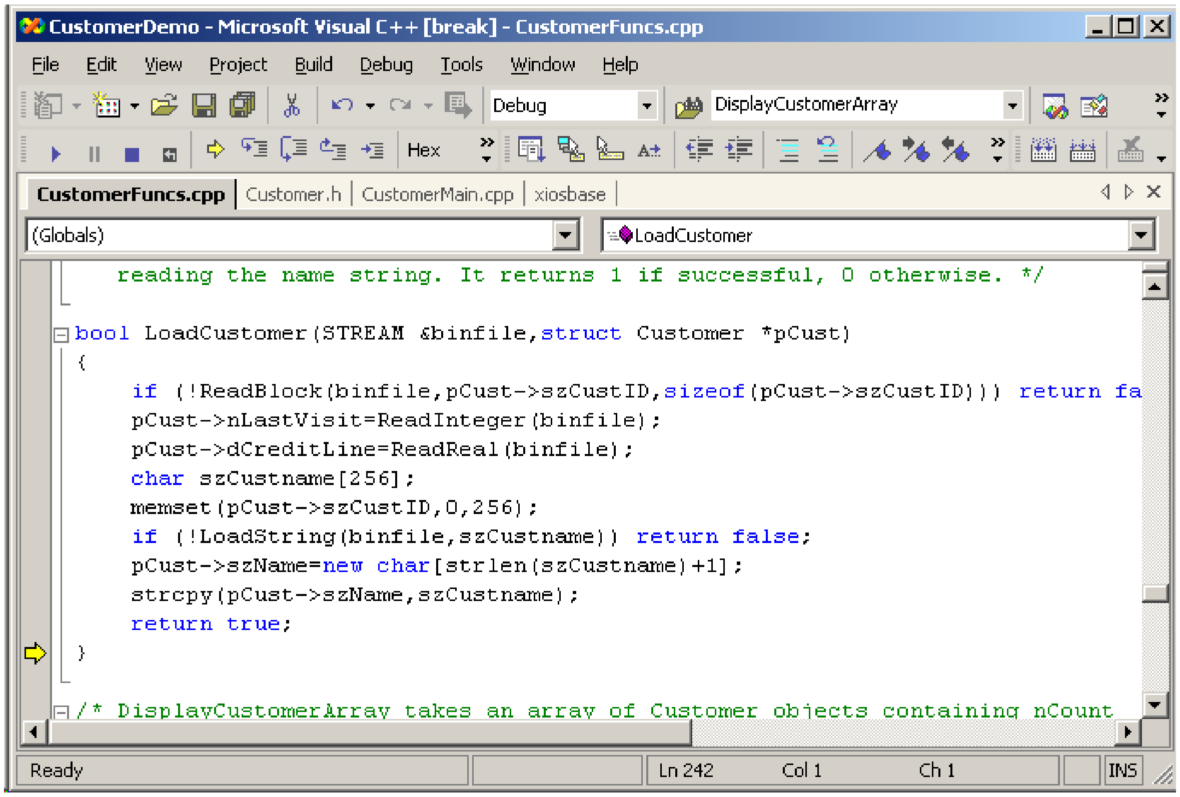
Runtime error: Problems with binary load and save (Options 5 and 6)
When we run Option 5—which prompts us for a binary file, followed by customer date to be saved to the file—the program appears to run properly (although w can't be sure until we reload the file). However, we try reloading by calling Option 6, we get a message that the customer could not be loaded, as shown in Figure 5.12.



```
Enter the text file name: testcust.txt

Type selection (? for menu,x to exit): 5
Enter the binary file name: testcust.bin

Customer name: Jane Doe
Customer ID: 105
Credit Line: 2500
Last visit (MM/DD/YYYY): 12/26/03

Type selection (? for menu,x to exit): 6
Enter the binary file name: testcust.bin
Customer could not be loaded!

Type selection (? for menu,x to exit):
```

Figure 5.12: Console display of failure to load binary function

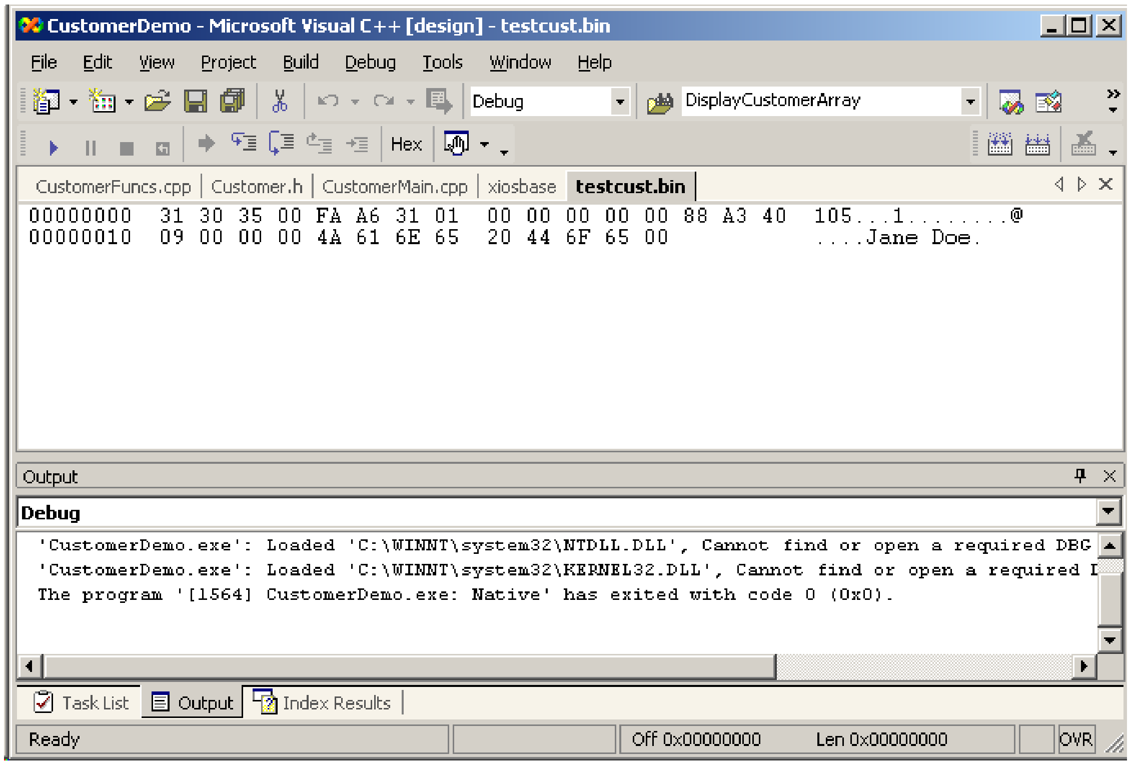
While the program is still running, we can then put a breakpoint at the top of the LoadCustomer() function and try Option 6 again. When we do so, we discover that the function fails in the initial ReadBlock() line, as shown in Figure 5.13. At this point it makes sense to try to look at the file we created to see if there's any data in it. Unfortunately, Visual Studio .NET won't let us open the file (testcust.bin in the example). At this point, it is time to stop the debugger and restart the program.


CustomerDemo - Microsoft Visual C++ [break] - CustomerFuncs.cpp
File Edit View Project Build Debug Tools Window Help
Debug
DisplayCustomerArray
Hex
CustomerFuncs.cpp Customer.h CustomerMain.cpp xiosbase
(Globals)
LoadCustomer
Ready
Ln 242 Col 1 Ch 1
INS

```
    reading the name string. It returns 1 if successful, 0 otherwise. */

bool LoadCustomer(STREAM &binfile,struct Customer *pCust)
{
    if (!ReadBlock(binfile,pCust->szCustID,sizeof(pCust->szCustID))) return fa
    pCust->nLastVisit=ReadInteger(binfile);
    pCust->dCreditLine=ReadReal(binfile);
    char szCustname[256];
    memset(pCust->szCustID,0,256);
    if (!LoadString(binfile,szCustname)) return false;
    pCust->szName=new char[strlen(szCustname)+1];
    strcpy(pCust->szName,szCustname);
    return true;
}

/* DisplayCustomerArray takes an array of Customer objects containing nCount
```

Figure 5.13: LoadCustomer() function returning after ReadBlock() fails

Interestingly enough, once we end the program, we find that we are able to open testcust.bin in Visual Stidio .NET. Surprisingly, at first glance, the file looks okay. The first 4 bytes are definitely the ID that was entered (105) and the end of the file contains the integer 9 followed by the 9 characters containing the name "Jane Doe" plus NUL terminator. This suggests that the file itself is not necessarily the problem.


CustomerDemo - Microsoft Visual C++ [design] - testcust.bin
File Edit View Project Build Debug Tools Window Help
Debug
DisplayCustomerArray
Hex
CustomerFuncs.cpp | Customer.h | CustomerMain.cpp | xiosbase | testcust.bin
Output
Debug
Task List | Output | Index Results
Ready
Off 0x00000000
Len 0x00000000
OVR

```
00000000  31 30 35 00 FA A6 31 01  00 00 00 00 00 88 A3 40  105...1.......@
00000010  09 00 00 00 4A 61 6E 65  20 44 6F 65 00           ....Jane Doe.
```

```
'CustomerDemo.exe': Loaded 'C:\WINNT\system32\NTDLL.DLL', Cannot find or open a required DBG
'CustomerDemo.exe': Loaded 'C:\WINNT\system32\KERNEL32.DLL', Cannot find or open a required I
The program '[1564] CustomerDemo.exe: Native' has exited with code 0 (0x0).
```

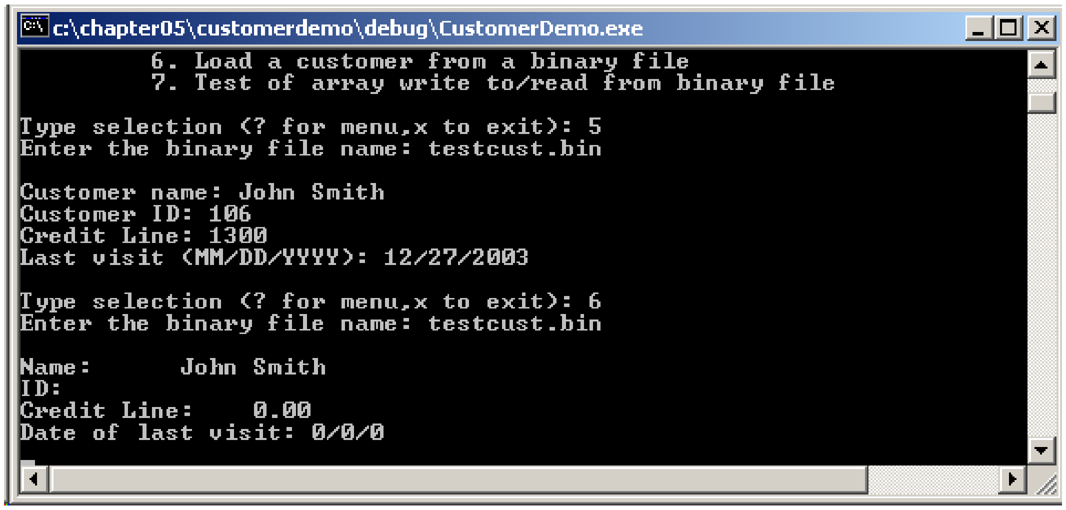
Figure 5.14: Contents of binary file created by SaveCustomer()

Whenever you have read or write failures, or you cannot access a file in another program, there is a likelihood of problems with how the file was opened. Looking at the Option 5 code, shown in Example 5.7, we might eventually notice that we open the strm file, but never close it. That explains two things: 1) why we couldn't examine the file while the program was running and 2) why the read function was having problems with the file (which was opened for writing only). To correct this, we add a Close(strm); statement just above the break.

Example 5.7: Code for Option 5 in main() function

```
case '5':
{
	DisplayFormatted("Enter the binary file name: ");
	InputString(buf1);
	Open(strm,buf1,false,true,true,false);
	if (!strm.is_open()) {
		DisplayFormatted("File could not be opened!\n");
		break;
	}
	val1=InputCustomer(&cust);
	if (val1==0) DisplayFormatted("Customer was illegal\n");
	else {
		if (!SaveCustomer(strm,&cust))
			DisplayFormatted("Customer could not be saved!\n");
		if (cust.szName!=NULL) delete [] cust.szName;
		cust.szName=NULL;
	}
	break;
}
```

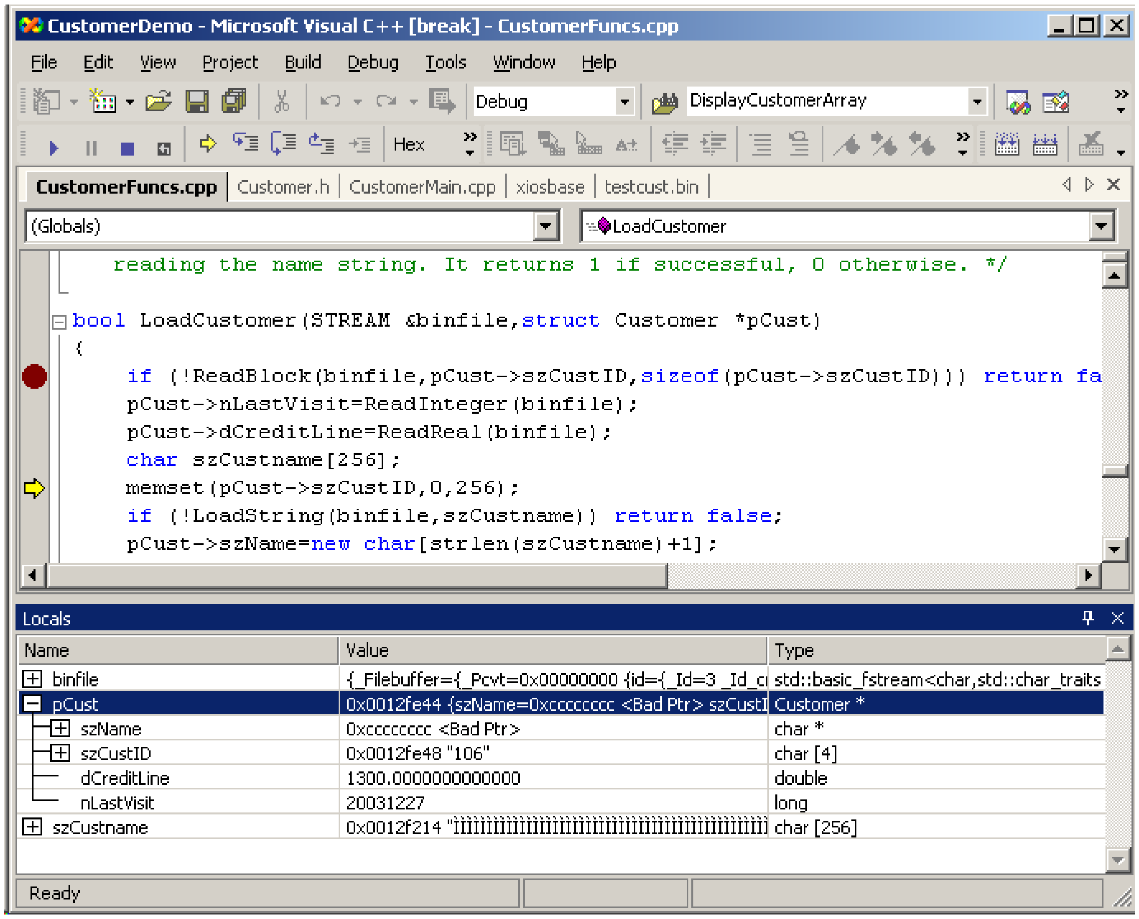
Making this change, we are now able to save a customer and get past the first line of the ReadCustomer() function. Unfortunately, our results are still not perfect, as shown in Figure 5.15. Although the name seems to be loading properly, the remaining data all shows up as zeros.



```
        6. Load a customer from a binary file
        7. Test of array write to/read from binary file

Type selection (? for menu,x to exit): 5
Enter the binary file name: testcust.bin

Customer name: John Smith
Customer ID: 106
Credit Line: 1300
Last visit (MM/DD/YYYY): 12/27/2003

Type selection (? for menu,x to exit): 6
Enter the binary file name: testcust.bin

Name:       John Smith
ID:
Credit Line:    0.00
Date of last visit: 0/0/0
```

Figure 5.15: Console display of loaded customer with corrupted data

In a situation like this, the first thing we should do is to verify the file. In this case, the file looks okay—or, to be more precise, at least it had the proper customer ID value. So the next step is to walk through the LoadCustomer() function with the inspection window open. When we do this, as shown in Figure 5.16, the data loads properly until we reach the memset() step, at which point it is erased. Since we already "fixed" this memset() step once—to remove a compiler error—our suspicions should be raised.


CustomerDemo - Microsoft Visual C++ [break] - CustomerFuncs.cpp
File Edit View Project Build Debug Tools Window Help
Debug
DisplayCustomerArray
Hex
CustomerFuncs.cpp | Customer.h | CustomerMain.cpp | xiosbase | testcust.bin
(Globals)
LoadCustomer

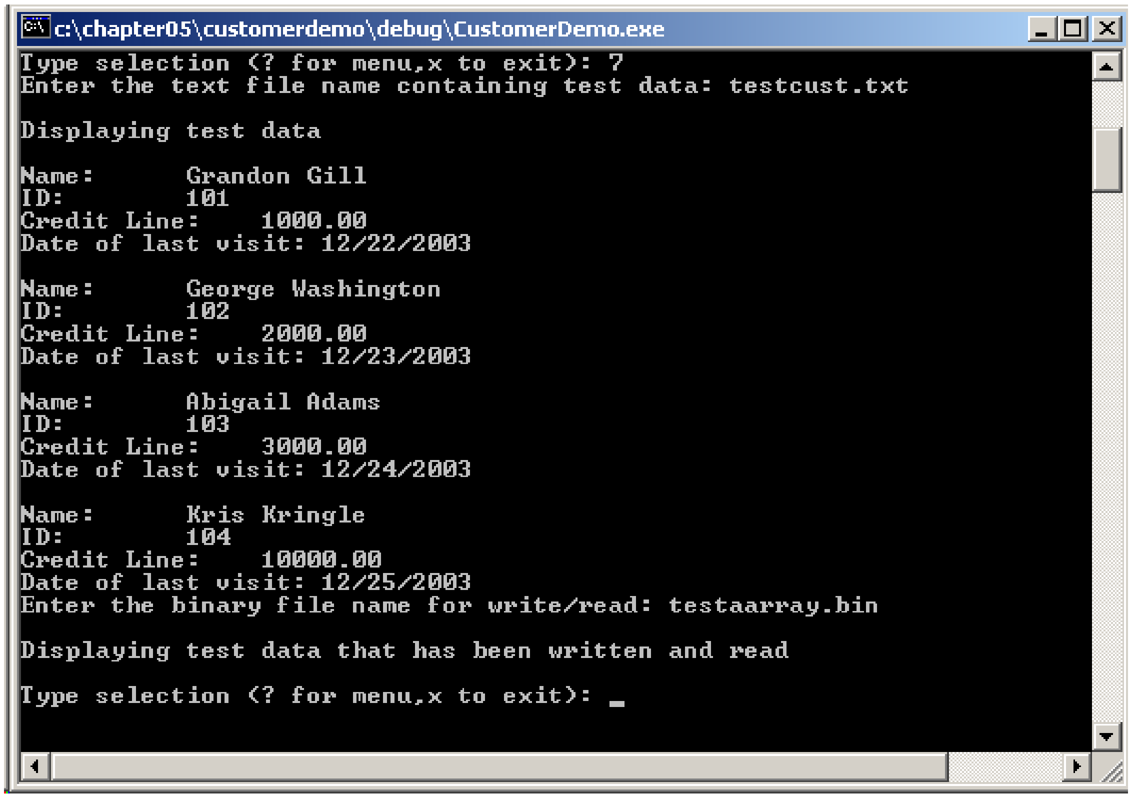
```
    reading the name string. It returns 1 if successful, 0 otherwise. */

bool LoadCustomer(STREAM &binfile,struct Customer *pCust)
{
    if (!ReadBlock(binfile,pCust->szCustID,sizeof(pCust->szCustID))) return fa
    pCust->nLastVisit=ReadInteger(binfile);
    pCust->dCreditLine=ReadReal(binfile);
    char szCustname[256];
    memset(pCust->szCustID,0,256);
    if (!LoadString(binfile,szCustname)) return false;
    pCust->szName=new char[strlen(szCustname)+1];
```

```
Name              Value                                                  Type
+ binfile         {_Filebuffer={_Pcvt=0x00000000 {id={_Id=3 _Id_c       std::basic_fstream<char,std::char_traits
- pCust           0x0012fe44 {szName=0xcccccccc <Bad Ptr> szCustI       Customer *
  + szName        0xcccccccc <Bad Ptr>                                   char *
  + szCustID      0x0012fe48 "106"                                       char [4]
    dCreditLine   1300.0000000000000                                     double
    nLastVisit    20031227                                               long
+ szCustname      0x0012f214 "ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ        char [256]
```

Figure 5.16: Loading of data in LoadCustomer()

Once encountered in the debugger, the problem becomes clear. We are getting ready to load the name in the function, but are initializing the szCustID element instead—an element we have already loaded successfully. To make matters worse, the ID buffer is only 4 bytes long, and we're initializing 256 bytes (wiping out everything else in the structure, and not doing the rest of memory much good either!). What we meant to do was to initialize the 256 byte szCustname array declared just above the memset statement. Replacing pCust->szCustID with szCustname in the memset statement, the program load works perfectly.

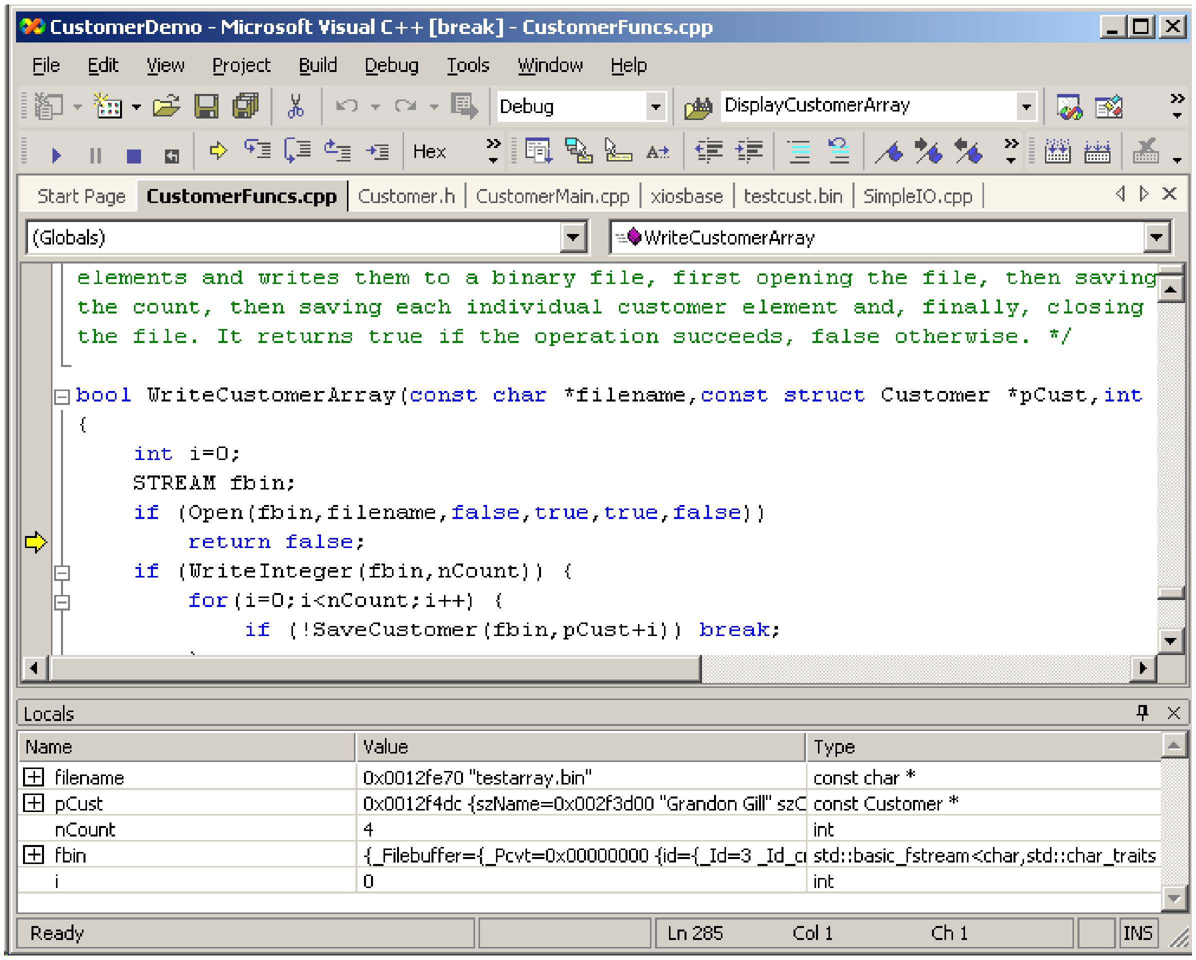
Runtime error: Problems with array display
The final option, Option 7, loads a test file into an array, displays it, then saves the array to a binary file, reloads the array, and displays it. When we try running the routine, however, it doesn't seem to work properly, as shown in Figure 5.17.



```
Type selection (? for menu,x to exit): 7
Enter the text file name containing test data: testcust.txt

Displaying test data

Name:      Grandon Gill
ID:        101
Credit Line:    1000.00
Date of last visit: 12/22/2003

Name:      George Washington
ID:        102
Credit Line:    2000.00
Date of last visit: 12/23/2003

Name:      Abigail Adams
ID:        103
Credit Line:    3000.00
Date of last visit: 12/24/2003

Name:      Kris Kringle
ID:        104
Credit Line:    10000.00
Date of last visit: 12/25/2003
Enter the binary file name for write/read: testaarray.bin

Displaying test data that has been written and read

Type selection (? for menu,x to exit): _
```

Figure 5.17: Display when Option 7 is first tested

The problem here does not appear to be the initial load into an array (which we already tested in Option 4). Thus, it must be somewhere between the binary load and save. To investigate the problem further, we check the binary file that was created. When we do so, we discover the file is empty—indicating a saving problem. Stepping into the WriteCustomerArray() function, we find one problem—we return false as soon as we open the file, as shown in Figure 5.18. Thinking about it for a few moments, the reason becomes clear: we should be returning on ! Open() instead of Open().



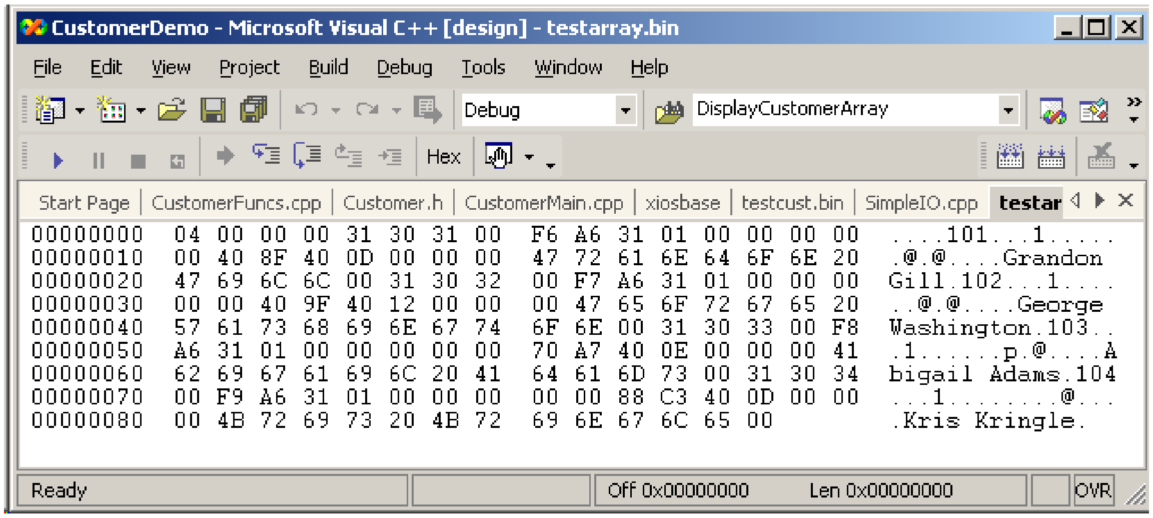
```
elements and writes them to a binary file, first opening the file, then saving
the count, then saving each individual customer element and, finally, closing
the file. It returns true if the operation succeeds, false otherwise. */

bool WriteCustomerArray(const char *filename,const struct Customer *pCust,int
{
    int i=0;
    STREAM fbin;
    if (Open(fbin,filename,false,true,true,false))
        return false;
    if (WriteInteger(fbin,nCount)) {
        for(i=0;i<nCount;i++) {
            if (!SaveCustomer(fbin,pCust+i)) break;
```

Locals

Name	Value	Type
filename	0x0012fe70 "testarray.bin"	const char *
pCust	0x0012f4dc {szName=0x002f3d00 "Grandon Gill" szC	const Customer *
nCount	4	int
fbin	{_Filebuffer={_Pcvt=0x00000000 {id={_Id=3 _Id_ci	std::basic_fstream<char,std::char_traits
i	0	int

Figure 5.18: Failing on wrong Open condition

After fixing the test condition in WriteCustomerArray() we rerun the program—and it still fails to load. Unfortunately, it is not unusual to have multiple errors leading to a failure. In this case, however, there is one promising indication. When we open the file that we created, it is no longer empty. In fact, as shown in Figure 5.19, it looks pretty good. It begins with an integer 4 (the number of customers) and, looking at the ASCII representation on the side, it looks like the customer data is coming in properly as well.


CustomerDemo - Microsoft Visual C++ [design] - testarray.bin
File Edit View Project Build Debug Tools Window Help
Debug
DisplayCustomerArray
Hex
Start Page | CustomerFuncs.cpp | Customer.h | CustomerMain.cpp | xiosbase | testcust.bin | SimpleIO.cpp | testar
Ready
Off 0x00000000 Len 0x00000000
OVR

```
00000000  04 00 00 00 31 30 31 00  F6 A6 31 01 00 00 00 00  ....101...1.....
00000010  00 40 8F 40 0D 00 00 00  47 72 61 6E 64 6F 6E 20  .@.@....Grandon 
00000020  47 69 6C 6C 00 31 30 32  00 F7 A6 31 01 00 00 00  Gill.102...1....
00000030  00 00 40 9F 40 12 00 00  00 47 65 6F 72 67 65 20  ..@.@....George 
00000040  57 61 73 68 69 6E 67 74  6F 6E 00 31 30 33 00 F8  Washington.103..
00000050  A6 31 01 00 00 00 00 00  70 A7 40 0E 00 00 00 41  .1......p.@....A
00000060  62 69 67 61 69 6C 20 41  64 61 6D 73 00 31 30 34  bigail Adams.104
00000070  00 F9 A6 31 01 00 00 00  00 00 88 C3 40 0D 00 00  ...1........@...
00000080  00 4B 72 69 73 20 4B 72  69 6E 67 6C 65 00        .Kris Kringle.
```

Figure 5.19: Contents of binary file created by WriteCustomerArray()

The problem seems to be with the ReadCustomerArray() function. As you may recall, that was the function that was misnamed when we were dealing with compiler problems, as shown in Example 5.5. Furthermore, there was a problem with the WriteInteger() function having only one argument. Which leads to a question of logic: why would we be writing an integer in a program that's supposed to be reading a file. Reexamining the logic, it becomes clear that what we should be doing is reading an integer, not writing it. Which means we need to replace WriteInteger(fbin,nCount) with:

```
nCount=ReadInteger(fbin);
```

Upon making this change, and recompiling, our display works perfectly. Our application has been debugged!

5.5.3: In Depth: Introduction to Automated Testing

By the time you've manually entered test data into the Customer application a few thousand times, you've developed an appreciation for why testing is not always a programmer's favorite activity. The problem is even worse than it first appears, however. Since one of the most common ways in which bugs are introduced into a program is fixing other bugs, you should really completely retest your application each time you modify it.

One way to reduce the burden imposed by testing is to automate as much of the process as possible. While there is a lot of pricey automated testing software in use, it is possible for us to get part of the way there just using the tools at hand.

The particular tool we will present here is *redirection*, the process of changing standard I/O so it goes to and from a file, instead of the console. To accomplish such redirection, we need to know a bit about the command line—which is the line the user types in (or, in the case of Windows, is incorporated into the shortcut) that tells the operating system to load and run the program.

In addition to the name of a program you want to run, additional information is often placed on the command line, following the program name. For example, if a document name is placed on the command line after the program name, many programs will load that document automatically when they start running. There are two symbols that have special meaning on the command line, however:

> \> Sends program standard output to the file name that follows, not the screen
>
> < Receives standard input by reading lines from a file, instead of the keyboard

For example, suppose we were to run a program as follows:

```
CustomerDemo < TestData.txt > TestOut.txt
```

Instead of reading input from the keyboard, the program would read a line from TestData.txt each time it needed input. Similarly, instead of sending text output to the screen, it would get written to TestOut.txt.

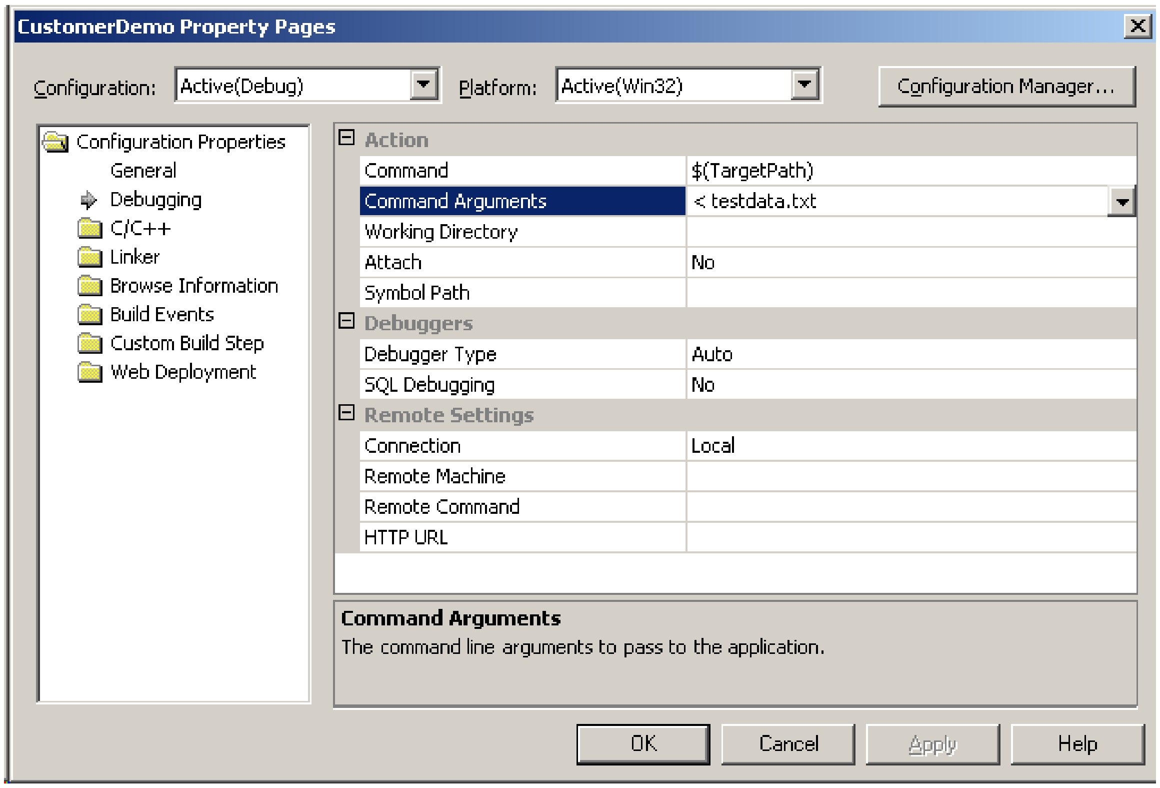
Suppose, then, we wanted to automate our tests of options 1, 2 and 3. We could use redirection in the following manner:

- We would prepare a text file so it contains *exactly* what we would type if we were doing the testing manually. Example 5.8 is an illustration of such a file, named TestData.txt.
- We would modify the debug command line arguments in Visual Studio .Net so that input was received from TestData.txt. This is done under Project Properties, as shown in Figure 5.20.

Example 5.8: TestData.txt file contents

```
1
Grandon Gill
101
1000
12/23/03
2
testauto.txt
Grandon Gill
101
1000
12/23/03
2
testauto.txt
Martha Washington
102
2000
12/24/03
2
testauto.txt
John Adams
103
3000
12/25/2003
3
testauto.txt
x
```

Test your understanding: What is the purpose of the line containing X at the bottom of TestData.txt?



Action	
Command	$(TargetPath)
Command Arguments	< testdata.txt
Working Directory	
Attach	No
Symbol Path	
Debuggers	
Debugger Type	Auto
SQL Debugging	No
Remote Settings	
Connection	Local
Remote Machine	
Remote Command	
HTTP URL	

Command Arguments
The command line arguments to pass to the application.

Figure 5.20: Modifying command line arguments

Once you have done this, you can run your program. Instead of prompting you for inputs, it will simply run then exit (only in debug mode, however, since the arguments were only set for debugging in Figure5.20). You could then put a breakpoint at the return in your main() function and examine the output window—noting it appears to done the tests that you asked for (although it looks a bit strange).

Since a long series of tests could exceed the capacity of your console window, the next step is to add an output file to your redirection. You can do this by changing the command line in Figure 5.20 to:

```
<TestData.txt >TestOut.txt
```

After doing this, nothing should appear in the console window when you run your program. Instead, all the output is sent to TestOut.txt. Although the fact that all the inputs are missing (including the carriage returns) makes the report a bit hard to read, it is still pretty easy to find the output of the tests.

5.5 Section Questions

1. What debugging capability tends to be most useful in detecting problems with improperly constructed case constructs?
2. What is often the first indication of an infinite loop when running a program in the debugger?
3. Under what circumstances is stepping through code while monitoring the inspection windows likely to be ineffective in finding defects?
4. What is the difference between the Locals and Auto inspection windows?
5. Under what circumstances is created an automated test script for a console program appropriate?

5.6: Debugging Lab

The debugging lab provides the reader with an opportunity to work on a different version of the Customer project with an entirely different set of embedded bugs.

5.6.1: Instructions

The Chapter05/Lab folder on the text CD contains a set of subfolders, each containing a project with a set of files with names identical to those used in this chapter. The bugs in these files, however, are entirely different from those presented in this chapter. In doing each lab, you should:

- Identify and fix all compiler errors
- Identify and fix all runtime errors
- Go through each function and identify and fix all runtime errors
- Create a test data file (such as that described in 5.5.3) to test all functions that you modified.

5.6.2: Documentation

In the event a lab is to be graded, you should create a table in your word processor or spreadsheet that holds the following information:

Source File	Original Line #	Description of problem	Description of fix

In doing this, make sure:

- you use the original line number (which you can get from the CD files, or from an unaltered copy that you keep on disk).
- You order the results by file, and by line number within each file.

This will facilitate correlating your results with an answer key.

5.7: Review and Questions

5.7.1: Review

Testing can often consume 50% or more of the development costs of a complex application. Debugging is a form of developer testing, one of many testing types that include:

- Design walkthroughs
- Developer testing
- Automated testing
- End-user testing

In debugging, the role of the programmer is to identify and repair defects in the code under development.

In getting an application to run, there are typically three categories of errors that need to be eliminated:

- *Compiler errors:* Errors in program constructs, syntax or usage (e.g., trying to evaluate variables or functions that have not be defined). These tend to be the easiest errors to find, because the compiler provides error messages identifying the line number where the error was detected. Compiler errors should always be fixed in order or appearance in each file, as many error types confuse the compiler sufficiently that error messages below the triggering error are essentially meaningless.
- *Linker errors:* Errors signifying that variables or functions have been declared, but never defined, or have been defined more than once. Linker errors are usually the easiest to correct, since they normally involve either writing a function you forgot to put in, or removing something that's been defined twice.
- *Runtime errors:* Errors that can only be detected through inspections of code or by running the program. There are by far the most difficult errors to deal with, since they can come in any shape or form, so there is no "guaranteed" method of detecting them.

In order to find and repair runtime errors, four IDE capabilities are normally employed:

- *Stepping though code line by line.* This approach is particularly useful for identifying structural problems in code, such as incorrect tests in branches, failure to iterate properly in loops and missing break statements in case constructs.
- *Setting breakpoints.* A breakpoint pauses the code while running under the debugger. It is particularly useful for localizing errors, as a breakpoint can be set

to avoid the tedious process of stepping to an error deeply nested within an application. Breakpoints can also be used to stop code where unexpected results occur (e.g., placing a breakpoint on a line of code in an if-construct that can only be reached if an error is encountered).

- *Inspection of values.* While stepping or paused at a breakpoint, variable values within each function active in the program can be examined. This is probably the single most powerful tool for detecting problems in program logic.
- *Examining and navigating the call stack:* The call stack is the list of functions active at a given time in the program. It is most useful when a problem—such as an exception or assertion failure—halts the program while running under the debugger, since it can be used to help the developer figure out the status of the program at the time of the failure. In addition, it can be a convenient tool for navigating around active functions, for the purpose of inspecting variables or setting breakpoints.

The use of each of these tools is illustrated in the chapter.

5.7.2: Glossary

Alpha testing – User testing performed when defects are known and design changes can still occur.
Assert statement – A statement that is designed to abort a given process when a condition or statement is true.
Assertion failure – Occurs when an assertion statement becomes true the statement assertion failure is used.
Automated Testing – Testing of an application through a script or some other method where one program tests the validity of a given component or components to see if the subroutine fits the appropriate standard.
Beta testing – User testing done prior to release when defects are still likely to be encountered but application design is frozen and no further design changes can occur.
Code inspection – The process of explaining your code to another expert developer in order to verify that the code is free of bugs or errors.
Component testing – Testing on a particular component of the overall application to ensure that it works according to specified standards.
Debugging – The process of removing syntax and logic errors out of a program in order for the program to function correctly.
Design testing – Testing done at the design phase of the coding process. This can show significant bottlenecks and serious holes in the overall structure of the program at this phase.
Developer testing – Usually associated with debugging activities the developer testing phase is very similar to that of system debugging processes.
End user testing – Testing where end users who are not known to be participants test a given software application to find subtle flaws and errors missed in other phases of the testing process.

Integration testing – Occurs when all components of the software are tested at the same time in one cohesive program to demonstrate that these components can function together.
sscanf – A function used to scan a line of text input and place the values in variables whose addresses are supplied as arguments. Works as an inverse to the sprintf function.
Syntax error – An error that occurs because of a misspelling, bad punctuation or misuse of the keywords of a computer language.
Test case – A case used to test a routine. Used in the debugging process and to discover when the
Undefined variable – A variable that has not been formally defined according to the complier.
Unrecognized function – A function where the compiler cannot recognize it because it has been declared incorrectly.

5.7.3: Questions

For each of the following defects, identify where in the testing cycle: a) it would ideally be detected , b) is most likely to be detected and c) what outcome is the likely result if it is not detected. Choose from the following list, explaining your answer:

A. Design Testing – Specification walkthroughs
B. Developer Testing – Compiling
C. Developer Testing – Linking
D. Developer Testing – Runtime Testing
E. Developer Testing – Code Inspections
F. Automated Testing – Component Testing
G. Automated Testing – Integration Testing
H. End-User Testing – Alpha Testing
I. End-User Testing – Beta Testing
J. During widespread use of the application

Question 5.1: A programmer has forgotten to write a key function required by the application

Question 5.2: A graphic user-interface crashed whenever a user clicks on the pixel-wide boundary of a button

Question 5.3: A program's financial calculations discount each cash flow by one extra period.

Question 5.4: A programmer changes the declaration of a function in a .c (or .cpp) file but fails to change the header file.

Question 5.5: A program crashed when invalid input to is provided to a particular prompt.

Question 5.6: A drawing application begins drawing wild lines across the screen whenever the picture it is drawing reaches a certain size.

Question 5.7: A programmer calls a function with the wrong types of arguments.

Question 5.8: The longer a program runs, the slower it becomes—along with all the other programs running on the system. Eventually, the computer locks up.

Question 5.9: The placement of a menu on the screen leads users to routinely press a button that clears the screen .

Question 5.10: A missing break statement in a currency conversion function causes Euros and UK Pounds to be converted from dollars using the same factor.

Question 5.11: An application that works perfectly on all your lab workstations will not run when you try to install it on your computer at work.

Question 5.12: An accounting application produces debit and credit totals that are a few cents apart, even though they should be identical.

Question 5.13: A drawing application works when polygons are constructed clockwise, but leads to an exception if they are drawn counterclockwise.

Question 5.14: A program that attempts to identify suitable customers from a complex database starts to get dramatically slower as the database grows in size.

Question 5.15: A programmer mismatches argument specifiers in a printf() statement intended alert the user to unusual characteristics in the data that he or she just entered.

Question 5.16: A function used to compute the *Internal Rate of Return* (IRR) of a series of cash flows doesn't return the expected answer—even though the programmer insists the answer is correct and demonstrates it using a spreadsheet.

Question 5.17: A program that has been translated to English occasionally pops up messages in Hebrew.

Question 5.18: A program that works perfectly under the debugger starts to crash routinely when release version is run.

Question 5.19: A programmer buddy of yours shows you that you can get into a password-protected application (for which you are not authorized) by pressing Ctrl-Alt-0, then waiting 4 seconds and pressing the shift key.

Question 5.20: A formerly reliable program starts to fail routinely after a new network application is placed on the system.

Questions 5.21-5.25: Perform one or more of the five debugging samples discussed in section 5.6. Once all compiler and linker errors have been removed, make copies of the source code (for use in *Question 5.26*).

Question 5.26: Develop a general test script for verifying the runtime routines in the sample projects. Attempt to run it against one of more of the debugging projects in Questions 5.21-5.25 and determine how many errors surface.

Part Two:

Object-Oriented Programming

Chapter 6

Encapsulation

Executive Summary

Chapter 6 examines how we can extend the concept of a structure to create encapsulated objects, also called abstract data types (ADTs). Organizing programs around encapsulated objects, rather than functions, is the primary difference between object-oriented programming and structured programming. The ability to define these objects also represents the most significant extension made to C in the design of C++.

The chapter begins with an overview of object-oriented programming, defining three characteristics of OOP design: encapsulation, inheritance and polymorphism. We then present the mechanisms for creating member functions and data, explaining member visibility (public, protected and private). Object construction and destruction are then discussed. Two commonly used member qualifiers, static and const, are then examined. The Unified Modeling Language (UML) representation of a class, generated using FlowC is then presented, and some principles of encapsulated design are introduced. Finally, a series of examples and lab exercises using an encapsulated version of the employee structure introduced in Chapter 4 are presented.

Learning Objectives

Upon completing this chapter, you should be able to:

- Explain the basic concepts of encapsulation, inheritance and polymorphism.
- Apply simple member functions
- Define inline and regular member functions
- Explain public, protected and private visibility
- Write constructor and destructor functions for C++ classes
- Understand what is meant by a const member, and explain appropriate uses
- Understand what is meant by a static member, and explain appropriate uses
- Draw a UML class diagram for a basic C++ class
- List some elementary principles of object-oriented design
- Create simple applications using encapsulated objects

6.1: Introduction to Objects and Object-Oriented Programming

In this section, we'll introduce the basic OOP concepts of encapsulation, inheritance and polymorphism—focusing on how to apply existing objects, rather than on how to define new ones. This presentation will be organized into three topics:

- Encapsulation
- Inheritance
- Polymorphism

These topics reflect the three main characteristics that distinguish OOP from structured programming. Before discussing them, however, it is useful to understand how OOP evolved.

6.1.1: Evolution of OOP

As we become more and more ambitious in the applications we design and build, the obstacle we ultimately face is always the same: complexity. Complexity limits the size of the applications we can build before we find them impossible to debug. It also limits our ability to share and maintain code. As a consequence, the evolution of programming technique has always reflected changing approaches to overcoming the complexity barrier. Roughly speaking, these techniques have evolved through three stages:

1. *Construct-oriented approaches:* These approaches attempt to reduce complexity by avoiding the use of constructs that make code confusing—such as the jump (goto statement) and multiple exit points (returns, breaks). Code developed using this approach often exhibits an almost linear flow, moving from construct block to construct block, and is relatively easy to read. The construct-oriented approach, often seen in programs written in older languages such as FORTRAN and COBOL, is typified by large functions that closely model the modular structure of an application.
2. *Function-oriented approaches*: Retaining the structured programming practices of the construct-oriented approach, the function-oriented approach also attempts to break modules into collections of small, self-contained, functions—often using data structures to pass complex arguments. This approach, which was summarized in Chapters 2 through 4, offers two major advantages over a pure construct-oriented approach: a) self-contained functions are easier to understand and document, and b) use of self-contained functions tends to promote reuse through the development of libraries. Programmers who are experienced using languages like C and Pascal nearly always adopt a function-oriented style.
3. *Object-oriented approaches*: The object-oriented approach employs all the practices of function-oriented programming, but changes the locus of organization from modules and functions to "objects". These objects incorporate a mixture of data and functions. Properly designed, they can be quite self-contained (helping to

manage complexity), offer the same benefits of reusability provided by libraries, and provide additional benefits in the form of extendibility, using a process called inheritance.

It is best to view these three approaches are evolutionary, not alternatives. The function-oriented approach can't be used effectively unless a construct-oriented approach is in place. Similarly, OOP employs the function-oriented approach—often to extremes—with the main distinction being its use of the "object" as an organizing entity.

The choice of programming approach tends to be dictated by three factors:

- Complexity of the problem being solved
- Tools (e.g., languages) to be used
- Experience of the developers or development team

When problems are simple, any of the three approaches is adequate. For such problems, tools and experience tend to dominate the choice of approach. As problems become more complex, however, the construct-oriented approach (e.g., writing the program in one, well-commented, 5000 line main() function) tends to give way to function-oriented approaches and, ultimately, object-oriented approaches .

We now turn to the some of the characteristics that C++ provides to support the object-oriented approach.

6.1.2: Encapsulation

Encapsulation is a design style whereby data and program elements are combined to form a single, self-contained object. C++ supports encapsulation by starting with a basic C structure (referred to as a class) and adding various capabilities, the most important of which are:

1. The ability to associate functions, as well as data elements, with class (structure) definitions. Collectively, data and function elements are referred to as *class members*. Frequently, we will refer to *data members* and *function members* when we need to distinguish between the two categories of members.
2. The ability to define functions that are automatically invoked when an object is created or destroyed. These functions are referred to as *constructor functions* and *destructor functions*.
3. The ability to specify accessibility of member data and member functions. Normally, accessibility is used to distinguish between the *interface* of an object—the set of features available to a developer who wants to embed that object within an application—and its *implementation*—the internal construction of the object that is normally of interest mainly to the programmer who developed the object.

We will now consider the first of these capabilities, class membership, from the object user's perspective.

The C++ concept of class members is a straightforward extension of the simple structure that we presented in Chapter 3 (indeed, the **struct** keyword in C++ can be used to create a C++ class). The data members of a class are exactly the same as the data elements of a structure. For example, the following declaration:

```
struct emp_pay0 {
    int nEmpId;
    double dGrossPay;
    double dBenefitContributions;
    double dFica;
    double dTax;
    double dNetPay;
};
```

would be a perfectly legal class in C++. The members of that class would be nEmpId, dGrossPay, dBenefitContributions, dFica, etc.—all of which would be data members.

Where C++ extends the simple structure is in its ability to make functions, as well as data, part of a class (or structure) definition. For example, in C++, we might see the structure definition revised as follows:

```
struct emp_pay1 {
    int nEmpId;
    double dGrossPay;
    double dBenefitContributions;
    double dFica;
    double dTax;
    double NetPay();
    void ComputeFica(double dRate);
};
```

In emp_pay1, then, the dNetPay data member in emp_pay0 has been replaced by what looks like (and is) a function prototype, NetPay(). A second function prototype, ComputePay(double dRate), is also embedded in the structure definition. These are not, however, the same type of function definitions and declarations that we have seen up to this point. They are, instead, member functions—meaning that they are meaningful only when applied to a specific object.

The emp_pay1 class would be a legal object declaration in C++, as we shall see shortly. Declared in this way, the process for applying member functions is almost exactly the same as that used for accessing structure data that we have already learned. For example:

```
int main(char argc,char *argv[])
{
    emp_pay1 e1,*pe1;
    e1.dGrossPay=1000;
    e1.dBenefitContribution=50;
    e1.ComputeFica(0.076);
    e1.dTax=0.15*e1.dGrossPay;
    pe1=&e1;
    cout << "The employee's net pay is " << pe1->NetPay() << endl;
    return 0;
}
```

The first thing you should notice in main() is the declaration line. Up to this point in the text, we would have written it as:

```
struct emp_pay1 e1,*pe1; // Also legal
```

Although we could have included the **struct** keyword, it is not necessary in C++ when we are creating objects (but it would be required in a C program). Since adding member functions to our structure makes the structure itself incompatible with C syntax anyway, we will use the more compact C++ notation for the rest of the text.

The next thing that you should notice is the similarity between member function access and structure data access, evident from the use of e1 above. For example, when we wanted to call the member function ComputePay() using the object e1, the same dot notation employed to get at structure data was used:

```
e1.ComputeFica(0.076);
```

Similarly, when we wanted to call the NetPay() function, and we wanted to use an object pointer (pe1), we used the following syntax:

```
pe1->NetPay()
```

The arrow (->) operator was used here because the left hand side was a pointer, just as was done in Chapter 3.

Naturally, when we start defining our own C++ classes, we'll need answers to a number of questions, the most obvious being: where would ComputerFica() and NetPay()actually be defined (since all we have is a prototype within the structure itself)? These types of questions will be addressed later in the chapter. Just knowing how to access member functions that have already been defined, however, is very powerful. It is, for example, sufficient to allow us to use the C++ file I/O system directly (the subject of Chapter 11).

6.1.3: Inheritance

Inheritance is the ability to define a class using members of another class as a starting point. To understand the types of benefits this capability provides, let us refine our example of an employee object. Suppose, we have an employee class defined as follows:

```
struct employee {
    int nEmpId;
    int nSSN;
    char szLName[30];
    char szFName[30];
    char cMI;
    double dGrossPay;
    double dBenefitContributions;
    double dFica;
    double dTax;
    double NetPay();
};
```

Here, we've just added some useful information—such as Name and SSN—to our earlier class. Now suppose, as well, that in addition to "standard" employees, we also have another special types of employee, an executive, who receives a bonus in stock options in addition to gross pay. One way we could accommodate this would be to create an independent structure for executive pay. For example, we could define an executive class as follows:

```
struct executive1 {
    int nEmpId;
    int nSSN;
    char szLName[30];
    char szFName[30];
    char cMI;
    double dGrossPay;
    double dBenefitContributions;
    double dFica;
    double dTax;
    double dOptionShares;
    double dEstValuePerOption;
    double NetPay();
    double OptionValue() ;
};
```

This class is, as should be evident, is quite similar to our earlier employee class. The only change is that we've added members to take care of the bonus information (shown in bold).

Alternatively, we could use composition to define our executive object. For example:

```
struct executive2 {
     struct employee emp;
     double dOptionShares;
     double dEstValuePerOption;
     double OptionValue() ;
};
```

While creating a brand new executive class using the executive1 approach would work, there are two significant disadvantages to defining the new class in this manner.

- If you ever wanted to keep a collection of all your employees, you'd need to maintain separate collections (e.g., lists or arrays) for your standard employees and your executives. Similarly, if you developed a function that took an employee as an argument, you'd need to create a separate version of the function to handle executives. The problem here is that while we know that an executive is a type of employee, by creating a separate class we hide that fact from our computer.
- If you ever made a change to your employee class (e.g., adding a health insurance ID number), you would need to make the same change to the executive class if you wanted to maintain consistency between them.

Similarly, the composition (executive2) suffers from the first of these disadvantages, and also a second disadvantage—the awkward syntax required to get at employee data within an executive2 object, e.g.,

```
executive2 e2;
char szLast=e2.emp.szLName;
```

Inheritance provides an elegant solution that is an improvement over both the executive1 and executive2 approaches. Using inheritance, we would define our executive class by specifying that an executive inherits from employee. This is done as follows:

```
struct executive :  public employee
{
     double dOptionShares;
     double dEstValuePerOption;
     double OptionValue();
};
```

The **: public employee** in the definition states that whenever we create an executive object, we automatically include all the members that are associated with an employee. As a result, the only members that we need to include with the definition are those that we are adding, or those that we are changing. (In our example, these are the same members that we bolded when we defined executive from scratch).

Test your understanding: If we defined executive as above, would the following code produce a compiler error:

```
executive e1;
e1.nSSN=0;
```

Why or why not?

The relationship we establish using inheritance is sometimes called an "is a" or "is-a" relationship, so named because an executive "is a" employee. Such a relationship solves the problems of defining executive independent of employee because:

- A collection containing pointers or references to employees can also include executives, because executives are a kind of employee.
- Functions that take employee pointers or references as arguments can also take executives, for the same reason.
- If we were to enhance our employee class, when we recompile our executive class it would automatically be enhanced as well—because the inheritance in the definition causes the revised version of the employee class to be used.

Naturally, our example would not limit us to inheriting executives. We could define other classes, such as salesperson, that inherit from employee but have additional information, such as sales and commission rate. C++ even allows for multiple inheritance—inheriting members from more than one class—that might be used to create classes such as sales_executive (inheriting both from executive and salesperson).

Naturally, there are many subtleties associated with inheritance. These are discussed in greater detail in Chapters 9 and 10.

6.1.4: Polymorphism

Polymorphism is the ability to define member functions that are called in a way that is context sensitive. In a Microsoft Foundation Classes (MFC) Windows program, for example, there may be dozens of functions named OnMouseMove() that define how various windows—all of which inherit from a common class called CWnd—behave when the mouse moves over them. Polymorphism allows the specific version of the function being called to vary according to what type of window the mouse is passing over (which could be a button, menu, dialog box, etc.).

To understand how polymorphism might be used, suppose that we wanted to create a member function for our employee objects (including inherited classes, such as executive) that returned total compensation. We could do this within the employee class relatively easily, e.g.,

```
        double TotalCompensation(); // would be defined to return the value of
dGrossPay
```

For executive objects, on the other hand, the function might be slightly more complex, because we'd need to include option value in addition to pay, e.g.,

```
        double TotalCompensation(); // would need to return dGrossPay+OptionValue();
```

Suppose, as well, that we have an array of pointers to employee objects (including pointers to executives and any other types of employees we might have defined) called arEmps containing nCount elements, and want to determine the total compensation for the whole company. A function to perform this computation might look as follows:

```
        double TotalCompanyCompensation(struct employee **arEmps,int nCount)
        {
                double dTotal=0.0;
                int i;
                for(i=0;i<nCount;i++)  dTotal=dTotal+ar[i]->TotalCompensation();
                return dTotal;
        }
```

Without polymorphism, this function would call the simple version of the TotalCompensation() function for every employee (regardless of whether or not it was an executive type of employee). Where polymorphism is present, on the other hand, each object would call its own version of the function (e.g., employee objects would call the simple version, executive objects would call the version that includes stock option value).

Polymorphism is present only when inheritance is used to define objects. For this reason, we will consider it, in greater depth, in Chapter 10.

6.1 Section Questions

1. Why is the transition from construct-oriented programming to object-oriented programming that took place between the early-1960s and mid-1990s best considered "evolutionary", rather than "revolutionary"?

2. What are some advantages that an encapsulated object provides over the use of separate functions and structures in terms of managing program complexity?

3. What are some advantages of inheritance in managing program complexity?

4. Under what circumstances is polymorphism likely to be particularly useful?

6.2: Defining Member Functions

In this section, we'll introduce the key capability required to implement encapsulation: the ability to define member functions. We begin by identifying similarities between member functions and the global functions that have been the focus of Chapters 2 through 4. We then look at some important differences between member functions and global functions: establishing member access, using the **this** pointer and overloading member functions. Our main focus in this section will be on describing the mechanics of implementing member functions. Towards the end of the chapter (Section 6.5) we will step back and introduce some general principles of object design.

6.2.1: Defining Member Functions

Defining member functions has both similarities and differences to the global C++ function definitions we have already seen. Initially, we will concentrate on the many similarities. For example—like the global C++ functions we have already seen—member functions:

- Have the standard *return-value name(arguments)* format we have already seen (with two important exceptions)
- Can have default arguments
- Can be overloaded
- Can be defined as either regular or inline functions

The last of these similarities warrants the most discussion, as we have not previously introduced talked much about inline functions. Whenever we define a C++ function, we have the choice of making it a regular or inline function, using the **inline** keyword. The difference determines how the function is stored and called, and its practical implications to the programmer relate mainly to efficiency.

When a regular function is compiled (as we have discussed in earlier chapters, starting with Chapter 2), the code for that function is placed in a single location. When a function call is encountered in your code, the compiler then generates the machine language instructions to call that function (pushing arguments on the stack and jumping to the function). The main advantage of this is that only one copy of each function needs to be stored. The main disadvantage is the overhead associated with making the call.

If an **inline** keyword is used when defining a function, this no longer applies. Instead, a copy of the entire function is made each time a call is encountered in your program. The resulting code can be slightly more efficient, but may also be quite a bit larger (if there are a lot of calls to the particular inline function). The way inline functions are implemented also has an important practical implication: since the definition of the function needs to be available to every source file that calls it, inline functions are normally defined in a .h (header file), rather than in a .cpp file.

Defining Inline Member Functions
Because this book focuses on clarity, rather than efficiency, there has been no need to introduce inline functions up to this point. In defining member functions, however, inline functions are often used. In fact, any function whose definition appears in the class/structure definition itself is—by default—going to be implemented as an inline function. For example, in the emp_pay2 structure below, the NetPay() member function definition has been implemented as an inline member:

```
struct emp_pay2 {
    int nEmpId;
    double dGrossPay;
    double dBenefitContributions;
    double dFica;
    double dTax;
    double NetPay() {return dGrossPay-dBenefitContributions-dFica-dTax;}
    void ComputeFica(double dRate);
};
```

This structure differs from the previously defined emp_pay1 (Section 6.1.2) in that the line prototyping the NetPay() function, i.e.,

```
double NetPay();
```

has been replaced by a full definition, i.e.,

```
double NetPay() {return dGrossPay-dBenefitContributions-dFica-dTax;}
```

By looking at the function, it is obvious that the variables used in the example have the same names as some of the emp_pay2 structure member variables. What may be less obvious is where they get their values from. To understand this, recall that NetPay(), as a member function, will always operate on some emp_pay2 object—it cannot be called by itself. In other words, within a global function such as main, a call such as:

```
emp_pay2 e2;
// some initialization
double dVal=e2.NetPay();
```

would be legal, whereas:

```
emp_pay2 e2a;
// some initialization
double dVal=NetPay();
```

would *not* be legal (just as writing dGrossPay by itself would not be legal, whereas e2a.dGrossPay would be fine).

The implications of this are that when we call a member function, the variables used inside the function can take their values from the object it is being applied to. Thus, the call:

```
double dVal=e2.NetPay();
```

would have the same effect as writing:

```
double dVal=e2.dGrossPay-e2.dBenefitContributions-e2.dFica-e2.dTax
```

given they way we have defined NetPay().

Just as we can reference member data with a member function, we can also reference other member functions. To see this, consider the emp_pay3 structure defined as follows:

```
struct emp_pay3 {
    int nEmpId;
    double dGrossPay;
    double dBenefitContributions;
    double dFica;
    double dTax;
    double dAutoPayments;
    double NetPay() {return dGrossPay-dBenefitContributions-dFica-dTax;}
    double PayCheckAmount(){return NetPay()-dAutoPayments;}
    void ComputeFica(double dRate);
};
```

In this structure, the NetPay() function is called from within the PayCheckAmount() function (from which automatic payments, such as insurance or mortgage deductions, have been subtracted). Applying the function in an example:

```
emp_pay3 e3;
// Some initialization
double dCheck=e3.PayCheckAmount();
// Accomplishes the same thing as…
dCheck=e3.NetPay()-e3.dAutoPayments;
// Accomplishes the same thing as…
dCheck= e3.dGrossPay-e3.dBenefitContributions-e3.dFica-e3.dTax –
                        e3.dAutoPayments;
```

Defining Regular Member Functions

A regular member function is defined in a .cpp file, the same as the global functions we have already seen. The main difference in how the definitions appear is in the use of the scope resolution operator to identify what class a function is associated with. For example, the ComputeFica() member might be defined as follows in a .cpp file:

```
void emp_pay3::ComputeFica(double dRate) {
    dFica=dGrossPay*dRate;
}
```

In this case, the *emp_pay3::* tells the compiler that we are defining a member function of the emp_pay3 structure. Having done so, it knows that the dFica and dGrossPay referenced within the function are member variables (and does not give us undefined variable errors).

Choosing Between Inline and Regular Member Functions
It usually does not make a great deal of difference to the compiler whether or not a member function defined to be inline or regular. In fact, if an inline function is too large, some compilers will implement the function as a regular function (regardless of how the programmer coded it).

There are three potential advantages to making all functions "regular" in nature:

- Greater consistency is achieved—you never have to wonder which file your implementation is in
- Your code is more pure if you don't mix function declarations and definitions
- When selling your code to others as a compiled library and header file, the less code you include in the header, the harder it is for the purchaser to reverse engineer your program

If performance suffers, you can also redefine the function by using the inline keyword and placing it in the header file outside the class definition. For example, if we redefined ComputeFica() as follows:

```
inline void emp_pay3::ComputeFica(double dRate) {
    dFica=dGrossPay*dRate;
}
```

and placed it under our emp_pay3 structure definition in the header file, we could accomplish the same thing as including the function definition within the structure definition.

There are also some arguments for making short member functions inline. These include:

- Potential performance benefits may be gained
- Many C++ member functions are very short (e.g., one line) and it is often saves time to include them in the class definition—as well as making it easier to see what they are doing
- Other languages, such as Java, incorporate all member function definitions within the overall class definition. Writing C++ classes in this way will often make it easier to convert your code to these other languages.

The bottom line is that the choice between inline and regular function definitions has little practical impact on the programmer. It is mainly one of style.

6.2.2: Access Specifiers

Walkthrough available in MAccess.avi

When defining a class or structure, the C++ programmer implicitly or explicitly specifies the accessibility each member function and data element. By controlling accessibility, class designers can distinguish between the interface of a class and its implementation. Using an automobile as an example:

- the "interface" consists of those elements of the car that the driver (e.g., user) interacts with, such as the steering while, pedals, controls and instrumentation panel.
- the implementation includes elements like the drive train and electrical system—those elements of the car that the typical driver doesn't need to know about and probably shouldn't play with.

Member Access
There are three keywords used to specify member access:

- **public:** Specifies that the member data or member function can be accessed any time an object is available.
- **private:** Specifies that the member data or member function can *only* be accessed within other member functions of the class
- **protected:** Specifies that the member data or member function can *only* be accessed within other member functions of the class (or classes that inherit from the class).

In this text, we'll nearly always use either public or protected access, which are by far the most common forms.

Example 6.1: struct Access demonstration

```
struct Person00 {
	char szFName[40];
	char szLName[40];
	char cMI;
	int nAge;
	int nWeight;
	void FullName(char *szBuf) {
		char szMI[2]={cMI};
		strcpy(szBuf,szFName);
		strcat(szBuf," ");
		if (szMI[0]>' ') {
			strcat(szBuf,szMI);
			strcat(szBuf," ");
		}
		strcat(szBuf,szLName);
	}
};

int P0test()
{
	Person00 p0={"Grandon","Gill",0,48,250};
	char buf[80];
	p0.FullName(buf);
	DisplayFormatted("First: %s, Last: %s, Full: %s\n",
	p0.szFName,p0.szLName,buf);
	return 0;
}
```

The easiest way to illustrate how access works is to compare a **struct** the same definition implemented as a **class**. In order to achieve backward compatibility, the default access to members of a **struct** is **public**. That means the structure and P0Test() function shown in Example 6.1 will run perfectly.

Replacing the **struct** keyword with the more commonly used **class** keyword—which defaults to **private** access—leads to many problems. The errors are listed in Example 6.2.

Example 6.2: Errors compiling Person00 defined as a class

```
Compiling...
PersonMain.cpp
C2552: 'p0' : non-aggregates cannot be initialized with initializer list
        'Person00' : Types with private or protected data members are not aggregate
C2248: 'Person00::FullName' : cannot access private member declared in class 'Person00'
        see declaration of 'Person00::FullName'
        see declaration of 'Person00'
C2248: 'Person00::szFName' : cannot access private member declared in class 'Person00'
        see declaration of 'Person00::szFName'
        see declaration of 'Person00'
C2248: 'Person00::szLName' : cannot access private member declared in class 'Person00'
        see declaration of 'Person00::szLName'
        see declaration of 'Person00'
```

These errors identify the following problems, all of which are a result of the fact that P0test() is not a member of the Person00 class, and therefore can only access members declared to be **public**:

1) `Person00 p0={"Grandon","Gill",0,48,250};`
You are not allowed to initialize data for which access is private

2) `p0.FullName(buf);`
You are not allowed to call a member function that is declared to be private

3)
```
DisplayFormatted("First: %s, Last: %s, Full: %s\n",
    p0.szFName,p0.szLName,buf);
```
You are not allowed to access data members that are declared to be private

We can eliminate these problems by inserting a single **public:** access specifier, as shown in Example 6.3. This also has the effect of rendering our class declaration *identical* to the structure definition in Example 6.1.

In C++, the only difference between a ***class*** *object and a* ***struct*** *object is the default access.*

For this reason, in this text we'll use **class** when declaring C++ objects with member functions, and reserve the use of **struct** for objects that only include data (such as the structures of Chapter 3).

Example 6.3: Person00 class with public access

```
class Person00 {
public: // changes access of all declarations that follow
      char szFName[40];
      char szLName[40];
      char cMI;
      int nAge;
      int nWeight;
      void FullName(char *szBuf) {
            char szMI[2]={cMI};
            strcpy(szBuf,szFName);
            strcat(szBuf," ");
            if (szMI[0]>' ') {
                  strcat(szBuf,szMI);
                  strcat(szBuf," ");
            }
            strcat(szBuf,szLName);
      }
};
```

Given that private/protected data access can lead to lots of errors—all of which go away when public access (or a **struct**) is used—it is reasonable to ask "why not just make everything public?" To understand why this is a bad idea, you must go back to why OOP evolved in the first place: to handle problems too complex for other techniques. A key challenge of managing complexity is uncertainty regarding where to direct your attention. If you are a programmer, using a class library designed by some other programmer, one of your first questions will always be:

> Which of the hundreds (or thousands) of available members should I be accessing in my program?

The public/protected declaration provides a useful starting point. Any member (function or data) declared to be public is intended for use by programmers who are using the class. Protected and private members, on the other hand, should not be accessed directly. By so declaring them, the class designer effectively stuck a "No user-serviceable parts inside" sticker on the member. While the specification (like the sticker) will do nothing to prevent a determined programmer from accessing the data or function, the wise programmer will take heed.

Based on this discussion, it seems like a bad idea to make our entire Person00 class public. So how do we get at the data we need (e.g., to print the name)? One approach is to define public member functions specifically designed to allow us to access whatever data we need. Not surprisingly, functions of this type are often referred to as *accessor functions*.

In our example, we had three problems:

- We could not initialize our structure
- We could not call the FullName() member, because it was private
- We could not get at the last name and first name data members in order to print them

In Example 6.4, we rewrite the class Person00 class as Person01 with some strategic functions defined.

Example 6.4: Person class with accessor functions defined

```
class Person01 {
protected: // changes access of all declarations that follow
      char szFName[40];
      char szLName[40];
      char cMI;
      int nAge;
      int nWeight;
public:
      void FullName(char *szBuf) {
            char szMI[2]={cMI};
            strcpy(szBuf,szFName);
            strcat(szBuf," ");
            if (szMI[0]>' ') {
                  strcat(szBuf,szMI);
                  strcat(szBuf," ");
            }
            strcat(szBuf,szLName);
      }
      void Initialize(const char *szF,const char *szL,
                  char cM=0,int nA=-1,int nW=-1) {
            strncpy(szFName,szF,40);
            strncpy(szLName,szL,40);
            cMI=cM;
            nAge=nA;
            nWeight=nW;
      }
      const char *First(){return szFName;}
      const char *Last(){return szLName;}
      char MI(){return cMI;}
      int Age(){return nAge;}
      int Weight(){return nWeight;}
};

int P1test()
{
      Person01 p1;
      p1.Initialize("Grandon","Gill",0,48,250);
      char buf[80];
      p1.FullName(buf);
      DisplayFormatted("First: %s, Last: %s, Full: %s\n",
            p1.First(),p1.Last(),buf);
      return 0;
}
```

To solve these problems, we've taken the following steps:

- Implemented a function Initialize() that allows us to set the values in our structure. The function includes default arguments, so that you only need to know first name and last name to call it.
- Set access to FullName() as **public**, since there's really no reason it needs to be private.

- Created a series of accessor functions that allow us to get the values of the different data elements—the First(), Last(), MI(), Age() and Weight() functions.
- In our new test function, P1test(), we only call the public functions.

If you look at the Person01 class, the most "dangerous" aspect of the class is probably the name arrays (szFName[] and szLName[]) , which could be overflowed by a programmer who didn't know they were limited to 40 characters. To protect our class, we do two things:

- In the Initialize() function, we use strncpy() to limit the characters copied to the available space (40 characters each).
- In the First() and Last() functions, we return the addresses of the arrays as constant character pointers (const char *) making it harder for the class user/programmer to write data to them.

Naturally, the cost of improving class safety is the time it takes to write the various accessor functions. Fortunately, such functions are so easy to write that the time it takes to write them is negligible. In section 6.5 we will further discuss the benefits of limiting member access.

In Depth: friend Specifier
From time to time, we find ourselves in situations where we want to maintain certain data/function members protected as a general rule, but really need access to them within a particular function or class. Such access can be provided using the **friend** keyword.

As an example, suppose we wanted to maintain the fully private nature of our Person00 class, but still wanted to call a function that accessed private members. Declaring that function to be a friend, within the class declaration, would allow us to accomplish this objective. This is illustrated in Example 6.5.

Example 6.5: Use of friend specifier to provide access to private members

```
class Person00r1 {
protected:
	char szFName[40];
	char szLName[40];
	char cMI;
	int nAge;
	int nWeight;
	void FullName(char *szBuf) {
		char szMI[2]={cMI};
		strcpy(szBuf,szFName);
		strcat(szBuf," ");
		if (szMI[0]>' ') {
			strcat(szBuf,szMI);
			strcat(szBuf," ");
		}
		strcat(szBuf,szLName);
	}
	friend int P0r1test();
};

int P0r1test()
{
	Person00r1 p0;
	strncpy(p0.szFName,"Grandon",40);
	strncpy(p0.szLName,"Gill",40);
	p0.cMI=0;
	p0.nAge=48;
	p0.nWeight=250;
	char buf[80];
	p0.FullName(buf);
	DisplayFormatted("First: %s, Last: %s, Full: %s\n",
	p0.szFName,p0.szLName,buf);
	return 0;
}
```

Since all members within the Person00r1 function are protected, the P0r1test function would generate a series of errors except for its declaration as a **friend**. (Even declared as a friend, the compiler would not allow the initialization list used in the original P0Test, presented in Example 6.1).

The friend specifier can also be used to specify access for a class or an individual function within a class. Example 6.6 shows a simple class (TestR2) defined with a single member (the P0r2Test() function). By making the class, (or just the function itself), a friend, access to protected members can be obtained. This is illustrated in Example 6.6.

Example 6.6: Granting friend access to a class or class member

```
class TestR2
{
public:
	int P0r2test();

};

class Person00r2 {
protected:
	char szFName[40];
	char szLName[40];
	char cMI;
	int nAge;
	int nWeight;
	void FullName(char *szBuf) {
		char szMI[2]={cMI};
		strcpy(szBuf,szFName);
		strcat(szBuf," ");
		if (szMI[0]>' ') {
			strcat(szBuf,szMI);
			strcat(szBuf," ");
		}
		strcat(szBuf,szLName);
	}
	// either option alone will work
	friend class TestR2;
	friend int TestR2::P0r2test();
};

// Declaration of class member function given friend access to
Person00r2 class
int TestR2::P0r2test()
{
	Person00r2 p0;
	strncpy(p0.szFName,"Grandon",40);
	strncpy(p0.szLName,"Gill",40);
	p0.cMI=0;
	p0.nAge=48;
	p0.nWeight=250;
	char buf[80];
	p0.FullName(buf);
	DisplayFormatted("First: %s, Last: %s, Full: %s\n",
	p0.szFName,p0.szLName,buf);
	return 0;
}
```

As a general rule, the friend specifier is not usually necessary (it is most commonly encountered in operator overloading, the topic of Chapter 7). Very often, the need for a friend specification can be avoided by defining additional accessor functions.

6.2.3: The *this* pointer

Because member functions operate on objects, and objects exist in memory, it stands to reason that the object to which a member function is being applied has an address. In C++, a variable named **this** containing that address is automatically defined whenever a regular member function is called.

For the most part, the **this** pointer serves relatively little useful purpose. Example 6.7 illustrates the point by showing how the Initialize() member function, originally defined in Example 6.4 could be rewritten as ExampleThis(), which uses the **this** pointer to access each member element.

Example 6.7: Using the *this* pointer

```
void Initialize(const char *szF,const char *szL,
   char cM=0,int nA=-1,int nW=-1) {
      strncpy(szFName,szF,40);
      strncpy(szLName,szL,40);
      cMI=cM;
      nAge=nA;
      nWeight=nW;
}

void InitializeThis(const char *szF,const char *szL,
   char cM=0,int nA=-1,int nW=-1) {
      strncpy(this->szFName,szF,40);
      strncpy(this->szLName,szL,40);
      this->cMI=cM;
      this->nAge=nA;
      this->nWeight=nW;
}
```

Similar to the **friend** specifier, the **this** pointer tends to very useful in some situations, even though it is not used routinely. We will see the most common use of the pointer in Chapter 7, Section 7.2, when we discuss assignment operator overloading, and in a number of operator overloading discussions throughout the book.

6.2.4: Member Function Overloading

Member functions have all the capabilities of normal C++ functions. In particular member functions can be:

- *Overloaded:* Different versions of a member function with the same name can be defined.
- *Assigned Default Argument Values:* Default values for acrguments, from right to left, can be specified (e.g., cM, nA and nW in Initialize(), as shown in Example 6.7)

As was noted in Chapter 2, the compiler must be able to distinguish which version of a function to call using its arguments.

6.2 Section Questions

1. Why is the *class-name::* scope resolution operator required when defining functions in the .cpp file but not within the class?
2. Why must class members be accessed using the . and -> operators outside of member functions, yet those operators are not required to access members within member functions?
3. Would it make sense to declare a large function that is called in many places from within you program as an inline function?
4. Under what circumstances would it matter if access for a particular member data element were declared to be private versus protected?
5. What are some good reasons for not making all class members (data and functions) public?
6. Explain why the need for a friend specification can often be avoided by defining additional accessor functions
7. Explain why this->nAge, (*this).nAge and nAge are all precisely equivalent when used within a Person01 member function
8. Why is any global function called with a structure or structure pointer as an argument a good candidate for a member function if you are moving to OOP?

6.3: Constructor and Destructor Functions

In addition to defining regular member functions, C++ allows us to define functions that are automatically invoked whenever an object is created (*constructor functions*) or destroyed (*destructor functions*). These functions can be very powerful tools, allowing the programmer to automate many allocation and cleanup functions that would otherwise require careful thought in a program.

6.3.1: Constructor Functions

Whenever we create an object from a C++ class, a constructor function is called to initialize the contents of that object. Such constructors can either be created implicitly (by the compiler) or explicitly (by the programmer—which is done by giving a member function the same name as the class itself).

Implicitly Created Constructors
If our class has no constructor functions explicitly defined by the programmer, two types of public constructors are created automatically:

- A *default constructor*, which simply creates the object and does no initialization
- A *copy constructor*, which makes a byte for byte copy of the object being created

Assume, for example, we've defined the following very simple class:

```
class MyClass
{
        int nData;
};
void MySimpleFunc(MyClass obj);
```

Since the programmer has not defined any constructor functions, situations where the default constructor would be invoked include:

```
MyClass obj1;
MyClass *pObj1=new MyClass;
```

The copy constructor, on the other hand, gets called whenever we need to copy an existing object in order to create a new object. Continuing the previous example, since no constructors have been defined, the automatically created copy constructor would be invoked in each of the following situations:

```
MyClass obj2=obj1;   // The contents of obj2 are initialized by copying obj1
MyClass pObj2=new MyClass(obj2); // Initializes the new object to obj2 contents
MySimpleFunc(obj1); // a copy of obj1 is made when passed into the function
```

In our discussions of structures in Chapter 3, we have already taken advantage of the fact that these two constructors are automatically created for us when we create pure structures. The real power of C++ constructors, however, derives from our ability to define our own.

Test your understanding: Would a copy constructor have been invoked if our function MySimpleFunc() had been prototyped in the following ways:

```
void MySimpleFunc(MyClass *pObj);
void MySimpleFunc(Myclass &obj);
```

Why or why not?

Explicitly Defined Constructors
If a programmer decides the implicit constructors do not accomplish what is needed (and it is rare that they do!), it is possible to define one or more constructors explicitly. This is done by prototyping one or more functions having *the same name as the class, and no specified return value*. In fact, constructor (and destructor) functions are the only C++ functions that do not have any return specification (i.e., not even void, to indicate that no value is returned).

To illustrate the explicit definition of constructors, it is useful to consider an example:

```
class emp_pay4 {
public:
    emp_pay4(){ nEmpId=0; }
    emp_pay4(int nID,double dPay=0.0) {
            nEmpId=nID;
            dGrossPay=dPay;
    }
protected:
    int nEmpId;
    double dGrossPay;
    double dBenefitContributions;
    double dFica;
    double dTax;
    double NetPay() {
            return dGrossPay-dBenefitContributions-dFica-dTax;
    }
    void ComputeFica(double dRate);
};
```

In the emp_pay4 structure defined above, there are two constructor functions (using the C++ function overload capability, which applies to constructors as well as other C++ functions) :

```
    emp_pay4(){ nEmpId=0; }
```

and

```
    emp_pay4(int nID,double dPay=0.0) {
            nEmpId=nID;
            dGrossPay=dPay;
    }
```

The first is invoked whenever an object is created with no argument. For example:

```
emp_pay4 e1;
emp_pay4 *pEmp=new emp_pay4;
emp_pay4 *parEmp=new emp_pay4[5];
```

The first line would cause e1 to be created with the first constructor (which would then initialize the nEmpID member of the object e1 to 0). The second line would cause a dynamic object (once again, with the nEmpID member set to 0) to be allocated. The last line would cause an array of emp_pay4 objects to be created, with the first constructor being called 5 times (for each of the 5 objects).

To invoke a constructor defined with arguments, parentheses are used—similar to a function call. For example, in the declarations below:

```
emp_pay4 e2(101,68000.00), e3(102);
emp_pay4 *pNew=new emp_pay4(103,75000);
emp_pay4 arEmp[5]={emp_pay4(104,97000),emp_pay4(105)};
```

both e2 and e3 would be created using the second constructor. In the case of e2, the nEmpId member would be initialized to 101, while the dGrossPay value would be initialized to 68000.00. In the case of e3, the nEmpId of the new object would be 102, while the default argument value of 0.00 would be used to initialize dGrossPay. In the second line, the dynamically allocated object assigned to pNew would have nEmpId initialized to 103 and dGrossPay initialized to 75000.

It is also possible to call the constructors with arguments when initializing arrays with an aggregate list, as shown in the line:

```
emp_pay4 arEmp[5]={emp_pay4(104,97000),emp_pay4(105)};
```

In this case, the second constructor is called on the first two elements of the 5-element array. The first constructor (with no arguments) is then called for the remaining 3 elements.

When the new operator is used to create an array, only a constructor with no arguments can be used. If one is not available, the operation will fail.

As soon as a programmer defines any explicit constructors, the implicit default constructor is no longer defined, although a copy constructor is still created. This can be important since certain operations (such as allocating an array of objects with new, as just noted) require a default constructor. For this reason, a programmer will normally want to define a default constructor (i.e., a constructor with no arguments) whenever *any* constructor is defined

Test your understanding: Explain why the following operation would fail, given our definition of emp_pay4:

```
emp_pay4 ep=e2;
```

6.3.2: Destructor Functions

Unlike constructor functions, only one destructor function can be declared for a given class. It is automatically invoked whenever an object is released. For example:

- When a local variable goes out of scope, such as occurs when a function returns.
- When a program ends—at which time destructor functions are called for all static variables
- When the **delete** operator is applied to objects that were created dynamically using the **new** operator.

Although you virtually never "call" a destructor function, you need to be aware of the types of activities that are performed during object release. In C++ file stream objects, for example, the destructor function is often defined so that it automatically closes the file associated with object. In many destructors, memory managed by the class will be released as part of the destruction process. This will be illustrated in Section 6.3.3.

The implicit destructor function does nothing. To define a destructor function, you use the class name preceded by a tilde. For example:

```
class emp_pay5 {
public:
    emp_pay5(){ nEmpId=0; }
    emp_pay5(int nID,double dPay=0.0) {
            nEmpId=nID;
            dGrossPay=dPay;
    }
    ~emp_pay5(){}
protected:
    int nEmpId;
    double dGrossPay;
    double dBenefitContributions;
    double dFica;
    double dTax;
    double NetPay() {
            return dGrossPay-dBenefitContributions-dFica-dTax;
    }
    void ComputeFica(double dRate);
};
```

In the emp_pay5 class above, the destructor (which doesn't actually do anything) is defined in the line:

```
~emp_pay5(){}
```

Empty destructors are relatively common (in which case they don't need to be overridden, since the implicit version works just fine). If a class manages memory—typically found in classes that contain pointers as members—the destructor will generally contain delete statements to return that managed memory to the heap.

6.3.3: Person Class with Dynamic Names

Walkthrough available in PersonDynamic.avi

To illustrate the use of constructors and destructors, we define a new version of the Person class, Person02. The implementation of Person02 differs from that of Person01 in several ways:

- Instead of storing first and last names in embedded arrays, the class uses pointers to dynamic memory. One advantage of doing so is eliminating the 40-character name limit in Person01.
- Explicit constructors and a destructor have been defined to manage the memory for the names.

The Person02 class is presented in Example 6.8.

Example 6.8: Person class managing dynamic names

```
class Person02 {
public:
      Person02(){
            m_pszFName=0;
            m_pszLName=0;
      }
      Person02(const char *szF,const char *szL,
         char cM=0,int nA=-1,int nW=-1){
            m_pszFName=0;
            m_pszLName=0;
            Initialize(szF,szL,cM,nA,nW);
      }
      ~Person02() {
            delete [] m_pszFName;
            delete [] m_pszLName;
      }
protected: // changes access of all declarations that follow
      char *m_pszFName;
      char *m_pszLName;
      char m_cMI;
      int m_nAge;
      int m_nWeight;
public:
      void FullName(char *szBuf) {
            char szMI[2]={m_cMI};
            strcpy(szBuf,m_pszFName);
            strcat(szBuf," ");
            if (szMI[0]>' ') {
                  strcat(szBuf,szMI);
                  strcat(szBuf," ");
            }
            strcat(szBuf,m_pszLName);
      }
      void Initialize(const char *szF,const char *szL,
                  char cM=0,int nA=-1,int nW=-1) {
            delete m_pszFName;
            m_pszFName=new char[strlen(szF)+1];
            strcpy(m_pszFName,szF);
            delete m_pszLName;
            m_pszLName=new char[strlen(szL)+1];
            strcpy(m_pszLName,szL);
            m_cMI=cM;
            m_nAge=nA;
            m_nWeight=nW;
      }
      const char *First(){return m_pszFName;}
      const char *Last(){return m_pszLName;}
      char MI(){return m_cMI;}
      int Age(){return m_nAge;}
      int Weight(){return m_nWeight;}
};
```

The changes made to the Person02 class include the following:

- Member variables were renamed so that each begins with m_ (e.g., nAge became m_nAge). While this change has no effect on program performance, it is consistent with Microsoft's convention for naming member variables. The benefit of doing so is that it becomes easier to keep track of what variables are member variables (vs. local variables and global variables) within member functions.
- The m_pszFName and m_pszLName members are redefined as pointers.
- Two constructor functions have been defined: a default constructor and a constructor that allows variables to be explicitly initialized.
- A destructor function has been defined that deletes the contents of the m_pszFName and m_pszLName arrays. This eliminates the possibility of memory leaks that would occur if a Person02 object were deleted without taking care of the name strings in memory.
- The Initialize() function was substantially modified.

The heart of the changes made to the Person02 class are in the Initialize() functions. Now that the class uses dynamic memory to hold the name strings, each time a string is changed, the old string is deleted and a copy of the new name is made and assigned to the appropriate pointer. For example:

```
delete m_pszFName;
m_pszFName=new char[strlen(szF)+1];
strcpy(m_pszFName,szF);
```

was used to create a new string m_pszFName string from the function argument szF. While this code would have problems if we ran out of memory (since we don't handle memory exceptions, which are not introduced until in Chapter 12), if the memory is available, we know the m_pszFName string will be long enough to handle the szF argument, so we use strcpy() to make the copy, as opposed to:

```
strncpy(szFName,szF,40);
```

previously used in the Person01 version.

> *Test your understanding:* Why is it critical that m_pszLName and m_pszFName be initialized to 0 before Initialize() is called in the second version of the Person02 constructor?

Despite the implementation differences between Person01 and Person02, the two classes are nearly identical as far as the interface is concerned. In other words, the public members available to a programmer using a Person02 object are essentially the same as those for a Person01 object. The practical implication of this is that anywhere a Person01 object was used, we should be able to change it to a Person02 object without affecting the program. Thus, given that the test code for the Person01 class (from Example 6.4) was as follows:

```
int P1test()
{
	Person01 p1;
	p1.Initialize("Grandon","Gill",0,48,250);
	char buf[80];
	p1.FullName(buf);
	DisplayFormatted("First: %s, Last: %s, Full: %s\n",
		p1.First(),p1.Last(),buf);
	return 0;
}
```

we should be able to replace the first line with:

```
Person02 p1;
```

without producing any errors (compiler or runtime). This, in fact, proves to be the case.

> *Test your understanding:* There is one interface enhancement made in transforming the Person01 to the Person02 class, and one interface capability has been lost. What are they? (Hint: examine all the public members and find one that can be called in Person02, but not Person01. Think about the implicitly defined members and figure which is no longer accessible).

6.3 Section Question

1. When MyClass was defined in Section 6.3.1, the two implicit constructors that were created could have been defined as follows:

```
MyClass(){}
MyClass(const MyClass &ref) { (*this)=ref;}
```

Explain why these two constructors do the same thing as the explicit constructors.

2. Assume the following class has been defined:

```
class Simple {
        public:
                Simple(int nVal){m_nVal=nVal;}
        protected:
                int m_nVal;
}
```

Explain which of the following lines will work and which will fail:

a) Simple s1;
b) Simple s2(5);
c) Simple *p3=new Simple(10);
d) Simple *p4[5]=new Simple[5];
e) Simple s5[]={Simple(15),Simple(20),Simple(30)};
f) Simple s6=s2;

3. Assume the following class is defined:

```
class AClass {
        AClass(){m_cVal=0;}
        char m_cVal;
}
```

What would be the problem with the following declaration:

```
int main(int argc,char argv[])
{
    AClass a;
    return 0;
}
```

How could this problem be solved?

4. What two problems could result if we wrote the following Person02 member function:

```
void Person02::First(const char *szF) {
    m_pszFName=szF;
}
```

5. Rewrite the First() function above to eliminate the problem.

6.4: const and static Qualifiers

There are three qualifiers that are only relevant to member functions: **const**, **static** and **virtual**. The last of these, **virtual**, is relevant only when inheritance is present, and is discussed in Chapter 10. We now turn our attention to the first two.

6.4.1: const Member Functions

Walkthrough available in ConstQual.avi

When declaring a member function, it is possible to append a **const** keyword to the declaration. For example, consider the following class:

```
class SimpleName {
public:
    SimpleName(const char *sz){Name(sz); }
    void Name(const char *sz){strncpy(m_szName,sz,40);}
    const char *Name() const {return m_szName;}
protected:
    char m_szName[40];
};
```

The second version of the Name() function is declared as follows:

```
    const char *Name() const {return m_szName;}
```

In this function, there are two **const** keywords. The first, beginning the declaration, we have already seen many times: it indicates that the return value is a constant character string. The second **const**, therefore, is the one of interest here.

> A member function followed by a const qualifier signifies that calling the member function will not change any data in the object to which the function is being applied.

The compiler takes a const declaration very seriously. Any member function that is so declared:

- Cannot change the values of any member data
- Cannot call any member functions that are not likewise declared to be const.

Furthermore, if an object, object reference or object pointer is declared to be const, only const member functions can be applied to it. For example, the following code would produce an error:

```
SimpleName nm("grandon");
const SimpleName *pN=&nm;
cout << pN->Name() << endl; // OK, displays "grandon" to the screen
pN->Name("Grandon"); // error is generated
```

The problem here is that pN is declared to be const and therefore attempting to call the Name(const char *) function, as is done on the last line, is illegal. Calling the Name() version, as is done on the third line, is fine however. Since the version of Name() with no arguments is defined as const, you can apply it to a const pointer.

If a member function is declared to be const in the header file, its definition in the .cpp must have the same qualifier. For example, if we chose not to define Name() inline, its definition would appear as follows:

```
    const char *SimpleName::Name() const
{
    return m_szName;
}
```

If you leave off the const in the definition header, a compiler error will result. That is because it is possible to define two member functions that differ only in const specification.

The const qualifier is a powerful tool in ensuring the integrity of your code. It puts the compiler to work for you—making sure that you only change your objects when you think you’re changing your objects. Used haphazardly, however, it can create intricate networks of error messages that can take hours (even days) to resolve. To illustrate how such problems can arise, let us refer back to our Person02 class (Example 6.8) and suppose we wanted to write a function to compare two person objects to see if they have the same name. Such a function (modeled after a strcmp() in return value) might be defined as follows:

```
int PersonCmp(const Person02 &p1,const Person02 &p2)
{
    int nCmp=stricmp(p1.Last(),p2.Last());
    if (nCmp!=0) return nCmp;
    return stricmp(p1.First(),p2.First());
}
```

The function is very simple: it first compares the last names and returns the difference unless they are equal, in which case it returns a comparison of the first names. Because PersonCmp() is not a member function or friend, we use the public member functions to get at the first and last names.

When we compile PersonCmp(), we get the errors listed in Example 6.9.

Example 6.9: Errors Compiling PersonCmp()

```
C2662: 'Person02::Last' : cannot convert 'this' pointer from 'const Person02' to 'Person02 &'
        Conversion loses qualifiers
C2662: 'Person02::First' : cannot convert 'this' pointer from 'const Person02' to 'Person02 &'
        Conversion loses qualifiers
```

These errors are pretty typical of const errors. The underlying problem is that the Person02::First() and Person02::Last() member functions are not defined to be const. As a result, we cannot apply them to const objects, such as the two arguments of the function, p1 and p2.

The way the errors are expressed by the compiler makes slightly more sense than:

> C2662: Jeepers, creepers! There's a lead pipe in the conservatory…

But only slightly… As we know from Section 6.2.3, a **this** pointer is created every time a member function is invoked. If a Person02 member has no specifier, the pointer is of type:

```
Person02 *this;
```

If, on the other hand, the member function is const qualified, the pointer is declared as follows:

```
const Person02 *this;
```

Since p1 and p2 are declared as const references in the function header, only the second type of this pointer can be created from them. That leads to an error whenever we try to create a **this** pointer that is not const.

Returning to our problem, a more practical question is "How do we solve it?". There are two possible approaches, the sleazy shortsighted approach and the thoughtful approach. As luck would have it, the first of these is the simplest. We redefine our function as follows:

```
int PersonCmpSleazy(Person02 &p1, Person02 &p2)
{
    int nCmp=stricmp(p1.Last(),p2.Last());
    if (nCmp!=0) return nCmp;
    return stricmp(p1.First(),p2.First());
}
```

By removing the two const qualifiers on the function arguments, we no longer limit ourselves to calling const member functions, and the errors go away. The problem with this approach is that we're making changes to our function that we know shouldn't be necessary—since a comparison function shouldn't be changing its arguments.

A much better approach would be to look at the function and see if the errors make sense given what the function is supposed to do. As it turns out, they don't make sense. That means our problem is really with the Person02 class itself. To correct this problem, we need to make sure that any function members that actually are const are declared that way. That leads to the Person03 class, shown in Example 6.10.

Example 6.10: Person03 Class with const Qualifiers

```
class Person03 {
public:
	Person03(){
		m_pszFName=0;
		m_pszLName=0;
	}
	Person03(const char *szF,const char *szL,
				  char cM=0,int nA=-1,int nW=-1){
		m_pszFName=0;
		m_pszLName=0;
		Initialize(szF,szL,cM,nA,nW);
	}
	~Person03() {
		delete m_pszFName;
		delete m_pszLName;
	}
protected: // changes access of all declarations that follow
	char *m_pszFName;
	char *m_pszLName;
	char m_cMI;
	int m_nAge;
	int m_nWeight;
public:
	void FullName(char *szBuf) const {
		char szMI[2]={m_cMI};
		strcpy(szBuf,m_pszFName);
		strcat(szBuf," ");
		if (szMI[0]>' ') {
			strcat(szBuf,szMI);
			strcat(szBuf," ");
		}
		strcat(szBuf,m_pszLName);
	}
	void Initialize(const char *szF,const char *szL,
						  char cM=0,int nA=-1,int
nW=-1) {
		delete m_pszFName;
		m_pszFName=new char[strlen(szF)+1];
		strcpy(m_pszFName,szF);
		delete m_pszLName;
		m_pszLName=new char[strlen(szL)+1];
		strcpy(m_pszLName,szL);
		m_cMI=cM;
		m_nAge=nA;
		m_nWeight=nW;
	}
	const char *First()const {return m_pszFName;}
	const char *Last()const {return m_pszLName;}
	char MI()const {return m_cMI;}
	int Age()const {return m_nAge;}
	int Weight()const {return m_nWeight;}
};
```

The changes in Person03 are relatively subtle. Six member functions could be const qualified:

```
void FullName(char *szBuf) const;
const char *First() const;
const char *Last() const;
char MI() const;
int Age() const;
int Weight() const;
```

Once we have made these changes, Person03 objects can be passed into the PersonCmp() function and there are no error messages.

> *Test your understanding:* Explain why four of the Person03 member functions could not be const qualified.

6.4.2: The static Qualifier

The term "static" is normally used to refer to data that is initialized and then remains available until a program ends. Global variables, for example, are static.

There are three places where the **static** keyword is commonly encountered in a C++ program. These are:

- In local variable declarations within a function
- In member data declarations within a structure or class
- Within class member function declarations

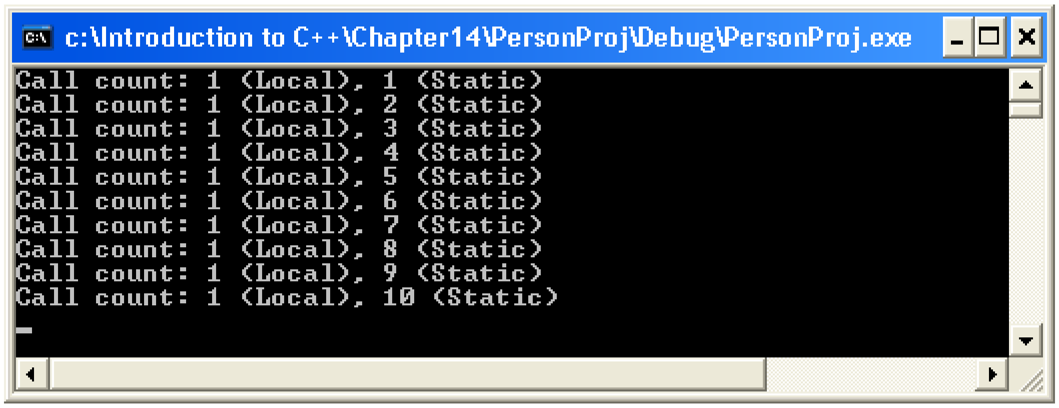
Before looking at how **static** applies to member functions, it is useful to examine its data-related implications.

static Variables in Functions
When a variable is declared static within a function, it is created and initialized the first time the function is called, and maintains its value even after the function returns. In other words, even though the variable goes out of scope when the function ends (i.e., we cannot access it by name), it still continues to exist in memory. To illustrate how this differs from local variables, consider the function defined in Example 6.11.

Example 6.11: SimpleCounter() function

```
void SimpleCounter()
{
	int nLocal=1;
	static int nStatic=1;
	cout << "Call count: " << nLocal << " (Local), "
			<< nStatic << " (Static)" << endl;
	nLocal++;
	nStatic++;
}
```

The first time the SimpleCounter() function is called, nLocal is created and initialized to 1, as is nStatic. Thereafter, each time the function is called (while the program continues running) the process is repeated for nLocal but nStatic retains its previous value. As a consequence, when the function is called 10 times from a loop, the output shown in Figure 6.1 results.


c:\Introduction to C++\Chapter14\PersonProj\Debug\PersonProj.exe

```
Call count: 1 (Local), 1 (Static)
Call count: 1 (Local), 2 (Static)
Call count: 1 (Local), 3 (Static)
Call count: 1 (Local), 4 (Static)
Call count: 1 (Local), 5 (Static)
Call count: 1 (Local), 6 (Static)
Call count: 1 (Local), 7 (Static)
Call count: 1 (Local), 8 (Static)
Call count: 1 (Local), 9 (Static)
Call count: 1 (Local), 10 (Static)
```

Figure 6.1: Output of SimpleCounter() called from a loop 10 times

static Variables in Structures and Classes

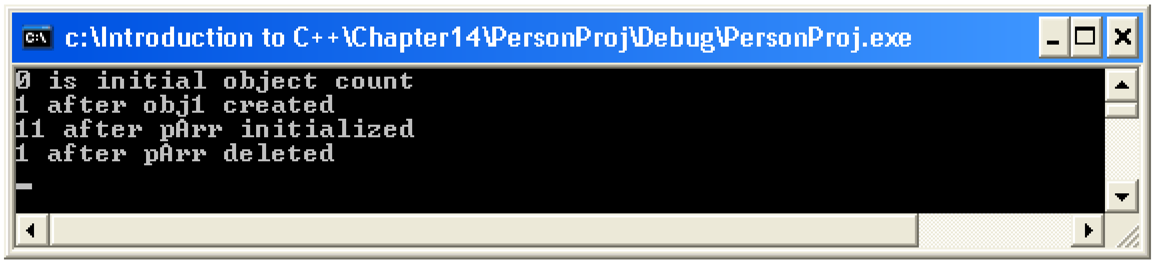
Data members in classes or pure structures declared to be static are somewhat similar in spirit to their function counterparts. They are constructed the first time any object of the class is created, and then remain until the program ends.

What makes static members unique (and occasionally useful) among date members is the fact that all objects share a single copy of the data member. This makes static data members useful for keeping track of global call information, such as the number of active objects in the class. This is illustrated by the ObjCounter class, in Example 6.11, where the m_nCount variable is declared as a public, static member. The output from running the test function, TestCount(), are presented in Figure 6.2.

Example 6.11: ObjCounter header and source files

```
// ObjCounter.h header file
#pragma once

class ObjCounter
{
public:
	ObjCounter(void);
	~ObjCounter(void);
	static int m_nCount;
};

void TestCount();
```

```
// ObjCounter.cpp source file
#include "objcounter.h"
#include <iostream>
using namespace std;

// Static members must be initialized outside the class definition
int ObjCounter::m_nCount=0;

ObjCounter::ObjCounter(void)
{
	m_nCount++;
}

ObjCounter::~ObjCounter(void)
{
	m_nCount--;
}

void TestCount()
{
	cout << ObjCounter::m_nCount << " is initial object count" <<
endl;
	ObjCounter obj1;
	cout << ObjCounter::m_nCount << " after obj1 created" << endl;
	ObjCounter *pArr=new ObjCounter[10];
	cout << ObjCounter::m_nCount << " after pArr initialized" <<
endl;
	delete [] pArr;
	cout << ObjCounter::m_nCount << " after pArr deleted" << endl;
}
```


c:\Introduction to C++\Chapter14\PersonProj\Debug\PersonProj.exe

```
0 is initial object count
1 after obj1 created
11 after pArr initialized
1 after pArr deleted
```

Figure 6.2: Results of running TestCount() function

As demonstrated by the output, when ObjCounter objects are constructed, the constructor function increments the m_nCount static member. Similarly, when objects are deleted, the counter is decremented. This applies to both individual objects and to arrays—where the constructor is called as many times as there are elements in the array (e.g., 10 times in the example).

One important consideration regarding static variables is the need to initialize them outside the class in which they are declared. In this respect, they are like global variables—the structure declaration allows their name and type to be shared but memory must be allocated for them in one, and only one, place. This was done in Example 6.11 in the ObjCounter.cpp file, with the line:

```
int ObjCounter::m_nCount=0;
```

If you do not have an in initialization statement in one (and only one) of your source files, an "unresolved external" linker error will result.

> *Test your understanding:* Given the manner in which static variables are declared, what are the implications for member access?

static Member Functions

A member function qualified as **static** is shared by all the objects in the class, just like a static data member. There are a number of practical implications of this:

- Since a static member function is not associated with any particular object, you cannot access non-static member data or member functions from within it without supplying an object
- The **const** qualifier cannot be applied to a static member function, since it can't—by its nature—make changes to an associated object.
- The **this** pointer cannot be accessed within a static member, since there is no associated object
- You don't need to have an object in order to call a static member function.

In all of these respects, a static member function is just like an ordinary function—not like a member function at all. There are, however, three reasons why it might make sense to define a static member function instead of a regular global function:

- Placing a static function within a class is like giving it the class as a namespace. This could prevent name collisions with other functions having the same name.
- Use of static functions can allow you to create applications where all functions and data (except the main() function) are associated with objects. Indeed, in languages such as Java, declaring static member functions is the *only* way to implement general functions.
- Unlike regular functions, private and protected members can be accessed within static functions. This can be useful when objects are passed into the function as arguments or are created within the function.

In Example 6.12, a static member function, Count(), has been added to the ObjCounter class. In addition, in order to call the function within TestCount(), the scope resolution operator needs to be applied.

Example 6.12: ObjCounter class with static member function

```
// ObjCounter.h header file, modified with Count() member
#pragma once

class ObjCounter
{
public:
	ObjCounter(void);
	~ObjCounter(void);
	static int m_nCount;
	static int Count(){return m_nCount;}
};

void TestCount();
```

```
// ObjCounter.cpp source file
#include "objcounter.h"
#include <iostream>
using namespace std;

// Static members must be initialized outside the class definition
int ObjCounter::m_nCount=0;

ObjCounter::ObjCounter(void)
{
	m_nCount++;
}
ObjCounter::~ObjCounter(void)
{
	m_nCount--;
}
void TestCount()
{
	cout << ObjCounter::Count() << " is initial object count" <<
endl;
	ObjCounter obj1;
	cout << ObjCounter::Count() << " after obj1 created" << endl;
```

```
      ObjCounter *pArr=new ObjCounter[10];
      cout << ObjCounter::Count() << " after pArr initialized" << endl;
      delete [] pArr;
      cout << ObjCounter::Count() << " after pArr deleted" << endl;
}
```

Section 6.4 Questions

1. Why wouldn't it make sense to declare a regular (non-member) function as const?
2. Why could failing to properly declare a single member function as const have a ripple effect on other member functions?
3. Can you ever assign values to a member variable within a const member function?
4. How are static member variables like global variables?
5. Explain why the const qualifier makes no sense for a static member function
6. If only public static member variables can be declared, does the same apply to static member functions?

6.5: Introduction to Object-Oriented Design and UML

In this section, we will introduce the concept of the unified modeling language (UML), summarize a few principles of object-oriented design and programming practice that have emerged in the chapter and, finally, show how FlowC can be used to design C++ classes.

6.5.1: What is UML?

The Unified Modeling Language, or UML, is an evolving standard for designing complex systems and object-oriented applications. Unlike flowcharting—which offers the opportunity for direct translation from diagram to code—UML designs usually operate at a higher level of abstraction. They serve to organize and clarify complex programming and systems problems without proving a mechanical means of implementing the solution.

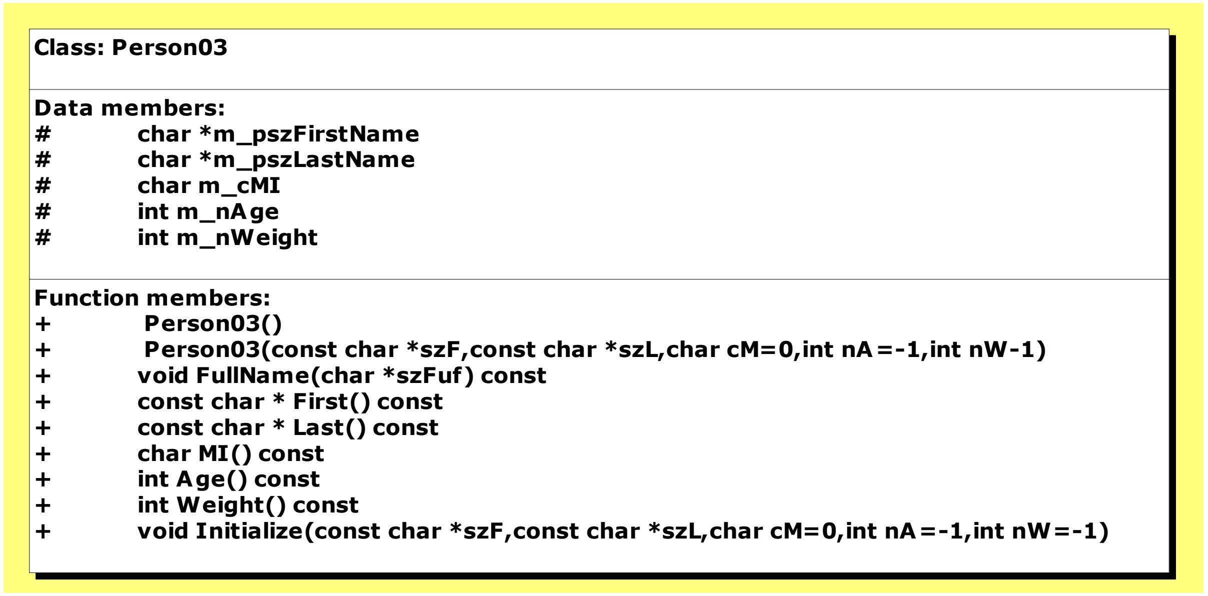
Although the UML standard continues to evolve, under the direction of the Object Management Group (OMG, at http://www.omg.org), certain vendors—such as Rational

Software, acquired by IBM in 2003—have taken a leading role in its development and in incorporating it into products. Although many diagrams exist, the OMG organizes them into three categories:

- *Structural Diagrams*, illustrating various aspects of the overall structure of an application or system.
- *Behavior Diagrams*, illustrating interactions between objects and transitions between object states.
- *Model Management Diagrams*, illustrating subsystems and models within the overall system or application.

The most commonly used diagrams in OOP design are the first two, structural diagrams—that illustrate how classes are defined and interrelate—and behavior diagrams, that illustrate how classes respond to events and messages from other classes.

In this text, we limit ourselves to considering some of the simple structural diagrams, the basic class diagram and the generalization/specialization (inheritance) diagram. The first of these, the basic class diagram, is a rectangular box divided into three parts: 1) the class name, 2) the data members of the class, and 3) the function members of the class. An example of such a diagram, created with FlowC, is presented in Figure 6.3. The second type of diagram relates to inheritance, and is introduced in Chapter 9.



```
Class: Person03

Data members:
#        char *m_pszFirstName
#        char *m_pszLastName
#        char m_cMI
#        int m_nAge
#        int m_nWeight

Function members:
+         Person03()
+         Person03(const char *szF,const char *szL,char cM=0,int nA=-1,int nW-1)
+        void FullName(char *szFuf) const
+        const char * First() const
+        const char * Last() const
+        char MI() const
+        int Age() const
+        int Weight() const
+        void Initialize(const char *szF,const char *szL,char cM=0,int nA=-1,int nW=-1)
```

Figure 6.3: Basic class diagram for Person03 class

In addition to listing the names of the data and function members, the FlowC version of the diagram provides additional information, such as function arguments. In addition, the characters on the left indicate member access, where # is protected and + is public (- is also commonly used to indicate private access). Where a member function or data is static, the $ symbol is also present[3].

6.5.2: Some Object-Oriented Design Principles for C++

During the course of this chapter, a number of comments have been made regarding reasonable design and programming practices for creating object-oriented C++ programs. Some of the most significant of these are as follows:

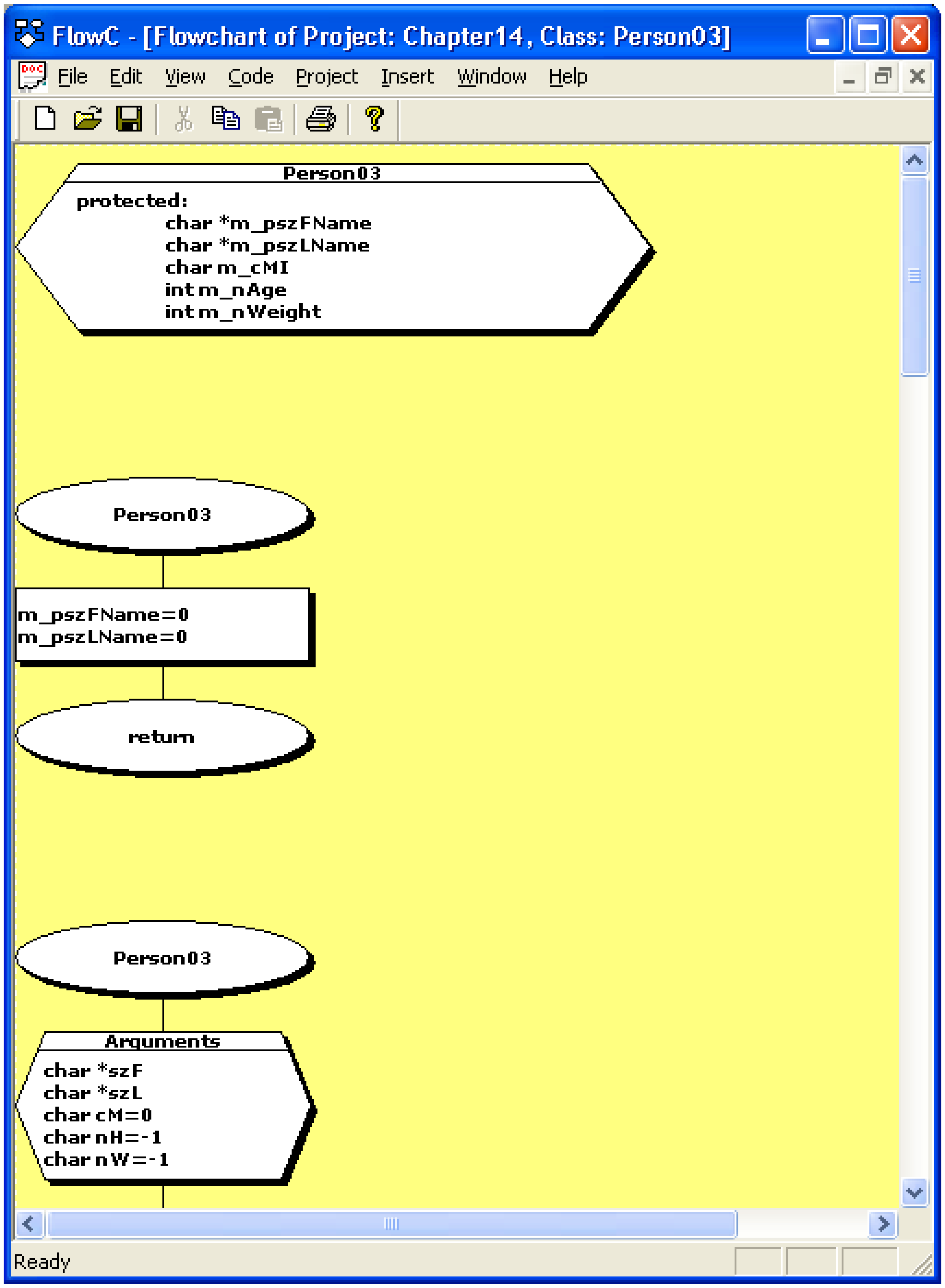
- *Think encapsulated:* The best C++ classes are designed as self-contained units. Such design encourages both abstract thinking and enhances reusability. One mechanical step you can take to help this along is to *write every class with its own .h file and .cpp file*. While this can lead to dozens (or hundreds) of files in even modest applications, it is much easier to move classes between applications when they are in separate files.
- *Differentiate between interface and implementation:* Use the public/protected access specifiers that C++ provides to make it clear what is internal to the class and what is available to programmers using the class. Also, don't assume you're only doing a favor for *other* programmers by designing your classes in this way. It can prove very useful when you are using your own classes, as well.
- *Design your classes as if the implementation could change at any minute:* In an "ideal" class, you can protect all your data, making only member functions part of the interface. It is easy to write accessor functions, and it is amazing how often they turn out to be useful. This was illustrated by the move from Person02 to Person03, where we changed almost everything about how the class was constructed without changing the interface.
- *Use naming conventions to reduce cognitive strain:* Simple things, such as beginning member variable names with m_ and data type abbreviations (as we have suggested here) can dramatically reduce the amount of thinking you need to do when you're programming. Remember, the reason OOP was developed was to help manage increasing program complexity. By choosing consistent names, you can avoid unnecessary complexity. Three weeks after you write a class you'll have forgotten nearly everything about the class you wrote and will be grateful for any help you can give yourself.
- *Design your classes so the compiler can help you find errors:* Capabilities that C++ offers, such as the const qualifier, can require some thought and their use may even slow down the initial writing of code. As code gets more and more complex, however, the initial writing time takes less and less of your time (as a percentage of total time). Locating bugs becomes the biggest challenge. Using tools such as const qualification can surface errors in your design—and attach a line number to them!

[3] The basic representation used to design this diagram was found at: http://www.rational.com/uml/resources/quick/plainposter.jsp, accessed 7/7/2002.

We will extend these principles further when we introduce the other main object-oriented programming capabilities (inheritance and polymorphism) in Chapters 9 and 10.

6.5.3: Creating in Classes in FlowC

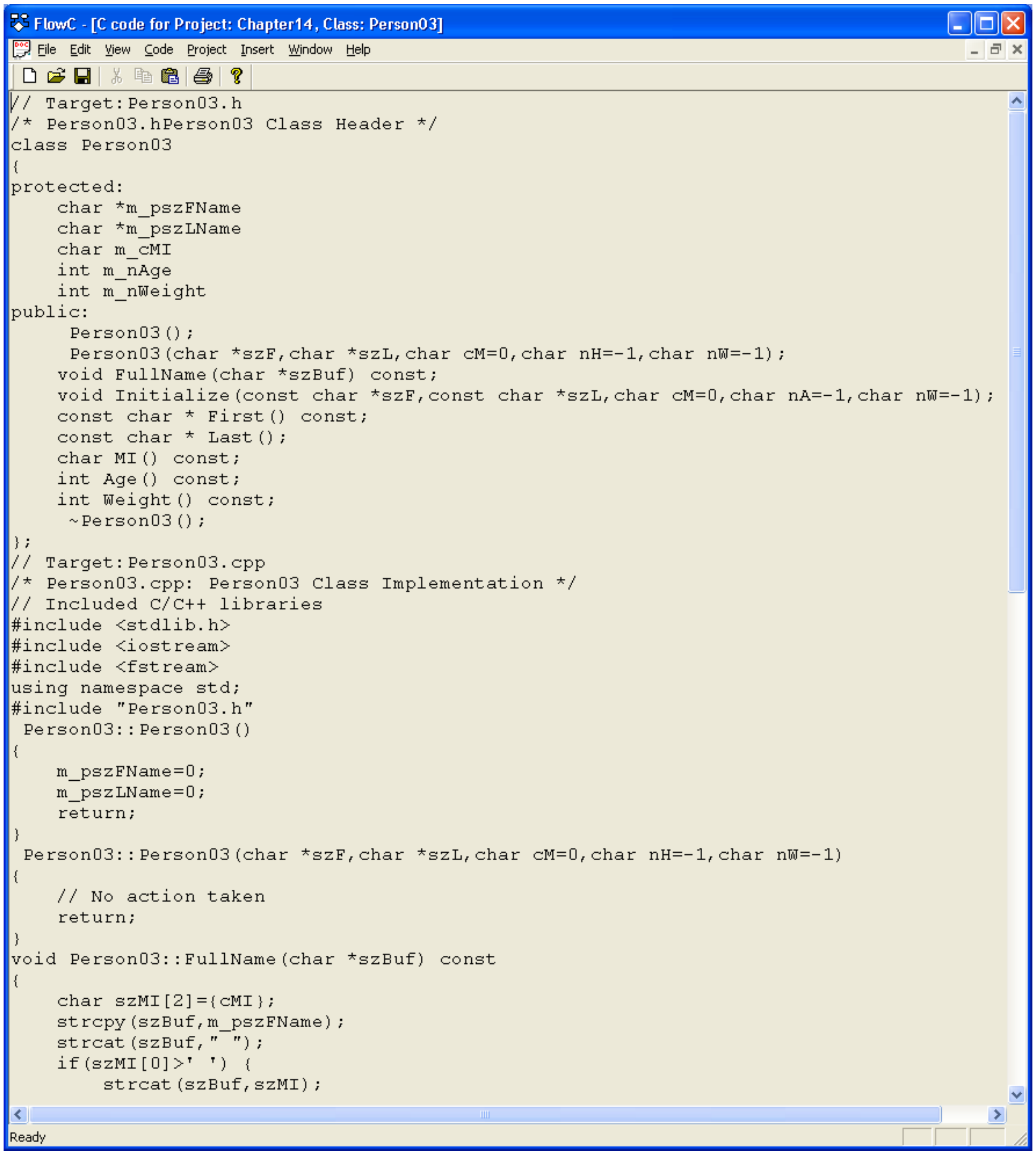
Walkthrough available in FlowCObject.avi


FlowC - [Flowchart of Project: Chapter14, Class: Person03]
File Edit View Code Project Insert Window Help
Ready

```
Person03
protected:
        char *m_pszFName
        char *m_pszLName
        char m_cMI
        int m_nAge
        int m_nWeight

Person03
m_pszFName=0
m_pszLName=0
return

Person03
Arguments
char *szF
char *szL
char cM=0
char nH=-1
char nW=-1
```

Figure 6.4: Class View in FlowC

Classes can be added to FlowC just as functions and structures can be added. Because classes are designed to be self-contained, however, they are only rendered in UML form in the project window (as shown in Figure 6.3). To edit a class, you must open a class window (from the main menu, the right-click menu or by double clicking the UML summary). The class view, shown in Figure 6.4, contains a declaration box that will display all member variables, and a collection of function flow charts for the function members. Once in a class view, you add and move between functions just as if it were a project view.


FlowC - [C code for Project: Chapter14, Class: Person03]
File Edit View Code Project Insert Window Help
Ready

```
// Target:Person03.h
/* Person03.hPerson03 Class Header */
class Person03
{
protected:
    char *m_pszFName
    char *m_pszLName
    char m_cMI
    int m_nAge
    int m_nWeight
public:
     Person03();
     Person03(char *szF,char *szL,char cM=0,char nH=-1,char nW=-1);
    void FullName(char *szBuf) const;
    void Initialize(const char *szF,const char *szL,char cM=0,char nA=-1,char nW=-1);
    const char * First() const;
    const char * Last();
    char MI() const;
    int Age() const;
    int Weight() const;
     ~Person03();
};
// Target:Person03.cpp
/* Person03.cpp: Person03 Class Implementation */
// Included C/C++ libraries
#include <stdlib.h>
#include <iostream>
#include <fstream>
using namespace std;
#include "Person03.h"
 Person03::Person03()
{
    m_pszFName=0;
    m_pszLName=0;
    return;
}
 Person03::Person03(char *szF,char *szL,char cM=0,char nH=-1,char nW=-1)
{
    // No action taken
    return;
}
void Person03::FullName(char *szBuf) const
{
    char szMI[2]={cMI};
    strcpy(szBuf,m_pszFName);
    strcat(szBuf," ");
    if(szMI[0]>' ') {
        strcat(szBuf,szMI);
```

Figure 6.5: FlowC code generation

FlowC can generate the C++ code for any class constructed within it, as shown in Figure 6.5. Consistent with common C++ practice, FlowC creates two files for each class: a header file containing the definition and a .cpp containing the implementation of any functions not declared to be inline. Classes included in a project can be exported to a Folder containing a Visual Studio .Net project and then brought into the project.

Although FlowC is not intended to be a full-fledged design tool, it can be useful for creating a skeletal project with class declarations and function stubs. It will also be used, from time to time, in the rest of the text for purposes of illustration.

6.5 Section Questions

1. What does it mean when we say that UML operates at a higher level of abstraction than flowcharting?

2. What key pieces of information are conveyed by a basic UML class diagram?

3. What recommended design and programming practices particularly help to reduce the cognitive strain on the programmer when designing and using C++ classes?

4. What recommended design and programming practices particularly help to reduce debugging time for C++ classes?

5. What does it mean to use FlowC to "create a skeletal project with class declarations and function stubs"?

6.6: Encapsulation Exercises

Walkthrough available in EmpClass.avi

In this section, we consider how to transform code that worked with structures into encapsulated objects. Specifically, we walk through the transformation of structure-based code for an employee database—originally developed in Chapters 3 and 4—into two classes. A lab is then presented for transforming the implementation of that database into one that utilizes dynamic memory.

6.6.1: A Simple employee Class

In Chapter 3, a simple structure was introduced to hold employee data. That structure, presented below in Example 6.13 was a pure structure. To work with the structure, we developed a series of functions, the prototypes of which are also presented in Example 6.13. Our first objective is to transform this collection of data and functions into an encapsulated class.

Example 6.13: employee Structure from Chapter 3

```
#define MAXNAME 30

struct employee {
      unsigned int nId;
      char szLastName[MAXNAME];
      char szFirstName[MAXNAME];
      unsigned int nYearHired;
      double dSalary;
      bool bActive;
};

void DisplayEmp(struct employee emp);
void InputEmp(struct employee *pEmp);
void SaveEmployee(STREAM &out,const struct employee *pEmp);
void LoadEmployee(STREAM &in,struct employee *pEmp);
void SaveString(STREAM &out,const char *sz);
void LoadString(STREAM &in,char *sz);
```

A number of aspects of the transforming Example 6.13 to a class are quite straightforward, and are applicable to almost any conversion from structured to object-oriented programs:

- Member data should be made protected by default, and it makes sense to rename it for consistency (e.g., using the m_ prefix convention often found in Microsoft code)
- Simple (one line) accessor functions can be written for setting and accessing member data.
- Any function that takes a structure as an argument can easily be rewritten as a member function. The procedure involves:
    - Removing the structure reference or pointer from the argument
    - If the reference or pointer argument was const, add the const qualifier to the member declaration
- Any function related to the class that does not take a structure as an argument can be made static.

In addition to these functions, it also makes sense to define one or more constructor functions for initializing data. As we will discuss further in Chapter 8, it is also frequently convenient to have a function that copies an object. It is not unreasonable to call such a function Copy(), and pass it a constant reference to the object we're making a copy of.

The result of making these changes are presented in Example 6.14

Example 6.14: employee Class after modifications

```
// Employee class
#define MAXNAME 30

class employee
{
public:
	employee();
	employee(int nId,const char *szL,const char *szF,
	unsigned int nY=2003,double dSalary=0.0,bool bActive=true);
	void Copy(const employee &emp);
	// Modified functions
	void Save(STREAM &out) const;
	void Load(STREAM &in);
	void Display() const;
	void Input();
	static void SaveString(STREAM &out,const char *sz);
	static void LoadString(STREAM &in,char *sz);

	// Accessor functions
	unsigned int Id() const {return m_nId;}
	void Id(int n){m_nId=n;}
	const char *LastName() const {return m_szLastName;}
	void LastName(const char *sz){strncpy(m_szLastName,sz,MAXNAME);}
	const char *FirstName() const {return m_szFirstName;}
	void FirstName(const char
*sz){strncpy(m_szFirstName,sz,MAXNAME);}
	unsigned int YearHired() const {return m_nYearHired;}
	void YearHired(int n){m_nYearHired=n;}
	double Salary() const {return m_dSalary;}
	void Salary(double d){m_dSalary=d;}
	bool Active() const {return m_bActive;}
	void Active(bool b){m_bActive=b;}

protected:
	unsigned int m_nId;
	char m_szLastName[MAXNAME];
	char m_szFirstName[MAXNAME];
	unsigned int m_nYearHired;
	double m_dSalary;
	bool m_bActive;
};
```

The revised class consists of the following:

- Two constructors, a default and one that allows all data elements to be initialized
- A Copy() function used to copy the contents of another object
- Rewritten Save(), Load(), Display() and Input() functions. In each case, we 1) dropped "Employee" or "Emp" from the name (since, being a member, it's obvious what type of data we are operating on), and 2) we removed the structure

reference from the argument. For two of the functions, Save() and Display(), we added const qualifiers since they are not going to change the object they are applied to (unlike Edit() and Load(), which will change the object by their very nature).

- SaveString() and LoadString() are brought into the class as static members, since they aren't applied directly to an object. It is debatable that they should be made part of the class at all—since they could be used to load and save strings from any source—but we do so here for the purposes of demonstration.
- A pair of accessor functions is defined for each member data element. The first (const qualified) is used to return the value from the object. The second is used to set the value.

Example 6.15: constructor and Copy() functions

```
employee::employee()
{
      m_nId=0;
      m_szLastName[0]=0;
      m_szFirstName[0]=0;
      m_nYearHired=0;
      m_dSalary=0;
      m_bActive=false;
}

employee::employee(int nId,const char *szL,const char *szF,
            unsigned int nY,double dSalary,bool bActive)
{
      m_nId=nId;
      strncpy(m_szLastName,szL,MAXNAME);
      strncpy(m_szFirstName,szF,MAXNAME);
      m_nYearHired=nY;
      m_dSalary=dSalary;
      m_bActive=bActive;
}

void employee::Copy(const employee &emp)
{
      Id(emp.m_nId);
      LastName(emp.m_szLastName);
      FirstName(emp.m_szFirstName);
      YearHired(emp.m_nYearHired);
      Salary(emp.m_dSalary);
      Active(emp.m_bActive);
}
```

Most of our member functions are quite straightforward. Three of our new functions, two constructors and the Copy() function, are presented in Example 6.15. These functions set member data values in two different ways—the constructor using the variables themselves and the Copy() function using the accessor functions that were defined. Of these two approaches, the accessor function approach is generally preferred. In this illustration, for example, using the accessor for FirstName() and LastName() allows you to avoid worrying about overrunning the m_szLastName and m_szFirstName arrays—since the accessor automatically ensure that too many characters are not copied.

The types of changes made to the earlier functions are all similar in nature. For this reason, we'll examine the "before" and "after" for just two of them: the InputEmp () function (which became the Input() member function) and the SaveEmployee() function (which became the Save() member function). The Input() group is shown in Example 6.16, the Save() group in Example 6.17.

Example 6.16: InputEmp () and Input() member functions

```
void InputEmp(struct employee *pEmp)
{
	char buf[MAXLINE];
	DisplayString("\nHit enter to accept existing values...\n\n");
	DisplayFormatted("Last Name [%s]: ",pEmp->szLastName);
	InputString(buf);
	if (strlen(buf)>0)strncpy(pEmp->szLastName,buf,MAXNAME);
	DisplayFormatted("First Name [%s]: ",pEmp->szFirstName);
	InputString(buf);
	if (strlen(buf)>0)strncpy(pEmp->szFirstName,buf,MAXNAME);
	DisplayFormatted("Salary [%.2f]: ",pEmp->dSalary);
	InputString(buf);
	if (strlen(buf)>0)pEmp->dSalary=atof(buf);
	DisplayFormatted("Year hired [%i]: ",pEmp->nYearHired);
	InputString(buf);
	if (strlen(buf)>0)pEmp->nYearHired=atoi(buf);
	DisplayFormatted("Still active (Y or N) [%s]: ",
	(pEmp->bActive) ? "Yes" : "No");
	InputString(buf);
	if (strlen(buf)>0)pEmp->bActive=(buf[0]=='y' || buf[0]=='Y');
}

void employee::Input()
{
	char buf[MAXLINE];
	DisplayString("\nHit enter to accept existing values...\n\n");
	DisplayFormatted("Last Name [%s]: ",m_szLastName);
	InputString(buf);
	if (strlen(buf)>0)strncpy(m_szLastName,buf,MAXNAME);
	DisplayFormatted("First Name [%s]: ",m_szFirstName);
	InputString(buf);
	if (strlen(buf)>0)strncpy(m_szFirstName,buf,MAXNAME);
	DisplayFormatted("Salary [%.2f]: ",m_dSalary);
	InputString(buf);
	if (strlen(buf)>0)m_dSalary=atof(buf);
	DisplayFormatted("Year hired [%i]: ",m_nYearHired);
	InputString(buf);
	if (strlen(buf)>0)m_nYearHired=atoi(buf);
	DisplayFormatted("Still active (Y or N) [%s]: ",
	(m_bActive) ? "Yes" : "No");
	InputString(buf);
	if (strlen(buf)>0)m_bActive=(buf[0]=='y' || buf[0]=='Y');
}
```

The two Input() functions are nearly identical except for a single change: wherever pEmp-> was present in the original function, it has been removed—since our member names now represent the values from the object we are operating on. The same applies for the Save() family. Of note here is the const qualifier applied to the definition of the Save() member—since saving an object should not normally require modifying it. The qualification makes the compiler check to be sure that we aren't using any improper functions inside of Save(), or setting member variable values.

Example 6.17: SaveEmployee () and Save() member functions

```
void SaveEmployee(STREAM &out,const struct employee *pEmp)
{
	WriteInteger(out,pEmp->nId);
	SaveString(out,pEmp->szLastName);
	SaveString(out,pEmp->szFirstName);
	WriteInteger(out,pEmp->nYearHired);
	WriteReal(out,pEmp->dSalary);
	WriteInteger(out,(int)pEmp->bActive);
}

void employee::Save(STREAM &out) const
{
	WriteInteger(out,m_nId);
	SaveString(out,m_szLastName);
	SaveString(out,m_szFirstName);
	WriteInteger(out,m_nYearHired);
	WriteReal(out,m_dSalary);
	WriteInteger(out,(int)m_bActive);
}
```

A slightly more significant change had to be made to the EmployeeTest() function used to test employee input and display. The old and new versions of these functions are presented in Example 6.18.

Example 6.18: Structure and Object versions of EmployeeTest()

```
// Structure version
void EmployeeTest()
{
	struct employee empSmith={101,"Smith","Joan",1999,100000,true};
	DisplayEmp(empSmith);
	NewLine();
	InputEmp(&empSmith);
	NewLine();
	DisplayEmp(empSmith);
}

// Object version
void EmployeeTest()
{
	employee empSmith(101,"Smith","Joan",1999,100000,true);
	empSmith.Display();
	NewLine();
	empSmith.Input();
	NewLine();
	empSmith.Display();
}
```

Among the changes to the object version:

- Since we cannot initialize protected members with an aggregate list, our multi-argument version of the constructor is invoked in its place. Specifically:

  ```
  struct employee empSmith={101,"Smith","Joan",1999,100000,true};
  ```

  becomes:

  ```
      employee empSmith(101,"Smith","Joan",1999,100000,true);
  ```

- Rather than passing our object in as an argument to various functions, we apply the member to it. For example:

  ```
      InputEmp(&empSmith);
  ```

  becomes

  ```
  empSmith.Input();
  ```

Naturally, not all transformations of structured programs to classes will be as mechanical as the above—which was facilitated by the fact the task of working with employees is naturally suited to object representation.

6.6.2: In Depth: An Employee Collection Class

The EmpData application (presented in Chapter 4) involves creating a simple database for loading and saving employee objects. The original structure being modified was prodigiously simple (shown in Example 6.18, along with function prototypes).

Example 6.18: Original EmpData structure

```
#define MAXEMPLOYEES 100

#include "Employee.h"

struct EmpData {
      int nCount;
      struct employee arEmp[MAXEMPLOYEES];
};

unsigned int PromptForId();
void ListEmployees(const struct EmpData *pData);
void AddEmployee(struct EmpData *pData);
void EditEmployee(struct EmpData *pData,unsigned int nId);
bool DeleteEmployee(struct EmpData *pData,unsigned int nId);
int FindEmployee(const struct EmpData *pData,unsigned int nId);
bool SaveAll(const char *szName,const struct EmpData *pData);
bool LoadAll(const char *szName,struct EmpData *pData);
```

The revised class, shown in Example 6.19, is fairly straightforward, but has a number of new member functions whose purpose is not immediately obvious.

Example 6.19: EmpData class

```
class EmpData
{
public:
      EmpData(){m_nCount=0;}
      employee *GetAt(int i) {
            if (i<0 || i>=m_nCount) return 0;
            return m_arEmp+i;
      }
      const employee *GetValue(int i) const {
            if (i<0 || i>=m_nCount) return 0;
            return m_arEmp+i;
      }
      int Count() const {return m_nCount;}
      static unsigned int PromptForId();
      void ListEmployees() const;
      void AddEmployee();
      void EditEmployee(unsigned int nId);
      bool DeleteEmployee(unsigned int nId);
      int FindEmployee(unsigned int nId) const;
      bool Save(const char *szName) const;
      bool Load(const char *szName);
protected:
      int m_nCount;
      employee m_arEmp[MAXEMPLOYEES];
};
```

The most important of these new member functions are the GetAt() and GetValue() members. The two functions have an identical purpose: accessing a pointer to a particular employee object in the m_arEmp[] array. What differentiates them is that one—GetValue()—returns a const employee object and is const qualified, while the other returns a regular pointer and is not const qualified. The implication is as follows:

- If you need to access an employee in a const qualified member function, use the GetValue() version
- If you need to access an employee, then change the value of that employee, use the GetAt() version.

This distinction is evident by comparing the Save() and Load() members, as shown in Example 6.20.

Example 6.20: Save() and Load()

```
bool EmpData::Save(const char *szName) const
{
	STREAM out;
	int i;
	// Open for writing, truncate
	if (!Open(out,szName,false,true,true,false)) {
		return false;
	}
	WriteInteger(out,m_nCount);
	for(i=0;i<Count();i++) {
		GetValue(i)->Save(out);
	}
	Close(out);
	return true;
}

bool EmpData::Load(const char *szName)
{
	STREAM in;
	int i;
	// Open for reading, don't truncate
	if (!Open(in,szName,true,false,false,false)) {
		return false;
	}
	m_nCount=ReadInteger(in);
	for(i=0;i<Count();i++) {
		GetAt(i)->Load(in);
	}
	Close(in);
	return true;
}
```

Since saving the employees does not require modifying them, GetValue() is used to iterate through the count of employees, so each can be saved to the stream. In Load(), however, GetAt() is used to acquire the pointer into which data from the stream will be loaded.

Most of the remaining member functions are the result of straightforward transformations of the original functions—similar to those shown in Section 6.6.1. Two functions, however, require us to do more major changes. The problem is that when we defined constructors for the employee object, we altered our implicit ability to assign one employee to another—a subject addressed in greater detail in the next chapter. As a result, functions that previously assigned one structure to another (e.g., DeleteEmployee(), AddEmployee(), etc.) no longer compile. To get around this, we replace the assignment operator with a call to the employee::Copy() function. The before and after versions of the AddEmployee() functions are presented in Example 6.21.

Example 6.21: Before and after versions of AddEmployee functions

```
/* AddEmployee() allows an employee to be added to
to the table and edited */
void AddEmployee(struct EmpData *pData)
{
        struct employee emp={0};
        bool bvalid=false;
        while(!bvalid) {
                unsigned int nId=PromptForId();
                if (FindEmployee(pData,nId)>=0) {
                        char buf[MAXLINE];
                        DisplayString("ID that you entered already exists. Try again? (Y or
N): ");
                        InputString(buf);
                        if (toupper(buf[0])=='N') bvalid=true;
                }
                else {
                        emp.nId=nId;
                        InputEmp(&emp);
                        pData->arEmp[pData->nCount]=emp;
                        pData->nCount++;
                        bvalid=true;
                }
        }
}
/* After: object version */
void EmpData::AddEmployee()
{
        employee emp;
        bool bvalid=false;
        while(!bvalid) {
                unsigned int nId=PromptForId();
                if (FindEmployee(nId)>=0) {
                        char buf[MAXLINE];
                        DisplayString("ID that you entered already exists. Try again? (Y or
N): ");
                        InputString(buf);
                        if (toupper(buf[0])=='N') bvalid=true;
                }
                else {
                        emp.Id(nId);
                        emp.Input();
                        m_arEmp[m_nCount].Copy(emp);
                        m_nCount++;
                        bvalid=true;
                }
        }
}
```

6.6.3: In Depth: Dynamic Employee Lab Exercise

In this lab exercise, the objective is simple:

- Reimplement the employee and EmpData classes so that they use dynamic arrays
- *Make no changes to the public interface of either class*

The new classes will be defined as shown in Examples 6.22 and 6.23. In both cases, the only public function change was the addition of a destructor function.

Example 6.22: Revised employee class

```
class employee
{
public:
      employee();
      employee(int nId,const char *szL,const char *szF,
      unsigned int nY=2003,double dSalary=0.0,bool bActive=true);
   ~employee();
      void Copy(const employee &emp);
      // Modified functions
      void Save(STREAM &out) const;
      void Load(STREAM &in);
      void Display() const;
      void Input();
      static void SaveString(STREAM &out,const char *sz);
      static void LoadString(STREAM &in,char *sz);

      // Accessor functions
      unsigned int Id() const {return m_nId;}
      void Id(int n){m_nId=n;}
      const char *LastName() const {return m_szLastName;}
      void LastName(const char *sz){strncpy(m_szLastName,sz,MAXNAME);}
      const char *FirstName() const {return m_szFirstName;}
      void FirstName(const char
*sz){strncpy(m_szFirstName,sz,MAXNAME);}
      unsigned int YearHired() const {return m_nYearHired;}
      void YearHired(int n){m_nYearHired=n;}
      double Salary() const {return m_dSalary;}
      void Salary(double d){m_dSalary=d;}
      bool Active() const {return m_bActive;}
      void Active(bool b){m_bActive=b;}
protected:
      unsigned int m_nId;
      char *m_szLastName;
      char *m_szFirstName;
      unsigned int m_nYearHired;
      double m_dSalary;
      bool m_bActive;
};
```

Example 6.23: Revised EmpData class

```
class EmpData
{
public:
      EmpData();
      ~EmpData();
      employee *GetAt(int i);
      const employee *GetValue(int i) const;
      int Count() const {return m_nCount;}
      static unsigned int PromptForId();
      void ListEmployees() const;
      void AddEmployee();
      void EditEmployee(unsigned int nId);
      bool DeleteEmployee(unsigned int nId);
      int FindEmployee(unsigned int nId) const;
      bool Save(const char *szName) const;
      bool Load(const char *szName);
protected:
      int m_nCount;
      employee *m_arEmp;
};
```

In implementing the classes, pay particular attention to:

- The changes made between Person01 (Example 6.4) and Person02 (Example 6.8) in this chapter
- The discussion of dynamic arrays in Chapter 13, Section 13.4.

6.7: Review and Questions

6.7.1: Review

The C++ programming language is based upon an OOP approach to programming. An OOP approach typically exhibits three characteristics:

- *Encapsulation:* The combination of data and functions into self-contained objects, operating independently, or nearly independently of each other.
- *Inheritance:* The ability to define objects using the characteristics and structure of other objects as a starting point.
- *Polymorphism:* The presence of functions or messages where the same function/message applied to different objects can produce distinctly different behaviors (depending on the type of object to which the function/member is applied).

In a C++ class or struct, every data member or function member is defined to be either accessible to programmers using the class (*public*) or inaccessible to programmers using the class (*private, protected*). In addition, other classes or functions can be declared to be *friend* functions/classes, meaning that they are granted access to non-public members when they use objects of the class.

By controlling accessibility to members, class designers can distinguish between the interface of a class and its implementation. One very common design choice that is often made by object-oriented programmers during class design is to hide the internal data structure of objects by making all data members private/protected, and including only member functions in the interface. Thus, programmers who use an existing class, such as the C++ standard library classes, will nearly always be applying member functions to class objects, rather than directly accessing the internal data of those objects. Designing classes in this way tends to promote long term stability—allowing internal data structures to change with system changes (e.g., hardware, operating system) while the original interface to the object is supported.

Member functions also have certain characteristics not associated with regular C++ functions. Member functions qualified as **const** do not change any data in the object to which they are applied. Member functions qualified as **static** are not applied to any particular object, making them roughly equivalent to regular C++ functions declared in a namespace.

Every C++ class has one or more constructor functions, used to initialize objects when they are created, and a single destructor function. If no constructor function is explicitly defined, two public constructors—one taking no arguments and one doing a byte-by-byte copy of an object of the same type—are implicitly defined. As soon as the programmer defines any constructor, the implicit default constructor ceases to be available. Constructor functions can be identified by virtue of the fact that they have the same name as the class itself, and no return values. These constructors can be used both in standard declarations and in dynamic memory allocation by supplying arguments in parentheses. Only a default constructor—a constructor that takes no arguments—can be used to allocate dynamic arrays of objects.

The destructor function for a class takes no arguments and is normally defined to perform cleanup operations, such as releasing dynamic memory allocated by the class itself and closing any open files. An implicit destructor that does nothing is defined if none is specified by the programmer. Destructor functions are not called explicitly, but instead get called as a side effect of operations such as deletion and variables going out of scope.

In designing object-oriented applications or systems, the Unified Modeling Language (UML) is sometimes used. This language consists of a variety of diagrams that can be used to describe application structure, activities and system organization. A common UML diagram is the basic class diagram, which identifies the data and function members of a class, along with their access specification (i.e., public, private, protected). FlowC can be used to create rudimentary basic class diagrams.

In the course of explaining the basic approach to encapsulation provided by C++, a number of recommendations for designing object-oriented applications were illustrated. These include:

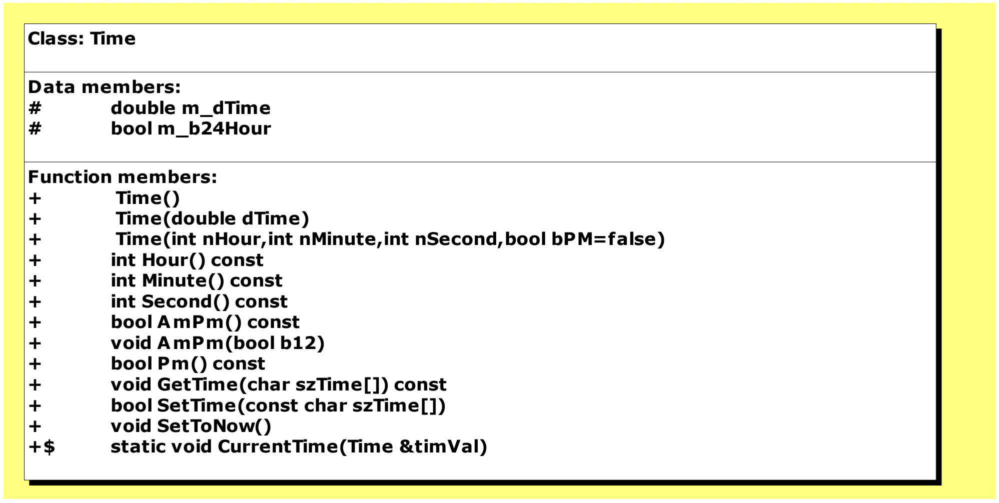
- *Think encapsulated:* The best C++ classes are designed as self-contained units.
- *Differentiate between interface and implementation:* Use the public/protected access specifiers to make it clear what is internal to the class and what is available to programmers using the class.
- *Design your classes as if the implementation could change at any minute:* In an "ideal" class, you can protect all your data, making only member functions part of the interface.
- *Use naming conventions to reduce cognitive strain:* Simple things, such as beginning member variable names with m_ and data type abbreviations (as we have suggested here) can dramatically reduce the amount of thinking you need to do when you're programming.
- *Design your classes so the compiler can help you find errors:* Capabilities that C++ offers, such as the const qualifier, can require some thought and their use may even slow down the initial writing of code.

6.7.2: Glossary

Accessor functions – Member functions specifically designed to provide access to data members within the class.
class – A definition of the organization of an object, similar to a structure definition in C/C++, but typically including functions as well as data elements.
const Qualifier – A keyword placed at the end of a member function's header declaration specifying that calling the function will not change object data.
Construct oriented programming – An approach to programming dominated by the use of branching and looping constructs. Programs organized in this fashion tend to rely heavily on the main() function.
Constructor function – A function used to initialize an object when it is declared. In C++, such functions can be identified by virtue of the fact that they have the same name as the class, and no return value.
Copy Constructor – A constructor function that initializes the object being constructed by copying the data in another object.
Data member – A member of a structure or class that contains defines data to be stored in all objects based upon that class.
Default constructor – A constructor function used to create an object if no arguments are specified in the object's declaration.

Destructor function – A function that is called whenever an object is released (e.g., goes out of scope, is deleted). Normally performs cleanup operations such as releasing memory from managed data members and closing any associated file streams. In C++, such functions can be identified by virtue of the fact that they have the same name as the class, preceded by a tilde (~), and no return value.
Encapsulation – The ability to combine data and code into a self-contained object.
friend – A function or class that is granted access to non-public members of another class as part of the declaration of the other class.
Function oriented programming – An approach to programming where the primary means of organizing the program is based around the creation of self-contained functions. Structured programming in C/C++ tends to be organized along these lines.
Implementation – The internal construction of an object, of greatest interest to the object's designer.
Inheritance – The ability to define new objects using existing objects as a starting point.
Interface – The collection of data and function members associated with an object that is available to any programmer using that object (much the way a user-interface represents the set of tools provided for a user's interaction with an application). Typically, an object's interface is the only part of a well-constructed object that is relevant to anyone not involved with the object's internal design.
Member function – A function that is defined as part of a class and operates upon objects declared to be of that class, applied using the same . and -> that are used to access elements of structures.
Object oriented programming – An approach to programming where the primary tool for organizing programs is the self-contained object, which allows data and functions to be encapsulated into a single unit.
Polymorphism – The ability to define objects that respond to the same message (e.g., a command sent by another object) differently according to the nature of the receiving object.
Private class member – A member of an object that cannot be altered by any other class or object, considered to be part of the object's implementation.
Protected class member – A private member of an object that can not be altered by any other class or object unless it that class inherits from the original class.
Public class member – A member of an object that can be accessed by other objects, considered to be part of the object's interface.
static Qualifier – A keyword placed before a member's declaration specifying that the data or function is shared by all objects in the class, and is not applicable to a single object (the way normal member functions and data are).
Unified Modeling Language (UML) – A language consisting of a variety of diagrams that can be used to describe application structure, activities and system organization.

6.7.3: Questions



```
Class: Time

Data members:
#         double m_dTime
#         bool m_b24Hour

Function members:
+          Time()
+          Time(double dTime)
+          Time(int nHour,int nMinute,int nSecond,bool bPM=false)
+         int Hour() const
+         int Minute() const
+         int Second() const
+         bool AmPm() const
+         void AmPm(bool b12)
+         bool Pm() const
+         void GetTime(char szTime[]) const
+         bool SetTime(const char szTime[])
+         void SetToNow()
+$        static void CurrentTime(Time &timVal)
```

Figure 6.6: UML diagram of Time class

Questions 1-6 involve the creation of an encapsulated class to hold time values. The basic UML class diagram for the completed class is presented in Figure 6.6.

6.1: Data and constructors. Create a basic class containing two protected data members:

- m_dTime should be a double whose decimal portion represents the percentage of the day complete (e.g., 0.3333 is 8 AM, 0.75 is 6 PM)
- m_b24Hour is a Boolean value signifying whether hour values returned are on a 12 hour (AM/PM) or 24 hour clock.

Create three constructors, 1) a default constructor that initializes the time to 12 AM (i.e., 0.00), 2) a constructor that takes a double and assigns it to the m_dTime value, and 3) a constructor that computes the time fraction for an hour, minute, second combination that is passed in. If bPM is true, the hour should be interpreted as PM on the AM/PM setting and m_b24Hour should be set to false. Otherwise, set m_b24Hour to true.

6.2: *AM/PM accessor functions.* Add a pair of accessor functions named AmPm for setting the value of m_b24Hour (which should be false if AmPm() is true).

6.3: *Some more accessor functions.* Add const-qualified member functions that return the following values:

- The hour associated with the m_dTime time value. *Note:* This should never be greater than 12 if m_b24Hour is true. Otherwise, it should never be greater than 23. Also, note that when m_b24Hour is true, the hour value can be impacted even in the AM time period (i.e., 12:30 AM)
- The minute associated with the current time (0 to 59)
- The second associated with the current time (0 to 59)
- Pm() is set to true if the hour is returned using the AM/PM system and it is a PM value.

6.4: Getting a Time String. Write a function GetTime() that takes the current time and places it, in text form, in a buffer that is passed in. You should format the string according to whether a 24 hour clock is used. (*Hint:* This is definitely a function where sprintf is quite convenient)

6.5: In Depth: Setting a Time String. Write a function SetTime() that takes a sting containing the time (e.g., "3:00:45 PM"), determines its fractional value then uses that to set the value of m_dTime. The function should return false if it cannot interpret the time string passed in.

6.6: In Depth: Getting System Time. Write a static member function, CurrentTime(), which retrieves the system time then translates it into a Time object (passed in as an object reference). Also use that function within a member function SetToNow() that sets the time value of the object to which it is applied to the current time. To access the system time, the basic approach is as follows:

- Include <time.h> in your source file
- To get the time, write the following code:

```
time_t theTime=time(NULL);
struct tm *pTm, tmData;
pTm=localtime(&theTime);
// Copying data from static structure
tmData=(*pTm);
```

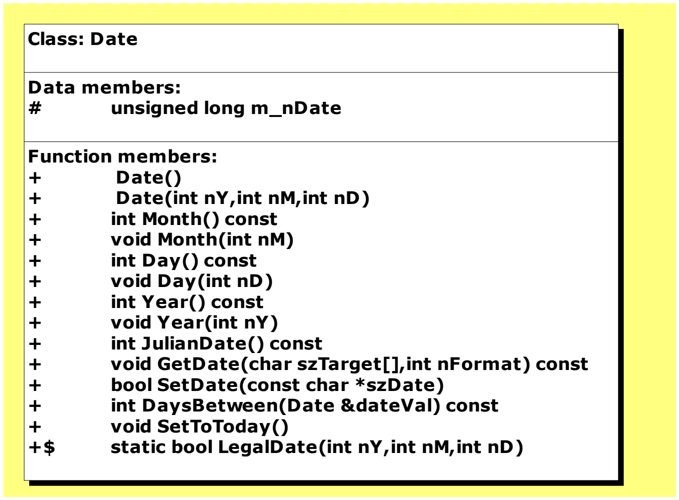
  - Use the Visual Studio .Net system to find out more about the tm structure(its elements are presented in Example 6.24). This structure contains the information you will need to determine your time value.

Note: after reading about the functions available in time.h, you may decide to change how you implement your GetTime() and SetTime() functions.

Example 6.24: tm Structure Definition

```
struct tm {
int tm_sec; /* seconds after the minute - [0,59] */
int tm_min; /* minutes after the hour - [0,59] */
int tm_hour; /* hours since midnight - [0,23] */
int tm_mday; /* day of the month - [1,31] */
int tm_mon; /* month number – [0,11] */
int tm_year; /* years since 1900 */
int tm_wday; /* days since Sunday - [0,6] */
int tm_yday; /* days since January 1 - [0,365] */
int tm_isdst; /* daylight savings time flag */
 };
```

Questions 7-10 involve creating an encapsulated Date class. The basic UML class diagram for this class is presented in Figure 6.7.



```
Class: Date

Data members:
#         unsigned long m_nDate

Function members:
+          Date()
+          Date(int nY,int nM,int nD)
+         int Month() const
+         void Month(int nM)
+         int Day() const
+         void Day(int nD)
+         int Year() const
+         void Year(int nY)
+         int JulianDate() const
+         void GetDate(char szTarget[],int nFormat) const
+         bool SetDate(const char *szDate)
+         int DaysBetween(Date &dateVal) const
+         void SetToToday()
+$        static bool LegalDate(int nY,int nM,int nD)
```

Figure 6.7: Basic UML class diagram for Date class

6.7: Data, constructors and accessors. Create a basic Date class containing a protected member m_nDate, and unsigned integer specifying the date in the form YYYYMMDD. (e.g., 23 July 2004 would be 20040723). As shown in Figure 6.7, create two constructors and accessor pairs for Month(), Day() and Year().

6.8: LegalDate function. Create a static LegalDate() member that returns true unless a specific year, month, day combination is not legal. Naturally, the challenge with this function is identifying leap years—since it changes what is legal for February. Use the following rule:

- If a year is divisible by 400, it is a leap year
- Otherwise, if it is divisible by 100 it is *not* a leap year
- Otherwise, if it is divisible by 4 it is a leap year
- Otherwise, it is not a leap year

You may want to call this function in your constructor and other accessor functions to ensure that invalid dates are not set.

6.9: JulianDate() and DaysBetween() members. Write a member function that computes the Julian date (i.e., numeric value of the day, where 1 Jan is 1, 2 Jan is 2, 1 Feb is 32, etc.) for a particular day. Once this function is complete, create a DaysBetween() function that returns the number of days between the object date and the functions argument, returning the value as an integer.

6.10: In Depth: GetDate(), SetDate() and SetToToday() functions. Write functions modeled after the GetTime(), SetTime() and SetToNow() functions in the Time class (Questions 6.4-6.6). For your GetDate() function, define at least two different string formats for the date, that can be passed in using the second argument (nFormat). Your SetDate() function should be able to successfully interpret both formats.

Questions 11 and 12 involve designing your own classes around loosely specified guidelines. They assume completion of 1-10 (although the design portion can be done without actually implementing these the Time and Date classes).

6.11. Create a DateTime class. Design and implement a class that composes a Date and Time object. There should be no public data members. (*Hint:* Much of the work on this class will involve writing simple accessor functions).

6.12. Create a TimeSpan class. This class should hold a double value, expressed in days and fractional days, that is computed when either Date, Time or DateTime objects are subtracted from each other.

Chapter 7

Operator Overloading

Executive Summary

Chapter 7 examines the rather unique capability that C++ provides for defining operators. Used appropriately, operator overloading can improve code clarity and speed the process of coding. Used excessively, it can render the programmer's code nearly incomprehensible. Although most operator overloading can be done at the discretion of the programmer, one type of overload—overloading the assignment operator—is virtually mandated in many programming situations. For this reason, we cannot ignore operator overloading.

The chapter begins by showing the mechanics of operator overloading, both within a class and outside of it. We then explore assignment operator overloads—the most critical overload—in some detail. In doing so, we also discuss the copy constructor, which is virtually always defined in conjunction with the assignment operator. We then proceed through a series of progressively more complex examples, beginning with an overload of the insertion and extraction operators, followed by an overload of the bracket operator used to create a void pointer array class. Finally, we develop a fully functional encapsulated NUL-terminated string class, called the GString class, which overloads almost a dozen different operators (including a type cast). These final exercises are intended to provide the reader with insights into the nature of the **string** and **vector<>** standard template library (STL) classes, which we begin exploring in Chapter 8.

Learning Objectives

Upon completing this chapter, you should be able to:

- Explain the relationship of the functional notation for operators that we adopted in Chapter 2 to operator overloading
- Create overloads for binary operators
- Create overloads for unary operators, including the typecast
- Discuss the specific challenges faced when overloading the assignment operator
- Explain the relationship between the assignment operator and the copy constructor and how they can be implemented together
- Explain how a bracket overload can make an object look like an array
- Overload the operators necessary to make a NUL-terminated string class look like a string object in other programming languages.

7.1: Overloading an Operator

Walkthrough available in Overload.avi

In this section, we'll consider why operator overloading can be a useful programming technique, then consider the mechanics involved with such overloading. The functional form we adopted for describing operators in Chapter 2 will facilitate our discussion of the latter topic.

7.1.1: Why Overload Operators?

To set up the operator overload problem, let's define a very simple class called MyInt that has a single data member—an integer. This class is presented in Example 7.1, along with a main() function that attempts to add two MyInt objects together.

Example 7.1: MyInt class and attempt to apply it in main()

```
class MyInt
{
public:
	MyInt(void){m_nInt=0;}
	MyInt(int nVal){m_nInt=nVal;}
	~MyInt(void){};

	int Value() const {return m_nInt;}

protected:
	int m_nInt;
};

int main(int argc,char *argv[])
{
	MyInt m1(7),m2(5),m3;
	m3=m1+m2;
	return 0;
}
```

Not surprisingly, when we tried to compile this program, the compiler had some harsh words for us. These include:

```
C2784: 'std::reverse_iterator<_RanIt> std::operator +(_Diff,const
       std::reverse_iterator<_RanIt> &)' : could not deduce template argument for
       'const std::reverse_iterator<_RanIt> &' from 'MyInt'
C2784: 'std::_Ptrit<_Ty,_Diff,_Pointer,_Reference,_Pointer2,_Reference2> std::operator
       +(_Diff,const std::_Ptrit<_Ty,_Diff,_Pointer,_Reference,_Pointer2,_Reference2>
       &)' : could not deduce template argument for 'const
```

```
        std::_Ptrit<_Ty,_Diff,_Pointer,_Reference,_Pointer2,_Reference2> &' from
        'MyInt'
C2676: binary '+' : 'MyInt' does not define this operator or a conversion to a type
        acceptable to the predefined operator
```

(At least, they'd probably be harsh if we could make any sense of them at all.)

In effect, what the compiler is telling us is that even though we may "understand" what should happen when you add two MyInt structures together, it certainly does not. Fixing this type of problem is what operator overloading is all about.

If a one-word justification for the necessity of overloading operators in C++ were required, the word would be "strings". Up to this point, we have been forced to use a number of distinctly programmer-hostile functions every time we wanted to work with text data, including:

- *strcpy* or *strdup*, every time we really wanted to say String1=String2
- *strcmp*, every time we wanted to test if String1>String2 or String1==String2 or String1<=String3 and so forth…
- *strcat*, every time we really wanted to write String1+String2

Most languages supply native support for string objects (as opposed to NUL-terminated strings)—meaning these operators can be applied to strings as part of the language. In C++, however, we need operator overloading to achieve the same effect.

The benefits of operator overloading are not entirely limited to implementing strings, however. Our ability to overload the assignment operator, for example, can be used to fine-tune how we manage memory and how our classes can be used. These benefits, however, are mainly relevant to advanced programming.

The "big picture" is that if a language supplies a nice set of built-in operators for handling strings, it probably doesn't need to support operator overloads.

7.1.2: Regular Operator Overloads

Regular operator overloads—overloads that are not defined within a class—use the same functional form we introduced in Chapter 2. For instance, in the MyInt class of Example 7.1, it is reasonable that when we add two MyInt objects together, we'd like to return a third object with the m_nInt members of the two original objects added together. Written as a function, this would look like:

```
MyInt operator+(const MyInt &v1,const MyInt &v2) {
        MyInt v3;
        v3.m_nInt=v1.m_nInt+v2.m_nInt;
        return v3;
    }
```

What we are saying here is the following:

- When we add two MyInt objects together, a third MyInt object will be returned
- The addition does not change the arguments on either side (which is why it makes sense to make them const references—although just making copies also would have worked)
- The returned MyInt object will have an m_nInt member equal to the sum of the m_nInt members of the two arguments.

Now, at this point it is reasonable to be a bit skeptical. After all, should we be using the + operator (inside the function) when the whole purpose is to overload the + operator? The answer is a resounding yes. The + operator used inside the function adds two elements that hold integer values—and, of course, adding integers is a capability that is built into C++. The + operator we just overloaded, on the other hand, allows us to add two MyInt objects—something that C++ would be clueless about otherwise.

It gets even better, however. Since working with MyInt objects might be a bit of a pain, maybe we should be able to add integers to them directly. Doing so, It turns out, is easy. Just overload the + operator a few more times, mixing int values and MyInt values, as shown in Example 7.2.

> *Test your understanding:* Why do the three operator overloads need to be defined as friend functions?

The fact that three overloads, and not four, need to be defined when mixing MyInt and primitive data elements is worth considering. Looking at the three overloads, we see:

```
    MyInt operator+(const MyInt &v1,const MyInt &v2);
    MyInt operator+(const MyInt &v1,int n2);
    MyInt operator+(int n1,const MyInt &v2);
```

Looking at these, we see the missing combination is:

```
    MyInt operator+(int n1,int n2);
```

The problems that such allowing such a fourth overload would cause should be immediately evident. If allowed, everywhere in your program where two integers were added, a MyInt object would immediately be produced. This would get ugly fast.

Example 7.2: MyInt with overloaded + operators and main() function

```
class MyInt
{
public:
	MyInt(void){m_nInt=0;}
	MyInt(int nVal){m_nInt=nVal;}
	~MyInt(void){};

	int Value() const {return m_nInt;}
	friend MyInt operator+(const MyInt &v1,const MyInt &v2) {
		MyInt v3;
		v3.m_nInt=v1.m_nInt+v2.m_nInt;
		return v3;
	}
	friend MyInt operator+(const MyInt &v1,int n2) {
		MyInt v3;
		v3.m_nInt=v1.m_nInt+n2;
		return v3;
	}
	friend MyInt operator+(int n1,const MyInt &v2) {
		MyInt v3;
		v3.m_nInt=n1+v2.m_nInt;
		return v3;
	}

protected:
	int m_nInt;
};

int main(int argc,char *argv[])
{
	MyInt m1(7),m2(5),m3,m4,m5;
	m3=m1+m2;
	m4=m3+9;
	m5=2+m4;
	return 0;
}
```

In fact, C++ prevents you from doing a number of overloads. These include:

- Overloads that would change the behavior of any of the built-in operators when applied to primitives
- Overloads involving operators that aren't predefined in C++ (i.e., you can't invent your own operators)
- Overloads that would really screw up the language, such as the . operator for selecting members and the scope resolution operator (::).

In addition, you can't alter operator precedence. This can make some overloads inconvenient—requiring strange use of parentheses to apply the overloaded operator.

For the most part, the operators most commonly overloaded as regular functions are:

- Insertion (<<) and extraction (>>) in the context of sending objects to and from streams
- Test operators (particularly == and !=) used to compare objects from the same or related classes

Examples of both types of overload for the MyInt class are presented in Example 7.3. The insertion/extraction overloads allow a MyInt object to be saved to a SimpleIO stream using the << operator, or read from the stream using a >> operator. Because we use the same operators for normal I/O in C++, overloading these operators makes perfect sense. Similarly, the ability to compare two objects for equality applies to most objects.

Example 7.3: MyInt stream and test overloads

```
fstream &operator<<(fstream &out,const MyInt &val)
{
      out.write((const char *)&val.m_nInt,sizeof(int));
      return out;
}

fstream &operator>>(fstream &in,MyInt &val)
{
      in.read((char *)&val.m_nInt,sizeof(int));
      return in;
}

bool operator==(const MyInt &v1,const MyInt &v2) {
      return (v1.m_nInt==v2.m_nInt);
}
bool operator==(const MyInt &v1,int n2) {
      return (v1.m_nInt==n2);
}
bool operator==(int n1,const MyInt &v2) {
      return (n1==v2.m_nInt);
}
```

Other operators (such as arithmetic operators) can—of course—be overloaded. The main issue here is one of whether or not the overload reduces, or adds to, confusion. What would it mean, for example, to overload the multiplication operator for two employee objects? Bottom line: operator overloading should be limited to situations where there is a reasonably natural interpretation of what the overload is supposed to be doing.

7.1.3: Class Member Operator Overloads

In addition to using regular or friend functions to overload operators, it is also possible to overload operators as member functions. In fact, some operators (such as the assignment operator) need to be overloaded in this way.

When defining member overloads, one less argument is normally required in the function header—since the object being operated on is implicitly available. For example, if we wanted to make our subtraction (-) operator overload a class member, it could be defined as follows:

```
MyInt operator-(const MyInt &v1) {
        MyInt v3;
        v3.m_nInt=m_nInt-v1.m_nInt;
        return v3;
}
```

In this case, in the following code fragment:

```
MyInt a1(5),a2(2),a3;
a3=a1-a2;
```

the subtraction would effectively be equivalent to the call:

```
a3=a1.operator-(a2);
```

In fact, the expression a1.operator-(a2) will compile and run properly (amazingly enough). Using the functional form instead of the operator form in actual code would, however, would be considered very odd programming practice.

One disadvantage of making binary operators a class member is that it limits your ability to mix data types on either side. For example, while we defined three overloads for our + operator, our – operator is limited to two overloads:

```
MyInt MyInt::operator-(const MyInt &v1);
MyInt MyInt::operator-(int n1);
```

For this reason, we normally define binary member functions outside of the class.

Increment/Decrement Operators
As first mentioned in Chapter 2, the increment and decrement operators are a bit tricky because they can appear on either side of the object they are applied to. For example:

```
x1++
++x1
```

To address this problem, the postfix version of the overload function is given an extra argument (of type int) for the sole purposes of distinguishing it from the prefix version. This is illustrated in the overload definitions shown in Example 7.4.

Example 7.4: Prefix and Postfix increment operator overloads

```
MyInt &MyInt::operator++() {
      m_nInt++;
      return *this;
}

MyInt MyInt::operator++(int){
      MyInt r;
      r.m_nInt=m_nInt;
      m_nInt++;
      return r;
}
```

The other difference between prefix and postfix overloads is that the prefix overload can return a reference to the original object (since the return value is made available after the increment is taken place), which requires the use of the **this** pointer. It also allows a prefix increment to be used on the left hand side of an assignment, as shown in last line of the code fragment below:

```
MyInt i1(10),i2(10),i3,i4,i5(25);
i3=i1++;
i4=++i2;
++i5=i1;
```

The fact that the code is legal, however, does not mean that it is particularly useful, as the assignment will wipe out the value just incremented—as well as rendering the code completely opaque.

Test your understanding: What are the values for the m_nInt members of i1, i2, i3, i4, and i5 going to be after the code fragment above executes?

Typecast operator

Another operator overloaded as a member is the typecast operator. The syntax for the overload is a bit odd, as what follows the operator keyword also serves as the return type. For example, we could overload the (int) typecast for MyInt (within the class declaration) as follows:

```
operator int () const {
      return m_nInt;
}
```

With this typecast in place, we could use a MyInt object virtually anywhere that an int would normally be used. For example:

```
    MyInt t1(50);
    int nVal=t1;
```

would compile and, when run, would result in nVal being set to 50.

Test your understanding: Explain how nVal becomes 50 in the above code fragment.

Parenthesis (function) operator
A particularly exotic operator that can be overloaded within a class is the parenthesis () operator, also known as the function operator. An example of such an overload is shown in the class below:

```
class MyLess {
public:
    bool operator () (const string &s1,const string &s2) const {
        return (stricmp(s1.c_str(),s2.c_str())<0);
    }
};
```

In this example, the MyLess class overloads the parenthesis operator for two strings. When the operator is invoked, it returns **true** if the first string is less than the second (case-insensitive), **false** otherwise.

The parenthesis overload is invoked by placing parentheses with the proper arguments after an object of the specified class. Consider the following code fragment, for example:

```
    string s1("hello"),s2("World");
    MyLess Lobject;
    if (Lobject(s1,s2)) cout << s1.c_str() << " is less than " << s2.c_str() << endl;
    else cout << s1.c_str() << " is greater than or equal to " << s2.c_str() << endl;
```

The expression Lobject(s1,s2) calls the overloaded () operator, returning true if s1 is less than s2 (case insensitive) and false if s1 is greater than or equal to s2.

At first glance, the practical utility of a () overload is not immediately obvious. As we explore template collections—in Chapters 14 and 15, in particular—we will discover that this technique can be used to modify the behavior of various collections, implementing case-insensitive sorting, for example.

7.1 Section Questions

1. Explain the two ways you could rewrite the + overload of two MyInt objects so it did not have to be a friend.
2. Why must you understand classes before operator overloading becomes useful?
3. What operator overloads should be applicable if it makes sense to sort a class (e.g., to perform binary search)?
4. Why wasn't the operator== overload, presented in Section 7.1.2, declared as const?
5. Explain why the **this** pointer may need to be used in some operator overloads.
6. Why is it generally better to pass const references as arguments for operator overloads than actual objects?

7.2: Assignment Operators and Copy Constructors

Walkthrough available in CopyAndAssign.avi

There are many situations in a C++ where a copy of an object is made. Examples include:

- When an assignment operation takes place
- When an object is created by initializing it to another object
- When an object is passed into a function
- When a function returns an object
- When an object is composed within another object being copied or assigned

Because copying is so common, the compiler automatically creates a copy constructor and an assignment operator for a class in most circumstances. The problem is, the versions it creates are often so flawed that they are counterproductive. As a result, the programmer will often need to write copy constructors and assignment overloads.

7.2.1: A Copying Problem

To illustrate the problem caused by improperly defined copying, we'll design a simple class to manage a NUL-terminated string that will become the basis of the Lab Exercise in Section 7.4. This class is presented in Example 7.5.

Example 7.4: The GString00 class

```
class GString00
{
public:
	GString00(void) {
		m_pBuf=0;
	}
	GString00(const char *szVal) {
		m_pBuf=0;
		Set(szVal);
	}
	~GString00(void) {
		delete [] m_pBuf;
	}
	void Display() const {
		if (m_pBuf) cout << m_pBuf;
		else cout << "(Null)";
	}
	void Set(const char *szVal) {
		delete [] m_pBuf;
		m_pBuf=new char[strlen(szVal)+1];
		strcpy(m_pBuf,szVal);
	}
protected:
	char *m_pBuf;
};
```

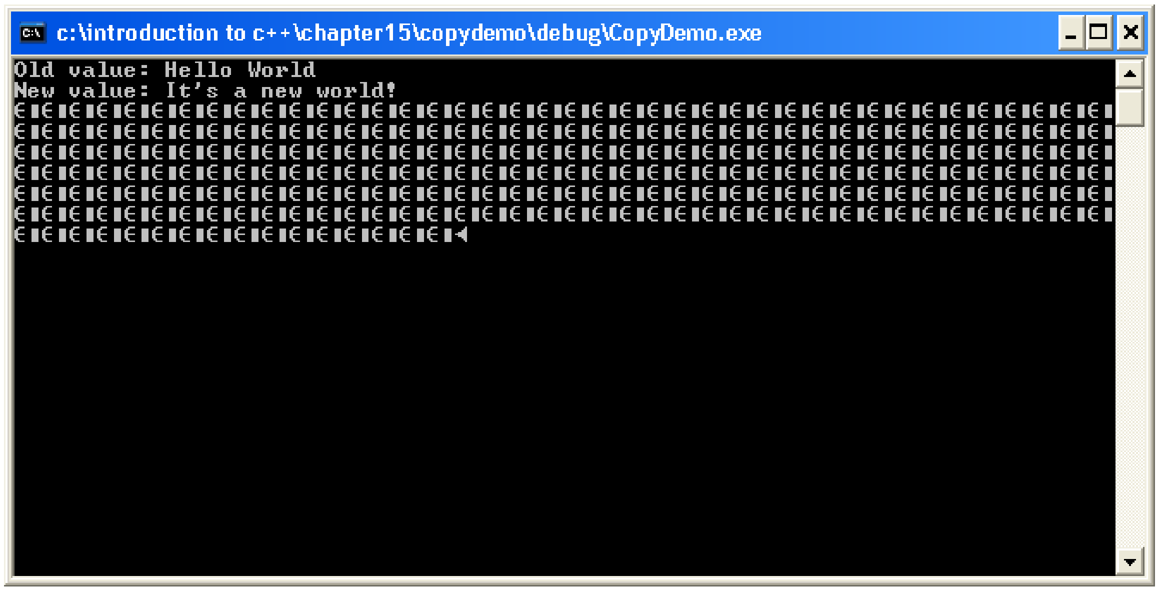
The class manages a character pointer buffer, m_pBuf, that points to a NUL terminated string. It consists of the following:

- Two constructor functions, a default constructor and one that initializes the string.
- A destructor function that releases the buffer when the object is destroyed
- A Display() member that displays the string value
- A Set() member that deletes the existing buffer, then allocates memory for a new one

At face value, the class seems perfectly reasonable. But run the code in Example 7.5 and the output in Figure 7.1 appears, followed immediately by an assertion failure. What is the problem?

Example 7.5: Main and TestSet() function

```
int main(int argc,char *argv[])
{
	GString00 str("Hello World");
	TestSet(str,"It\'s a new world!");
	str.Display();
}

void TestSet(GString00 str,const char *pStr)
{
	cout << "Old value: ";
	str.Display();
	cout << endl;
	str.Set(pStr);
	cout << "New value: ";
	str.Display();
	cout << endl;

}
```



c:\introduction to c++\chapter15\copydemo\debug\CopyDemo.exe

```
Old value: Hello World
New value: It's a new world!
€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█
€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█
€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█
€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█
€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█
€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█
€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█€█◄
```

Figure 7.1: Output from Example 7.5

The behavior of the program initially seems to defy the laws of logic. Stepping though it in the debugger, you'd find that the TestSet() function seems to work perfectly (as is evident from the output in Figure 7.1) and yet when we return from the function, the str variable has been corrupted. This is particularly surprising since when we called TestSet(), str was passed in by value—so nothing we did inside the function should have affected it.

Welcome to the world of bad copies! What actually happened is as follows:

- When the function was called, since str was passed in by value—and since we did not define a copy constructor—the implicit copy constructor was used to make a copy for use within the function. The implicit copy was, byte-for-byte, the same as str, meaning that we then had two copies of the GString00 object that pointed to the same buffer.
- While in the function everything worked fine except for one thing: when we called SetAt() we deleted the old buffer (which, unfortunately was the buffer that the str in main() still pointed to).
- When we called str.Display() in main, it pointed to the corrupted buffer—leading to the peculiar display and the assertion failure.

At this point, we could easily blame our call to Set() within the TestSet() function. To see if this was really the problem, we comment it out and rerun the code with the functions shown in Example 7.6.

Example 7.6: Revised Test Code

```
int main(int argc,char *argv[])
{
      GString00 str("Hello World");
      TestSet(str,"It\'s a new world!");
      str.Display();
}

void TestSet(GString00 str,const char *pStr)
{
      cout << "Old value: ";
      str.Display();
      cout << endl;
//    str.Set(pStr);
//    cout << "New value: ";
//    str.Display();
//    cout << endl;
}
```

The result? Exactly the same output (excepting the "It's a new world!" part) followed by the same crash! This seems really odd, since we didn't call any function that could have modified str. Or, at least, that's what we think…

The explanation is as follows:

- As noted previously, the local copy of str in TestFunction() pointed to the same buffer as the version of str in main.
- When we returned from TestSet(), the destructor was called on the local object—as always happens when local variables go out of scope. The result: our hapless buffer was deleted by the destructor function, instead of by Set().
- When we returned to main(), str again pointed to memory that had been released.

So, how do we solve this problem? We need to write our own copy constructor—one that makes a duplicate copy of the buffer instead of duplicating the address in m_pBuf. For reasons we'll see shortly, it is convenient to do this with two functions (although it could easily be done in one). These functions are shown in Example 7.7.

Example 7.7: Copy constructor and Copy() member

```
GString00(const GString00 &str) {
      m_pBuf=0;
      Copy(str);
}
void Copy(const GString00 &str) {
      Set(str.m_pBuf);
}
```

As soon as the compiler sees a constructor function with a single reference to a const object of the type being constructed, its implicit copy constructor is discarded. What *our* copy constructor does is to set m_pBuf to 0 (to ensure we don't try to delete an uninitialized pointer) then calls Set(), which makes a copy of the buffer. When we pass an object with this copy constructor into SetTest(), the copy within SetTest() has its own buffer—a copy of the original buffer—to play with, so nothing that happens within the function will affect the str object in main().

So, when do you need to define your own copy constructor? The general rule for implicit copy constructors is as follows:

- If composed objects have their own copy constructors, these are used
- If they do not, a byte-by-byte copy is made

There are also some hard-to-keep-track-of circumstances under which implicit copy constructors are not generated (e.g., if any composed member has a private or protected copy constructor).

For practical purposes, almost any time your class members call **new** and **delete** to manage embedded data, if you want a copy made, you'll need to write your own copy constructor. And, chances are, you will need one:

- If you intend to pass objects into functions or return them from functions (pointers and references don't require copy constructors, however).
- If you intend to place them in collections, such as the vector template, discussed in Chapter 16.
- You intend to initialize them with other objects
- You intend to assign them to other objects using an assignment operator overload

We now turn the last of these.

7.2.2: Overloading the Assignment Operator

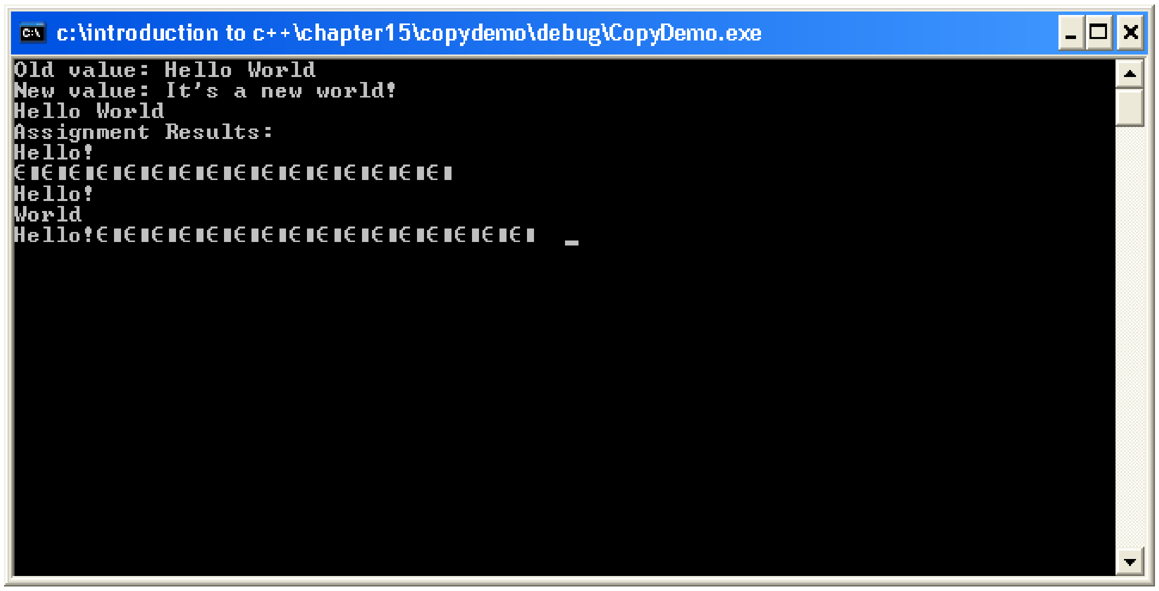
Since assignment and copying are very similar, it stands to reason that once a copy constructor has been defined, the assignment operator definition will be very similar. In fact, if you've put the major copying work into some function (that we have chosen to call Copy() here), you can just call that function from both places. In fact, it is tempting to define the GString00 assignment operator as follows:

```
const GString00 &GString00::operator=(const GString00 &rhs){
        Copy(rhs);
        return *this;
}
```

This particular function will work nearly all of the time. But one problem can make it fail. Consider, for example, the code in Example 7.8, which produced the output in Figure 7.2.

Example 7.8: main() and TestAssign() functions

```
int main(int argc,char *argv[])
{
      GString00 str("Hello World");
      TestSet(str,"It\'s a new world!");
      str.Display();
      TestAssign();
}

void TestAssign()
{
      GString00 a1("Hello!"),a2("World"),a3,a4,a5;
      GString00 *pa=&a1,*pb=&a2;
      a3=a1;
      a4=a2;
      a5=*pa;
      *pb=a2;
      cout << endl;
      cout << "Assignment Results:";
      cout << endl;
      a1.Display();
      cout << endl;
      a2.Display();
      cout << endl;
      a3.Display();
      cout << endl;
      a4.Display();
      cout << endl;
      pa->Display();
      pb->Display();
}
```


c:\introduction to c++\chapter15\copydemo\debug\CopyDemo.exe

```
Old value: Hello World
New value: It's a new world!
Hello World
Assignment Results:
Hello!
€▮€▮€▮€▮€▮€▮€▮€▮€▮€▮€▮€▮€▮€▮€▮€▮
Hello!
World
Hello!€▮€▮€▮€▮€▮€▮€▮€▮€▮€▮€▮€▮€▮€▮€▮€▮  _
```

Figure 7.2: Assignment failure output when running code in Example 7.8

Tracing the two bad output lines, we find the problem is in a2 and pb. As it happens, these refer to the same object. With the line of code:

```
*pb=a2;
```

we were, in effect, writing a2=a2 (which would have led to the same problem). The source of the difficulty is as follows:

- When an object is assigned to itself, the Set() function is called with a pointer to the same buffer that is to be modified.
- That buffer is immediately deleted, which corrupts it
- When memory is allocated, we are just making a copy of corrupted memory of indeterminate length—which is not good.

There are two ways of handling this problem, a thoughtful way and a simple mechanical way that works for virtually every assignment overload. The "thoughtful" way involves changing the Set() function so the copy is made before the deletion occurs. Unfortunately, it does not generalize well to other assignment situations.

The simple mechanical way involves testing to see if a self-assignment is being made and, if it is, simply returning. This could be implemented as follows:

```
const GString00 &GString00::operator=(const GString00 &rhs){
if (this==&rhs) return *this;
        Copy(rhs);
        return *this;
}
```

Since the & operator, applied to a reference, gives the address of the referenced object, we are not self-assigning *unless* the address of the reference matches the **this** pointer. Furthermore, if a Copy() function has been written, this particular overload can be used for almost any class. And, of course, with the modification our test code now runs flawlessly.

7.2.3: General Framework for Assignment and Copy Constructors

Combining what we found in 7.2.1 and 7.2.2, we can propose a general skeleton that can be used for virtually all assignment/copy constructor overloads:

- Write a function Copy() that does the appropriate element-by-element copying. Its prototype should be as follows:

```
void Copy(const ClassName &ref);
```

- Use the following framework for the copy constructor:

```
ClassName::ClassName(const ClassName &ref)
{
    // Initialize any pointers that might be deleted to 0
    Copy(ref);
}
```

- Use the following framework for the assignment operator overload

```
ClassName &ClassName::operator=(const ClassName &ref)
{
    if (this==&ref) return *this;
    Copy(ref);
    return *this;
}
```

Finally, if you *don't* want a copy constructor defined (e.g., because you see no need for you complex object ever to be copied), place an empty copy constructor as a private or protected member,. Doing so prevents an implicit copy constructor from being used (and the compiler will give you an error if you accidentally try to copy your object).

7.2 Section Questions

1. Would the problems initially encountered with TestSet() have occurred if we had passed str in as a reference?
2. If constructor problems are encountered, can they be solved by eliminating the delete operations in the destructor function?
3. What is the principal pitfall of implementing assignment overloads?
4. What is the main benefit of having an assignment overload return a reference to the object being assigned?
5. What makes understanding assignment overloads particularly important?
6. Why does it make sense to always overload both the assignment operator and copy constructor if you are going to overload either one?

7.3: A Quick Introduction to C++ I/O

One of the most common types of operator overloads is the extraction (>>)/insertion (<<) operator overloads used for I/O. This section presents a "quick and dirty" summary of the practical information required to use the C++ for text and binary I/O. A systematic look of the STL I/O system is presented in Chapter 11.

7.3.1 Opening a C++ File Stream

C++ file I/O is accomplished using various I/O objects. These objects are declared exactly the way we declare any other types of objects.

Text Streams
There are two types of objects that we will most commonly use for text I/O in C++ I/O:

ifstream – Input file streams
ofstream – Output file streams

If we want to use these objects in our code, we need to place:

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

at the top of our .cpp file.

The process of declaring a file object closely corresponds to the way that we define any other object. Once a file object is created, we can then open it, applying the same . operator we learned about for working with structures to the open(). For example:

```
    ofstream myout;
    myout.open("SampleFile.txt");
```

After this code fragment, the myout object would be an open file stream to which we could send output.

Failure of file member functions can be tested using another member function: fail(). For example:

```
    ifstream myin;
    myin.open("SampleInput.txt");
    if (myin.fail()) {
        cerr << "The file did not open…" << endl;
        // take appropriate action
    }
```

The is_open() member can also be used to check if a file stream has been opened successfully.

To close a C++ stream, the close member can be used. For example:

```
        myin.close();
```

In many cases, however, C++ streams don't have to be closed explicitly. The reason is that when a C++ object goes out of scope, the associated file is automatically closed. For example, consider the block of code below:

```
    {
        ofstream myout;
        myout.open("SampleFile.txt");
        myout << "Hello, World!" << endl;
    }
```

In this example, which writes "Hello World\n" to a text file named SampleFile.txt, myout does not need to be closed explicitly, since—upon leaving the block—myout goes out of scope and therefore closes automatically. As a consequence of this, the close() member is normally used when a) we want to reopen another stream using an existing object, or b) we are concerned that another process might want to access the open file, so we need to close it as quickly as possible.

C++ predefines a number of text stream objects that are automatically available:

cout – an *ostream* object used for standard output
cin – an *istream* object used for standard input
cerr – an *ostream* object (that defaults to standard output) used for error messages

These objects (*istream* and *ostream*) actually reside one level above the *ofstream* and *ifstream* in the C++ I/O hierarchy. This proves to be unimportant to programmers, since anything they can do with their own *ofstream* and *ifstream* objects can also be done with *ostream* and *istream* objects (except opening and closing the stream).

Binary Streams
There are three types of objects most commonly used for binary I/O C++, our first two streams plus a third:

ifstream – Input file streams
ofstream – Output file streams
fstream – Input/Output file streams

The process of opening an *fstream* object is slightly more complicated because we need to be specific about the mode used to open in. This is done by specifying a second argument to the open() member function. C++ uses defined constants to specify opening mode, that are bitwise or'd together. Common combinations of these constants (which must be written in their full form, i.e. ios_base::*constant_name*) are presented in Table 7.1.

C++ equivalent	Meaning
ios_base::in	Existing file is to be opened for input
ios_base::in | ios_base::out	Existing file is to be opened for input and output
ios_base::out | ios_base::trunc *or* ios_base::out	File is to be opening for writing and erased if it exists
ios_base::out | ios_base::in | ios_base::trunc	File is to be opening for reading and writing and erased if it exists
ios_base::out | ios_base::app	File is opened for appending in write mode
ios_base::out | ios_base::in | ios_base::app	File is opened for appending in read and write mode
ios_base::binary	File is opened in binary mode, with no end of line translations

Table 7.1: Opening modes for C++

An example of opening an existing binary file for I/O operations would therefore be:

```
fstream myio;
myio.open("IOExample.bin", ios_base::in | ios_base::out | ios_base::binary);
if (myio.fail()) {
    // Error handling
}
```

The second open() argument can also be specified for ifstream and ofstream objects (it defaults to ios_base::in and ios_base::out, respectively, if not specified). When specifying a second argument for these objects, the programmer must make sure that the argument is consistent with the stream type (i.e., don't open an ifstream object with an ios_base::out argument, or the operation will fail).

7.3.2 C++ Text I/O

Text I/O using the predefined objects **cin**, **cout** and **cerr** can be performed using the >> (insertion or input) operator and << (extraction or output) operators for all the primitive C++ types. The usage is identical when using file streams for text I/O.

Text Output
The operator overloads for output are roughly of the form:

```
ostream &operator<<(ostream &out, data-type val);
```

The ostream object can be 1) a predefined output object (such as cout and cerr), 2) an open ofstream object, or 3) an fstream object that has been opened with ios_base::out. Allowable *data-type* arguments include all the primitive data types (e.g., char, int, short, long, float and double) and NUL-terminated strings. In addition, the **endl** manipulator can be used to for a safe newline, e.g.,

```
cout << endl;
```

Because the operator returns another stream reference, output can be chained together, e.g.,

```
cout << "The value of pi is " << 3.14159 << endl;
```

Many formatting techniques for text output are available. These are covered in greater detail in Chapter 11. For the time being, however, when formatting is desired, we can use the sprintf() function to achieve formatting, e.g.,

```
double pi=3.14159;
char buf[80];
sprintf("The value of pi to 4 significant digits is %7.4lf",pi);
```

```
        cout << buf << endl;
```

Text Input
The operator overloads for input are roughly of the form:

```
        istream &operator<<(istream &out, data-type &val);
```

The istream object can be 1) a predefined input object (e.g., cin), 2) an open ifstream object, or 3) an fstream object that has been opened with ios_base::in. Allowable *data-type* arguments include all the primitive data types (e.g., char, int, short, long, float and double) and NUL-terminated strings. In addition, the **ws** manipulator can be used to "eat" leading white space characters, e.g.,

```
        int myInt;
        cin >> ws >> myInt;
```

Like the << operator, the >> operator can be chained.

When reading into character arrays, the >> operator assumes each string ends when white spaces is encountered. As a result—as we argued in Chapter 4—it is usually better to get a line of text from the user, rather than bringing in strings variable by variable. A member function available for all istream objects, called getline(), is available for this purpose. For example:

```
    buf[MAXSIZE];
    cin.getline(buf,MAXSIZE);
    // buf can now be processed to extract the necessary information
```

7.3.3 C++ Binary I/O

C++ binary I/O is normally performed on fstream objects. In SimpleIO, this corresponds to the STREAM data type.

Reading and Writing Data
C++ provides read() and write() members very similar to the SimpleIO ReadBlock() and WriteBlock() functions. These functions take the following arguments:

```
        .read(char *buf,int nCount);
        .write(const char *buf,int nCount);
```

There is only one major difference the C++ versions of the functions and their SimpleIO ReadBlock() and WriteBlock() counterparts: the C++ functions require a character buffer instead of a void *.

The difference in arguments is significant because it means that a typecast will have to be used any time a .read() or .write() function is called on non-character data. For example, to save and load an integer in binary format, the following code could be used:

```
int nTest1=25,nTest2;
ofstream out;
ifstream in;
out.open("TestOut.bin",ios_base::out | ios_base::trunc | ios_base::binary);
if (!out.fail()) out.write((char *)&nTest1,sizeof(int));
out.close(); // closing out because we want to reopen it for input
in.open("TestOut.bin",ios_base::in | ios_base::binary);
if (!in.fail()) in.read((char *)&nTest2,sizeof(int));
// nTest2 should now contain the integer 25
```

File Positioning Functions
C++ provides fstream positioning members equivalent to the SimpleIO SeekReadPosition() and SeekWritePosition() functions:

```
.seekg(unsigned int nPosition);  // Moves to specific position in input stream
.seekp(unsigned int nPosition);  // Moves to specific position in output stream
```

A failure of a seek operation can be tested with the fail() member function.

There are also three member functions that are commonly used to identify our current position in a file stream, all of which are applicable to both text and binary files:

- *unsigned int tellg()*: returns the current offset from the origin of the file for an input stream.
- *unsigned int tellp()*: returns the current offset from the origin of the file for an output stream.
- *bool eof()*: returns true value if we have reached the end of a file stream, and have tried to read past it.

To remember which version of each function is used on each type of stream, think of g for “getting” (i.e., input) and p for “putting” or “printing” (i.e., output).

7.3 Section Questions

1. What is the typical method we use to check if an I/O operation has been successful?

2. How is opening a C++ file stream object different from opening a file stream using SimpleIO?

3. If cin, cout and cerr are different object types than ifstream and ofstream, can we still use the same operators and functions to access our files as we do for the standard objects?

4. How does the buffer (i.e., memory area) used for binary I/O used in C++ differ from that used in SimpleIO? What are the practical implications of this difference for the programmer?

5. Why might .getline() be preferable to a series of input/extraction (>>) operators when acquiring complex data from a user?

6. When repositioning a file pointer in a binary file, what do you need to remember in C++ about how the file was opened?

7.4: Operator Overloading Demonstrations

In this section we will walk through two operator overloading examples. Both involve building on previously developed classes. The first is a simple insertion/extraction overload for employee objects—complicated by the lack of a copy constructor. The second involves the creation of a fully functional encapsulated void pointer array class—demonstrating an overload of the [] (bracket) operator.

7.4.1: Adding Operators to employee Objects

Walkthrough available in EmployeeOps.avi

In Chapter 14, Sections 14.6.1 and 14.6.2, we walked through the encapsulation of *employee* objects and an *EmpData* class to manage a collection of employees. In doing so, we glossed over some rather awkward code.

Assignment and Copy Constructor Overload

For example, in the EmpData::AddEmployee() member (originally presented in Example 14.21, presented below in Example 7.9), when we appended the new employee to the array, the code was the following:

```
emp.Id(nId);
emp.Input();
m_arEmp[m_nCount].Copy(emp);
m_nCount++;
bvalid=true;
```

Example 7.9: EmpData::AddEmployee function

```
void EmpData::AddEmployee()
{
	employee emp;
	bool bvalid=false;
	while(!bvalid) {
		unsigned int nId=PromptForId();
		if (FindEmployee(nId)>=0) {
			char buf[MAXLINE];
			DisplayString("ID that you entered already exists. Try again? (Y or
N): ");
			InputString(buf);
			if (toupper(buf[0])=='N') bvalid=true;
		}
		else {
			emp.Id(nId);
			emp.Input();
			m_arEmp[m_nCount].Copy(emp);
			m_nCount++;
			bvalid=true;
		}
	}
}
```

In the line:

```
m_arEmp[m_nCount].Copy(emp);
```

what we really wanted to write was:

```
m_arEmp[m_nCount]=emp;
```

To write the clearer version, however, we'd need an assignment operator overload.

As it turns out, this is prodigiously easy to do, since we had the foresight to write a Copy() function already. All we need to do is follow the pattern presented in Section 7.2.3, as shown in Example 7.10.

Example 7.10: employee Copy Constructor and Assignment Overload

```
employee(const employee &emp) {
      Copy(emp);
}
const employee &operator=(const employee &emp) {
      if (&emp==this) return *this;
      Copy(emp);
      return *this;
}
```

Since the employee class, as implemented in Chapter 14, Section 14.6.1, stored names in arrays declared within the class, there aren't even any member pointers that need to be set to 0 in the copy constructor. (Overloading assignment and copy constructors for the EmpData class are left as part of an end-of-chapter exercise).

Insertion/Extraction Overloads

By making it a practice of overloading the >> and << operators for binary loading and saving of any objects we intend to serialize, we can eliminate the mental strain of remembering "What did we call that function"?

Insertion/Extraction overloads have to be done outside our class, since common practice dictates the stream object will always be on the left when these operators are applied (which means an class member overload would need to be done in the stream 's class, whatever that is).

> *Test your understanding:* Explain why a member overload, as opposed to a regular function overload, would have to be in the stream class.

As a first attempt, we simply try a naïve overload along the following lines:

```
fstream operator>>(fstream in,employee &emp) {
            emp.Load(in);
            return in;
}
fstream operator<<(fstream out,employee &emp) {
            emp.Save(out);
            return out;
}
```

When we try these, however, we get the following error message from the compiler:

```
error C2558: class 'std::basic_fstream<_Elem,_Traits>' : no copy constructor
available or copy constructor is declared 'explicit'
    with
```

```
        [
            _Elem=char,
            _Traits=std::char_traits<char>
        ]
```

The specifics of the message become (slightly) clearer after we've talked about C++ I/O in Chapter 11. For now, however, what should catch out eyes is the words "copy constructor". What the message says, in effect, that we can't copy an fstream object that is passed into a function or returned from a function.

How do we address this problem? If we look back to Example 7.3, we discover we've already got the answer. Since references don't involve making copies, we can pass in and return references. The modified overloads are:

```
fstream &operator>>(fstream &in,employee &emp) {
                emp.Load(in);
                return in;
}
fstream &operator<<(fstream &out,employee &emp) {
                emp.Load(out);
                return out;
}
```

What is particularly remarkable about this process is that we can do it having no idea how fstream objects are implemented.

7.4.2: The PtrArray class

Walkthrough available in PtrArray.avi

In this walkthrough, we create a PtrArray class that encapsulates a dynamically resizing array of void pointers and overloads the [] and assignment operators. The resulting class has an interface nearly identical to that of the CPtrArray that is provided as part of the Microsoft Foundation Classes. It also provides a useful starting point for understanding the vector<> template, introduced in Chapter 8.

Conceptually, the PtrArray class is illustrated in Figure 7.3. There are three data members in the class:

- ***m_nSize:*** Keeps track of the number of elements currently *active* in the array
- ***m_nCount:*** Keeps track of the number of elements currently *available* in the array. This number will always be greater than or equal to m_nSize. It is not the same, however, since we allocate new space for our array in blocks. This prevents

us from having to reallocate memory for the array every time we add, insert or remove elements.

- ***m_pData:*** An array of void pointers containing m_nCount elements.

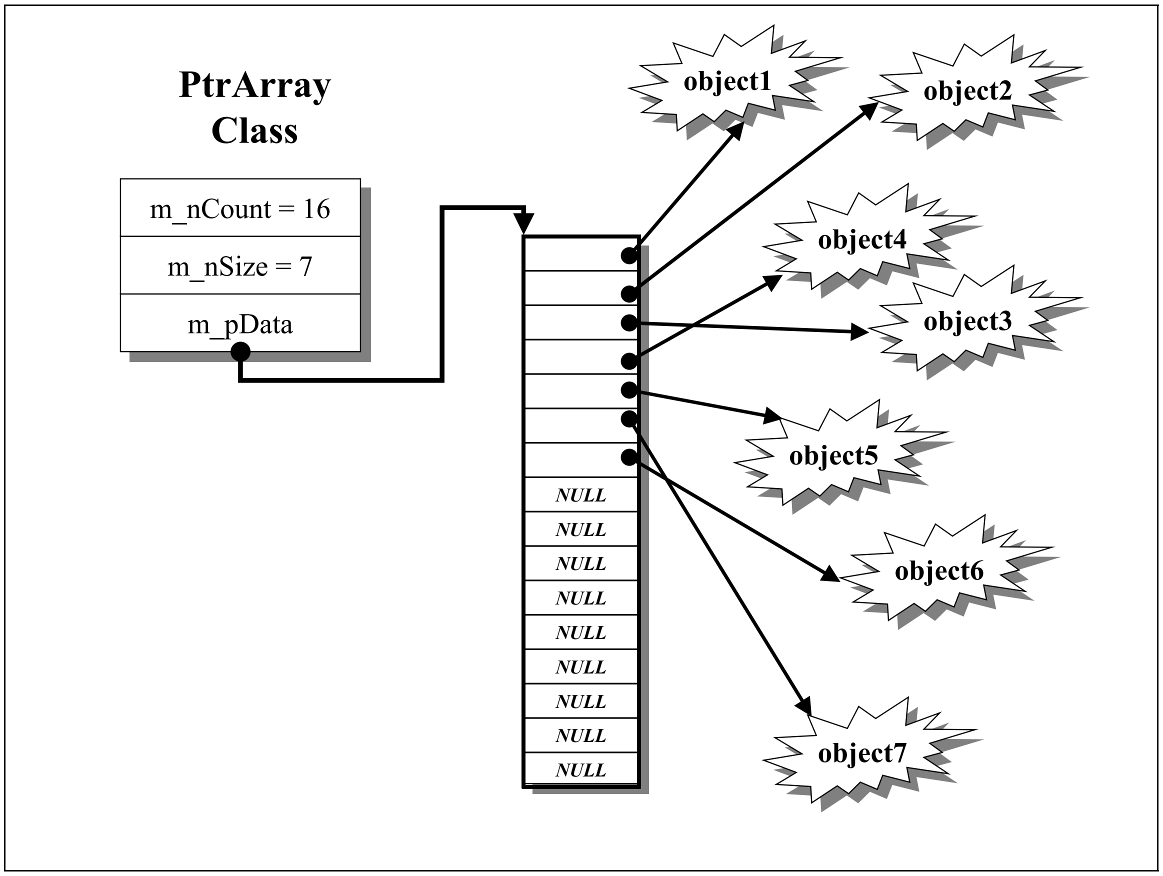


Figure 7.3: Conceptual view of PtrArray class

The typical operation of the PtrArray class is as follows:

- Elements are added to the array using Insert() or Add() members.
- Periodically, the number of active pointers (m_nSize) reaches the count of available pointers (m_nCount). At this time, a new void pointer array—larger than the old one by some specified block size—is allocated and the old pointers are copied over. The original array is then deleted and the address of the new array is placed in m_pData.
- The size of the pointer array can also be set in advance by calling SetSize()
- Elements are usually accessed using the [] operator, although the SetAt() and GetAt() member functions can also be used. Every time an element is accessed, the coefficient value is checked to ensure it is a legal coefficient—thereby eliminating the common C/C++ problem of accessing elements outside of array boundaries.

Declaration
The declaration of the class is presented in Example 7.11.

Example 7.11: PtrArray class declaration

```
class PtrArray
{
public:
      PtrArray();
      PtrArray(const PtrArray &ar);
      ~PtrArray();
      const PtrArray &operator=(const PtrArray &ar);
      void Copy(const PtrArray &ar);
      void *&operator[](int);
      void *&operator[](int) const;
      void *GetAt(int i) const;
      void SetAt(int i,void *pData);
      void Add(void *pData);
      void InsertAt(int i,void *pData);
      void SetSize(int i);
      int GetSize() const;
      void RemoveAt(int i);
      void RemoveAll();
protected:
      int m_nCount;
      int m_nSize;
      void **m_pData;
      void ReallocArray(int nNewSize);
};
```

Interface
The PtrArray interface includes the following:

- A default and copy constructor, along with a destructor. A desrructor is critical, in this regard, because the class manages its own member (an array of void pointers)
- A Copy() function and assignment overload
- Two overloads of the bracket operator—for use in const and non-const operations
- A RemoveAll() function, to clean out the array
- Add() and InsertAt() members to add elements to the array
- A RemoveAt() member to remove elements from the array
- A GetSize() member to return the number of elements in the array
- A SetSize() member to specify the number of active pointers in the array—resizing if necessary
- A protected ReallocArray() function—meaning we don't want programmers to call it directly (which is fine, since the public SetSize() accomplishes the same purpose)

A demonstration of the interface is presented in Example 7.12.

Example 7.12: PtrArray demo function

```
void PtrArrayDemo()
{
      int i;
      char buf[80];
      class PtrArray ar;
      ar.Add("Hello");
      ar.Add("World!");
      ar.Add("me");
      for(i=0;i<ar.GetSize();i++) {
            sprintf(buf,"%s ",(char *)ar[i]);
            cout << buf;
      }
      cout << endl;
      ar.InsertAt(2,"It\'s");
      for(i=0;i<ar.GetSize();i++) {
            sprintf(buf,"%s ",(char *)ar[i]);
            cout << buf;
      }
      cout << endl;
      ar.RemoveAt(1);
      for(i=0;i<ar.GetSize();i++) {
            sprintf(buf,"%s ",(char *)ar[i]);
            cout << buf;
      }
      cout << endl;
      ar.SetAt(2,"I");
      for(i=0;i<ar.GetSize();i++) {
            sprintf(buf,"%s ",(char *)ar[i]);
            cout << buf;
      }
      cout << endl;
}
```

Implementation

The simplest of the member functions for the PtrArray class are those for data access, shown in Example 7.13. All data access is accomplished using the [] operators, either directly or by using the GetAt() and SetAt() members. Some general observations on these functions are as follows:

- We define both const and non-const versions of the overloads to permit [] to be used in const member functions (such as the GetAt() function).
- By forcing all access to occur using the [] operators, we limit the number of places where bounds chacking must be performed.
- The assert() function—which stops a debug version of the program—is used in the bounds checking. This is appropriate because attempting accesses outside of array boundaries are always indicative of a programming problem, rather than a user-induced problem (such as mistyped file name). The topic of appropriate error

handling techniques is discussed further in Chapter 12, which focuses on debugging.

Example 7.13: Member functions for data access

```
void *&PtrArray::operator[](int i) {
       assert(i>=0 && i<m_nSize);
       return m_pData[i];
}
void *&PtrArray::operator[](int i) const {
       assert(i>=0 && i<m_nSize);
       return m_pData[i];
}
// Returns the pointer value at position i
void *PtrArray::GetAt(int i) const
{
       return (*this)[i];
}
// Sets the pointer value at position i to pData
void PtrArray::SetAt(int i,void *pData)
{
       (*this)[i]=pData;
}
// Returns number of active pointers in the array
int PtrArray::GetSize() const {
       return m_nSize;
}
```

Test your understanding: Why do we use the syntax (*this)[i] in the GetAt() and SetAt() functions?

Example 7.14: New member functions in the PtrArray class

```
PtrArray::PtrArray(){
       m_pData=0;
       m_nCount=0;
       m_nSize=0;
}
PtrArray::PtrArray(const PtrArray &ar) {
       m_pData=0;
       Copy(ar);
}
PtrArray::~PtrArray() {
       RemoveAll();
}
const PtrArray &PtrArray::operator=(const PtrArray &ar) {
       if (&ar==this) return *this;
       Copy(ar);
       return *this;
}
void PtrArray::Copy(const PtrArray &ar) {
       RemoveAll();
       SetSize(ar.GetSize());
       for(int i=0;i<ar.GetSize();i++) {
              SetAt(i,ar[i]);
       }
}
```

Constructor, destructor and assignment overload functions are presented in Example 7.14. These follow the pattern introduced in this chapter perfectly.

Example 7.15: Member functions for adding and removing elements

```
// Adds pointer pData to the end of the array
void PtrArray::Add(void *pData)
{
       if (m_nCount==m_nSize) ReallocArray(m_nSize+1);
       else m_nSize++;
       if (m_nSize>=m_nCount) return; // If reallocation failed...
       SetAt(m_nSize-1,pData);
}
// Inserts the pointer pData at position i, moving existing element up
// Asserts i is legal
void PtrArray::InsertAt(int nPos,void *pData)
{
       int i;
       assert(nPos>=0 && nPos<=m_nSize);
       if (nPos<0 || nPos>m_nSize) return;
       if (m_nSize==m_nCount) ReallocArray(m_nSize+1);
       else m_nSize++;
       if (m_nSize>=m_nCount) return; // If reallocation failed...
       for(i=m_nSize-1;i>nPos;i--) {
              SetAt(i,GetAt(i-1));
       }
       SetAt(nPos,pData);
}
// Removes the pointer at position i, moving existing element down
// Asserts i is legal
void PtrArray::RemoveAt(int nPos)
{
       int i;
       assert(nPos>=0 && nPos<m_nSize);
       if (nPos<0 || nPos>=m_nSize) return;
       for(i=nPos;i<m_nSize-1;i++) {
              SetAt(i,GetAt(i+1));
       }
       m_nSize--;
}
// Sets the array to the specied size
void PtrArray::SetSize(int i)
{
       ReallocArray(i);
}
```

The most interesting functions in the class involve the addition and removal of elements. These are presented in Example 17.15. The two functions that involve element insertion (the Add() and InsertAt() members) both begin by checking to see if the array is full, calling ReallocArray() to resize it if necessary. The Add() function then places the element at the end of the array, incrementing m_nSize. The InsertAt() function, on the other hand, needs to move existing elements out of the way before putting the new pointer into the array. This is done with a loop that starts at the end of the array and moves down towards the insertion point.

RemoveAt() removes an element from the array. In moving existing elements, the loop used is somewhat similar to that used in the InsertAt() function—except it iterates from the insertion point to the end of the array, setting each element to the value that was previously one higher.

The final member functions in the class server to resize the actual void pointer array managed by the class. These are shown in Example 7.16.

Example 7.16: PtrArray resizing functions

```
void PtrArray::ReallocArray(int nNewSize)
{
	if (nNewSize==0) {
		RemoveAll();
	}
	else {
		int nRealSize=DYNBLOCKSIZE*((nNewSize+DYNBLOCKSIZE-
1)/DYNBLOCKSIZE);
		void **pNew=new void *[nRealSize];
		int nCopySize=(nNewSize<m_nCount) ? nNewSize : m_nCount;
		if (m_nSize>0) {
			for(int i=0;i<nCopySize;i++) {
				pNew[i]=m_pData[i];
			}
			delete m_pData;
		}
		m_pData=pNew;
		m_nCount=nRealSize;
		m_nSize=nNewSize;
	}
}

void PtrArray::RemoveAll() {
	delete [] m_pData;
	m_pData=0;
	m_nCount=0;
	m_nSize=0;
}
```

The ReallocArray() function always allocates pointers in blocks that are even multiples of DYNBLOCKSIZE (a defined constant). Once it computes what will become the new count (nRealSize), it allocates a new array of pointers (pNew). It then computes the number of pointers to be copied from the original array (the smaller of the new size and the old count) and copies those pointers from the old array over the new array. It then deletes old array and attaches the new array to m_pData.

RemoveAll() is even simplier, since it deletes the pointer array, then sets both the counter and the size members to 0. This member can be called by the programmer directly, but is also called in the PtrArray destructor function.

> *Test your understanding:* Explain why, from the class user's perspective, calling SetSize(0) is equivalent to calling RemoveAll().

7.5: Lab Exercise: The GString Class

Walkthrough available in GString.avi

Near the outset of the chapter, the assertion was made that if it weren't for strings, we probably wouldn't have operator overloads. In this section, the reader can develop a complete class that encapsulates a NUL-terminated string and provides a lot of bells and whistles. Indeed, in some ways its features compare favorably the to those of the STL **string** class that will be introduced in the next chapter.

7.5.1: Overview

Before the STL was widely adopted as the C++ standard, Visual C++ developers using strings in their applications frequently used CString objects, defined as part of the MFC. The GString class presented here is an attempt to develop a string implementation that closely mimics the CString class. There are two advantages of doing so:

- The MFC CString class is quite well designed, and provides an excellent vehicle for practicing operator overloads
- After completing the GString class, the reader will be completely conversant in the CString interface, should he or she ever decide to engage in MFC programming

Among the nice features of GString objects:

- Automatic assignment from NUL-terminated strings
- Automatic typecasting to constant NUL terminated strings
- Support for comparison operators (e.g., >, ==, !=) with other GStrings and NUL-terminated strings
- Ability to concatenate with other GStrings and NUL-terminated strings using the + operator
- Access of individual characters using the [] operator
- >> and << support for text output
- >> and << support for serializing to a binary stream
- Various substring generation members, such as Left(), Right() and Mid()

The class declaration is shown in Example 7.17.

Example 7.17: GString declaration

```
class GString
{
public:
	GString(){m_szBuf=0;}
	GString(const char *p);
	GString(const GString& sz);
	~GString();
	const GString &operator=(const GString &stringSource);
	const GString &operator=(const char *pszSource);
	void Copy(const GString &sz);

	operator const char * () const {return m_szBuf;}

	friend bool operator==(const GString& s1,const GString& s2);
	friend bool operator==(const GString& s1,const char *s2);
	friend bool operator==(const char *s1,const GString& s2);
	friend bool operator!=(const GString& s1,const GString& s2);
	friend bool operator!=(const GString& s1,const char *s2);
	friend bool operator!=(const char *s1,const GString& s2);
	friend bool operator<(const GString& s1,const GString& s2);
	friend bool operator<(const GString& s1,const char *s2);
	friend bool operator<(const char *s1,const GString& s2);
	friend bool operator>(const GString& s1,const GString& s2);
	friend bool operator>(const GString& s1,const char *s2);
	friend bool operator>(const char *s1,const GString& s2);
	friend bool operator<=(const GString& s1,const GString& s2);
	friend bool operator<=(const GString& s1,const char *s2);
	friend bool operator<=(const char *s1,const GString& s2);
	friend bool operator>=(const GString& s1,const GString& s2);
	friend bool operator>=(const GString& s1,const char *s2);
	friend bool operator>=(const char *s1,const GString& s2);

	friend ostream &operator<<(ostream &out,const GString &str1);
	friend istream &operator>>(istream &in,GString &str1);
	friend fstream &operator<<(fstream &out,const GString &str1);
	friend fstream &operator>>(fstream &in,GString &str1);

	int GetLength() const;
	char operator[](int i) const;
	char GetAt(unsigned int i) const;

	void Empty();
	bool IsEmpty() const;

	friend GString operator+(const GString& str1,const GString& str2);
	friend GString operator+(const GString& str1,const char *s2);
	friend GString operator+(const char *s1,const GString& str2);

	int Find(const char *str1) const;

	GString Mid(int i) const;
	GString Mid(int i,int j) const;
	GString Left(int i) const;
	GString Right(int i) const;

protected:
	char *m_szBuf;
	void Copy(const char *bufnew);
	void SubstituteBuf(char *bufnew);
	void Concat(const char *s1,const char *s2);

};
```

7.5.2: Interface and Implementation Issues

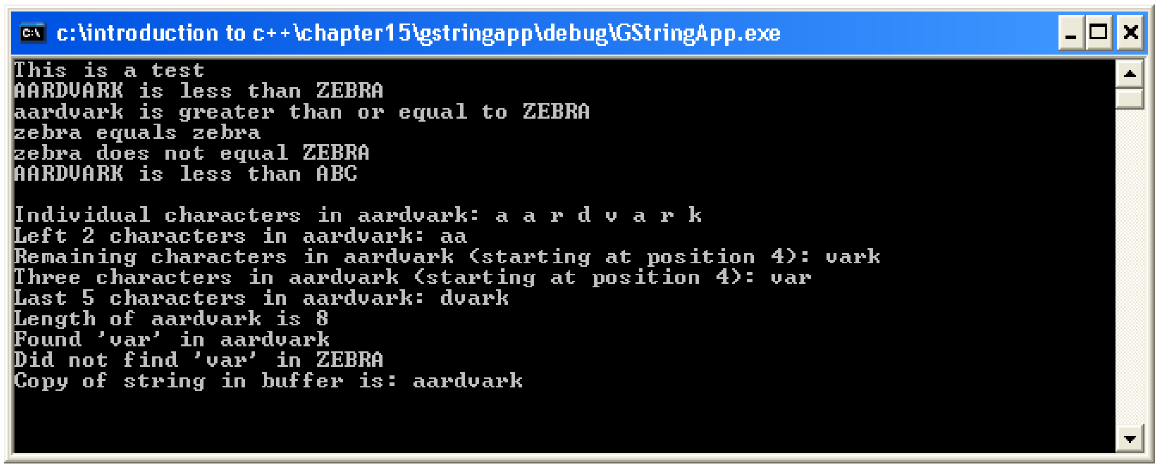
In this section, we consider the interface of the GString class, then address some implementation issues.

Interface
The GString interface is relatively straightforward. The GStringDemo function, presented in Example 7.18, demonstrates all but the I/O features.

Example 7.18: GStringDemo function

```
void GStringDemo()
{
        char buf[80]={0};
        GString s1,s2,s3,s4,s5,s6,s7,s8;
        s1="This is ";
        s2="a test";
        s3=s1+s2;
        cout << s3 << endl;
        s4="AARDVARK";
        s5="ZEBRA";
        s6="aardvark";
        s7="zebra";
        s8=s7;
        if (s4<s5) cout << s4 << " is less than " << s5 << endl;
        else cout << s4 << " is greater than or equal to " << s5 << endl;
        if (s6<s5) cout << s6 << " is less than " << s5 << endl;
        else cout << s6 << " is greater than or equal to " << s5 << endl;
        if (s7==s8) cout << s7 << " equals " << s8 << endl;
        else cout << s7 << " does not equal " << s8 << endl;
        if (s7!=s5) cout << s7 << " does not equal " << s5 << endl;
        else cout << s7 << " equals " << s5 << endl;
        if (s4<"ABC") cout << s4 << " is less than " << "ABC" << endl;
        else cout << s4 << " is greater than or equal to " << "ABC" << endl;
        // basic_string members
        cout << endl << "Individual characters in " << s6+": ";
        for(int i=0;i<s6.GetLength();i++) {
                cout << s6[i] << " ";
        }
        cout << endl << "Left 2 characters in " << s6+": " << s6.Left(2);;
        cout << endl << "Remaining characters in " << s6+
                " (starting at position 4): " << s6.Mid(4);
        cout << endl << "Three characters in " << s6+
                " (starting at position 4): " << s6.Mid(4,3);
        cout << endl << "Last 5 characters in " << s6+": " << s6.Right(5);
        // setting up not found constant
        cout << endl << "Length of " << s6 << " is " << s6.GetLength();
        int nPos=s6.Find("var");
        if (nPos!=-1) cout << endl << "Found \'var\' in " << s6;
        else cout << endl << "Did not find \'var\' in " << s6;
        nPos=s5.Find("var");
        if ((int)nPos!=-1) cout << endl << "Found \'var\' in " << s5;
        else cout << endl << "Did not find \'var\' in " << s5;
        strcpy(buf,s6);
        cout << endl << "Copy of string in buffer is: " << buf << endl;
        return;
}
```

The output of running GStringDemo() is presented in Figure 7.3.


c:\introduction to c++\chapter15\gstringapp\debug\GStringApp.exe

```
This is a test
AARDVARK is less than ZEBRA
aardvark is greater than or equal to ZEBRA
zebra equals zebra
zebra does not equal ZEBRA
AARDVARK is less than ABC

Individual characters in aardvark: a a r d v a r k
Left 2 characters in aardvark: aa
Remaining characters in aardvark (starting at position 4): vark
Three characters in aardvark (starting at position 4): var
Last 5 characters in aardvark: dvark
Length of aardvark is 8
Found 'var' in aardvark
Did not find 'var' in ZEBRA
Copy of string in buffer is: aardvark
```

Figure 7.3: Output after running GStringDemo function

Implementation Issues

Most of the functions required to implement the GString class are just a line or two long. A few, however, warrant some additional explanation. This is particularly true of the protected members, which—by their very nature—are defined largely at the discretion of the programmer.

friend ostream &operator<<(ostream &out,const GString &str1);
friend istream &operator>>(istream &in,GString &str1);
friend fstream &operator<<(fstream &out,const GString &str1);
friend fstream &operator>>(fstream &in,GString &str1);

The ostream and isteam streams are objects of the same type as **cout** and **cin**, as mentioned in Section 7.3.2. These overloads therefore provide us with text I/O. The fstream overloads should therefore load and save the data in binary form. (Hint: the LoadString() and SaveString() functions introduced in Chapter 4 accomplish the type of loading and saving required).

void Copy(const char *bufnew);

Deletes the existing GString buffer, then creates a copy of *bufnew* to assign to *m_szBuf*. It can be called from many other member functions (including the Copy(const GString) function), but is not necessary for the class user—since assignment can be used its place.

void SubstituteBuf(char *bufnew);

Works like the protected copy but simply assigns *bufnew* to *m_szBuf*, without copying it. You'd never want to make a function like this accessible to a programmer just using the class, but it can quite convenient when writing some of the substring and concatenation functions.

void Concat(const char *s1,const char *s2);

Sets the GString object's buffer to a newly concatenated string, consisting of s1 followed by s2. Would not serve a useful purpose for a class user, but can be used in all the + operator overloads.

7.5.3: Procedure

A reasonable procedure for programming the GString class would be the following:

- *Start with the final version of GString00 (from 7.2.2).* This will handle copying and assignment. You might also want to implement the protected Copy()—which can then be called from your public copy and many other functions.
- *Implement the typecast operator.* Use the operator in 7.1.3 as a guide—yours will be every bit as trivial.
- *Implement the comparison operators.* Done properly, all of these will be nearly identical and a single line of code. Don't hesitate to use library functions. By this time, you've probably got enough code to begin testing.
- *Implement the stream operators for text and binary I/O.* If it is helpful, you may assume that no string will be more than 32000 characters long. Use the SimpleIO functions unless you've already become familiar with Chapter 18.
- *Implement GetLength().* Another one-line call to a library function.
- *Implement the [] operator.* This operator should return a character from the GString object at the specified position. Make it return 0x00 if an invalid position is specified. The GetAt() function does the same thing.
- *Implement Empty() and IsEmpty().* Empty() releases a string, setting m_szBuf to 0. IsEmpty() tells if a string is empty, returning true if it is, false otherwise.
- *Implement Find().* This function returns the position where it finds a substring matching its argument in the object it is applied to. It returns –1 if no matching substring is found.
- *Look at the Chapter 13 end-of-chapter exercises, questions 1-10.* These discuss implementing structure versions of the remaining functions you'll be writing.
- *Implement the + operator.* In order to do this efficiently, you can define a protected Concat() function, such as that described in 7.5.2, that can be called from all three overloads.
- *Implement the substring members.* If you write the Mid() member first, it can be used inside the Left() and Right() members.

7.6: Review and Questions

7.6.1: Review

C++ allows you specify how its operators will work when applied to the classes that you create. This is a powerful capability that makes it possible to implement classes that behave like flexible strings and arrays.

Operator overloads are specified using the operator functional form introduced in Chapter 4. Thus, an overload of the + operator would tend to look like:

```
ClassType operator+(const ClassType &a1,const ClassType &a2);
```

The function body would then be called whenever an expression was encountered where two ClassType objects were separated by a plus sign. In general, it is better to use references than actual objects in defining operator overloads, since they involve less copying overhead when called and don't need an available copy constructor.

Operators can either be overloaded as regular functions (often specified as friend functions in the class to which they apply) or as member functions. When an operator is overloaded as a member, one less argument is specified—as the object to which the operator is applied is implicitly available to class members. For binary operators, this means the LHS of the operator needs to be an object of that class. This can be a limitation for binary operators that relate different types of values (e.g., const char * and string objects), so such operators tend to be defined as regular or friend functions.

There are some limitations on operator overloading. These include:

- Overloads that would change the behavior of any of the built-in operators when applied to primitives
- Overloads involving operators that aren't predefined in C++ (i.e., you can't invent your own operators)
- Overloads that would really screw up the language, such as the . operator for selecting members and the scope resolution operator (::).
- Overloads can't alter operator precedence.

Certain overloads require special syntax. Three important examples are:

- The postfix ++ and -- operators, which have an extra int argument that is not used in order to distinguish them from the prefix versions

- The typecast overload, where the operator name and return value are combined, as in:
  ```
  operator const char*()
  ```
- The (), or function, overload, where the parenthesis operators are overloaded, and the function body is invoked by placing the arguments, enclosed in parentheses, after an object variable.

The most critical (and common) operator overload is that of the assignment operator. Although the compiler normally creates an implicit version of the assignment operator, this version is typically "wrong" for classes that manage their own pointers and can cause quite subtle bugs. Part of the problem is that assignment and copying operations may not always be self-evident to the programmer—often occurring invisibly as functions are called and return and during the use of template classes.

Because the assignment operator and copy constructor are nearly identical, they are virtually always implemented together (and should be). A general framework for their implementation is as follows:

- Write a function Copy() that does the appropriate element-by-element copying. Its prototype should be as follows:

  ```
  void Copy(const ClassName &ref);
  ```

- Use the following framework for the copy constructor:

  ```
  ClassName::ClassName(const ClassName &ref)
  {
          // Initialize any pointers that might be deleted to 0
          Copy(ref);
  }
  ```

- Use the following framework for the assignment operator overload

  ```
  ClassName &ClassName::operator=(const ClassName &ref)
  {
          if (this==&ref) return *this;
          Copy(ref);
          return *this;
  }
  ```

A common place for operator overloads in is file I/O. The STL I/O library overloads the primitive C++ data types for text input and output as follows:

```
istream &operator>>(istream &in, data-type &dat);
ostream &operator<<(ostream &out, data-type dat);
```

Because these operators return the same stream reference that they are applied to, they can be chained together, e.g.,

```
        cout << "The value of pi is " << 3.14159 << endl;
```

Input streams can also be chained together. Text user input, however, is normally done with the getline() member of istream—which avoids having strings unexpectedly terminated by white characters.

The C++ STL does not overload any operators for binary I/O, which is typically accomplished with read() and write() member functions of the fstream object. It is fairly common, however, to overload file stream extraction and insertion operators for binary I/O (e.g., as is done in the MFC).

7.6.2: Glossary

Assignment operator – The operator used to assign a value to its LHS argument. It is nearly always overloaded in conjunction with a copy constructor.
Copy Constructor – A constructor function that initializes the object being constructed by copying the data in another object.
CString – An encapsulated string implementation provided with the MFC.
Extraction operator – Another name for the >> or input operator.
fstream – An STL file stream object, which can be opened either for input or output or both.
Function operator overload – A overload of the () operator within a class, allowing a parentheses placed after an object to be used to call a function.
INPUT – The name used by SimpleIO to refer to an input text stream.
Insertion operator – Another name for the << or output operator.
ifstream – An STL file stream object that is normally opened by the user input.
istream – A general input stream object, which can be attached either to a file stream or to a predefined standard input object such as **cin** and is normally used for text input.
Microsoft Foundation Classes (MFC) – A library of C++ classes provided by Microsoft as part of Visual Studio that is particularly well suited for developing MS-Windows applications.
ofstream – An STL file stream object that is normally opened by the user for output.
ostream – An STL general output stream object, which can be attached either to a file stream or to a predefined standard input object such as **cout** or **cerr** and is normally used for text input.
OUTPUT – The name used by SimpleIO to refer to an output text stream.
string Object – An object instantiating the STL string class. More generally, can refer to any encapsulated string object, such as MFC CString objects.
Standard Template Library (STL) – The most commonly used standard C++ library, offering various collection objects, strings and file stream encapsulations.

STREAM – The name used by SimpleIO to refer to an file stream, input or output, typically used for serializing data in binary form.
Typecast overload – An operator that is called when an object is typecast explicitly or implicitly (e.g., by being passed as an argument to a function).

7.6.3: Questions

Questions 7.1-7.2 involve enhancing the MyInt (Section 7.2.1) class

7.1. MyInt += operator. Overload the += operator within the MyInt class.

7.2. MyInt bool typecast. Write a typecast that casts a MyInt object to bool using the normal C++ rules for true and false.

Questions 7.3-7.7 involve enhancing the PtrArray (Section 7.3.2) and GString (Section 7.4) classes

7.3. Array of integers. Create a new class by modifying PtrArray class that holds an array of integers.

7.4. Array of strings. Create a new IntArray class by modifying PtrArray class that holds an array of GString objects.

7.5. Case insensitive strings. Create a new IString class by modifying GString class to make it case insensitive. Make sure all members are case insensitive.

7.6. GString minus operator. Overload the – operator for the GString class. It should takes its RHS and remove it, if found, from the LHS string, returning a new string with the RHS removed. Implementations for mixed of GString and const char * arguments should be supplied.

7.7. GString += and -= operators. Overload the += and -= operators (consistent with the + operators already defined and – operators specified in Question 7.6) for the GString class.

Questions 7.8-7.10 involve enhancing the employee and EmpData classes as originally presented in Chapter 14, Sections 14.6.1 and 14.6.2. It assumes the modifications in Section 7.3.1 have been made.

7.8. Overloading EmpData operators. Add a copy constructor, assignment operator and insertion/extraction (<< and >>) operator overloads to the EmpData class, making its interface consistent with the employee class. (*Hint:* you'll also want to define a Copy() member, naturally).

7.9. Replacing the EmpData embedded array. Replace the embedded array of employees in the EmpData class with a PtrArray member (as defined in 7.3.2). Implement the change so that the EmpData interface does not change.

7.10. Replace employee embedded name string arrays with GString objects. Change the employee class implementation so that the last name and first names are stored as GString arrays. The only interface changes you should make is to have the LastName() and FirstName() functions return GString objects, i.e.:

```
GString LastName() const;
GString FirstName() const;
```

Note: as long as the employee interface is maintained, you should not have to change EmpData.

Questions 11 through 14 involve overloading operators related to the Date, Time, DateTime and TimeSpan classes developed in Chapter 14, Questions 7.1-7.12.

7.11. Overload subtraction for the time class. Add a friend operator overload to the Time class (Chapter 14, Questions 7.1-7.6) that returns a real number when one time class is subtracted from another. The result should correspond to a fraction of a day, and may be positive or negative.

7.12. Overload addition and subtraction for the Date class. Add a series of friend overloads to the Date class (Chapter 14, Questions 7.7-7.10) that perform the following operations:

Arg1	***Op***	***Arg2***	***Result***
Date	-	Date	**int**, indicating days between the two dates
Date	-	**int**	Date, the LHS date with the number of days subtracted
Date	+	**int**	Date, the LHS date with the number of days added
int	+	Date	Date, the RHS date with the number of days added

7.13. Overload addition and subtraction for the DateTime class. Add a series of friend overloads to both the DateTime class (Chapter 14, Questions 7.11) and TimeSpan (Chapter 14, Questions 7.11) class that perform the following operations:

Arg1	***Op***	***Arg2***	***Result***
DateTime	-	DateTime	TimeSpan, indicating distance between the two dates
DateTime	-	TimeSpan	DateTime, the LHS date with the TimeSpan subtracted
DateTime	+	TimeSpan	DateTime, the LHS date with the TimeSpan added
TimeSpan	+	DateTime	DateTime, the RHS date with the TimeSpan added

7.14. Overload += and -= for the DateTime class. Add a series of member overloads to both the DateTime class (Chapter 14, Questions 7.11) and TimeSpan (Chapter 14, Questions 7.11) class that perform the += and -= operations. The results should be consistent with table in 7.13.

Chapter 8

Strings and Vectors

Executive Summary

Chapter 8 presents two common template classes from the C++ Standard Template Library (STL): the **string** and the **vector<>**. The practical realities of programming dictate that library classes such as these will be used extensively in our programs. The benefits of doing so include enhanced safety (particularly for objects that would otherwise be implemented as pointers), greater portability, and reduced programming time. Indeed, home grown classes that handle the mechanics general purpose tasks—such as maintaining collections of other objects—should generally be avoided.

The chapter begins by introducing the STL **string** object, a useful templated class that allows use to reduce our reliance on NUL-terminated C-style strings. The **string** member functions are also contrasted with those of the GString object, presented as a lab exercise in Chapter 7, Section 7.5. The **vector<>** template collection class is then discussed, and is contrasted with the PtrArray class introduced in Chapter 7, Section 7.4. Two general classes are then developed in walkthroughs: a Person class that holds data related to an individual, and a PersonArray, a **vector<>**-like class that holds a collection of Person pointers. A tokenizing class, used extensively later in the text, is then presented. Finally, a Company class, that holds **string** and other data related to a company—including a **vector<>**-based collection of employees—is described as a lab exercise. (The classes developed here later become the starting point for inheritance walkthroughs and lab exercises in Chapters 9 and 10).

Learning Objectives

Upon completing this chapter, you should be able to:

- Identify the key elements of the STL **string** interface and use them
- Contrast the costs and benefits of using **string** objects with NUL-terminated strings
- Create **vector<>**-based objects
- Contrasts the costs and benefits of using **vector<>**-based objects with standard C++ arrays
- Embed **string** objects within other classes
- Compose **vector<>**-based collections inside of other classes
- Break a string into a vector<string> array of tokens.

8.1: The string STL Class

Walkthrough available in STLString.avi

The **string** object is probably the most commonly used class in C++ programs. Although we treat string just as if it were a normal class name, it is actually a synonym for the **basic_string<char>** template class—the <*char*> signifying that **string** objects deal with 1-byte character elements. The practical significance of this is the possibility of strings being implemented using non-ASCII characters (such as short integers, which would be able to represent UNICODE). Such implementations will become increasingly common, and probably the norm, in the coming decade as the need to write code that supports international users grows. For now, however, we'll just think of **string** objects as substitutes for ASCII strings.

Using **string** objects, the NUL-terminated strings that originated in C can be avoided in many situations. Moreover, a member is available for converting string objects to constant NUL-terminated strings, offering some backward compatibility.

8.1.1: string Overview

Among the characteristics that differentiate **string** objects from NUL-terminated strings are the following:

- Support for assignment from other **string** objects and NUL-terminated strings
- Support for comparison operators (e.g., >, ==, !=) with other **string** objects and NUL-terminated strings
- Ability to concatenate with other **string** objects and NUL-terminated strings using the + operator
- Access of individual characters using the [] operator
- >> and << support for text output

Overall, the most important difference between **string** objects and NUL terminated strings is that **string** objects can be treated just like the other primitive data types (e.g., integers and real numbers). This not only simplifies C++ programming, it also makes it much more like programming in other languages, such as Basic or Java.

Whenever you use **string** objects in your program, you must include the C++ STL string library file with the statement:

```
#include <string>
```

```
    using namespace std;
```

Omitting the .h is particularly critical in this case, as "string.h" refers the to C-string library (with prototypes of strcpy, strlen, strcat, etc.) and not the **string** object.

8.1.2: string Interface

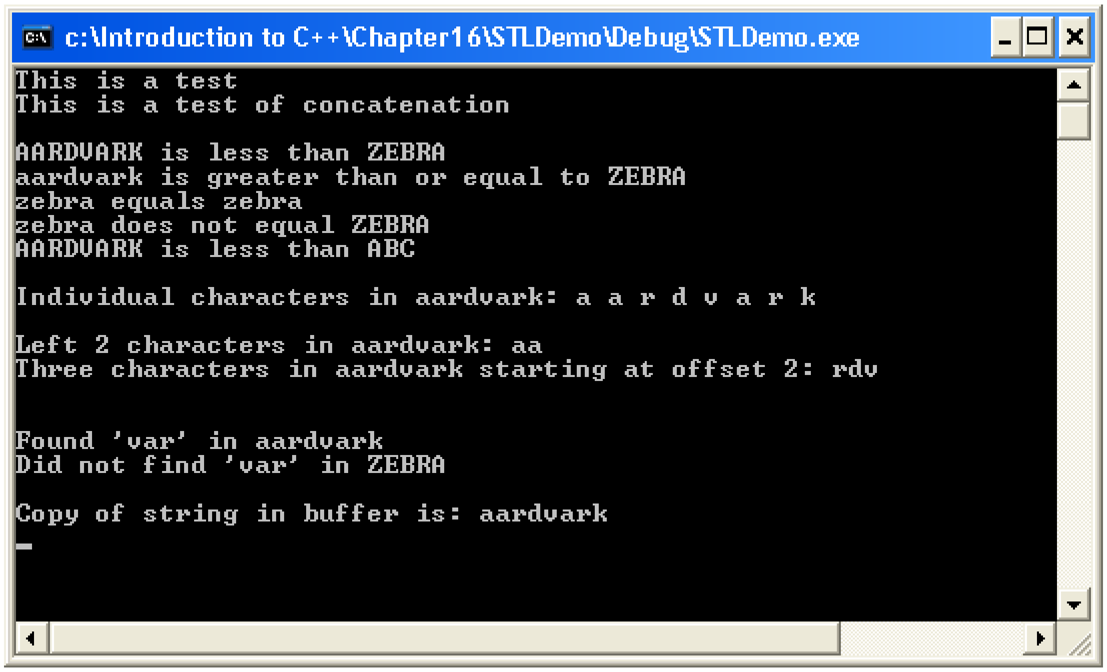
The STL **string** interface is very like the GString interface presented as a lab exercise in Chapter 7 (Section 7.5). A demonstration routine illustrating the use of string objects is presented in Example 8.1. The output from running the routine is presented in Figure 8.1.

Some differences from the GString interface (aside from naming conventions) include:

- There is no automatic typecast from **string** objects to const char *. The c_str() member function needs to be called instead.
- The **string** object substring() function copies characters into a buffer instead of returning a new string.
- The **string** object copy() function copies characters into a buffer instead setting the string to its argument.
- The **string** object empty() function tests if a string is empty, instead of emptying it
- The **string** object has a generous collection of searching functions, including find(), find_ first_not_of(), find_ first_of(),find_last_not_of(), and find_ last_of()

Example 8.1: Demonstration of string features

```
void StringDemo()
{
	char buf[80]={0};
	string s1,s2,s3,s4,s5,s6,s7,s8;
	// initializing from NUL terminated string
	s1="This is ";
	s2="a test";
	// demonstrating concatenation
	s3=s1+s2;
	cout << s3 << endl;
	cout << s3 + " of concatenation" << endl << endl;
	// comparison demonstration
	s4="AARDVARK";
	s5="ZEBRA";
	s6="aardvark";
	s7="zebra";
	s8=s7;
	if (s4<s5) cout << s4 << " is less than " << s5 << endl;
	else cout << s4 << " is greater than or equal to " << s5 << endl;
	if (s6<s5) cout << s6 << " is less than " << s5 << endl;
	else cout << s6 << " is greater than or equal to " << s5 << endl;
	if (s7==s8) cout << s7 << " equals " << s8 << endl;
	else cout << s7 << " does not equal " << s8 << endl;
	if (s7!=s5) cout << s7 << " does not equal " << s5 << endl;
	else cout << s7 << " equals " << s5 << endl;
	if (s4<"ABC") cout << s4 << " is less than " << "ABC" << endl;
	else cout << s4 << " is greater than or equal to " << "ABC" << endl;
	cout << endl << "Individual characters in " << s6+": ";
	// using length() member
	for(unsigned int i=0;i<s6.length();i++) {
		// accessing individual characters
		cout << s6[i] << " ";
	}
	// using copy() member to get a substring
	s6.copy(buf,2);
	cout << endl << endl << "Left 2 characters in " << s6+": " << buf << 
endl;
	cout <<  "Three characters in " << s6+" starting at offset 2: "
		<< s6.substr(2,3) << endl << endl;
	// testing find() member function
	unsigned int nPos=s6.find("var");
	if ((int)nPos!=-1) cout << endl << "Found \'var\' in " << s6;
	else cout << endl << "Did not find \'var\' in " << s6;
	nPos=s5.find("var");
	if ((int)nPos!=-1) cout << endl << "Found \'var\' in " << s5;
	else cout << endl << "Did not find \'var\' in " << s5 << endl;
	// converting string object to NUL-terminated string
	strcpy(buf,s6.c_str());
	cout << endl << "Copy of string in buffer is: " << buf << endl;
	return;
}
```



```
This is a test
This is a test of concatenation

AARDVARK is less than ZEBRA
aardvark is greater than or equal to ZEBRA
zebra equals zebra
zebra does not equal ZEBRA
AARDVARK is less than ABC

Individual characters in aardvark: a a r d v a r k

Left 2 characters in aardvark: aa
Three characters in aardvark starting at offset 2: rdv


Found 'var' in aardvark
Did not find 'var' in ZEBRA

Copy of string in buffer is: aardvark
_
```

Figure 8.1: Output from StringDemo() function of Example 8.1

8.1.3: string Operators and Members

As we begin to use more and more library classes, it becomes increasingly important that the that reader start to use the excellent online language documentation provided by Visual Studio .Net (and on the MSDN web site) for the purpose of understanding class members, as opposed to relying too heavily on a textbook for reference purposes. Unlike any textbook, the language documentation supplied by .Net tends to be complete and up-to-date—at least as far as the Microsoft version of the compiler is concerned.

Unfortunately, the nature of some of the STL class objects makes it easy to miss parts of the relevant description. This is particularly true for string and standard I/O objects where many of the templated classes have synonyms, or where inheritance is present (a particular issue for the standard I/O classes).

In the case of the **string** object, to find all the relevant members and operators supported by the class, you need to look in two places: under **string** and under **basic_string** (which is the template class used to derive strings).

Using this documentation, we can identify the operators overloaded for string objects. These are summarized in Table 8.1.

Operator	Arguments	Purpose
== != > < >= <=	LHS & RHS: string const char * (at least one string must be present)	Performs a character comparison, using ASCII character sequence
=	LHS: string RHS: string const char *	Assigns a value to the string on the LHS
+	LHS & RHS: string const char * (at least one string must be present)	Returns a new string that is the result of concatenating the LHS and RHS arguments
+=	LHS: string RHS: string const char *	Equivalent to LHS=LHS+RHS
<<	LHS: ostream (OUTPUT) RHS: string	Sends a string to a text output stream, such as cout
>>	LHS: istream (INPUT) RHS: string	Extracts a string from a text input stream, such as cin
[]	integer argument	Returns the character at the specified position in the string. Will not generate an exception.

Table 8.1: string Operators, compiled from the Visual Studio .Net documentation

There are over 30 string member functions, a number of which are of quite limited use. Some of the more commonly used of these are summarized in Table 8.2.

Operator	Arguments	Purpose
append push_back	1. string | const char *	Concatenates argument to the end of the string (like += operator)
c_str	None	Returns a const char * to a buffer whose contents are the same as those of the string. Similar to a const char * typecast.
clear erase	None (clear) 1. offset=0 (erase) 2. count=0 (erase)	Empties the contents of a string Erase version allows an optional specified range of the string to be emptied
copy	1. char * 2. count 3. offset=0	Copies number of characters (arg2) from a string to a character buffer (arg1), starting at a specified position (arg3)
find find_first_not_of find_first_of find_last_not_of find_last_of rfind	1. char | const char * | string 2. offset=0	Performs various searches for a substring within the string, starting at position offset. Returns character offset where found or value < 0 (when typecast to signed integer) (rfind starts searching at the end of the string and moves forward)
insert	1. offset 2. string | const char *	Inserts a string at the specified offset position within the string object. (*Note*: other overloads also exist)
length size	None	Returns length of the string
replace	1. offset 2. count 3. string | const char *	Replaces count characters, starting a position offset with a string (arg3)
substr	1. offset 2. count	Returns a string that is a substring of the object, starting at offset with up to count characters
swap	1. string	Swaps contents with its argument

Table 8.2: Selected string Member functions, compiled from Visual Studio .Net documentation

Section 8.2 Questions

1. Why does it make sense to create the string object from a template?

2. What do you need to do in order to pass string objects into NUL terminated string functions such as strcmp()?

3. Can string objects be saved directly to binary files?

4. Why do string objects appear in two places in the Visual Studio .Net reference documentation?

5. Why is it better to use string objects than your own home-grown string class (e.g., GString)?

6. What's the problem with using #include <string.h> instead of #include<string> when you want to use string objects?

8.2: The vector<> STL Class

Walkthrough available in Vector.avi

The **vector<>** template class can be used to encapsulate an array of virtually any type of object for which an assignment operator is available. Its flexibility allows it to substitute for most arrays normally used in programs—greatly increasing code integrity in the process.

8.2.1: vector Overview

Among the characteristics that differentiate **vector<>** objects from standard arrays are the following:

- Dynamic resizing as elements are added (using the insert() and push_back() members).
- Ability to remove elements (using the remove() and pop_back() members)
- Bounds checking performed as elements are accessed using the [] operator
- size() operator returns the number of elements in the **vector<>**
- Access can be performed using an iterator

Whenever you use **vector** objects in your program, you must include the C++ STL vector library file with the statement:

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

8.2.2: Iterators

In order to understand the **vector<>** interface, we need to introduce the general topic of iterators—a topic to be presented in greater depth in Chapter 13. The idea behind iterators is straightforward: in C++ we'll be dealing with many different forms of collections, referred to as collection *shapes*. Wouldn't it be nice it there were a uniform way to access all the elements in each collection, regardless of how it is constructed?

STL collection classes automatically define an iterator object type whenever a template class is created. Iterator variables can be declared as follows:

template-name<*parameters*>::iterator *variable-name*;
template-name<*parameters*>::const_iterator *variable-name*;
// use const_iterator when you won't be changing collection values

The interface of the iterator well known to C++ programmers—it is used exactly like a pointer. In other words, if nPos is declared as an iterator:

- ***nPos** accesses the collection element at that position
- **nPos[offset]** accesses collection elements at an offset position from the iterator (assuming the collection supports random access, as an array-like class such as a **vector<>** does)
- ***(nPos+offset)** also accesses collection elements at an offset position (collections supporting random access), and is equivalent to **nPos[offset]**—just as it is for pointers.
- **nPos++** moves the iterator to the next element in the collection
- **nPos--** moves to the previous element in the collection

In addition, two class member functions are always defined for use with iterators:

- begin() returns the iterator value at the start of the collection.
- end() returns a special iterator value signifying we have incremented the last element of the collection

As an example, consider **string** objects. Because a **string** is an array of characters, individual characters can be accessed using iterators in addition to the [] notation or the at() member. In fact, the two code fragments below accomplish precisely the same result:

```
// character access using length and [] overload
for(unsigned int i=0;i<s6.length();i++) {
        // accessing individual characters
        cout << s6[i] << " ";
}
```

and

```
// character access using iterator
string::iterator iter;
for(iter=s6.begin();iter!=s6.end();iter++) {
        // accessing individual characters
        cout << *iter << " ";
}
```

The principle advantage of using iterators to access class elements is their consistency across shapes. This means if you know how to iterate through a **vector<>** using an iterator, you can do the same thing in a **list<>** or **map<>**. In addition, the STL provides various flavors of iterators (e.g., reverse iterators, random access iterators) whose availability varies by shape. The consideration of these, however, will be delayed until Chapter 13.

8.2.3: vector<> Interface

The **vector<>** interface is very similar to that of the *PtrArray* class that was presented in Chapter 7, Section 7.4. Some differences include:

- the vector<> template defines an iterator in addition to supporting the [] overload. This allows us to go through the elements of an array many different ways. For example:

```
vector<string> arText;
// initialization omitted
for(i=0;i<arText.size();i++) {
        cout << arText[i] << " ";
}
```

  and

```
// iterator using "pointer arithmetic" style
for(i=0;i<arText.size();i++) {
        cout << *(arText.begin()+i) << " ";
}
```

  and

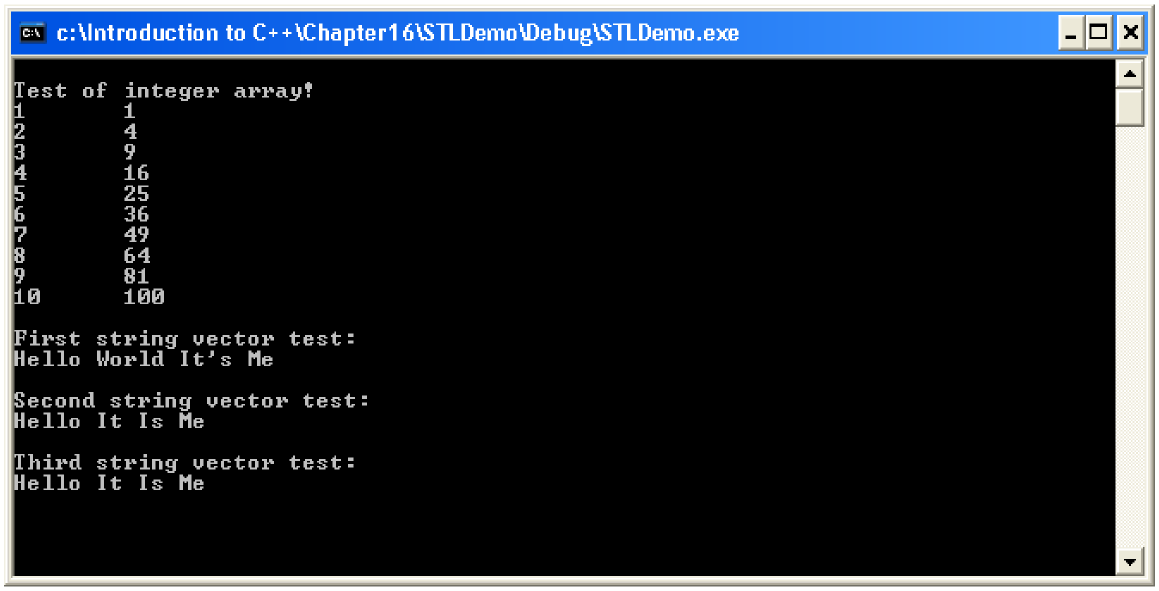
```
vector<string>::iterator iter;
// using same approach as string example
for(iter=arText.begin();iter!=arText.end();iter++) {
        cout << *iter << " ";
}
```

- vector<> template insertion requires an iterator to identify the insertion point.

A demonstration of the **vector<>** interface, creating then performing operations an int vector, followed by a string vector, is shown in Example 8.2. The output from running the demonstration function is presented in Figure 8.2.

Example 8.2: Demonstration of vector<> template class

```
void VectorDemo()
{
	vector<int> arSquares;
	// ading values to end of vector
	for(unsigned int i=1;i<=10;i++) {
		arSquares.push_back(i*i);
	}
	cout << endl <<  "Test of integer array!" << endl;
	for(i=0;i<arSquares.size();i++) {
		cout << i+1 << '\t' << arSquares[i] << endl;
	}
	vector<string> arText;
	// inserting values at the start of the vector
	string s0("Me"),s1("It\'s"),s2("World"),s3("Hello");
	arText.insert(arText.begin(),s0);
	arText.insert(arText.begin(),s1);
	arText.insert(arText.begin(),s2);
	arText.insert(arText.begin(),s3);
	cout << endl << "First string vector test:" << endl;
	for(i=0;i<arText.size();i++) {
		cout << arText[i] << " ";
	}
	cout << endl;
	// inserting values at position 3
	arText.insert(arText.begin()+3,"Is");
	// replacing a value
	arText[2]="It";
	// removing an element
	arText.erase(arText.begin()+1);
	cout << endl << "Second string vector test:" << endl;
	// iterator using "pointer arithmetic" style
	for(i=0;i<arText.size();i++) {
		cout << *(arText.begin()+i) << " ";
	}
	vector<string>::iterator iter;
	cout << endl << endl << "Third string vector test:" << endl;
	// using same approach as string example
	for(iter=arText.begin();iter!=arText.end();iter++) {
		cout << *iter << " ";
	}
	cout << endl;
	return;
}
```



```
Test of integer array!
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
9       81
10      100

First string vector test:
Hello World It's Me

Second string vector test:
Hello It Is Me

Third string vector test:
Hello It Is Me
```

Figure 8.2: Output of VectorDemo() function

8.2.4: vector<> Member Functions

The **vector<>** member functions are similar to those of the **string** and other collection classes. Some of the more important members are summarized in Table 8.3.

Operator	Arguments	Purpose
assign	1. iterator 2. iterator	Replaces the contents of the vector with a collection range Arguments specify the begin and end point of assignment range from the source collection, which is not necessarily another vector<>
begin	None	Returns an iterator that can be used for random access (i.e., supports [] operators) to the first element of the vector
capacity	1. None	Number of available elements before the vector must resize
clear erase	None (clear) 1. iterator (erase) 2. iterator=end() (erase)	Empties the contents of a vector Erase version allows an optional specified range of the vector to be emptied
end	None	Returns an iterator representing the position beyond the data in the array
insert	1. iterator 2. element-data-type	Inserts an element at the specified position within the vector (*Note*: other overloads also exist)
pop_back	1. element-data-type	Removes last element in the vector
push_back	1. element-data-type	Adds element to the end of the vector
reserve	1, element-count	Sets the minimum size of the vector before resizing must occur
size	None	Returns number of elements in the vector

Table 8.3: Selected vector<> Member functions, compiled from Visual Studio .Net documentation

Section 8.2 Questions

1. Why would we want to use vector<> objects in place of true C++ arrays wherever we can? Is there any cost of doing so?

2. What does it mean when we say a vector<> collection is type-safe?

3. If iter is the name of an iterator, what does iter[3] refer to? Are there any restrictions on the use of brackets with iterators?

4. Explain the arText.begin()+3 in the expression arText.insert(arText.begin()+3,"Is");

5. Where do the member names pop_back() and push_back() likely derive from?

6. What is the difference between a vector's capacity and its size?

8.3: Template Object Exercises

In this section, we will develop two relatively simple classes, a Person class and a collection class that maintains a sorted array of pointers to Person objects. These two classes will later be used extensively in the Video Store lab exercise presented at the end of Chapters 9 and 10. Before beginning with these, however, we will overload the >> and << operators for loading and saving string objects.

8.3.1: Loading and Saving String Objects

In order to the exercises in this chapter and the next, it is convenient to overload the output/insertion (<<) and input/extraction (>>) operators for loading and saving strings to a file. To make them easy to access, we make them inline functions in a file called utils.h, the contents of which are presented in Example 8.3.

Example 8.3: string Object Overloads in Utils.h

```
// Utils.h: contains useful utility functions
#define MAXSTRINGLEN 32001

inline fstream &operator<<(fstream &strm,const string &str) {
      int nLen=(int)str.length();
      nLen=(nLen>MAXSTRINGLEN-1) ? MAXSTRINGLEN-1 : nLen;
      nLen++;
      strm.write((const char *)&nLen,sizeof(int));
      char buf[MAXSTRINGLEN]={0};
      str.copy(buf,nLen);
      strm.write(buf,nLen+1);
      return strm;
}

inline fstream &operator>>(fstream &strm,string &str) {
      int nVal;
      char buf[MAXSTRINGLEN]={0};
      strm.read((char *)&n=nVal,sizeof(int));
      nVal=(nVal>MAXSTRINGLEN-1) ? MAXSTRINGLEN-1 : nVal;
      strm.read(buf,nVal);
      str=buf;
      return strm;
}
```

In the output (insertion) overload we do the following:

- Save the length of the string (nLen++ used to include the NUL terminator) after truncating it to no more than MAXSTRINGLEN-1 characters
- Copy the (truncated, if necessary) characters into a text buffer using the **string** copy() member.
- Save the buffer to the strm

The input (extraction) proceeds in similar fashion:

- We read the integer length
- We truncate (if necessary) to protect our buffer.
- We read the string into our buffer
- We assign our buffer to the string

Test your understanding: Why should it never be necessary to truncate the strings we are reading in?

8.3.2: A Person class

Walkthrough available in Person.avi

The Person object we're creating here a very simple object—having only first name and last name data members. In Chapter 9, however, we will be taking this class and using it as a basis for an inheritance network. For this reason, we're including quite a number of member functions, in addition to normal accessor functions and overloads. Among these:

- Input() and Display(): Intended for console-oriented I/O
- Get() and Print(): Intended for I/O from a fixed text file
- Load() and Save(): Intended for I/O from a binary file
- Key(): returns a combination of last name and first name intended to be a unique identifier for collections
- FixedString(): Generates a fixed length string containing last name and first name of a type consistent with what the Get() function reads from a file
- Overloads of << and >> for binary file load and save
- Overloads of << and >> for fixed text file load and save

The header file for the class is presented in Example 8.4.

Example 8.4: Person class declaration (Person.h)

```
class Person
{
public:
	Person(void);
	virtual ~Person(void);
	Person(const Person &pers) {
		Copy(pers);
	}
	const Person &operator=(const Person &pers) {
		if (&pers==this) return *this;
		Copy(pers);
		return *this;
	}
	void Copy(const Person &pers) {
		m_szLastName=pers.m_szLastName;
		m_szFirstName=pers.m_szFirstName;
	}

	string LastName() const {return m_szLastName;}
	void LastName(const char *szName){m_szLastName=szName;}
	string FirstName() const {return m_szFirstName;}
	void FirstName(const char *szName){m_szFirstName=szName;}

	string Key() const {return m_szLastName+","+m_szFirstName;}

	bool IsValid() const {return !m_szLastName.empty();}
	void Input();
	void Display() const;
	void Get(istream &in);
	void Print(ostream &out) const;
	string FixedString() const;
	void Load(fstream &strm);
	void Save(fstream &strm) const;

protected:
	string m_szLastName;
	string m_szFirstName;
};

istream &operator>>(istream &in,Person &pers);
ostream &operator<<(ostream &out,const Person &pers);
fstream &operator>>(fstream &strm,Person &pers);
fstream &operator<<(fstream &strm,const Person &pers);
```

By now, virtually everything in the header should be familiar. A few comments:

- The Key() function uses concatenation to develop a lookup key for our Person. This will be used in Section 8.3.3
- The IsValid() function is used to detect a valid load—in this case a "valid" person cannot have an empty last name.

- The >> and << operator overloads don't need to be friend functions, as we're just wrapping them around public member functions

The Person functions themselves are also, for the most part, very simple. The binary file functions could not be simpler, as shown in Example 8.5.

Example 8.5: Person Binary Loading and Saving Members

```
void Person::Load(fstream &strm)
{
	strm >> m_szLastName;
	strm >> m_szFirstName;
}


void Person::Save(fstream &strm) const
{
	strm << m_szLastName;
	strm << m_szFirstName;
}

fstream &operator>>(fstream &strm,Person &pers)
{
	pers.Load(strm);
	return strm;
}
fstream &operator<<(fstream &strm,const Person &pers)
{
	pers.Save(strm);
	return strm;
}
```

Because we overloaded the << and >> operators for binary load/save of **string** objects in 8.3.1, our Load() and Save() members take advantage of this fact. And, as already noted, we just wrap our Person << and >> operators around those functions.

The Input() and Display() functions, presented in Example 8.6, are equally trivial. Input() uses the SimpleIO functions to read in the data. To make it useful as an editing function, it displays the existing values for the data members (if any). This is done by concatenating [and] around the existing value to create a new string, e.g.,

```
if (!m_szLastName.empty()) cout << "[" << m_szLastName << "]: ";
```

If the user hits return (causing an empty string to be placed in buf), the function does not modify the existing value.

Example 8.6: Person Input and Display Members

```
void Person::Input() {
      char buf[1024];
      cout << "Last Name: ";
      if (!m_szLastName.empty()) cout << "[" << m_szLastName << "]: ";
      cin.getline(buf,1024);
      if (buf[0]!=0) m_szLastName=buf;
      cout << "First Name: ";
      if (!m_szFirstName.empty()) cout << "[" << m_szFirstName << "]: 
";
      cin.getline(buf,1024);
      if (buf[0]!=0) m_szFirstName=buf;
}
void Person::Display() const
{
      cout << "Last Name:\t" << LastName() << endl;
      cout << "First Name:\t" << FirstName() << endl;
}
```

The fixed format members, Get(), Print(), and FixedString() require slightly more explanation. The idea behind them is to create paired functions for writing and reading fixed text. The write function, Print(), calls FixedString() which, in turn:

- Creates a buffer filled with 40 space characters
- Assigns the buffer to a string, str
- Uses the replace() member to replace however many spaces are required with the last name, starting at position 0. The call LastName().substr(0,20) prevents more than 20 characters from the last name being used.
- Uses the replace() member to replace however many spaces are required with the first name, starting at position 20.
- Returns the resulting string

The reading function, Get(), does the following:

- reads a line from the stream
- assigns that line to a string (szBuf)
- uses the substr() member to extract the last name and first name strings from the buffer.
- calls the Trim() function (defined in utils.h and shown in Example 8.8), which removes any trailing white characters in the two strings.

These functions are presented in Example 8.7.

Example 8.7: Person Members for Fixed Format I/O

```
string Person::FixedString() const
{
      char buf[1024]={0};
      int i;
      for(i=0;i<40;i++) buf[i]=' ';
      string str=buf;
      str.replace(0,LastName().substr(0,20).length(),
            LastName().substr(0,20));
      str.replace(20,FirstName().substr(0,20).length(),
            FirstName().substr(0,20));
      return str;
}
void Person::Get(istream &in) {
      char buf[1024];
      in.getline(buf,1024);
      string szBuf=buf;
      m_szLastName=szBuf.substr(0,20);
      Trim(m_szLastName);
   if (szBuf.length()>20) m_szFirstName=szBuf.substr(20,20);
      Trim(m_szFirstName);
}
void Person::Print(ostream &out) const
{
      out << FixedString() << endl;
}
istream &operator>>(istream &in,Person &pers) {
      pers.Get(in);
      return in;
}
ostream &operator<<(ostream &out,const Person &pers)
{
      pers.Print(out);
      return out;
}
```

Example 8.8: Trim() function in Utils.h

```
inline void Trim(string &str) {
      int nLen=(int)str.length()-1;
      while(nLen>=0 && str[nLen]<=' ') nLen--;
      str.erase(nLen+1);
}
```

8.3.3: A Sorted Person Array

Walkthrough available in PersonArray.avi

The PersonArray class is designed to maintain an array of keyed Person objects—allowing no duplicate objects in the array.

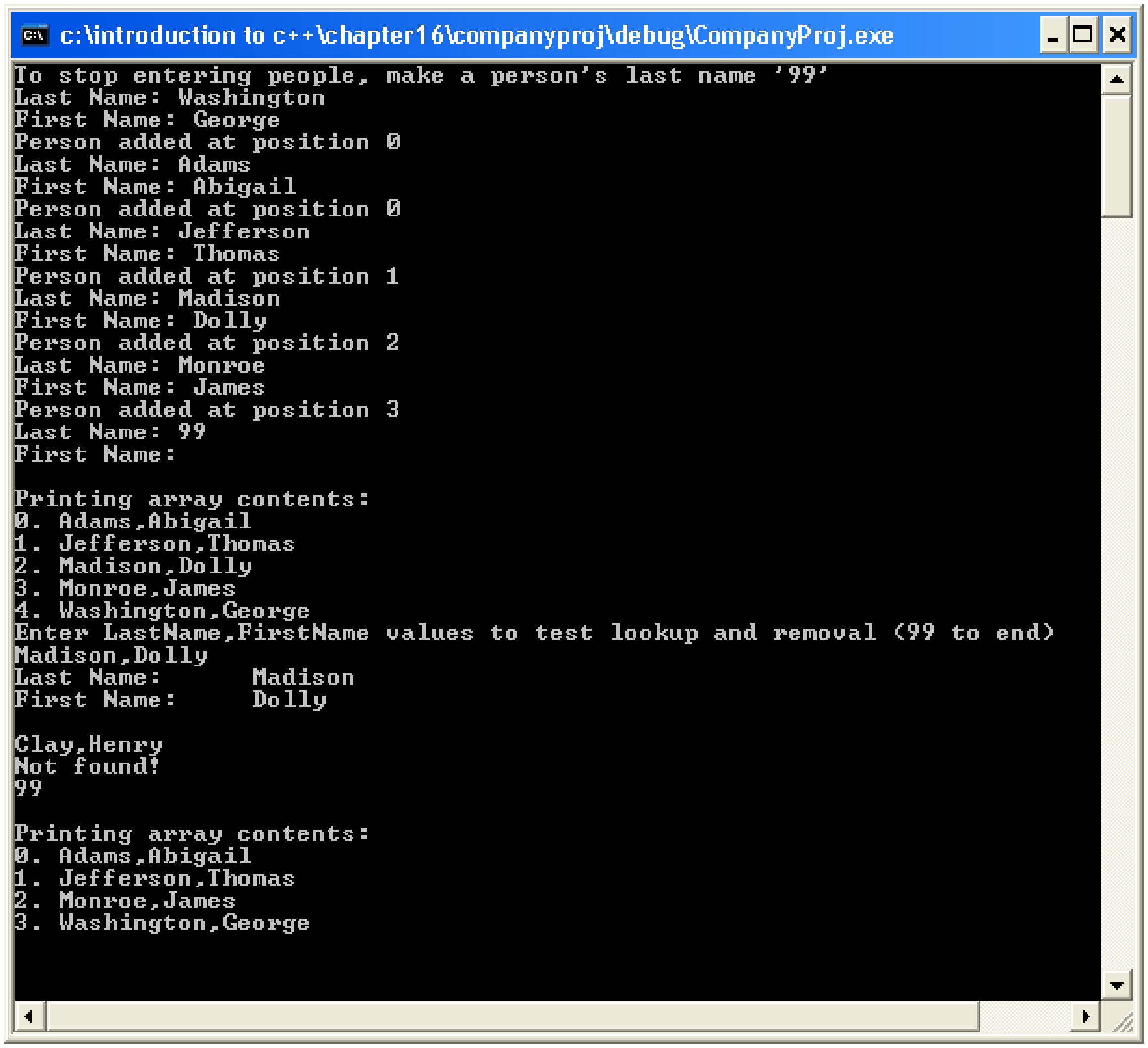
Interface
The PersonArray interface includes a variety of feature, including:

- An Add() member used to add a Person pointer to the array. Other insertion members are not included in the interface because in a sorted array, a given element should only be in one position.
- Several different overloads of the Find() member, causing a binary search of the array to be conducted and a Person pointer returned if a match is found.
- Two types of overloads of the [] operator:
    - [int] returns the element at the specified position in the array
    - [string] returns the element with the key matching the string (or 0 if no match is found).
- A Remove() member that removes a specified key, deleting the associated Person object
- A RemoveAll() member that removes all Person elements and deletes them
- Load() and Save() members for binary serialization, as well as overloads of the >> and << operators.

Some test code that uses the class is presented in Example 8.9. The output from a run where 5 names were entered is presented in Figure 8.3.

Example 8.9: Test function using PersonArray class

```
void TestPersonArray()
{
	PersonArray ar;
	cout << "To stop entering, make last name \'99\'" << endl;
	Person pers;
	while(pers.LastName()!="99")
	{
		pers.Input();
		if (pers.LastName()=="99") continue;
		int nPos=ar.Add(pers);
		if (nPos<0) cout << "Duplicate person encountered!" << endl;
		else cout << "Person added at position " << nPos << endl;
	}
	int i;
	cout << endl << "Printing array contents:" << endl;
	for(i=0;i<ar.Size();i++) {
		cout << i << ". " << ar[i]->Key() << endl;
	}
	// Test [] lookup and remove
	cout << "Enter LastName,FirstName values to test lookup/removal (99 to 
end)"
		<< endl;
	string LName;
	while(LName!="99")
	{
		char buf[256];
		cin.getline(buf,255);
		LName=buf;
		if (LName=="99") continue;
		const Person *p=ar[LName];
		if (p==0) cout << "Not found!" << endl;
		else {
			p->Display();
			cout << endl;
			ar.Remove(p->Key().c_str());
		}

	}
	cout << endl << "Printing array contents:" << endl;
	for(i=0;i<ar.Size();i++) {
		cout << i << ". " << ar[i]->Key() << endl;
	}
}
```


c:\introduction to c++\chapter16\companyproj\debug\CompanyProj.exe

```
To stop entering people, make a person's last name '99'
Last Name: Washington
First Name: George
Person added at position 0
Last Name: Adams
First Name: Abigail
Person added at position 0
Last Name: Jefferson
First Name: Thomas
Person added at position 1
Last Name: Madison
First Name: Dolly
Person added at position 2
Last Name: Monroe
First Name: James
Person added at position 3
Last Name: 99
First Name:

Printing array contents:
0. Adams,Abigail
1. Jefferson,Thomas
2. Madison,Dolly
3. Monroe,James
4. Washington,George
Enter LastName,FirstName values to test lookup and removal (99 to end)
Madison,Dolly
Last Name:      Madison
First Name:     Dolly

Clay,Henry
Not found!
99

Printing array contents:
0. Adams,Abigail
1. Jefferson,Thomas
2. Monroe,James
3. Washington,George
```

Figure 8.3: Output from a TestPersonArray() function

Implementation
The implementation of the PersonArray class is similar to what we've seen before except for its use of insertion sort to keep its elements in order. This is accomplished by making the actual collection that holds the data—a vector<Person*> object—a protected member, then limiting modifications to additions and removals. All functions that return a Person * return that pointer as a const value—meaning it should not be changed. (To change a person object, you'd need to remove it, modify it, then add it back).

The class declaration, in PersonArray.h, is presented in Example 8.10. One aspect of the interface design was that many versions of the same member are provided—the only difference being variation in how the argument is presented (e.g., const char * vs. string&). The main purpose this performs is to make access for the programmer more convenient.

Example 8.10: PersonArray class declaration (PersonArray.h)

```
class PersonArray
{
public:
      PersonArray(void);
      ~PersonArray(void);

      const Person *operator[](int) const;
      const Person *operator[](const char *) const;
      const Person *operator[](const string &szKey) const;
      const Person *Find(const char *szKey,int &nPos) const;
      const Person *Find(const char *szKey) const;
      const Person *Find(const string &szKey) const;

      int Add(Person *);
      int Add(const Person &pers);
      void Remove(const char *szKey);
      void RemoveAt(int nVal);
      void Remove(Person *p);
      int Size() const;
      void RemoveAll();

      void Load(fstream &strm);
      void Save(fstream &strm) const;

protected:
      vector<Person*> m_arp;
      int FindPos(const string &szKey) const;
      int FindPos(const char *szKey) const;
};

fstream &operator>>(fstream &strm,PersonArray &ar);
fstream &operator<<(fstream &strm,const PersonArray &ar);
```

The most critical function of the class is the protected FindPos() function, both versions being shown in Example 8.11. The function employs binary search. The algorithm—which can be adapted to any *sorted* array—works as follows:

- Two variables, nLower and nUpper are defined to hold the range being searched. nLower starts at 0 (the "lowest" element in the array), nUpper starts at the size of the array—which will be 1 higher than the largest legal coefficient (since all arrays in C++ are zero-based).
- We check the key we are looking for with the lowest element. If it is less than or equal to that element—or if the array size is 0—we return 0.
- We then enter a loop which does the following:
    - We create a test position half-way between the upper and lower bounds
    - We check the key at that test position.

    - If what we are looking for is greater than the test key, the test position becomes our new lower bound (since we know the key we're looking for can't be in the bottom half or our range).
    - If what we are looking for is less than the test key, the test position becomes our new upper bound.
    - If the test key matches what we are looking for, we return the test position.
- Once we have reset our bounds within the loop, we repeat the process with our new upper or lower bound. Each time the loop repeats, the area we search becomes half as large as the previous range.
- The loop ends when either: 1) we return because a match was found, or 2) the upper and lower bound difference is 1—meaning that the item is not there. If this is the case, we return the upper bound.

The significance of its return value, no matter how the function ends, is as follows:

- If it finds the key, it returns the position of the key
- If it doesn't find the key, it returns the position where the key would be located if we were to insert it.

Example 8.11: FindPos() Member Functions of PersonArray class

```
// Implements binary search
int PersonArray::FindPos(const char *szKey) const {
      int nLower=0,nUpper=(int)m_arp.size();
      if (m_arp.size()==0 || m_arp[0]->Key()>=szKey) return 0;
      while(nUpper-nLower>1) {
            int nTest=(nUpper+nLower)/2;
            if (m_arp[nTest]->Key()<szKey) nLower=nTest;
            else if (m_arp[nTest]->Key()>szKey) nUpper=nTest;
            else return nTest;
      }
      return nUpper;
}
```

Many other member functions then use the result of the search. The public functions for locating keyed values, both Find() functions and text overloads of the [] operator, are presented in Example 8.12. All return 0 if the key is not found, and all do little more than provide convenient packaging around FindPos(). This ensures that if there is an error in our binary search routine, it only needs to be fixed in one place.

Example 8.12: Find() Member Functions of the PersonArray class

```
const Person *PersonArray::Find(const string &szKey) const{
      return Find(szKey.c_str());
}

const Person *PersonArray::Find(const char *szKey,int &nPos) const
{
      nPos=FindPos(szKey);
      if (nPos==Size() || m_arp[nPos]->Key()!=szKey) return 0;
      return m_arp[nPos];
}

const Person *PersonArray::Find(const char *szKey) const {
      int nPos;
      return Find(szKey,nPos);
}

const Person *PersonArray::operator[](const char *szKey) const {
      return Find(szKey);
}

const Person *PersonArray::operator[](const string &szKey) const {
      return Find(szKey.c_str());
}
```

The routines for addition and deletion just provide a wrapper around the vector<Person*> object's own interface. They are presented in Exhibit 8.13.

Exhibit 8.13: PersonArray addition and removal member functions

```
int PersonArray::Add(Person *pers) {
	int nPos;
	if (Find(pers->Key().c_str(),nPos)) return -1;
	vector<Person*>::iterator iter=m_arp.begin();
	m_arp.insert(iter+nPos,pers);
	return nPos;
}
int PersonArray::Add(const Person &pers) {
	Person *p=new Person(pers);
	int nRet=Add(p);
	if (nRet<0) delete p;
	return nRet;
}
void PersonArray::Remove(const char *szKey) {
	int nPos;
	if (!Find(szKey,nPos)) return;
	RemoveAt(nPos);
}
void PersonArray::Remove(Person *p) {
	assert(p!=0);
	Remove(p->Key().c_str());
}
void PersonArray::RemoveAt(int nPos)
{
	assert(nPos>=0 && nPos<(int)m_arp.size());
	if (nPos<0 || nPos>Size()) return;
	delete m_arp[nPos];
	vector<Person*>::iterator iter=m_arp.begin();
	m_arp.erase(iter+nPos);
}
void PersonArray::RemoveAll()
{
	for(int i=0;i<(int)m_arp.size();i++) {
		delete m_arp[i];
	}
	m_arp.clear();
}
```

In the case of the two overloads of the Add() member, two different purposes are being accomplished:

- The Add(Person *) takes a Person pointer to the actual object that is being added to the collection—which should have been created with the **new** operator.
- The Add(const Person&) overload, in contrast actually creates a new object by making a copy of the argument that was passed. This would be useful where a Person has been edited, for example.

Both return –1 if the addition fails.

The Add(Person *) member performs what is called an insertion sort. Instead of adding elements at the end of the array, it finds the position where they would be if they were present (the value returned by then FindPos() member), then inserts them there. This is actually a reasonably efficient sort and will be discussed further in Chapter 15, where the question of how to sort collections is addressed.

In the case of the Remove() members, the RemoveAt(int) does the actual removal and deletion of the Person being removed. The various Remove() overloads just package the call to RemoveAt() in different ways.

RemoveAll() does a complete removal: deleting all elements then calling clear on the vextor<Person*> object itself. This function is useful for calling in the destructor and before loading data from a binary file. As we have emphasized throughout the text, whenever the same task needs to be performed in more than one place, it makes sense to define a function.

The Load() and Save() members, along with their >> and << overloads, are presented in Example 8.14. In each case, they take advantage of the fact that we have already implemented serialization on our Person object. So, consistent with our general philosophy for collections, we just save/load the collection size, the loop through the elements calling save/load on each. The >> and << overloads, in turn, are identical to those implemented in the Person class.

Example 8.14: Load and Save Members of PersonArray

```
void PersonArray::Load(fstream &strm)
{
	RemoveAll();
	int nVal;
	strm.read((char *)&nVal,sizeof(int));
	for(int i=0;i<nVal;i++) {
		Person *pers=new Person;
		strm >> *pers;
		m_arp.push_back(pers);
	}
}

void PersonArray::Save(fstream &strm) const
{
	int nVal=Size();
	strm.write((const char *)&nVal,sizeof(int));
	for(int i=0;i<Size();i++) {
		strm << *(m_arp[i]);
	}
}
fstream &operator>>(fstream &strm,PersonArray &ar)
{
	ar.Load(strm);
	return strm;
}
fstream &operator<<(fstream &strm,const PersonArray &ar)
{
	ar.Save(strm);
	return strm;
}
```

One capability not added to the PersonArray class (although presented as an end-of-chapter exercise) is the copy constructor/assignment operator. This decision was intentional, for three reasons. First, collections such as this one could, potentially, get quite large, so we should be reluctant to see them copied when a reference would suffice. Second, there's no obvious reason why we should need to make a copy of this particular collection. Third, the way we will be enhancing the class in Chapters 9 and 10 will introduce further subtleties to copying.

As a result of this decision, both the constructor and destructor functions for the class are very simple, as shown in Example 8.15, along with the two remaining overloads/functions, both of which are trivial in nature.

Example 8.15: Constructor, Destructor and Remaining Functions

```
PersonArray::PersonArray(void)
{
}

PersonArray::~PersonArray(void)
{
      RemoveAll();
}

const Person *PersonArray::operator[](int i) const {
      assert(i>=0 && i<(int)m_arp.size());
      if (i<0 || i>Size()) return 0;
      return m_arp[i];
}

int PersonArray::Size() const {
      return (int)m_arp.size();
}
```

In Section 8.5, we perform a lab that incorporates the PersonArray into a Company class.

8.4: Tokenize Class Walkthrough

Walkthrough available in Tokenize.avi

Tokenizing is the name commonly given to the process of taking a string of text and breaking it up into its elemental components, called tokens (or, sometimes, atoms). In this walkthrough, we develop a class, Tokenize, which can be used for this purpose.

8.4.1: Overview of Tokenizing

One of the great strengths of C/C++ is its ability to work with low-level data, such as raw strings. As a result, we often use the language to process input strings from files or users. One important step that is nearly always required in such applications is tokenizing—the breaking up of strings into component prices (known as tokens). For example, before we could begin to interpret an expression such as:

"CAPITAL[T]= (1+INT_RATE)*CAPITAL[T—1] + INCOME[T]"

we'd probably want to break it up into its component prices, e.g.,

"CAPITAL","[","T","]","=","(","1","+","INT_RATE",")", etc.

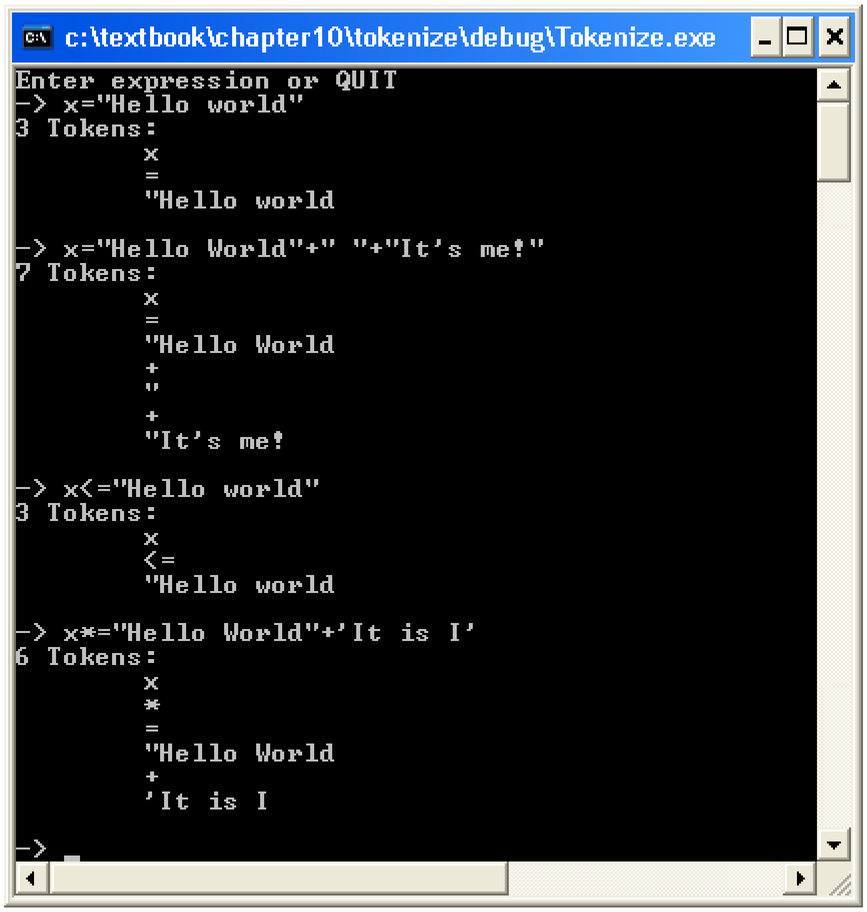
How we decide to break it up is likely to depend on what we plan to use the results for. Any time we break a string up, however, there are likely to be certain issues that we need to address. These include:

- Are there special break characters? So we don't have to write spaces between every token, certain characters can be defined to automatically break up tokens (e.g., '+' or ','). Alternatively, we might choose to identify characters that *don't* break up tokens, e.g., letters, numbers and special characters, such as '_'.
- Are there characters that keep tokens together. For example, the " and ' are often used to allow tokens containing white space and break characters to be defined, such as "Hello, World"
- Are there certain break characters that can be combined? For example, in C/C++, the '<' and '=' characters are both break characters. When they appear together, however, it's easier to work with them if they come out as a single token (i.e., as "<=" instead of "<" and "=").

With this in mind, we'll create a fairly powerful tokenizing class that can be customized to break up many different types of expressions. The tokenizing process will be based on a number of assumptions:

- Any non-printing character (excepting those within quoting delimiters, such as " or ') is assumed not to be part of a token
- Any non-white character that is not between 'A' and 'Z', 'a' and 'z' or '0' and '9' is assumed to start a new token—unless the character is explicitly defined to be a non-breaking character (default non-breaking characters are '.' and '_')
- If a delimiting character is encountered (defaults are " and '), all characters that follow are assumed to be part of the same token until a matching delimiting character is encountered. Since the ending delimiter must match the starting delimiter, it is possible to enclose apostrophes (') within double-quotes, or double-quotes within apostrophes.
- The leading delimiter in a delimited token (e.g., a token within quotes) is included in the token, while the ending delimiter is NUL'ed out. Thus, the quoted string "Hello" within an expression would come back as the token "Hello (see Figure 8.4). This proves to be useful device when translating expressions—since it's easy to ignore the leading character of a string if it's not wanted (i.e., just add 1 to the address), but it can tedious to strip off the trailing delimiter.

The operation of the class is illustrated in Figure 8.4.



```
Enter expression or QUIT
-> x="Hello world"
3 Tokens:
        x
        =
        "Hello world

-> x="Hello World"+" "+"It's me!"
7 Tokens:
        x
        =
        "Hello World
        +
        "
        +
        "It's me!

-> x<="Hello world"
3 Tokens:
        x
        <=
        "Hello world

-> x*="Hello World"+'It is I'
6 Tokens:
        x
        *
        =
        "Hello World
        +
        'It is I

-> _
```

Figure 8.4: Output of Tokenize() function test

An example of a main() function using the Tokenize class to produce the output in Figure 8.4 is presented in Example 8.16.

Example 8.16: Application of Tokenize Class

```
int main(int argc,char *argv[])
{
	char buf[256]={0};
	vector<string> arToks;
	Tokenize tok;
	bool bContinue=true;
	cout << "Enter expression or QUIT" << endl;
	do
	{
		cout << "-> ";
		cin.getline(buf,255);
		if (stricmp(buf,"Quit")==0) bContinue=false;
		else {
			if (tok.TokenizeString(buf,arToks)) {
				int i;
				cout << (int)arToks.size();
				cout << " Tokens:" << endl;
				for(i=0;i<(int)arToks.size();i++) {
					cout << "\t" << arToks[i].c_str() <<
endl;
				}
			}
			else {
				cout << "Illegal input line..." << endl;
			}
			cout << endl;
		}

	}
	while(bContinue);
	return 0;
}
```

8.4.2: Tokenize Class

The Tokenize class is designed to allow us to create a customizable tokenizer. The class, declared in Example 8.17, consists of two key components:

- Data members that customize the tokenizing behavior
- A function, TokenizeString(), that takes a NUL-terminated input string and creates a vector<string> collection of tokens.

Example 8.17: Tokenize Class declaration

```
// Default global variables
extern const char *szNonBreaks;
extern const char *szDelimiters;
extern const char *arJoins[];
extern const int nJoinCount;

class Tokenize
{
public:
	Tokenize();
	Tokenize(const char *szNonB,const char *szDelim,
		const vector<string> &arJn,bool bHide=false)
	{
		Initialize(szNonB,szDelim,arJn,bHide);
	}
	void Initialize(const char *szNonB,const char *szDel,
		const vector<string> &arJn,bool bHide=false);

	bool IsBreak(char cChar) const;
	bool TokenizeString(const char *buf,vector<string> &arDest)
const;
	size_t TokenLength(const char *szTok) const;

	// Accessors
	string NonBreaks() const {return m_szNonBreaks;}
	void NonBreaks(const char *szNB){m_szNonBreaks=szNB;}
	string Delimiters() const {return m_szDelimiters;}
	void Delimiters(const char *szD){m_szDelimiters=szD;}
	void SetJoinedBreaks(const vector<string> &arB) {
		m_arJoins.assign(arB.begin(),arB.end());
	}
	void SetJoinedBreaks(const char *arB[],int nCount) {
		m_arJoins.clear();
		for(int i=0;i<nCount;i++) {
			m_arJoins.push_back(arB[i]);
		}
	}
	void GetJoinedBreaks(vector<string> &arDest) const {
		arDest.assign(m_arJoins.begin(),m_arJoins.end());
	}
	bool HideBreaks() const {return m_bHideBreaks;}
	void HideBreaks(bool bH){m_bHideBreaks=bH;}

protected:
	string m_szNonBreaks;
	string m_szDelimiters;
	vector<string> m_arJoins;
	bool m_bHideBreaks;
};
```

The external (global) variables—which provide default values for the various member variables when the default constructor is used—have the following significance:

- *szBreaks:* a string containing all the non-alphanumeric characters (i.e., characters that aren't letters and numbers) that are not break characters. This turns out to be a more economical way to handle breaking characters than specifying them individually.
- *szDelimiters:* a string containing the characters that can be used to delimit a token that contains blanks or break characters. These characters would normally be quotes or apostrophes, but might also include special charters so that one could delimit strings using notation such as %This is a string%. The tokenizer requires that the same character start and end each string.
- *arJoins[]:* A lexically sorted (i.e., using ASCII ordering) array containing break strings of break characters that we want to combine, such as "<=" or "&&"
- *nJoinCount:* a constant integer that contains the number of join strings in arJoins.

The values for these global variables are presented in Example 8.18. It is worth noting that nJoinCount is computed by the compiler—using the sizeof operator—so that additional join strings (such as "++" or "+=") could be added without modifying the count. It is critical, however, that any new strings be added in the right order (using ASCII sorting) because binary search is used to find out if a given operator encountered in the expression being tokenized is present.

Example 8.18: Values for Tokenize() default values

```
const char *szNonBreaks="_.";
const char *szDelimiters="\"\'";
// arJoins must be lexically sorted...
const char *arJoins[]= {
      "!=","&&","<=","==",">=","||"
};
const int nJoinCount=sizeof(arJoins)/sizeof(char *);
```

Two member functions, TokenLength() and IsBreak() are defined to make the TokenizeString() function itself more compact. TokenLength(), shown in Example 8.18, is passed in the address of the start of a token. It then determines token length using the following rule:

- If the token begins with a string delimiter, it finds the matching delimiter, returning the length of the string including the starting delimiter (but not the ending one). It returns 0 (signifying an error) if a matching delimiter is not found.
- If the token begins with any other non-break character, it returns the number of characters until the next white (' ' or less) or break character is encountered
- If the token begins with a break character, it searches the m_arJoins array (using a binary search routine) to find out if a matching multi-character break is present. If not, it returns 1. If a match is present, it returns the length of the matching break string.

The IsBreak() function, also shown in Example 8.19, takes a single character as an argument. It returns false if the character is alphanumeric (a letter or a number) or matches one of the specified non-breaking characters. Otherwise, it returns true.

Example 8.19: Tokenize helper functions

```
size_t Tokenize::TokenLength(const char *szTok) const
{
      const char *pDel=(m_szDelimiters.size()==0) ? 0 :
            strchr(m_szDelimiters.c_str(),*szTok);
      size_t nLen;
      // Check first for delimiters
      if (pDel!=0) {
            for(nLen=1;szTok[nLen]!=0 && szTok[nLen]!=*pDel;nLen++){};
            if (szTok[nLen]==0) return 0;
      // Reaching NUL terminator implies unbalanced delimiter
            return nLen;
      }
      // Check next for normal token
      else if (!IsBreak(*szTok)) {
            size_t nLen=1;
            for(nLen=1;szTok[nLen]>' ' &&
!IsBreak(szTok[nLen]);nLen++){}
            return nLen;
      }
      else if (m_arJoins.size()==0) return 1;
      // Checking for joined breaks
      int nStart=0;
      int nEnd=(int)m_arJoins.size();
      int nFlag;
      nFlag=memcmp(szTok,m_arJoins[nStart].c_str(),
            m_arJoins[nStart].length());
      if (nFlag<0) return 1;
      else if (nFlag==0) nEnd=nStart;
      while(nEnd-nStart>1) {
            int nTest=(nStart+nEnd)/2;
            int nFlag=memcmp(szTok,m_arJoins[nTest].c_str(),
                  m_arJoins[nTest].length());
            if (nFlag>0) nStart=nTest;
            else if (nFlag<0) nEnd=nTest;
            else nStart=nEnd=nTest;
      }
      if (nStart==nEnd) return m_arJoins[nEnd].length();
      else return 1;
}

bool Tokenize::IsBreak(char cChar) const
{
      if ((cChar>='A' && cChar<='Z') || (cChar>='a' && cChar<='z') ||
            (cChar>='0' && cChar<='9') ||
            (strchr(m_szNonBreaks.c_str(),cChar)!=0)) return false;
      return true;
}
```

The TokenizeString() function, presented in Example 8.20, takes two arguments—the NUL-terminated string being tokenized and the vector<string> array where the tokens are to be placed. It returns true if the string being tokenized is valid, false otherwise (most likely as a result of non-matching delimiters).

Example 8.20: TokenizeString() function

```
bool Tokenize::TokenizeString(const char *buf,
                                 vector<string> &arDest) const
{
      arDest.clear();
      size_t nPos=0,nLen=strlen(buf);
      if (nLen==0) return false;
    string src=buf;
      const char *szDel=m_szDelimiters.c_str();
      while(nPos<nLen)
      {
            while ((buf[nPos]<=' ' || (IsBreak(buf[nPos]) &&
HideBreaks()))
                  && nPos<nLen) nPos++;
            if (nPos==nLen) break;
            size_t nTokLen=TokenLength(buf+nPos);
            if (nTokLen==0) return false; // Illegal token
            arDest.push_back(src.substr(nPos,nTokLen));
            if (strchr(m_szDelimiters.c_str(),src[nPos])!=0) {
                  // If token begins with string delimiter
                  // need to skip past matching delimiter
                  nTokLen++;

            }
            nPos=nPos+nTokLen;
      }
      return true;
}
```

The basic operation of the function is as follows:

- We begin by emptying the destination vector<string> object (arDest).
- *nPos* is used to keep track of the current position in the expression string, and *nLen* holds the length of the expression. *str* holds a **string** copy of our expression, which simplifies extracting tokens from the expression.
- We loop until nPos>=nLen, the end of the string. Within the loop:
    - We skip over non-token characters. These will always include white characters (' ' or less) but may also include break characters if m_bHideBreaks is set to **true**. This might be the case if we were tokenizing a comma-delimited string, for example. The loop keeps skipping characters until the beginning of a new token (or the end of the expression) is encountered.
    - We call TokenLength() to find out how many characters are in the current token.

- We use the **string** substr() member to extract the token from the string, the **vector<>** push_back() member to add it to our destination vector<string> object.
- We add the token length to the current position to position us past the end of the current token. If a delimited token was extracted, we add one to the token length to skip the matching delimiter.

The remaining functions, Initialize() and the constructor functions, are presented (for the sake of completeness) in Example 8.21.

Example 8.21: Tokenize Class Initializer and Constructor functions

```
Tokenize::Tokenize()
{
   // Uses external values
      m_szNonBreaks=szNonBreaks;
      m_szDelimiters=szDelimiters;
      SetJoinedBreaks(arJoins,nJoinCount);
      m_bHideBreaks=false;
}

void Tokenize::Initialize(const char *szNonB,const char *szDel,
            const vector<string> &arJn,bool bHide)
{
      m_szNonBreaks=szNonB;
      m_szDelimiters=szDel;
      SetJoinedBreaks(arJn);
      m_bHideBreaks=bHide;
}
```

8.5: Lab Exercise: Company Class

Walkthrough available in Company.avi

In this section, we specify a simple class that we'll call the Company class. Its primary purpose is to manage a collection of employees who, for the time being, will be implemented as Person objects. This class is relatively straightforward to build, and will be an important building block of the inheritance lab exercises at the end of Chapters 9 and 10.

8.5.1: Class Overview

The Company class is, for the most part, a class that manages a collection of employees. Furthermore, for the time being, we will just treat our newly create Person object as an employee, although we will be extending what constitutes an employee in Chapter 9.

The basic functionality we want to extend to the class is the following:

- The ability to add and remove employees
- The ability to find an employee using a lookup key
- The ability to edit employees
- The ability to load and save company data to a binary file stream
- The ability to import and export employees from a fixed length text file format

Most of these functionalities already exist within the PersonArray class. The major job in the lab is therefore to implement the Company interface through which PersonArray capabilities are accessed. In a very real sense, this is similar to what the PersonArray implementation itself did for the vector<Person*> class.

8.5.2: Class Specifications

The key functions to be implemented for the Company class are described in Table 8.4.

Menu Option No.	Top Level Company Functions
1	**void ChangeName() const** Used to change the name of the Company object. Prompts user for new name.
10	**void DisplayAll() const** Displays the company name and all the Employee objects.
11	**void LoadFromFile()** Prompts the user for a file name, opens the file for reading, then loads the Company data from the file.
12	**void SaveToFile() const** Prompts the user for a file name, opens the file for writing, then saves the Company data to the file.
101	**void AddEmployee()** Prompts the user for an Employee ID and, if the ID is valid (i.e., not already present), allows the user to edit and add the Employee to the PersonArray collection.
102	**void RemoveEmployee()** Calls FindEmployee(true) to access the Employee object, displaying the data. Prompts the user for a confirmation and, if the user confirms, removes the Employee object from the PersonArray collection.
103	**void EditEmployee()** Calls FindEmployee(false) to access the Employee object, then allows the user to edit the Employee.
104	**Person *FindEmployee(bool bDisplay=true) const** Prompts the user for an Employee ID and, if the ID is valid returns a pointer to the employee. If bDisplay is true, displays the Employee object's data.
105	**void ListEmployees() const** Displays the data for all employees in the PersonArray, one employee per line.
106	**void ImportEmployees()** Prompts the user for a file name (presumably a text file), then loads the fixed format employee records into the PersonArray.
107	**void Export Employees() const** Prompts the user for a file name (presumably a text file), then saves the employee records from the PersonArray in a fixed text format.
N/A	**void Interface()** Enters a loop that displays a menu (i.e., calling the Menu() function described below), prompts the user for an option number, then selects one of the options above (i.e., calling the Option() function described below).

Table 8.4: Top Level Menu Options for Company Class

Most of the "useful" class members perform tasks comparable to those of the PersonArray class. There are, however a few additional members:

virtual void Menu() const;
The Menu() option should display the options (10-12, 101-107) and descriptions to the screen. e.g.,

1 – Change Name
10 – Display all data
11 – Load
12 – Save
101 – Add an Employee
102 – Remove an Employee
103 – Edit an Employee
etc.

This particular implementation will help us when we start inheriting the Company class in Chapter 9.

virtual bool Option(int nOption);
Takes the integer value of an option typed in by the user and calls the appropriate top-level function (e.g., implemented with a case statement). The function should return **true** unless the option selected is not one of the valid option numbers (i.e., 101-108), in which case it should return **false**. e.g.,

```
switch(nOption)
{
    case 101:
    {
        AddEmployee();
        break;
    }
    case 102:
    {
        RemoveEmployee();
        break;
    }
    // More options
    default:
    {
        return false;
    }
}
return true;
```

void ImportEmployees(istream &in);
void ExportEmployees(ostream &out) const;
Uses the argument to save or load employees using a text stream in fixed text format. Obviously, this should be the same format already supported by the Person object. These functions can be used with sample data provided in the lab folder.

const Person *GetEmployee(int i);
int EmployeeCount() const;
Used to allow random access of employees, in place of defining an iterator. Although we could overload the [] operator to do this as well, thinking of a company as an array of employees doesn't really clarify the problem and, in Chapter 10, we'll make enhancements to the class (though inheritance) that would cause the notation to become positively confusing.

The declaration that should be used for the Company class is presented in Example 8.22. Only simple accessor functions have been omitted, which should be defined according to the conventions you have already seen many times. No copy constructor or assignment overload is required, for the same reasons presented in the design of the PersonArray class.

Example 8.22: Company class Declaration

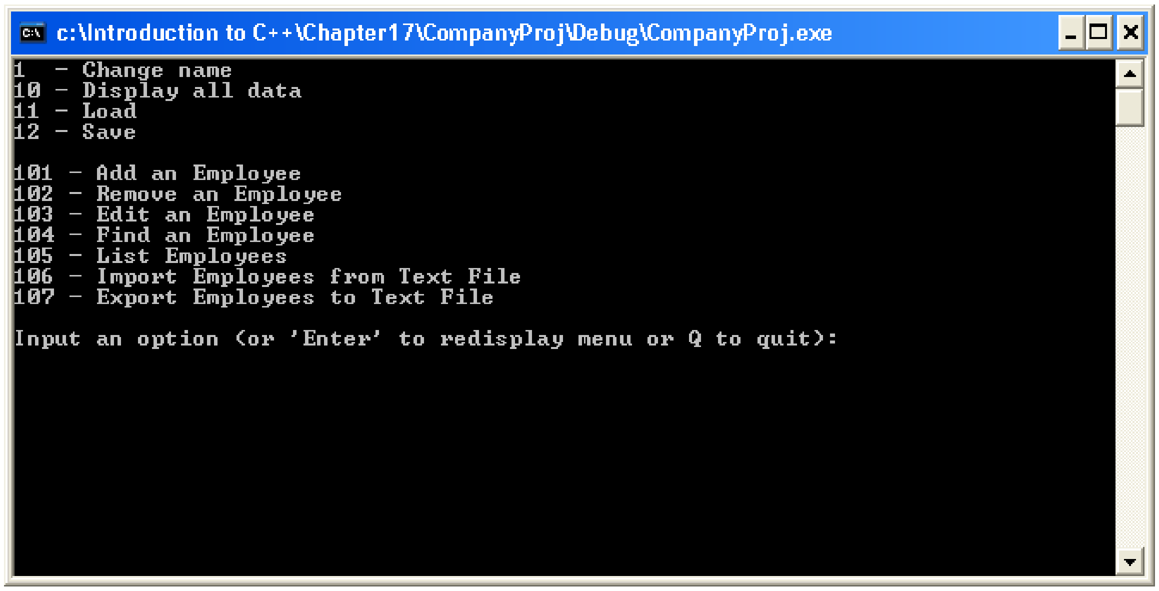
```
class Person;

class Company
{
public:
	Company(void);
	virtual ~Company(void);

	void RemoveAll();

	bool AddEmployee(const Person &pers);
	void RemoveEmployee(const char *szKey);
	void RemoveEmployee(const string &szKey);
	const Person *FindEmployee(const char *szKey);
	const Person *FindEmployee(const string &szKey);
	bool EditEmployee(const char *szKey);
	bool EditEmployee(const string &szKey);

	void Load(fstream &strm);
	void Save(fstream &strm);

	void Display(ostream &out);

	void LoadFromFile();
	void SaveToFile() const;

	void ImportEmployees(const char *szFileName);
	void ExportEmployees(const char *szFileName) const;

	const Person *Employee(int i);
	int EmployeeCount() const;

protected:
	PersonArray m_arPerson;
	string m_szCompanyName;
	string m_szFileName;
};

fstream &operator>>(fstream &strm,Company &ar);
fstream &operator<<(fstream &strm,const Company &ar);
```

8.6.3: Instructions

The Company class has enough members so that you need to be a bit methodical about testing. You should begin writing a simple Menu() function that lists the specified options, and an Option() function that implements the selection of the option. You can then write an Interface() function should contain a non-terminating loop that works roughly as follows:

- Calls Menu() to display the options if some local display variable (e.g., bDisplay) is true.
- Sets the display variable to **false**.
- Displays the prompt, which is not part of the Menu() (the reason for this involves facilitating inheritance in Chapter 17, as we shall see).
- Prompts the user for the option, read in as a NUL-terminated string
- Checks for an empty line—if so, it sets the display variable to **true** (causing the menu to refresh the next iteration of the loop)
- Checks for the 'Q' option and returns **true** if found.
- If not 'Q', sends the integer value to the Option() function
- If Option() returns **false**, informs the user that the option was illegal and sets the display variable to **true**.

The screen display for a Company object when the Interface() member is called should be roughly the same as Figure 8.5.



c:\Introduction to C++\Chapter17\CompanyProj\Debug\CompanyProj.exe

```
1  - Change name
10 - Display all data
11 - Load
12 - Save

101 - Add an Employee
102 - Remove an Employee
103 - Edit an Employee
104 - Find an Employee
105 - List Employees
106 - Import Employees from Text File
107 - Export Employees to Text File

Input an option (or 'Enter' to redisplay menu or Q to quit):
```

Figure 8.5: Example Company::Interface() display

Once your Interface() function is implemented, you are free to choose the order of implementation. A reasonable order would be:

- Adding an employee
- Employee() and EmployeeCount(), useful in many of the other functions
- Listing employees
- Loading and saving employees (saving you lots of subsequent typing)
- Deleting employees

- Editing employees. Recall that when you edit an employee, you could be changing its name. As a result, a common procedure is to delete then add the employee after the edit has been performed.
- Exporting employees. Check the file with a text editor once you've created it.
- Importing employees.

Whatever you do during this process, don't change how the Company class is declared. We will be modifying it in the next chapter.

8.7: Review and Questions

8.7.1: Review

The STL **string** is a templated **basic_string<char>** object designed to work with 1-byte characters. It supports all the standard string operators, including comparison (>, <, <=, >=, != and ==), assignment (=, +=), concatenation (+), character element access([]) and insertion/extraction using a text stream (<< and >>). It also provides member functions for searching and generating substrings. With **string** objects available, it is possible to avoid the awkward syntax that has been required for working with strings that has been around since the introduction of C in the late 1960s and early 1970s.

STL collections support a template-specific iterator data type, creating a common means of sequentially accessing collection elements regardless of how the collection is constructed (sometimes referred to as its shape). Once an iterator has been declared, it supports using an interface closely reminiscent of standard pointer arithmetic. For example, if nPos is an iterator set to a particular collection position:

*nPos accesses the collection element at that position
nPos[offset] accesses collection elements for shapes supporting random access
*(nPos+offset) also accesses collection elements for shapes supporting random access
nPos++ moves the iterator to the next element in the collection
nPos-- moves to the previous element in the collection

In addition, two class member functions are defined for use with iterators:

- begin() returns the iterator value at the start of the collection.
- end() returns a special iterator value signifying we have incremented the last element of the collection

The two primary advantages of using iterators are: 1) the programmer can use the same approach to gain access to elements in different shapes, and 2) using iterators can reduce the number of code changes required if the programmer changes an object's underlying collection shape.

The **vector<>** template implements a type-safe dynamic array object for data types supporting assignment and copying. Its interface includes a [] overload to get at individual elements, along with a variety of insertion and deletion members. There are a number of advantages of using vector<> objects over true C++ arrays. These include:

- Dynamic resizing as elements are added (using the insert() and push_back() members).
- Ability to remove elements (using the remove() and pop_back() members)
- Bounds checking performed as elements are accessed using the [] operator
- size() operator returns the number of elements in the **vector<>**
- Access can be performed using an iterator

To create an iterator for use with a vector<> collection, the declaration:

vector<*type*>::iterator *iter-var-name* ;

is made, with *type* matching the type used to create the template object you intend to apply the iterator to. Thereafter, the begin() member of the template object can be used to initialize the iterator. This same procedure applies to other shapes, including **string** objects (where the iterator accesses characters within the string).

The only drawback of using vector<> and string<> objects in place of true arrays and NUL-terminated strings is one of performance. On today's computers, however, the performance costs tend to be slight when contrasted with the benefits of code clarity and safety gained from using STL objects.

8.7.2: Glossary

Atom – Another term sometimes used for a token.
Bounds Checking – Ensuring that array positions are not accessed that are outside of the valid coefficients of the array.
Collection – A set of objects that is managed by some other object, the collection object.
Collection shape – The set of performance properties for collection access and modification that determine a collection's most appropriate uses.
Dynamic Resizing – The ability of a collection to change its size, as necessary, to accommodate the addition and removal of elements.
Iterator – An object that can be uses to traverse the elements of a collection using a pointer-style interface. Each STL collection template defines an iterator data type as part of its implementation, and a common interface is used for all iterators.
Random access – The ability to access elements in a collection without traversing other elements, often implemented with the [] operator.
Serialization – A term frequently used to refer to loading and saving objects in binary form.
Shape – see collection shape.
Singly Linked List – A linked list shape that supports processing from front to back only.
Standard Template Library (STL) – A library of template-based objects that began to replace the original C++ standard libraries in the mid-1990s and became part of the ISO standard in 1998.

string – An STL template class that implements an encapsulated string object.
Template – A parameterized way to declare a class or function in C++ that causes source code to be generated when a template object/function is created with specified parameters. Templates are sometimes likened to code factories.
Token – A substring in an expression that is not meant to be further decomposed, such as a variable name, operator or keyword
Tokenizing – The process of breaking string expressions into component substrings, referred to as atoms or tokens.
vector<> – An STL template class that implements a dynamic array. It takes a single parameter, the data type of the array elements.

8.7.3: Questions

8.1: Counting characters in a string. Write a function, CountChars(), that returns the number of matching characters in a string, and is prototyped as follows:

```
int CountChars(const string &str,char cTarget);
```

where str is the string being searched and cTarget is the character we're looking for.

8.2: String trimming. Implement an LTrim() function, modeled after Trim() in Example 8.8, that trims leading blanks from a string and returns the result.

8.3: Case change. Implement MakeUpper() and MakeLower() functions that are prototyped as follows:

```
void MakeUpper(string &str);
void MakeLower(string &str);
```

The functions should make their argument strings entirely upper/lower case.

8.4: Search and replace strings. Write a function, ReplaceAll() that is prototyped as follows:

```
ReplaceAll(vector<string> &ar,const char *s1,const char *s2);
```

and replaces all occurrences of s1 within ar with the string s2.

8.4: Search and replace vectors. Write a function, ReplaceAll() that is prototyped as follows:

```
ReplaceAll(vector<string> &ar,const vector<string> &v1, const vector<string>
&v2);
```

and replaces all occurrences of vector v1 within ar with the vector v2. An occurrence of v1 is present when a sequence of elements in ar matches the contents v1 for all elements in v1.

8.6: Removing strings from a vector<> object. Write a function, RemoveAll() that is prototyped as follows:

```
ReplaceAll(vector<string> &ar,const char *s1);
```

and removes all occurrences of s1 within ar.

8.7: Reversing a vector<> object. Write a function, Reverse() that reverses the elements in a vector<string> object and is prototyped as follows:

```
ReplaceAll(vector<string> &ar);
```

Question 8.8-8.10 involve variations of the PersonArray class.

8.8: Copying PersonArray. Implement a copy constructor and assignment operator for the PersonArray class, implemented in 8.3.3.

8.9: Finding matching first names. Implement a PersonArray member function, FindMatchingFirstNames(), that finds all people with first names matching the target name and places them in matching target array. The function should be prototyped as follows:

```
void PersonArray::FindMatchingFirstNames(const char *szName,
            PersonArray &ar) const;
```

where szName is the first name being searched for and ar is the array where the matching elements will be placed.

8.9: Finding matching last names. Implement a PersonArray member function, FindMatchingLastNames(), that finds all people with first names matching the target name and places them in matching target array. The function should be prototyped as follows:

```
void PersonArray::FindMatchingLastNames(const char *szName,
            PersonArray &ar) const;
```

where szName is the first name being searched for and ar is the array where the matching elements will be placed. This function should do an "intelligent" search for matching last names, and should not just iterate through the entire array.

Question 8.11-8.15 involve enhancements to the Tokenize class.

8.11: Handling escape (\) characters. Modify the Tokenize() function so that escape character sequences *within a delimited string* (e.g., \t, \x0A, \\) are translated in a manner comparable to that used in C++. (*Hint:* You'll probably want to write a separate member function that processes tokens prior to adding them to the destination vector).

8.12: Specified break characters. Modify the Tokenize class so that either specific break *or* non-break characters can be specified. There are many ways that this can be implemented (e.g., a Boolean member that identifies whether m_szNonBreaks refers to breaks on non-breaks, two separate members holding break strings, etc.)

8.13: Joining string segments. In C++, any time double quoted expressions are adjacent to each other, without any other intervening tokens, they are combined. For example:

```
char *pStr="Hello"
        " World";
```

will cause pStr to point to "Hello World". Modify the tokenizer so that adjacent delimited tokens, beginning with the same delimiter, are combined.

8.14: Matching up container break characters. Modify the tokenizer class to take another data member containing break characters that must be matched (e.g., "()[]{}"). Once an expression has been tokenized in the TokenizeString() function, you should ensure that appropriate matching (including nesting) is present—returning false if improper nesting is detected. For the present time, you can allow improper nesting of different types (e.g., "(example[is) bad]") to escape detection. This will be remedied in an end-of-chapter question 15.10 in Chapter 15 (Recursion and Sorting).

8.15: Regenerating a string from a token array. Write a function that creates the shortest possible string from an vector<string> containing tokens. The function, prototyped as follows:

```
string Tokenize::GenerateString(const vector<string> &arToks) const;
```

should insert spaces only where necessary (e.g., not before or after break characters) and should place the proper training delimiter on delimited string objects.

Chapter 9

Inheritance

Executive Summary

Chapter 9 introduces the use of inheritance in C++. Inheritance provides important representational and computational capabilities not otherwise offered by structured programming languages. It is particularly well suited for promoting code reuse and providing a means of representing certain types of abstract relationships between different types of data.

The chapter begins with brief overview of inheritance in general terms, contrasting it with composition. We then explore how inheritance relationships can be specified in a C++ program, and how initialization is accomplished when inherited objects are constructed. The **static_cast** and **dynamic_cast** C++ operators are also explained. The chapter concludes with a lab exercise that builds upon the Person/Company classes introduced in Chapter 8.

Learning Objectives

Upon completing this chapter, you should be able to:

- Explain the benefits of inheritance
- Explain how inheritance differs from composition
- Create a simple UML generalization/specialization diagram using FlowC
- Create inherited classes in C++
- Control construction in inherited classes
- Differentiate between polymorphic and regular member functions
- Call base class member functions within other member functions
- Use inheritance to create a network of classes as part of an application

9.1: Introduction to Inheritance

Inheritance provides a powerful tool for defining abstract data types (ADTs). Instead of starting with a "blank slate" when we create a class, we can use one or more previously defined classes as a starting point. Doing so provides two important benefits:

1. It promotes reuse of existing objects
2. It implements a new form of relationships between objects that can be very useful in visualizing certain types of programming situations

In this section, we explore inheritance from a general perspective.

9.1.1 What is Inheritance?

Inheritance is the ability to define a class using a previously defined class as a starting point. In creating an inherited definition, there are two types of objectives we are usually trying to accomplish:

- *Adding* capabilities that are not present in the original class
- *Overriding* behaviors in the original class that are not appropriate in the inherited class

To illustrate these capabilities, let us suppose we wanted to define a series of classes to encapsulate different animals. In doing so, we might want to define a series of member functions to access key data elements relevant to all animals, such as:

- Body temperature (e.g., warm blooded or cold blooded)
- Does it lay eggs? (true or false)
- Does it give live birth?

In addition, we'd probably need some members relevant to specific animals or groups of animals. These might include things like:

- Running speed
- Swimming speed
- Hair color

We might also find it useful to view our classes in taxonomy, organizing our classes into groups and subgroups with common characteristics. This is, in fact, what biologists do when they define a genus, species, subspecies, etc and assign different organisms to it. It is also the same principle used in a C++ inheritance hierarchy. An example is presented in Figure 9.1.

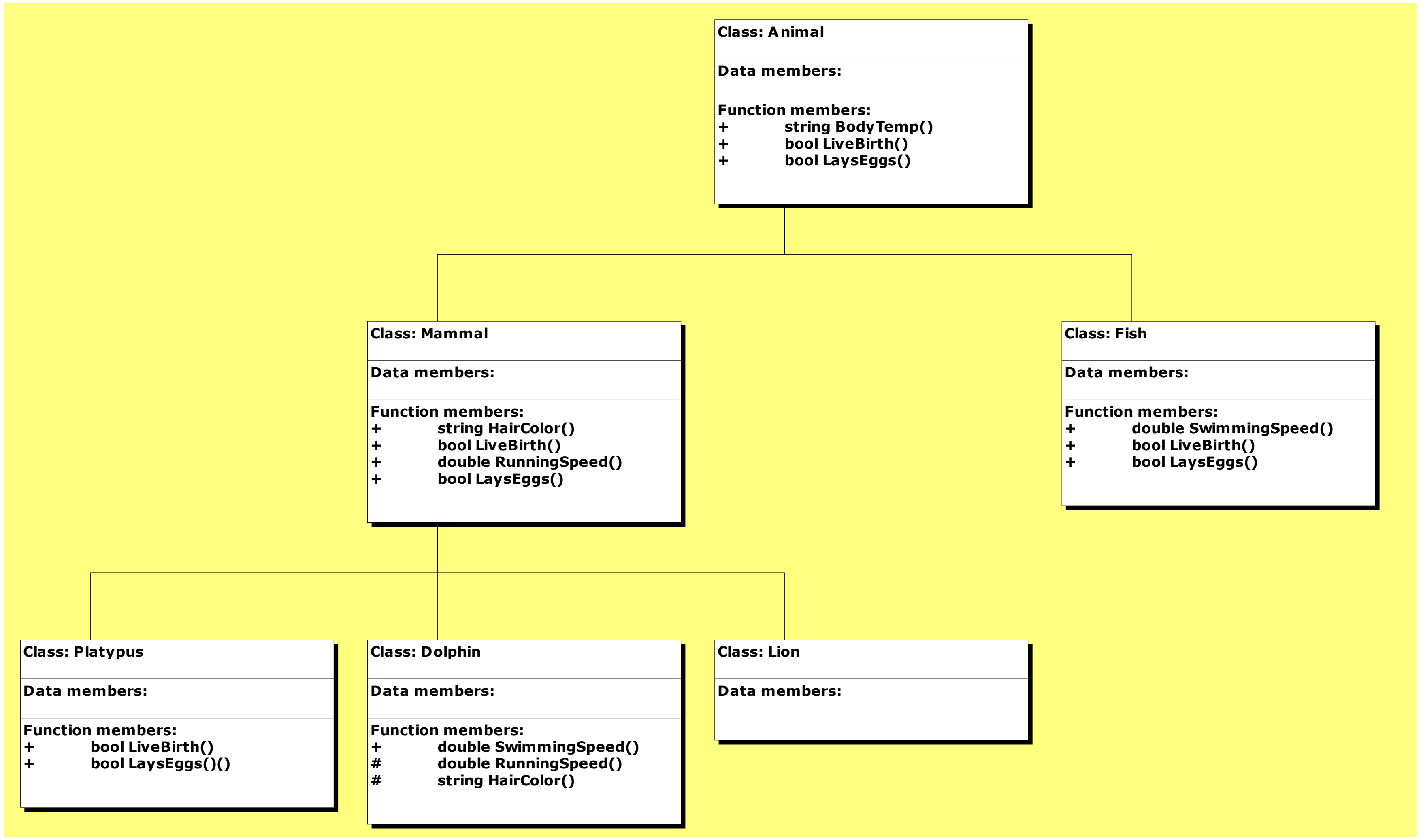


Figure 9.1: Animal inheritance network

In this hierarchy, we start with a class called Animal having three function members, BodyTemp() (e.g., "warm-blooded" or "cold-blooded"), LiveBirth() (e.g., true or false) and LaysEggs() (e.g., true or false). Although these are attributes of the class—meaning that all Animal objects have values for them—there is no meaningful default values for them—since hundreds of thousands of animals fall into both categories of each value.

When we specialize to the Mammal and Fish classes, on the other hand, we may override the values for certain attribute. For example, if we have a Mammal object, we can generally say:

- LiveBirth() returns true
- LaysEggs() returns false
- BodyTemp() is "Warm Blooded"

It might also make sense to add another member, HairColor(), since mammals generally have hair and, perhaps, RunningSpeed(), reflecting the fact that the vast majority of mammals live on land. These are both examples of adding capabilities not present in the base class.

Similarly, for the Fish objects, we'd probably default to:

- LiveBirth() returns false
- LaysEggs() returns true
- BodyTemp() is "Cold Blooded"

It might also make sense to add another member, SwimmingSpeed(), reflecting the fact that the vast majority of fish live in water.

Now, if we were to move down a level to actual species—done in this example for a three select mammals—we can start to see the need for override. In the case of a Lion object we're done—lions fit the default for the mammal class very well. All we'd need to do initialize relevant members with data reflecting things like running speed and the color of the lion's fur. When we specialize to a Platypus object, on the other hand, we need to override the default returns for LiveBirth() and LaysEggs()—since the duck-billed platypus is one of a handful of mammals that actually lays eggs.

Similarly, in the case of the Dolphin, the RunningSpeed() member is no longer applicable and HairColor() doesn't really seem that applicable either (although they might have a strand of hair here or there). Removing an inherited member is usually not allowed in inheritance mechanisms so we might do the next best thing: make it protected to remove it from the class interface. It would make sense to add a SwimmingSpeed() attribute to the Dolphin object.

9.1.2 Benefits of Inheritance

There are three principal benefits to using inheritance in writing programs: conceptual, reuse and organizational. We briefly consider all three.

Conceptual
Taxonomies, such as the Animal taxonomy of Figure 9.1, were used for centuries before OOP was invented as a means of organizing and clarifying relationships between entities. The ability to model such relationships in our code can provide the same type of clarification benefits to both class designers and class users.

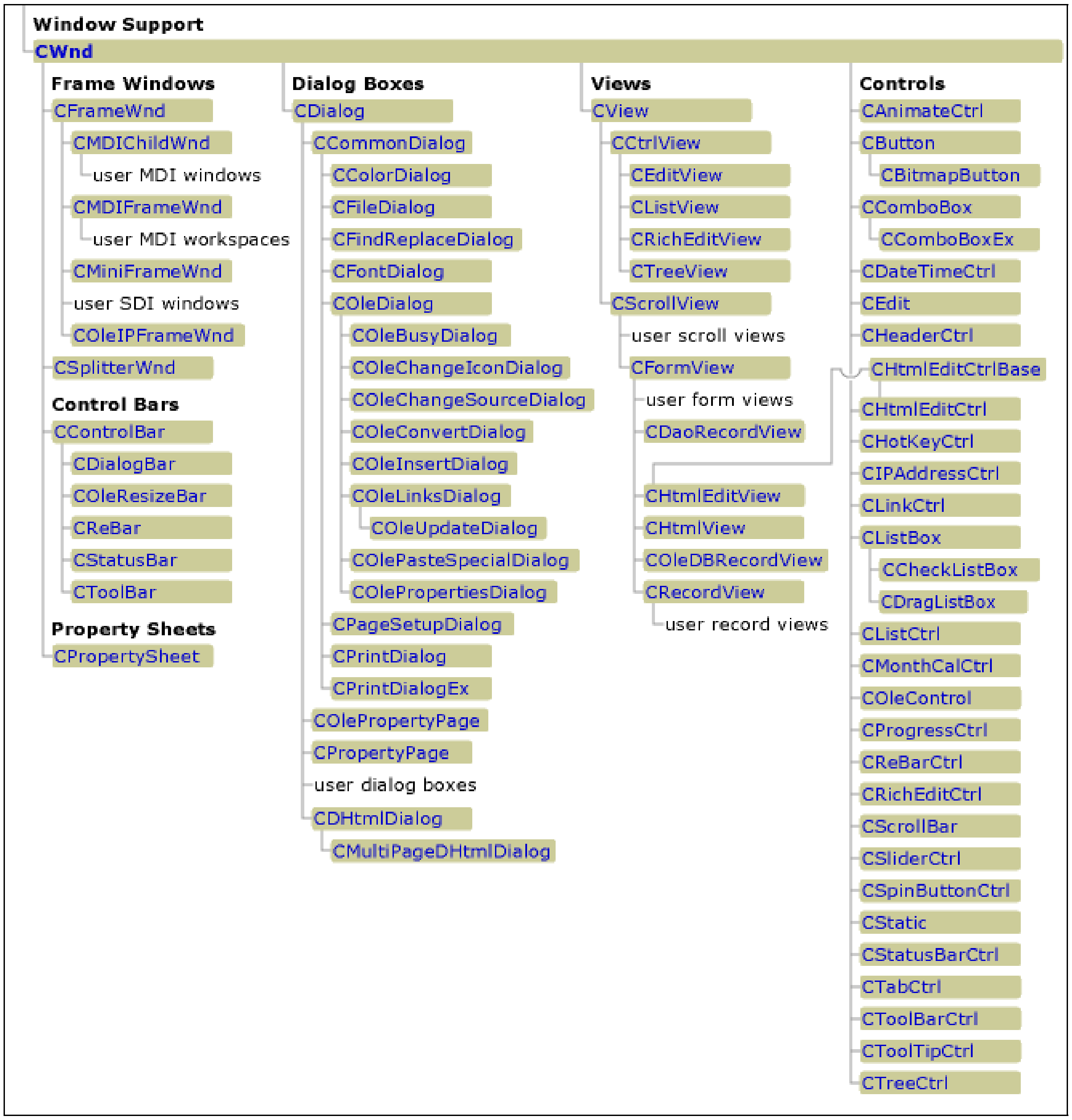


Figure 9.2: MFC Window-Object Hierarchy (from MS Visual Studio .Net documentation)

An example of such a hierarchy can be found in the graphic objects provided by the Microsoft Foundation Classes (MFC). The principal drawing object is a window, encapsulated using the CWnd class. A programmer almost never uses a pure CWnd object, however. Instead, selections from the dozens of specialized window classes—all of which inherit from CWnd—are actually used in the programs we write. Since we know these specialized windows inherit from CWnd, however, we don't have to learn a new interface for each object. Since members for common activities, such as repositioning a window, are established at the CWnd class level, we know that the process for sizing and positioning a combo box will be essentially the same as that for sizing and repositioning our main view window, or the frame window that encloses it. Use of a hierarchy also tends to simplify documentation—as only the additions and overrides made to each class need to be explained.

Reuse
The other related benefit of inheritance is the way it promotes reuse of code. Throughout this text, using existing code wherever possible has been an important guiding principle. Unfortunately, the way such reuse has typically been accomplished—copying the code from one place, dropping it into another place, and then editing the copy—does not scale well. As we reuse more and more code, from more and more different sources, we experience some inherent weaknesses of copies. For example:

- In working on our copy, we discover some ways to improve the original code and make the changes. Unfortunately, these changes are *only* made in the copy, since we don't have time to go back and change the original.
- While we are working on our copy, someone else improves the original—perhaps removing a defect or adding an enhancement. We don't get the word of the improvement, so our edited copy continues to be based on substandard code.

Although inheritance doesn't entirely solve problems like these, it is a good start. Using inheritance, we don't copy the original code, we inherit from it. The changes we make are therefore in the form of the additions or overrides. This means that whenever we need a capability built into the object we're inheriting from—known as the base class—we call the original code, not some copy.

There are some important implications of this process:

- When we discover a defect in the base class, we fix the original base class code, not some copy of that code we have made.
- When the base class is enhanced, as soon as we recompile our program those enhancements become available to our code.

Naturally, using inheritance to promote reuse comes with an important caveat: the interface of all classes in our hierarchy must be preserved. As soon as some programmer comes along an "enhances" a class by pulling out some outdated member function, our code may stop working because we happened to be using that function. Its fine to add members to a class—but removing them is a no-no. The need to preserve the interface also helps to explain why public member functions are so much preferable to public member data in a class. If your data elements are public, you can't change how they are implemented (e.g., moving from 1-byte to 2-byte characters) without changing the interface. When data access is implemented using member functions, on the other hand, we can build necessary conversions into our rewritten member functions to support the old interface, while adding new member functions (or new overloads) to support the additional functionality provided by the new data type. In fact, you will often see member functions with an Ext appended to their name in the MFC. On most cases, these are the result of extensions to the MFC classes that were renamed to avoid conflicts with the original interface.

Organizational
A particularly strong case for inheritance exists where an application can benefit from maintaining a collection of related objects of different types in a single collection. (This, in turn, can allow us to exploit polymorphism, discussed in Chapter 10).

Suppose, for example, we are building a program to keep track of the residents of a zoo. Somewhere in the application, we are likely to have a collection (e.g., a vector) containing all the animals. If all inherit from a common source (e.g., Animal) this is easily done with a:

```
vector<Animal*> arAll;
```

The reason this is possible is that any time inheritance is used, base class pointers can also point to inherited objects. Thus, the following assignments would all be legal and would not generate any errors or warnings:

```
Lion myLion;
Lion *pLion=&myLion;
Mammal *pMammal=&myLion;
Animal *pAnimal=&myLion;
```

That means that if you have a collection of objects all of which inherit from some base class, you can use the base class pointers to hold all the elements in the collection.

The ability to use a base class pointer (or reference) to keep track of many objects is so useful that we sometimes create *abstract classes* at the top levels of the hierarchy to act as position holders and to help define and organize lower levels. What makes a class abstract is that we never intend to create objects from it—only from its child classes.

Examples of classes that would likely be abstract classes in the Figure 9.1 hierarchy are Animal, Mammal and Fish. Whereas Lion, Dolphin and Platypus refer to actual types of animals—images of which spring into your mind when you hear the names—the term mammal typically generates no such mental image. Rather, it serves to organize our collection of animals, along with other classes such as Insect, Amphibian and Primate.

9.1.3 Inheritance vs. Composition

Inheritance is not the only way to take advantage the capabilities of one class within another. We have already seen many examples of another technique, first introduced in Chapter 3, for standard structures, called composition. In composition, we embed one object within another—thereby making it possible to use the embedded object's data and members within the outer object.

Although inheritance and composition offer many of the same benefits, there are also some important differences:

- When inheriting from another class, the default is for the interface of the base class to become almost entirely available within the class being created. In other words, all public members and data in the base class become public members and data in the new class.
- When composing a class, it is more typical to make the embedded object a private data member, then implement whatever interface members are appropriate.

The choice between inheritance and composition is not always clear-cut. The first question to ask is normally: *does an is-a relationship exist*? The can be determined by making the statement:

A(n) X object is a(n) Y object

where X is the new object and Y is the base object. For example, in Figure 9.1, we could make the statement:

A Lion is a Mammal

and the statement makes sense. This would argue for inheritance, rather than composition.

An is-a relationship is not the only criteria for choosing between the inheritance and composition. Another practical issue is the degree to which we actually want to inherit the entire interface of the base class. For example, in the PersonArray walkthrough presented in Chapter 8 (Section 8.3.3) we composed a vector<Person*> object within our PersonArray class to implement a sorted array. Now, it makes perfect sense to say:

A PersonArray is a vector<Person*>

which suggests inheritance could have (or, possibly, should have) been used. The problem here is we specifically did not want a number of the vector<> members to be accessible—such as insert()—since their improper use by the programmer could lead to the array getting out of order. Thus, if we chose to inherit from vector<Person*>, we'd need to make a lot of members private or use the "private" inheritance capability of C++ (to be discussed), both of which tend to negate many of the benefits of inheritance.

Another potential barrier to using inheritance is where many is-a relationships exist. For example, we might have a species taxonomy (such as Figure 9.1) and another taxonomy based on preferred environment (e.g., sea-creature, land-creature, air-creature) and yet another based on preferred food source (e.g., fish-eater, meat-eater, plant-eater). Now each of the following statements makes sense:

A Dolphin is a Mammal
A Dolphin is a Sea-Creature
A Dolphin is a Fish-Eater

To implement them in an inheritance network, however, you'd need multiple inheritance—meaning the ability of a class to inherit characteristics from more than one base class. While such multiple inheritance is supported by C++ (unlike Java and C#, which do not support multiple inheritance), its use is fraught with perils one you start trying to do it (as will be discussed in Chapter 10). For this reason, where many such relationships need to be preserved, composition is often a better alternative.

9.1.3 Inheritance in FlowC

FlowC allows elementary inheritance relationships to be diagrammed, in a form similar to the UML generalization/specialization diagram (e.g., Figure 9.1). The steps to accomplish this are relatively simple:

- To establish an inheritance relationship, you simply specify one or more base classes in the same dialog where member variables are added
- To display the diagram, select a particular class, right click it, and chose inheritance net

How the net will be displayed depends, to some extent on what class is clicked. Figure 9.1, for example, was the inheritance net for Animal. If you selected Lion, on the other hand, the net in Figure 9.3 would be produced. The reason for the difference is that FlowC displays the direct parents and all children for the class selected. Thus, since Lion has only parents, the appearance is linear, whereas Animal provides the entire taxonomy.

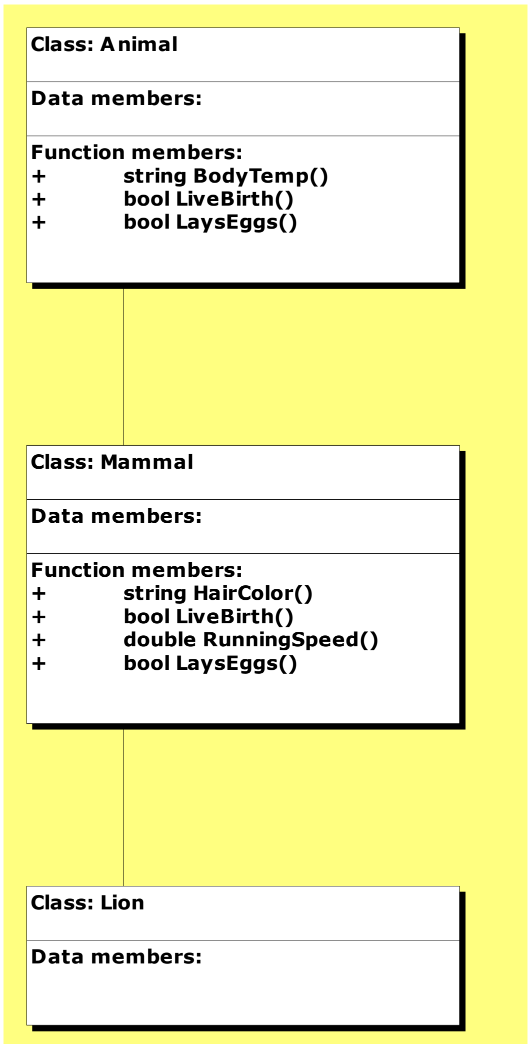


Figure 9.3: Lion inheritance net

Section 9.1 Questions

1. Why might a zoologist argue that Dolphin is really an abstract class?

2. In the statement:

    The choice between inheritance and composition is not always clear-cut (unless the need to maintain a polymorphic collection exists)

    why does the polymorphic collection make the choice more clear-cut?

1. If you need to override a lot of members to make an inherited class work, what might that tell you about your hierarchy?

2. Explain the concerns you might have in creating a sorted array (e.g., the PersonArray) by inheriting from a vector<>?

3. Does making a member protected remove it from the class?

4. How would multiple inheritance impact the display of a class such as Lion (Figure 9.3) in FlowC?

9.2: Inheritance in C++

Walkthrough available in Employee1.avi

In this section we examine how single inheritance relationships can be established in C++. We begin by presenting a simple inheritance scenario, then consider some variations, including:

- Public vs. Private inheritance
- Issues in overriding member functions and data
- Controlling initialization in inheritance

Later, in Chapter 10, we address the additional issues of implementing polymorphism and multiple inheritance.

9.2.1: Establishing an Inheritance Relationship

To demonstrate how easy it is to establish an inheritance relationship, we return to the Person class introduced in Chapter 8, Section 8.3.2, and the PersonArray class, introduced in Chapter 8, Section 8.3.3. As you may recall:

- The Person class implemented a very simple object, with a last name and a first name.
- The PersonArray implemented a sorted array of person objects.

As a lab exercise (Chapter 8, Section 8.4), we then embedded the PersonArray in a Company class, intended to represent the employees in our company.

Unfortunately, our Person class is a pretty lame excuse for an employee object. (Actually, it's a pretty lame excuse for a Person object as well, but we'll let that slide for now). At the very minimum, we'd want our employees to have an employee ID and a title (or position name). At this point, we have two choices:

- We could create a brand new Employee class from scratch (perhaps starting with a renamed copy of the Person.cpp and Person.h files). This would mean we'd also need to create a brand new EmployeeArray class collection.
- We could create an Employee class by inheriting from the Person class. Since that would mean that an Employee object was—by definition—also a Person object, we could still use our PersonArray for storing the collection.

Naturally, the second approach seems like less work and therefore is likely to be the more attractive.

Implementing inheritance is very simple in C++. When declaring the class, we simply add a : after the class name, followed by a comma-separated list of classes we intend to inherit from with access specifiers (usually public or private). In the case of our Employee class, we're only inheriting from one class and we'll want public access to all members (other access types are considered in Section 9.2.3). The Employee class definition is presented in Example 9.1.

Example 9.1: Employee class, first version

```
#include "person.h"

class Employee :
      public Person
{
public:
      Employee(void);
      virtual ~Employee(void);

      string ID() const {return m_szID;}
      void ID(const char *sz){m_szID=sz;}
      string Position() const {return m_szPosition;}
      void Position(const char *sz){m_szPosition=sz;}

protected:
      string m_szID;
      string m_szPosition;
};
```

Looking at the definition—our first attempt—we see that we're defining an Employee to be a Person with the following data members added:

- An ID string (m_szID)
- A Title/Position string (m_szPosition)

After making this change, we can then go into our PersonArray test function (originally presented in Chapter 8, Example 8.9) and change the local pers object from a Person to an Employee, as shown in Example 9.2. After adding:

```
#include "Employee.h"
```

to the top of the .cpp file, the program will then compile and run.

Example 9.2: Change of Person to Employee in demonstration function

```
void TestPersonArray()
{
	PersonArray ar;
	DisplayString("To stop entering people, make a person\'s last name 
\'99\'");
	NewLine();
	Employee pers; // Changed from Person to Employee
	while(pers.LastName()!="99")
	{
		pers.Input();
		if (pers.LastName()=="99") continue;
		int nPos=ar.Add(pers);
		if (nPos<0) {
			DisplayString("Duplicate person encountered!");
			NewLine();
		}
		else {
			DisplayFormatted("Person added at position %i",nPos);
			NewLine();
		}
	}
	// etc...
```

If you run the program, however, you'll quickly see that running and doing what we want are two different things. Since our change to the Person class to create an Employee only involved adding members, our member functions such as Input() still operate as if we were dealing with a Person object. In other words, while we may be able to hold ID and position information in our new Employee class, we don't have any way of getting that information into the class if we use inherited Person members. Thus, we need to discuss overriding members.

9.2.2: Overriding Member Functions

Whenever you inherit from a class, you will typically find that some member functions work just fine, while others need to be modified. In the case of creating our new Employee class, Person members the only members that work just fine (for the time being) in the new class are the simple accessor functions: LastName(), FirstName(), Key() and IsValid(). All our remaining functions—Input(), Display(), Get(), Print(), Load(), Save(), even our constructors—really need to be overridden if they are going to be appropriate for handling Employee objects.

To override a member function, all you need to do is declare one or more member functions with the same name as function you're overriding. You should also be aware that as soon as you override one function with the same name, any overloaded versions in the base class cease to be part of the public interface. As a consequence, one way to "hide" inherited members that aren't relevant (e.g., RunningSpeed() for a Dolphin) is to make a single private or protected version of the function, with no arguments.

Simple Overrides

Given all the changes that need to be made, it is reasonable to question the benefits of inheritance at this point. But overriding a member function is usually very different from rewriting from scratch. Take, for example, our Person::Display() function. That function was originally written as:

```
void Person::Display() const
{
        cout << "Last Name:\t" << LastName() << endl;
        cout << "First Name:\t" << FirstName() << endl;
}
```

What we want to do is not so much throw away that function as to add to it in our Employee::Display() version of the function. Fortunately, C++ allows us to call the base class version of the function within an override by using the scope resolution operator. As a result, out new Employee::Display() member can be written as:

```
void Employee::Display() const
{
        cout << "ID:\t" << ID() << endl;
        Person::Display();
        cout << "Position:\t" << Position() << endl;
}
```

What we have done here is to sandwich the Person::Display() function call between the ID display and the position display. In fact, we can use the same approach in many of our I/O functions, as shown in Example 9.3. The approach fails, however, for one function: Get(), which is used to read a line from a fixed format text file. The problem here is that the Person::Input() function reads a line from a text file but does not give us access to the line that was read. As a result, if we call the base class from a fixed-format file containing one employee per line, we'll lose access to the ID and position data. We'll return to this problem shortly.

Test your understanding: In all our Employee I/O functions, we chose to sandwich the Person data between our ID and position members. Is it necessary that all I/O functions handle the data in the same order?

Example 9.3: Employee I/O functions calling the base class

```
void Employee::Input() {
      char buf[1024];
      DisplayString("ID: ");
      InputString(buf);
      m_szID=buf;
      Person::Input();
      DisplayString("Position: ");
      InputString(buf);
      m_szPosition=buf;
}
void Employee::Display() const
{
      cout << "ID:\t" << ID() << endl;
      Person::Display();
      cout << "Position:\t" << Position() << endl;
}
void Employee::Load(fstream &strm)
{
      strm >> m_szID;
      Person::Load(strm);
      strm >> m_szPosition;
}
void Employee::Save(fstream &strm) const
{
      strm << m_szID;
      Person::Save(strm);
      strm << m_szPosition;
}
string Employee::FixedString() const
{
      char buf[1024]={0};
      int i;
      for(i=0;i<30;i++) buf[i]=' ';
      string str=buf;
      str.replace(0,ID().substr(0,10).length(),ID().substr(0,10));
      str.replace(10,Position().substr(0,20).length(),Position().substr(0,20));
      string pers=Person::FixedString();
      str.insert(10,pers);
      return str;
}

void Employee::Print(ostream &out) const
{
      PrintLine(out,FixedString().c_str());
}
```

The same basic approach of calling the base class can also frequently be applied to constructor functions and assignment overloads—which are not inherited (more about inheriting constructors will be presented in the "In Depth" Section 9.2.4). In the case of our Employee object, we clearly needed to do something about these—since in our

Person base class version, only the names are copied. The modified versions are presented in Example 9.4—with the only real change being made to the Copy() function, which calls the Person::Copy() base class. We need, however, to duplicate the assignment and copy constructor code, however, since—as we noted at the start of the paragraph—these functions are not inherited.

Example 9.4: Employee Overloaded Constructor and Assignment functions

```
	Employee(const Employee &pers) {
		Copy(pers);
	}
	const Employee &operator=(const Employee &pers) {
		if (&pers==this) return *this;
		Copy(pers);
		return *this;
	}
	void Copy(const Employee &pers) {
		Person::Copy(pers);
		m_szID=pers.m_szID;
		m_szPosition=pers.m_szPosition;
	}
```

Benefits of Base Class Calls

The benefits of using base class member functions extend beyond saving us a bit of coding. They can dramatically enhance our ability to modify code. As an example, suppose we decided to add a phone number to our original Person class. This would, of course, entail modifications to nearly all of our original Person member functions, to accommodate assignment, loading, saving, displaying and input of the new piece of data. If, however, our Employee class always calls the base class in its functions, we may be able to enhance the Person class with a phone number *without making any changes to the Employee class*.

This brings us back to the Get() member of the Employee class. In the Person class, Get() was defined as shown in Example 9.5. Since we could not call this function in Employee since we'd loose access to the file line, our first attempt at rewriting the function (also shown in Example 9.5) looks like a more complicated version of the original.

> *Test your understanding:* Why will the Employee version of the Get() function in Example 9.4 cease to work properly when we introduce a phone number into the Person class?

Example 9.5: Person and Employee Get() Member Functions, Version 1

```
void Person::Get(istream &in) {
      char buf[1024];
      in.getline(buf,1024);
      string szBuf=buf;
      m_szLastName=szBuf.substr(0,20);
      Trim(m_szLastName);
      m_szFirstName=szBuf.substr(20,20);
      Trim(m_szFirstName);
}

void Employee::Get(istream &in) {
      char buf[1024];
      in.getline(buf,1024);
      string szBuf=buf;
      m_szID=szBuf.substr(0,10);
      Trim(m_szID);
      m_szLastName=szBuf.substr(10,20);
      Trim(m_szLastName);
      m_szFirstName=szBuf.substr(30,20);
      Trim(m_szFirstName);
      m_szPosition=szBuf.substr(50,20);
      Trim(m_szPosition);
}
```

The problems that can be caused by this type of design are sufficiently severe that it is probably worth redesigning the base class to ensure greater consistency. One way to do this is to look at the Person::Print() function, which broke the writing process into two steps: preparing a string (the FixedString() function) and writing the data to the file (the Print() function). This two-stage process allowed us to call the Person::FixedString() within our Employee::FixedString() function, as previously shown in Example 9.2.

We can modify the Get() member of the Person class using the same process, using another overload of the FixedString() function, as shown in Example 9.6 (with the m_szPhone member already added).

Example 9.6: Revised Person Get(), FixedString() and FixedLength() Members

```
void Person::Get(istream &in) {
      char buf[1024];
      in.getline(buf,1024);
      string szBuf=buf;
      FixedString(szBuf);
}

void Person::FixedString(string &str)
{
      m_szLastName=str.substr(0,20);
      Trim(m_szLastName);
      m_szFirstName=str.substr(20,20);
      Trim(m_szFirstName);
      m_szPhone=str.substr(40,15);
      Trim(m_szPhone);
}
```

With the base class revised in this fashion, we can then make effective use of the base class FixedString() member within our Employee class, as illustrated in Example 9.7. This function:

- Extracts the ID from positions 0-9.
- Creates a substring of the Person information, then calls Person::FixedString() to extract that information
- Extracts the Position/Title information from the portion of the string following the Person segment.

Example 9.7: Revised Get() and FixedString() Members of the Employee Class

```
void Employee::Get(istream &in) {
      char buf[1024];
      in.getline(buf,1024);
      string szBuf=buf;
      FixedString(szBuf);
}


void Employee::FixedString(string &str) {
      int nPersonStr=(int)Person::FixedString().length();
      m_szID=str.substr(0,10);
      Trim(m_szID);
      Person::FixedString(str.substr(10,nPersonStr));
      m_szPosition=str.substr(10+nPersonStr,20);
      Trim(m_szPosition);
}
```

Test your understanding: Explain the role of nPersonStr in the Employee::FixedString() function. How well will this function accommodate changes to the Person base class?

Overriding Design Principles
In considering this example, there are a number of good programming practices relating to overriding that are illustrated. These include:

- In a child class, always call the base class member function whenever it is feasible to do so. This shortens code and enhances reliability.
- Design every class with overriding by child classes in mind. It is surprising how often a class that you thought was only a child becomes a parent. (e.g., the Dolphin class in mentioned in the end-of-section exercises for Section 9.1).
- When you discover yourself overloading a function where you can't call the base class—but would like to—consider redesigning the base class if you have access to it. No matter how experienced you are, you almost never get all the members constructed exactly right the first time.

9.2.3: Public vs. Private Inheritance

Walkthrough available in Protected.avi

When the **public** keyword is used in specifying an inheritance derivation, as shown in Example 9.1, the access specifiers of the resulting class are as follows:

- **public** base-class members remain public
- **protected** base-class members remain protected
- **private** base-class members become inaccessible

protected differs from **private** only in that we can access inherited protected members in inherited classes. Since this is normally the behavior we want, **protected** access is more common than **private** access.

It is also possible to specify protected or private derivation, e.g.,

```
class MyClass : protected BaseClass {…etc…};
```

or

```
class MyClass : private BaseClass {…etc…};
```

When the **protected** derivation specifier is used:

- **public** base-class members become protected
- **protected** base-class members remain protected
- **private** base-class members become inaccessible

When the **private** derivation specifier is used:

- **public** base-class members become private
- **protected** base-class members remain private
- **private** base-class members become inaccessible

As a practical matter, then, any time you use a derivation other than public, you hide the entire interface of the derived class. The only exception to this occurs when no constructor is declared—in which case the implicit constructor will be public.

Although it is not immediately obvious why one would want to hide the entire interface of a child class, we have actually seen an example where this could have been used. In the PersonArray class (Chapter 8, Section 8.3.3), we managed a vector<Person*> object within the class. As noted in Section 9.1.3, however, it makes sense to say:

A PersonArray is a vector<Person*>

The problem with public inheritance, however, was that there were too many vector<Person*> members that could mess up our order. By using private or protected inheritance, however, we could hide those members from the class user, then expose only those that were save. This is illustrated in Example 9.8, where only those vector members we'd allow are exposed. As is also evident from the example, many of these implementations consist of little more than calls to the base class (vector<Person*>) members.

Example 9.8: Example PersonArray Class Using Protected Inheritance

```
class PersonArray : protected vector<Person*>
{
public:
       PersonArray(void);
       ~PersonArray(void);

       const Person *operator[](int) const;
       const Person *operator[](const char *) const;
       const Person *operator[](const string &szKey) const;
       int push_back(Person *);
       int push_back(const Person &pers);
       void erase(int nVal);
       void erase(const char *szKey);
       void erase(Person *p);
       int size() const;
       void clear();

       const Person *Find(const char *szKey,int &nPos) const;
       const Person *Find(const char *szKey) const;
       const Person *Find(const string &szKey) const;
       void Load(fstream &strm);
       void Save(fstream &strm) const;

protected:
       int FindPos(const string &szKey) const;
       int FindPos(const char *szKey) const;
};

// Examples of implemented functions
const Person *PersonArray::operator[](int i) const {
       return vector<Person*>::operator[](i);
}
int PersonArray::size() const {
       return (int)vector<Person*>::size();
}
void PersonArray::erase(int nPos)
{
       assert(nPos>=0 && nPos<(int)size());
       if (nPos<0 || nPos>size()) return;
       delete (*this)[nPos];
       iterator iter=begin();
       vector<Person*>::erase(iter+nPos);
}
void PersonArray::clear()
{
       for(int i=0;i<(int)size();i++) {
              delete (*this)[i];
       }
       vector<Person*>::clear();
}
int PersonArray::push_back(Person *pers) {
       int nPos;
       if (Find(pers->Key().c_str(),nPos)) return -1;
       vector<Person*>::iterator iter=begin();
       insert(iter+nPos,pers);
       return nPos;
}
```

Because private and protected derivation do not offer many coding or reusability benefits over composition—and are generally less well understood by programmers—we'll use composition throughout this text in most situations where it might be possible to use private or protected inheritance.

9.2.4: Initialization During Inheritance

Up to this point, where we have needed to initialize member variables we have done so in the body of our constructor functions. For example, we could define a constructor to initialize the members of our Person object as follows:

```
Person(const char *szL,const char *szF,const char *szP) {
        m_szLastName=szL;
        m_szFirstName=szF;
        m_szPhone=szP;
}
```

Unfortunately, this approach has two limitations that present problems from time to time:

- It provides us with no way to invoke base class constructors other than the default constructor—which is called as part of the construction process for the child class.
- It will not allow us to initialize any data member declared const.

To get around these limitations, the new ISO standard for C++ allows us to specify initializations that take place during construction as part of an initialization list that precedes the body of the constructor. The basic format is as follows:

ClassName(*arguments*) : *initializer-list* { *constructor-body* }

Two types of items can be contained in this list:

1. Base class constructors
2. Member variable names

Argument lists that are valid for a constructor appropriate to the data being initialized can follow each initializer list item. For example, suppose we wanted to write a constructor for our Employee object that could be used to initialize all the values. Two approaches to writing the constructor—without and with an initializer list—are shown in Example 9.9. The initializer version calls the three-argument Person constructor defined above, and then initializes the two remaining member variables.

Test your understanding: Rewrite the three argument Person constructor above using an initialization list.

Example 9.9: Employee Constructor Without and With Initializer List

```
// Using body of the constructor
Employee(const char *szI,const char *szL,
      const char *szF,const char *szP,const char *szT)
{
      m_szID=szI;
      m_szLastName=szL;
      m_szFirstName=szF;
      m_szPhone=szP;
      m_szPosition=szP;
}

// Using initializer list
Employee(const char *szI,const char *szL,
      const char *szF,const char *szP,const char *szT) :
            Person(szL,szF,szP),
            m_szID(szI),
            m_szPosition(szT)
{
// No initializations in body are required
}
```

The primary advantage of using initializer lists—which is increasingly common in commercial C++ code—is that they allow the base class to be called even for constructor functions. As noted previously, they also *must* be used if a class contains const member variables.

9.2.5: Typecasting Inherited Objects in C++

The technique we have thus far used for typecasting has been the (type-specifier) format that originated in the C programming language. While this technique remains widely used, C++ offers two templated operators that are much safer to use for typecasting. These are:

- static_cast<*type-specifier*>
- dynamic_cast<*type-specifier*>

The static_cast<> operator is used to do a typecast on any type of expression, but performs no runtime checking. As a consequence, it is similar in effect to the old C-style use of parentheses, and should only be used where the casting is sure to work. For example:

```
Person *p1;
Employee *e1=new Employee;
p1=static_cast<Person*>(e1);
```

In this illustration, we can be confident that the typecast is valid because we know an Employee object is, by inheritance, also a Person object. This is referred to as an *upcast*.

The dynamic_cast<> operator, in contrast, is only used on pointers and references. It performs a runtime check to ensure that the pointer or reference being checked can be cast to the type-specifier template argument. If the typecast is consistent, it is performed. If not, and the type-specifier pointer, a NULL value (0) is returned. If the type-specifier is a reference, an exception is generated.

Suppose, for example, we have a Customer class that inherits from Person the Employee does (this will be part of the lab exercise in Section 9.6). Then consider the following code fragment:

```
Person *p1,*p2;
Employee *e1=new Employee,*e2,*e3;
Customer *c1=new Customer;
p1=static_cast<Person*>(e1);
p2=static_cast<Person*>(c1);
e2=dynamic_cast<Employee*>(p1); // e2 will now have the same address as e1
e3= dynamic_cast<Employee*>(p2); // e3 will now have the address 0
```

In the case of the dynamic cast of p1, the address in p1 points to an Employee object so that address is returned. This process illustrates a *downcast*. In the case of the dynamic cast of p2, however, the runtime check done by the program determines that p2 does not point to an Employee object, so a 0 (NULL value) is returned.

Section 9.2 Questions

1. If emp is an Employee object created from the definition in 9.1, would emp.LastName() be a valid expression?

2. Given that the Person and Employee copy constructors are identical, why do we need to override the copy constructor for an Employee?

3. Why does making a single overridden member function private have the effect of making all member functions of that name inaccessible.

4. Following the rules we have already been given, what would happen if you declared a member variable in the child class with the same name as one in the base class?

5. What OOP design principle from Chapter 14 do the principles for overriding in Section 9.2.2 particularly relate to?

6. Why would protected/private inheritance tend to be particularly relevant to specialized collections?

7. Aside from obvious use in initializing const members, why are constructor initialization lists desirable from a design perspective?

8. Implement a simple static member of the Employee class, prototyped as follows:

   ```
   bool Employee::IsEmployee(void *p);
   ```

   that uses dynamic casting to return true or false depending upon whether or not the argument p is pointer to an employee object.

9.3: Employee Array Demonstration

Walkthrough available in EmpArray.avi

In this demonstration, we're going to show how we can use inheritance to make classes more general. Our specific focus will be on taking the PersonArray, developed in Chapter 8 (Section 8.3.3) and making suitable for storing sorted collections Employees.

9.3.1: EmployeeArray Class Declaration

The EmployeeArray class inherits from PersonArray and adds serialization of objects—which we know will be Employee objects—along with a Display() member that shows its contents and various << and >> overloads. The class declaration is shown in Example 9.10.

Example 9.10: EmployeeArray Declaration

```
class EmployeeArray :   public PersonArray
{
public:
      EmployeeArray(void);
      virtual ~EmployeeArray(void);

      int Add(const Employee &emp){
            Employee *pEmp=new Employee(emp);
            return PersonArray::Add(pEmp);
      }

      void Load(fstream &strm);
      void Save(fstream &strm) const;
      void Display() const;

};

fstream &operator>>(fstream &strm,EmployeeArray &emp);
fstream &operator<<(fstream &strm,const EmployeeArray &emp);
```

Besides adding I/O to the EmployeeArray class, there was one other major change: an override of the two Add() members to take Employee objects as arguments, instead of the more general Person objects. Since the Add() member is the only way to get objects into the collection, this ensures we'll only have Employee objects in the vector<Person*> collection m_arp. This is critical, because our serialization operations will crash and burn if non-Employee objects are present. The overrides themselves are very simple, however. They just call the base class versions.

One defect in the class, as Add() is currently implemented, is that Employee objects are added in Last Name, First Name order, just as Person objects were. Even though we have an ID() member defined, we can't use that as the ordering mechanism for our array. This issue will be further addressed in Chapter 10, when we cover polymorphism.

The members that we added to support I/O are nearly identical to those that we developed for the original PersonArray class. These are presented in Example 9.11.

Example 9.11: EmployeeArray I/O Members

```
void EmployeeArray::Load(fstream &strm)
{
	RemoveAll();
	int nVal;
	strm.read((char *)&nVal,sizeof(int));
	for(int i=0;i<nVal;i++) {
		Employee *pers=new Employee;
		strm >> *pers;
		m_arp.push_back(pers);
	}
}
void EmployeeArray::Save(fstream &strm) const
{
	int nVal=Size();
	strm.write((const char *)&nVal,sizeof(int));
	for(int i=0;i<Size();i++) {
		strm << *((Employee *)(m_arp[i]));
	}
}
void EmployeeArray::Display() const
{
	for(int i=0;i<Size();i++) {
		cout << *((Employee *)(m_arp[i])) << endl;
	}
}
fstream &operator>>(fstream &strm,EmployeeArray &ar)
{
	ar.Load(strm);
	return strm;
}
fstream &operator<<(fstream &strm,const EmployeeArray &ar)
{
	ar.Save(strm);
	return strm;
}
```

The critical changes we needed to make to the original PersonArray versions were as follows:

- We create a new Employee object (instead of a Person object) when we are loading
- We typecast the Person pointers to Employee pointers in the Save() and Display() members, since we know that only Employee objects are being stored.

We now have a specialized array for storing Employee data.

Test your understanding: Could any of the C++ typecast operators have been beneficial instead of the old C syntax in the expression ((Employee *)(m_arp[i]))?

9.4: Store Lab Exercise

Walkthrough available in Store.avi

The Video Store application is designed to demonstrate the construction of an application that makes extensive use of inheritance and collections of objects. It builds upon the Company class, developed in a Chapter 8 lab exercise (Section 8.5).

9.4.1: Overview of Entire Video Store Application

At the heart of the Video Store application, which continues from Chapter 8 and concludes in Chapter 10, are three major classes that manage collections of objects. The Company class, already discussed, manages a collection of Employee objects. The Store class—the focus of this section—inherits from Company and manages an additional collection of Customer objects. Finally, the VideoStore class—the focus of the lab exercise at the end of Chapter 10—manages two additional collections—Film objects and Rental objects—allowing the user to check out and return films. The basic structure of the application (data members only) is shown in Figure 9.10.

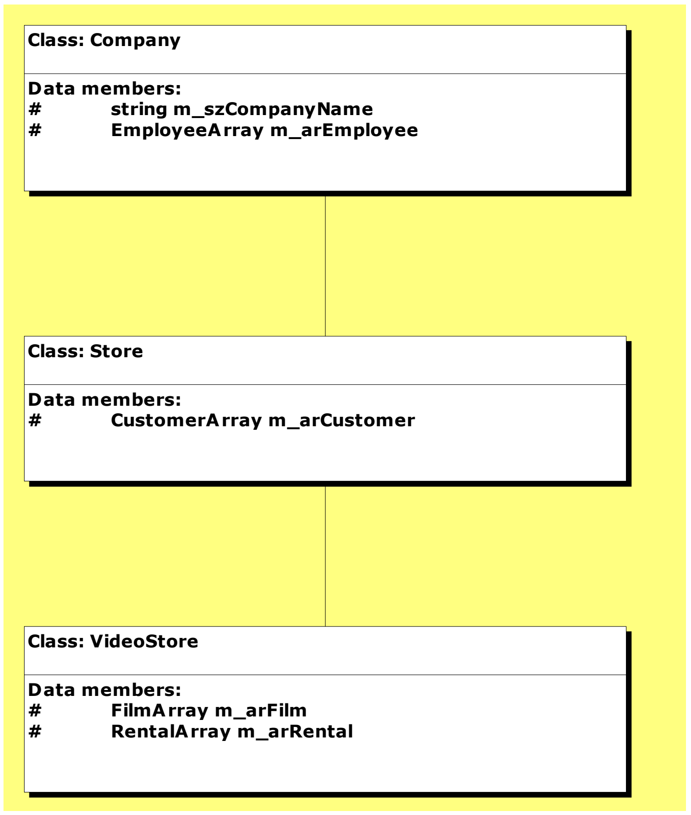
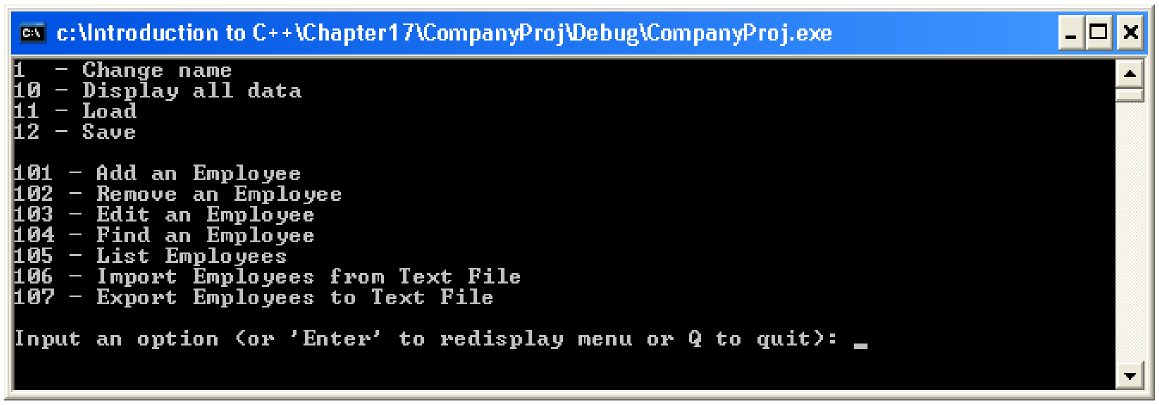


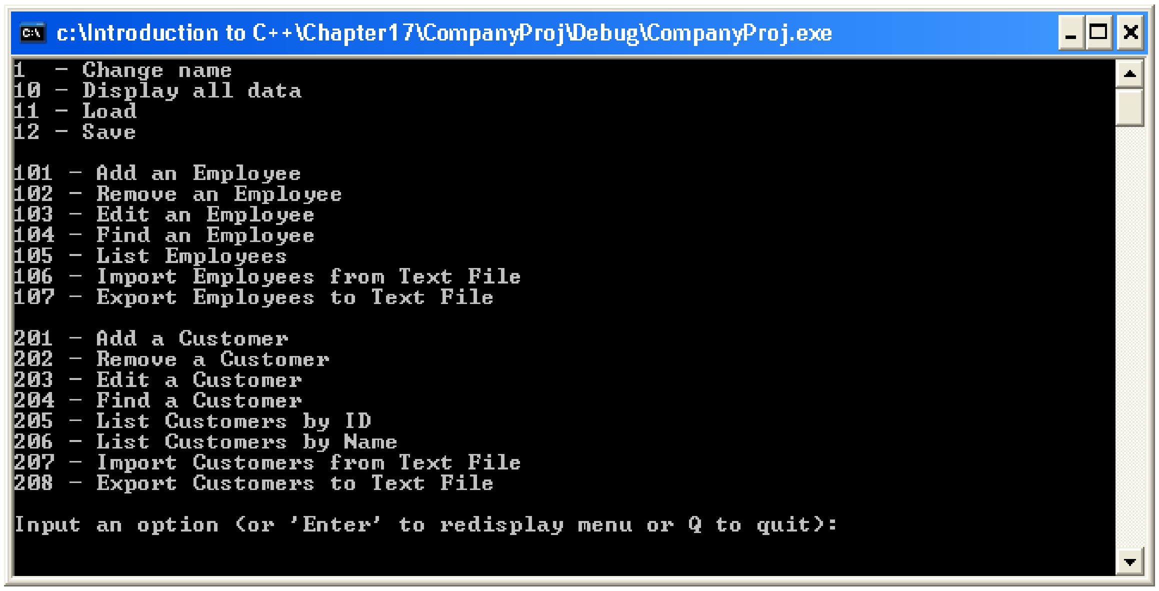
Figure 9.10: Overview of Collection Classes

Each one of the three collection classes has its own interface, implemented though an Interface() function that calls a Menu() function to display a list of options available, then calls an Option() function to call the relevant member function. Because of the inheritance relationships present, the number of options supported by the menus grows from parent to child, as shown in Figures 9.11, 9.12 and 9.13.


c:\Introduction to C++\Chapter17\CompanyProj\Debug\CompanyProj.exe

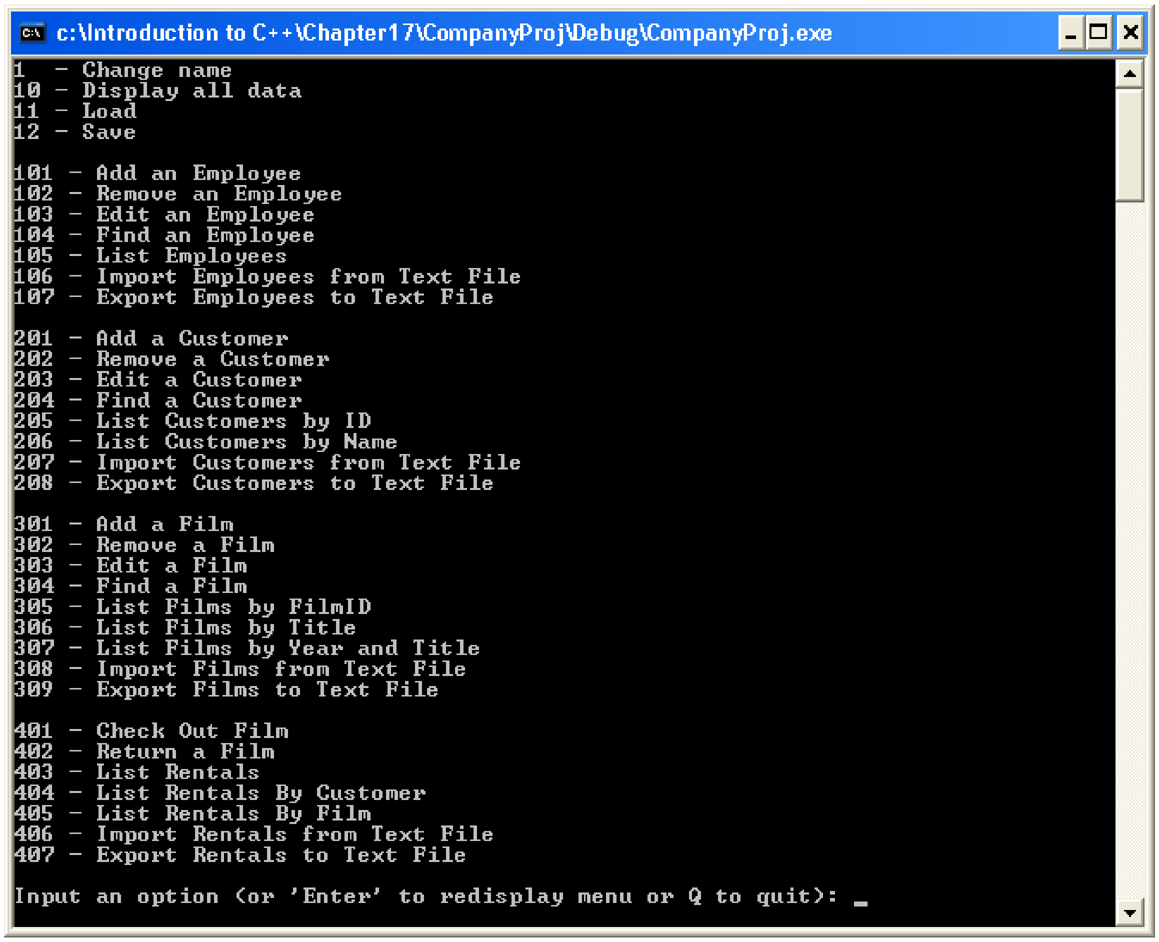
```
1   - Change name
10 - Display all data
11 - Load
12 - Save

101  - Add an Employee
102  - Remove an Employee
103  - Edit an Employee
104  - Find an Employee
105  - List Employees
106  - Import Employees from Text File
107  - Export Employees to Text File

Input an option (or 'Enter' to redisplay menu or Q to quit): _
```

Figure 9.11: Interface() menu for Company object



```
c:\Introduction to C++\Chapter17\CompanyProj\Debug\CompanyProj.exe
1  - Change name
10 - Display all data
11 - Load
12 - Save

101 - Add an Employee
102 - Remove an Employee
103 - Edit an Employee
104 - Find an Employee
105 - List Employees
106 - Import Employees from Text File
107 - Export Employees to Text File

201 - Add a Customer
202 - Remove a Customer
203 - Edit a Customer
204 - Find a Customer
205 - List Customers by ID
206 - List Customers by Name
207 - Import Customers from Text File
208 - Export Customers to Text File

Input an option (or 'Enter' to redisplay menu or Q to quit):
```

Figure 9.12: Interface for Store object



```
c:\Introduction to C++\Chapter17\CompanyProj\Debug\CompanyProj.exe
1  - Change name
10 - Display all data
11 - Load
12 - Save

101 - Add an Employee
102 - Remove an Employee
103 - Edit an Employee
104 - Find an Employee
105 - List Employees
106 - Import Employees from Text File
107 - Export Employees to Text File

201 - Add a Customer
202 - Remove a Customer
203 - Edit a Customer
204 - Find a Customer
205 - List Customers by ID
206 - List Customers by Name
207 - Import Customers from Text File
208 - Export Customers to Text File

301 - Add a Film
302 - Remove a Film
303 - Edit a Film
304 - Find a Film
305 - List Films by FilmID
306 - List Films by Title
307 - List Films by Year and Title
308 - Import Films from Text File
309 - Export Films to Text File

401 - Check Out Film
402 - Return a Film
403 - List Rentals
404 - List Rentals By Customer
405 - List Rentals By Film
406 - Import Rentals from Text File
407 - Export Rentals to Text File

Input an option (or 'Enter' to redisplay menu or Q to quit): _
```

Figure 9.13: Interface for VideoStore object

9.4.2: Store Specifications

The complete Store application requires that we implement 3 new classes:

- *Customer,* inheriting from Person and providing an additional customer ID
- *CustomerArray,* inheriting from PersonArray, to be used manage the collection of Customer objects
- *Store,* inheriting from Company, which manages a customer collection in addition to the employee collection managed by the Company class. We will implement all members of this class except ListCustomersByName(), to be done in Chapter 10.

Customer Class
There are four fundamental object types that are managed within the complete Video Store application, which we call *unit classes*. These are Employee, Customer, Film and Rental. The Employee class has been developed within the text of this chapter. The Customer class is the simplest of the remaining classes to implement, since it is nearly identical to the Employee class.

Like the Employee class, the Customer class must support a common set of interface members (initiated in the Person class) that are listed in Table 9.1.

Members Supported by Unit Classes
string Key() const Returns a unique key for maintaining the object in a KeyArray-based collection. The nature of the key depends on the object type.
void Input() Prompts the user for input for each member of the class
void Display() const Displays the data values for the object.
string FixedString() const Returns a fixed field-width representation of the object's data, suitable for export to a text file.
void FixedString(const string &str) Retrieves data from a string containing a fixed-width representation of the object, consistent with what was produced by FixedString().
void Load(STREAM &strm) Loads a unit object stored in a binary file.
void Save(STREAM &strm) const Saves a unit object to a binary file

Table 9.1: Members to be implemented for unit classes

CustomerArray
The CustomerArray should be identical in structure and interface to EmployeeArray (Section 9.3).

Company and Store Classes
As noted in the overview, the main classes are Company, Store and VideoStore. Most of the members of these classes relate to the collections they manage. The Company class, originally developed in Chapter 8, Section 8.5, requires only minimal modifications from its original form. Its top-level interface (essentially repeated from Chapter 8, Table 8.5) is presented in Table 9.2. There are only two changes required from the earlier version: changing the return type of FindEmployee() to an Employee* (instead of a Person*) and changing the underlying collection from a PersonArray to an EmployeeArray.

Menu Option No.	Top Level Company Functions
1	**void ChangeName() const** Used to change the name of the Company object. Prompts user for new name.
10	**virtual void DisplayAll() const** Displays the company name and all the Employee objects.
11	**void LoadFromFile()** Prompts the user for a file name, opens the file for reading, then loads the Company data from the file.
12	**void SaveToFile() const** Prompts the user for a file name, opens the file for writing, then saves the Company data to the file.
101	**void AddEmployee()** Prompts the user for an Employee ID and, if the ID is valid (i.e., not already present), allows the user to edit and add the Employee to the EmployeeArray collection.
102	**void RemoveEmployee()** Calls FindEmployee(true) to access the Employee object, displaying the data. Prompts the user for a confirmation and, if the user confirms, removes the Employee object from the EmployeeArray collection.
103	**void EditEmployee()** Calls FindEmployee(false) to access the Employee object, then allows the user to edit the Employee.
104	**Employee *FindEmployee(bool bDisplay=true) const** Prompts the user for an Employee ID and, if the ID is valid returns a pointer to the employee. If bDisplay is true, displays the Employee object's data.
105	**void ListEmployees() const** Displays the data for all employees in the EmployeeArray, one employee per line.
106	**void ImportEmployees()** Prompts the user for a file name (presumably a text file), then loads the fixed format employee records into the EmployeeArray.
107	**void Export Employees() const** Prompts the user for a file name (presumably a text file), then saves the employee records from the EmployeeArray in a fixed text format.
N/A	**void Interface()** Enters a loop that displays a menu (i.e., calling the Menu() function described below), prompts the user for an option number, then selects one of the options above (i.e., calling the Option() function described below).

Table 9.2: Top Level Menu Options for Company Class

The Store class is very much like the Company class—particularly so because Employee objects are so similar to Customer objects. Its top-level menu options are presented in Table 9.3.

Menu Option No.	Top Level Store Functions
201	**void AddCustomer()** Prompts the user for an Customer ID and, if the ID is valid (i.e., not already present), allows the user to edit and add the Customer to the CustomerArray collection.
202	**void RemoveCustomer()** Calls FindCustomer(true) to access the Customer object, displaying the data. Prompts the user for a confirmation and, if the user confirms, removes the Customer object from the EmployeeArray collection.
203	**void EditCustomer()** Calls FindCustomer(false) to access the Customer object, then allows the user to edit the Customer.
204	**Customer *FindCustomer(bool bDisplay=true) const** Prompts the user for a Customer ID and, if the ID is valid returns a pointer to the customer. If bDisplay is true, displays the Customer object's data.
205	**void ListCustomers() const** Displays the data for all customers in the CustomerArray, one customer per line.
206	**void ListCustomersByName() const** *To be implemented in Chapter 10.*
207	**void ImportCustomers()** Prompts the user for a file name (presumably a text file), then loads the fixed format customer records into the CustomerArray.
208	**void Export Customers() const** Prompts the user for a file name (presumably a text file), then saves the customer records from the CustomerArray in a fixed text format.

Table 9.2: Top Level Menu Options for Store Class

In implementing the Store class, a variety of helpful functions can also be implemented, just as they were for the Company class. In addition to overloading the virtual functions from the Company class, a full set of functions that do not require user I/O should be implemented. For example, in addition to

```
void AddCustomer();
```

which prompts the user for customer information, we should have:

```
bool AddCustomer(const Customer &pers);
```

which performs the actual addition to the array. The main benefit of having two different functions—one that involves the user and one that doesn't—is that it would allow use to use our Store class in contexts where customers were not coming from the user (e.g., imported from a database), or were not coming from standard input (e.g., a Windows program, where customer data was gathered in a dialog box). Furthermore, having variations on the functions isn't much extra work, since we can call the non-user version from the user version once the user has entered the appropriate data.

The declaration used to create the sample version is presented in Example 9.12.

Example 9.12: Declaration of Store Class

```
class Store :     public Company
{
public:
      Store(void);
      virtual ~Store(void);

      virtual void RemoveAll();

      // Top Level Functions
      // All company functions plus
      virtual void DisplayAll() const;
      void AddCustomer();
      void RemoveCustomer();
      void EditCustomer();
      Customer *FindCustomer(bool bDisplay=true) const;
      void ListCustomers() const;
      // void ListCustomersByName() const; (Done in Chapter 10)
      void ImportCustomers();
      void ExportCustomers() const;

      // Helpful functions
      bool AddCustomer(const Customer &pers);
      void RemoveCustomer(const char *szKey);
      void RemoveCustomer(const string &szKey);
      Customer *FindCustomer(const char *szKey) const;
      Customer *FindCustomer(const string &szKey) const;

      virtual void Load(fstream &strm);
      virtual void Save(fstream &strm) const;

      Customer *GetCustomer(int i) const;
      int CustomerCount() const;

      virtual void Menu() const;
      virtual bool Option(int nOption);

      void ImportCustomers(istream &in);
      void ExportCustomers(ostream &out) const;
protected:
      CustomerArray m_arCustomer;
};

fstream &operator>>(fstream &strm,Store &ar);
fstream &operator<<(fstream &strm,const Store &ar);
```

9.4.3: Instructions

Although the number of functions to be implemented in the lab exercise is large, few of the individual functions are particularly complex. Indeed, the longest functions are those that do mechanical activities such as prompting the user for object information, displaying a menu and routing menu selections.

A good general approach to the exercise is the following:

- Begin by modifying and testing the Company class. Don’t move on until your confident you've got the Company working well, and as intended.
- Write the Customer class, using Employee as a starting point. In fact, this class is so simple that a few search-and-replace operations, followed by some deletions (since a Customer doesn't have a position member) will get the job done.
- Develop a CustomerArray class, using the EmployeeArray as a model. Again, the similarity is so strong you can consider living dangerously and postponing testing.
- Develop to the Store class. As it turns out, since a Store and Company manage such similar data types (i.e., Customers and Employees), the classes will be very similar.
    - A good starting point is to build the interface first, implementing the Menu() and Option() members. In both cases, you should be able to call the base class and then add the additional data for the Store.
    - The key virtual functions, DisplayAll(), Load() and Save() (all of which begin by calling the base class) can then be implemented. Be sure to use the serialization capabilities of your unit-collection class (CustomerArray) for the last two of these.
    - All the Customer management functions can then be implemented, excepting ListCustomersByName().
    - You should now do extensive testing of the Store class to make sure it is working

9.5: Review and Questions

9.5.1: Review

Inheritance is a powerful programming capability that promotes abstract thinking, reuse of code and the ability to handle diverse collections of related objects using polymorphism. The inheritance relationship is also sometimes referred to as the "is-a" relationship, because when ClassB can benefit from inheriting properties from ClassA, it usually makes sense to say:

A ClassB object is a ClassA object.

In an inheritance relationship such as the above, ClassA is often referred to as the base-class or parent class, while ClassB can be called the child class.

In C++, inheritance is implemented by declaring a class, then following the class name with a colon, some inheritance-specifiers and name of the base class. For example:

```
class Child : public Parent {
// ClassB member are declared, just as they would be in a normal class
};
```

When Child is declared in this way, all of the members of Parent (except private members) can be used within Child objects. Other inheritance derivation specifiers (e.g., protected, private) can be used. These hide various Parent members within the Child class, and are considerably less common in their use.

When a Child inherits from a Parent, there are two types of changes that are typically made in the Child class:

- *Additions:* new data and function members can be added to the Child class
- *Overrides:* The behavior of Parent member functions are changed in the Child class.

Whenever a given function name is overridden in the Child, all Parent overloads of that function become invisible—although they can be called using the scope resolution operator (e.g., Parent::OverrideMemberName() will call the function defined in the Parent class).

A special case of inheritance involves constructor functions. Normally, when a Child object is created, the default (no argument) version of the Parent constructor is called in the process of constructing the Child. To allow greater flexibility, an initialization list may be specified as part of a constructor function. Within that list may be Parent constructor overloads and data member names. For example, the Employee constructor:

```
    Employee(const char *szI,const char *szL,const char *szF,
                                                          const char
*szP,const char *szT) :
                        Person(szL,szF,szP),
                        m_szID(szI),
                        m_szPosition(szT)
                {
                // No initializations in body are required
                }
```

calls a three-argument version of its base class (Person) constructor, and then species constructors for two of its string data members (m_szID and m_szPosition).

When inheritance is present, it can also be useful to replace standard C-style typecasting (using parenthesis) with C++ template operators:

- **static_cast<***target-type***>**: Performs a typecast with no runtime checking
- **dynamic_cast<***target-type***>**: Determines if the typecast is valid, returning 0 (pointers) or generating an exception (references) if it is not.

9.5.2: Glossary

Base class – A class whose properties are being inherited in an inheritance relationship (same as parent class).
Child class – A class that inherits properties from one or more parent classes.
Downcast – A typecast to a child type.
dynamic_cast<> – A C++ templated operator used as a substitute for the C typecast syntax. A dynamic_cast<> operation does runtime checking to ensure the cast is valid, returning a 0 (pointer) or an exception (reference) if it is not. Often used for downcasts.
Hierarchy – A collection of class parent-child relationships.
Initialization List – A comma-separated of constructor functions and member names that can be used to initialize objects during construction, and is specified as part of a constructor function.
Override – Redefining a base class member function in a child class. Once a given function name has been overridden, all other overloads that may exist in the base class are hidden.
Parent class – A class whose properties are being inherited in an inheritance relationship (same as base class).
Private inheritance – A form of inheritance in which all public and protected members of the base class become private in the child class.

Protected inheritance – A form of inheritance in which all public and protected members of the base class become/remain protected in the child class.
Public inheritance – A form of inheritance in which all public members of the base class remain public in the child class. Private members of the base class are always inaccessible in the child class.
static_cast<> – A C++ templated operator used as a substitute for the C typecast syntax. A static_cast<> operation does no runtime checking (see dynamic_cast<>), but can be safely used for upcasts.
Upcast – A typecast to a parent type.

9.5.3: Questions

9.1: Motorized Vehicle Taxonomy. Create a hierarchy of motorized vehicles, with attributes such as number of wheels, passenger seating, etc.

9.2: Public vs. Protected Inheritance: Explain why the amount of code that must be written when choosing between public inheritance and composition is often very different, while the amount of code that must be written when choosing between protected inheritance and composition is often very similar.

9.7: Enhanced Employee Array. In the EmployeeArray class presented in Section 9.3, overload the [] operators so they return Employee * objects.

9.9: Case insensitive string. Create an iString class that inherits from string but does comparisons in case insensitive fashion. One benefit of inheriting here is so that you can do the necessary operator overloads.

9.10: Uppercase string class. Create a uString class that inherits from iString (Question 9.5) but only allows uppercase characters. You'll have to override a number of the iString members here so that each time the string is initialized, all the characters are transformed to uppercase.

Chapter 10

Polymorphism and Multiple Inheritance

Executive Summary

Chapter 10 introduces two important inheritance-related capabilities offered by C++: polymorphism and multiple inheritance. Polymorphism is the ability to call different versions of a function depending upon the type of object to which it is applied. It is a capability that is critical when we start to implement collections of objects. Multiple inheritance is the ability to define a class based upon more than one base class.

The chapter begins with brief overview of the polymorphism problem in general terms, contrasting non-polymorphic behavior with polymorphic behavior. We then show how the **virtual** keyword can be used to determine whether polymorphism is implemented or not implemented for a specific class member function. We then turn to the subject of multiple inheritance, which is presented with a series of cautions relating to the types of problems it engenders. The chapter concludes with an extended Video Store lab exercise, building upon the Person/Company classes introduced in Chapters 8 and 9.

Learning Objectives

Upon completing this chapter, you should be able to:

- Describe the types of problems polymorphism is intended to solve
- Explain why polymorphism is particularly relevant to collections
- Explain the use of the **virtual** keyword in polymorphism
- Differentiate between appropriate polymorphic and regular member functions
- Explain the pros and cons of multiple inheritance
- Explain how the **virtual** keyword affects multiple inheritance
- Use inheritance to create a network of classes as part of an application

10.1: Polymorphism

In this section, we present an example that demonstrates what polymorphism is, and why it is important. In Section 10.2, we examine its implementation in C++.

10.1.1: The Polymorphism Problem

In Chapter 9, we introduced a hierarchy of animals (repeated in Figure 10.1) as a way of illustrating the benefits of inheritance. In that hierarchy, we also noted the need for overriding members. For example, it makes sense to have the mammal class return a value of **true** for a member LiveBirth(), since it is a nearly universal characteristic of mammals that they give live birth. On the other hand, we have the strange case of the duck-billed platypus, which lays eggs. So, we override live birth within that the platypus class to return false. Such classes are illustrated in Example 10.1.

Example 10.1: Mammal, Lion and Platypus Classes

```
class Mammal : public Animal
{
public:
	Mammal(void){}
	virtual ~Mammal(void){}

	string Name() const {return string("(Generic mammal)");}

	bool LiveBirth() const {return true;};
	bool LaysEggs() const {return false;}
};

class Lion : public Mammal
{
public:
	Lion(void){}
	virtual ~Lion(void){}
	string Name() const {return string("Lion");}
};

class Platypus : public Mammal
{
public:
	Platypus(void){}
	virtual ~Platypus(void){}

	bool LiveBirth() const {return false;};
	bool LaysEggs() const {return true;}
	string Name() const {return string("Duck-billed Platypus");}
};
```

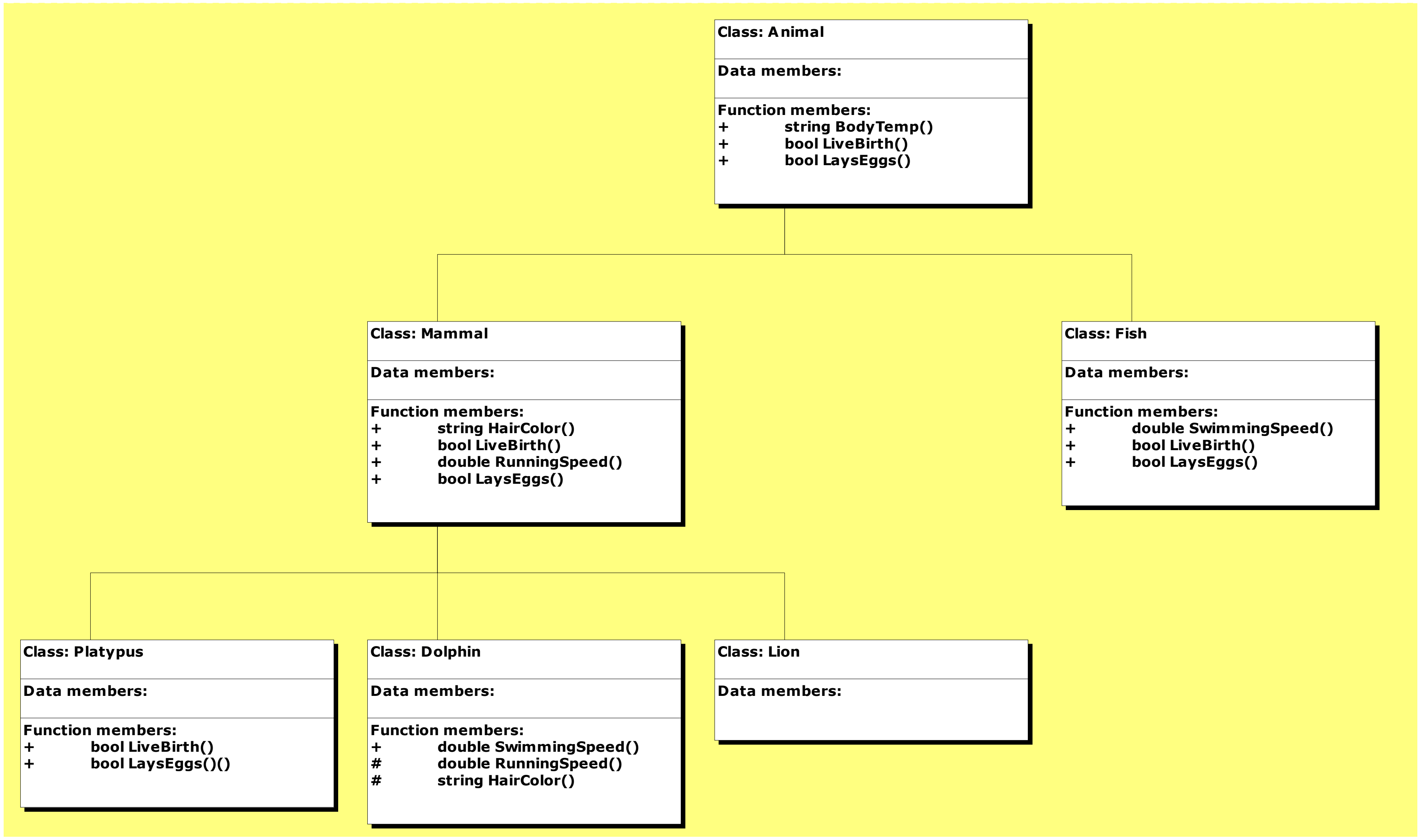


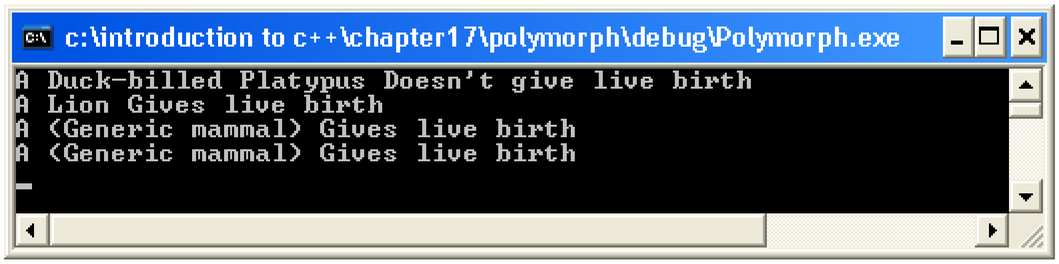
Figure 10.1: Animal inheritance network

The polymorphism problem—or, rather, the problem you run into when there is no polymorphism—is demonstrated by the test code shown in Example 10.2.

Example 10.2: Animal demonstration test code

```
void TestAnimal()
{
	Platypus p;
	Lion l;
	Mammal *pM1,*pM2;
	cout << "A " << p.Name() <<
		(p.LiveBirth() ? " Gives live birth" : " Doesn't give live birth")
		<< endl;
	cout << "A " << l.Name() <<
		(l.LiveBirth() ? " Gives live birth" : " Doesn't give live birth")
		<< endl;
	pM1=&p;
	cout << "A " << pM1->Name() <<
		(pM1->LiveBirth() ? " Gives live birth" : " Doesn't give live 
birth")
		<< endl;
	pM2=&l;
	cout << "A " << pM2->Name() <<
		(pM2->LiveBirth() ? " Gives live birth" : " Doesn't give live 
birth")
		<< endl;
}
```

The test code defined a platypus and a lion, the sends output to cout specifying whether or not the object gives live birth. When use the platypus object (p) and the lion object (l) directly, the proper responses are received, as shown in Figure 10.2. When, however, we use Mammal pointers (pM1 and pM2), however, we don't get the proper answer. In fact, we get the values for the Mammal class—not for the Lion and Platypus classes.


c:\introduction to c++\chapter17\polymorph\debug\Polymorph.exe

```
A Duck-billed Platypus Doesn't give live birth
A Lion Gives live birth
A (Generic mammal) Gives live birth
A (Generic mammal) Gives live birth
```

Figure 10.2: Output from TestAnimal()

The source of the polymorphism problem was that the compiler—when generating code for the calls using pM1 and pM2—saw that the pointers were defined as Mammal pointers, and therefore generated code that called the Mammal-defined members. There really wasn't anything else the compiler could do. Although, in this simple example, it might have been able to figure out what types of mammal pM1 and pM2 pointed to (just as we could), this will generally not be the case. The problem is particularly acute with collections (e.g., a vector<Animal*> collection used to keep track of all the occupants of the zoo), where the compiler would have no conceivable way of predicting what type of animal would be stored in each position of the collection.

10.1.2: The Polymorphism Solution

From the above discussion, we can see that polymorphism is the ability to call different versions of a function depending upon the type of object to which it is applied. The discussion also explains why the compiler will not be able to help us much in making polymorphic function calls—at the time a program is compiled, there is just no way to tell what types of objects a pointer will point to or a reference will refer to.

Given that the compiler can't tell what version of a polymorphic function to call, the only other choice is to make the decision at runtime. This means a major change in the way polymorphic objects are constructed when contrasted with, for example, pure structures. Structures can be implemented as plain blocks of memory—they don't need to keep track of what they are because that's the compiler's job. Polymorphic objects, on the other hand, need to know enough about themselves to determine what version of a function to call at runtime.

10.2: Polymorphism in C++

Walkthrough available in Animals.avi

In this section, we examine a simple polymorphism example, then consider the subject of abstract classes, ending with some general rules regarding when to specify polymorphism.

10.2.1: Implementing Polymorphism in C++

Most programming languages supporting polymorphism make the capability a "done deal". C++ is different, however. It lets you specify whether or not a particular member function is going to be polymorphic. This is accomplished by specifying virtual functions.

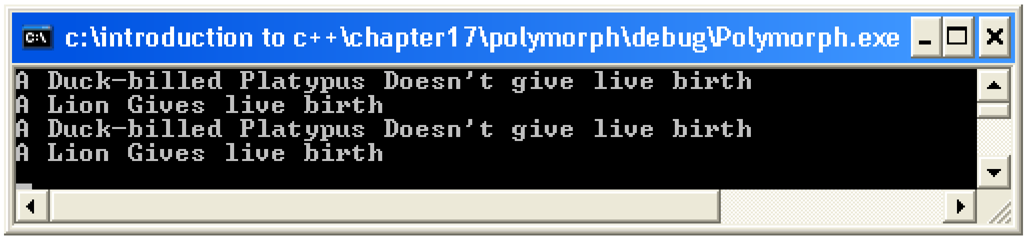
virtual Functions
To implement polymorphic behavior in a member function, we need to use the **virtual** keyword when declaring that function. Thereafter, that function (and any versions of the function in inherited classes, whether or not the keyword is used) will be polymorphic.

Continuing our example from Section 10.1.1, this means that all we need to do to fix our problem is to use the virtual keyword when we declare the Name() and LiveBirth()

functions in the Animal class. These changes are presented in Example 10.3, the output of Figure 10.5 results.

Example 10.3: Revised Mammal Definition

```
class Mammal : public Animal
{
public:
	Mammal(void){}
	virtual ~Mammal(void){}

	virtual string Name() const {return string("(Generic mammal)");}

	virtual bool LiveBirth() const {return true;};
	virtual bool LaysEggs() const {return false;}
};
```


c:\introduction to c++\chapter17\polymorph\debug\Polymorph.exe

```
A Duck-billed Platypus Doesn't give live birth
A Lion Gives live birth
A Duck-billed Platypus Doesn't give live birth
A Lion Gives live birth
```

Figure 10.5: Output of TestAnimal () with revised Mammal class definition

Implementation Details

What does declaring a function virtual actually do? As we noted in Section 10.1.2, polymorphic behavior requires information the compiler typically does not have. When we declare a function virtual, the following occurs in a typical C++ implementation:

- The compiler creates a pointer—sometimes referred to as a *thunk*—that gets embedded in each object of that class (or any child classes) just as if it were member data. The thunk points to the specific function to be called. The collection of thunk associated with a given object is frequently implemented in an array called a *vftable* (virtual function table).
- When the compiler encounters a call to a virtual function, instead of coding the call directly, it writes code that calls whatever function is pointed to by the thunk embedded in the object.

Knowing the particular mechanics of virtual function calls, which can vary across compiler implementations, is not particularly critical. It is important, however, to be aware of two practical consequences of this approach. First, it means that you never want to bypass the object construction process that initializes these pointers (e.g., reading a whole object as a single block from a file, as is sometimes done with pure structures). Second, you should be aware that there is some performance cost associated with the virtual function lookup process. Not enough of a hit to justify not using polymorphism where it is warranted. But, perhaps, enough to keep you from declaring all of your member functions virtual—whether or not they need to be.

Common Polymorphism Errors

Although the appropriate use of the virtual keyword implements polymorphic behavior very nicely, there are a few issues it's good to be aware of. First, any functions inherited from a virtual function need to match the function prototype exactly. Subtle differences, such as differing return types or a missing/extra const qualifier totally negate the process. For example, if the Platypus member:

```
bool LiveBirth() const {return false;}
```

were changed to:

```
int LiveBirth() const {return false;}
```

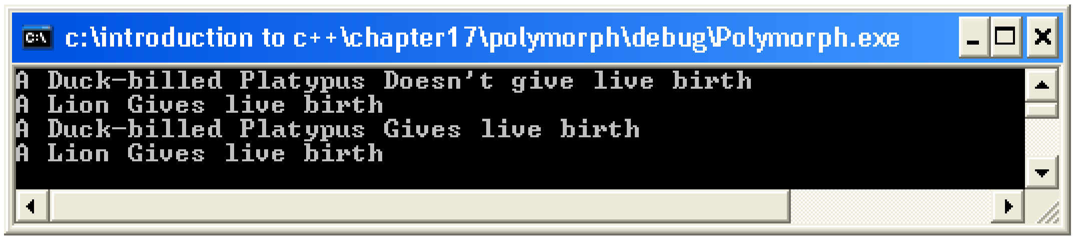
the result would be a fairly common error message from the compiler:

```
error C2555: 'Platypus::LiveBirth': overriding virtual function return type differs
and is not covariant from 'Mammal::LiveBirth'
      see declaration of 'Mammal::LiveBirth'
```

If, on the other hand, we committed the sin of omitting the **const** qualifier in our Platypus LiveBirth() function, making it:

```
bool LiveBirth() {return false;}
```

we'd get the output in Figure 10.6.


c:\introduction to c++\chapter17\polymorph\debug\Polymorph.exe

```
A Duck-billed Platypus Doesn't give live birth
A Lion Gives live birth
A Duck-billed Platypus Gives live birth
A Lion Gives live birth
```

Figure 10.6: Erroneous output resulting from missing const qualifier

The problem here is that the compiler thinks the new LiveBirth() (missing the const) is a different overload from our original virtual function in Mammal. As a result, it calls the Mammal version since there is no const override in the Platypus class.

Disabling Polymorphism

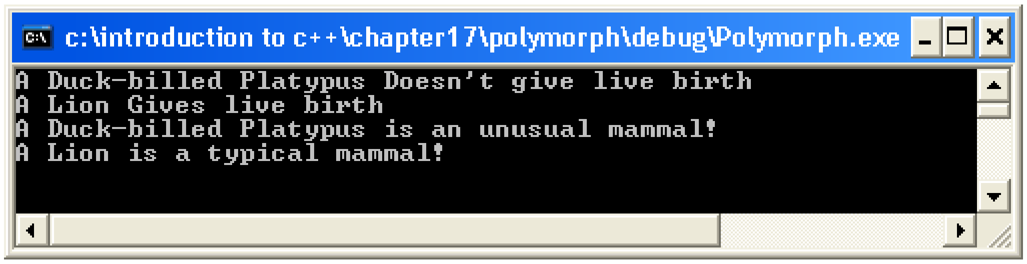
The second polymorphism issue we're going to discuss involves situations where we want to call the base class member, instead of calling the virtual member. Suppose, for example, we had a collection of Mammal objects and wanted to identify any unusual members. To do this, we might want to compare each object's LiveBirth() and LaysEggs() members with the default values for Mammals.

To disable polymorphic behavior, the scope resolution operator can be used to specify the version of the polymorphic function to be called. As a demonstration, consider the revised version of TestAnimal() presented in Example 10.4.

Example 10.4: Revised TestAnimal() Function

```
void TestAnimal()
{
        Platypus p;
        Lion l;
        Mammal *pM1,*pM2;
        pM1=&p;
        cout << "A " << pM1->Name() <<
                (pM1->LiveBirth() ? " Gives live birth" : " Doesn't give live
birth")
                << endl;
        pM2=&l;
        cout << "A " << pM2->Name() <<
                (pM2->LiveBirth() ? " Gives live birth" : " Doesn't give live
birth")
                << endl;

        if (pM1->LaysEggs()==pM1->Mammal::LaysEggs() &&
                pM1->LiveBirth()==pM1->Mammal::LiveBirth()) {
                        cout << "A " << pM1->Name() << " is a typical mammal!" <<
endl;
        }
        else {
                        cout << "A " << pM1->Name() << " is an unusual mammal!" <<
endl;
        }
        if (pM2->LaysEggs()==pM2->Mammal::LaysEggs() &&
                pM2->LiveBirth()==pM2->Mammal::LiveBirth()) {
                        cout << "A " << pM2->Name() << " is a typical mammal!" <<
endl;
        }
        else {
                        cout << "A " << pM2->Name() << " is an unusual mammal!" <<
endl;
        }
}
```

The two if constructs at the end compare the polymorphic LaysEggs() and LiveBirth() return values with their return values as specified in the Mammal class. The output that results is shown in Figure 10.7.


c:\introduction to c++\chapter17\polymorph\debug\Polymorph.exe

```
A Duck-billed Platypus Doesn't give live birth
A Lion Gives live birth
A Duck-billed Platypus is an unusual mammal!
A Lion is a typical mammal!
```

Figure 10.7: Output from revised TestAnimal()

The virtual Destructor Function
Whenever you are going to be handling polymorphic collections of objects, one function you will definitely want to make virtual is the destructor function. The reason for this is that the destructor is called whenever the **delete** operator is applied. If the destructor is not polymorphic, the destructor for whatever pointer type that is used to hold the collection (e.g., Mammal) will be used, as opposed to the destructor for the object type (e.g., Platypus). This can be particularly problematic where a child class manages memory data that is not managed by the base class.

10.2.2: Abstract Classes

Although our Mammal hierarchy doesn't exactly qualify us to be zoologists, it does provide a rather powerful demonstration of polymorphism in C++. There are, however, a few conceptual problems with the hierarchy:

- The "(Generic Mammal)" name specified in the Mammal class is a really lousy default value for classes that inherit from Mammal. If the programmer doesn't know the name of a mammal, gosh darn it, he or she shouldn't be creating a class for it!
- There's nothing in our code that prevents us from creating a Mammal object and using it in our program. The question is, do we really want to mix classification objects with actual animal objects?

These conceptual problem suggest that we need to be able to define an "abstract" class, a class to be used to organize an inheritance network but *not* for creating actual objects.

C++ gives us a very easy mechanism for creating an abstract class—just specify one or more virtual member functions to be a *pure virtual function*. This is done by writing = 0 after the function declaration in the header file.

The creation of pure virtual functions is illustrated in Example 10.5, which includes an Animal class from which Mammal inherits. Among the features that should be noted:

- Animal is an abstract class, with the member functions Name(), LiveBirth() and LaysEggs() all declared as pure virtual functions.
- Mammal no longer has a Name() function. It can be omitted because the pure virtual function in the Animal base class is—in effect—a promise that any class that can be turned into an object will have a Name() function implemented. Since Mammal doesn't have a name, it follows that we should get an error if we try to create a Mammal object. If we try, in fact, we get the following compiler error:

```
error C2259: 'Mammal' : cannot instantiate abstract class
        due to following members:
        'std::string Animal::Name(void) const' : pure virtual function was not defined
        see declaration of 'Animal::Name'
```

Thus, the ability the use of pure virtual functions has solved both of our conceptual concerns.

Test your understanding: Why didn't we pull :LiveBirth() and LaysEggs() from the Mammal definition the way we removed Name()?

Example 10.5: Animal Class Hierarchy with Abstract Animal and Mammal classes

```
class Animal
{
public:
	Animal(void);
	virtual ~Animal(void);

	virtual string Name() const = 0;

	virtual bool LiveBirth() const = 0;
	virtual bool LaysEggs() const = 0;
};

class Mammal : public Animal
{
public:
	Mammal(void){}
	virtual ~Mammal(void){}

	virtual bool LiveBirth() const {return true;};
	virtual bool LaysEggs() const {return false;}
};

class Lion : public Mammal
{
public:
	Lion(void){}
	virtual ~Lion(void){}
	string Name() const {return string("Lion");}
};

class Platypus : public Mammal
{
public:
	Platypus(void){}
	virtual ~Platypus(void){}

	bool LiveBirth() const {return false;};
	bool LaysEggs() const {return true;}
	string Name()const {return string("Duck-billed Platypus");}
};
```

So far, we've shown why it is conceptually beneficial to be able to declare Animal and Mammal as abstract classes. What we haven't yet explained is why declaring pure virtual functions is a good way to accomplish this.

Declaring a pure virtual function does two things:

- As we saw with Mammal error, it sets up a “contract” to implement the function for all objects we intend to create, giving an error if we try to create a class missing one or more pure virtual member definitions.

- It allows us to call that member function on any pointer or reference object that is of the base class type (or a child of that type). For example, since we know the Name() function is going to be available for any Animal object we create, the compiler lets us call it on Animal or Mammal pointers—even though we know those classes don't have an actual definition for Name() in place.

It is this second capability that makes polymorphism so powerful when collections of objects are being maintained. If, for example, we had a vector<Animal*> collection, we could call the Name() function on each member to get a list of names. We could also create an IsUnusual() function for each member that could identify if it is typical of its parent class.

10.2.3: Polymorphism Design Issues

As we have seen, using the virtual specifier C++ makes it possible to specify which members are polymorphic and which are not. An obvious question to ask, therefore, is why not make every member virtual? That would, in fact, be similar to what many other OOP languages (such as Java) default to.

There are, essentially, two reasons why making every member virtual would be considered a sloppy programming practice. These are:

1. *Overhead:* It takes slightly more time to call a virtual function than it does a non-polymorphic function. While the difference is usually fairly trivial, why be wasteful?
2. *Conceptual:* When you declare a function virtual you are stating, in effect, "I believe I'll eventually declare some child class that will override this function". If that is, in fact, the case, then fine. If, on the other hand, you don't think you'll ever be overriding a particular function, you can make that assertion in your code by *not* making it virtual. Then, when you come back to your class a year later, you can use the member declarations to help clarify (in your own mind) the thought processes going through your head when you designed the class.

The bottom line is that a function should generally be virtual if:

- It is likely to be overridden at some future time
- It is a function you are likely to want to use in a collection of objects based upon some abstract class
- If it is a destructor function, unless you know the object will never be used as part of a collection and doesn't manage any memory internally.

Any member function that doesn't meet any of these three criteria can—and probably should—be left as a non-virtual function for reasons of efficiency and clarity.

Section 10.2 Questions

1. Why are polymorphic functions particularly useful in collections?
2. Why does the presence of virtual functions render the memory-mapped view of a structure inaccurate?
3. Why are compiler error messages about undefined pure virtual functions a good thing?
4. In an intermediate abstract class, when does it make sense to write a definition for a pure virtual function that has been inherited, rather than leaving it to be defined in child classes?
5. If you were to define a Fish class, what would it probably look like?
6. Of the reasons not to make every member function polymorphic, which is more convincing: the performance or the conceptual?
7. Why is the destructor function usually polymorphic?

10.3: Multiple Inheritance in C++

Walkthrough available in Multi.avi

Multiple inheritance is the ability to define classes that inherit characteristics from more than one base class. Although it can be a powerful tool, its use also leads to a number of subtle problems that have led to it being dropped from many other languages, such as Java and C#. In this section, we'll look at how to define multiple inheritance, some of the problems associated with it, and techniques for minimizing the problems associated with it.

10.3.1: What is Multiple Inheritance?

To illustrate the type of scenario where multiple inheritance might be useful, let us return to our animal hierarchy and consider the case of the mule. A mule is an animal that results from breeding a horse and a jackass. Mules are also born sterile, meaning they neither give live birth nor lay eggs. As a consequence of their origin, it might make sense to define a Mule object as inheriting from both Horse and Jackass classes, as illustrated in Figure 10.8, overriding the LiveBirth() member.

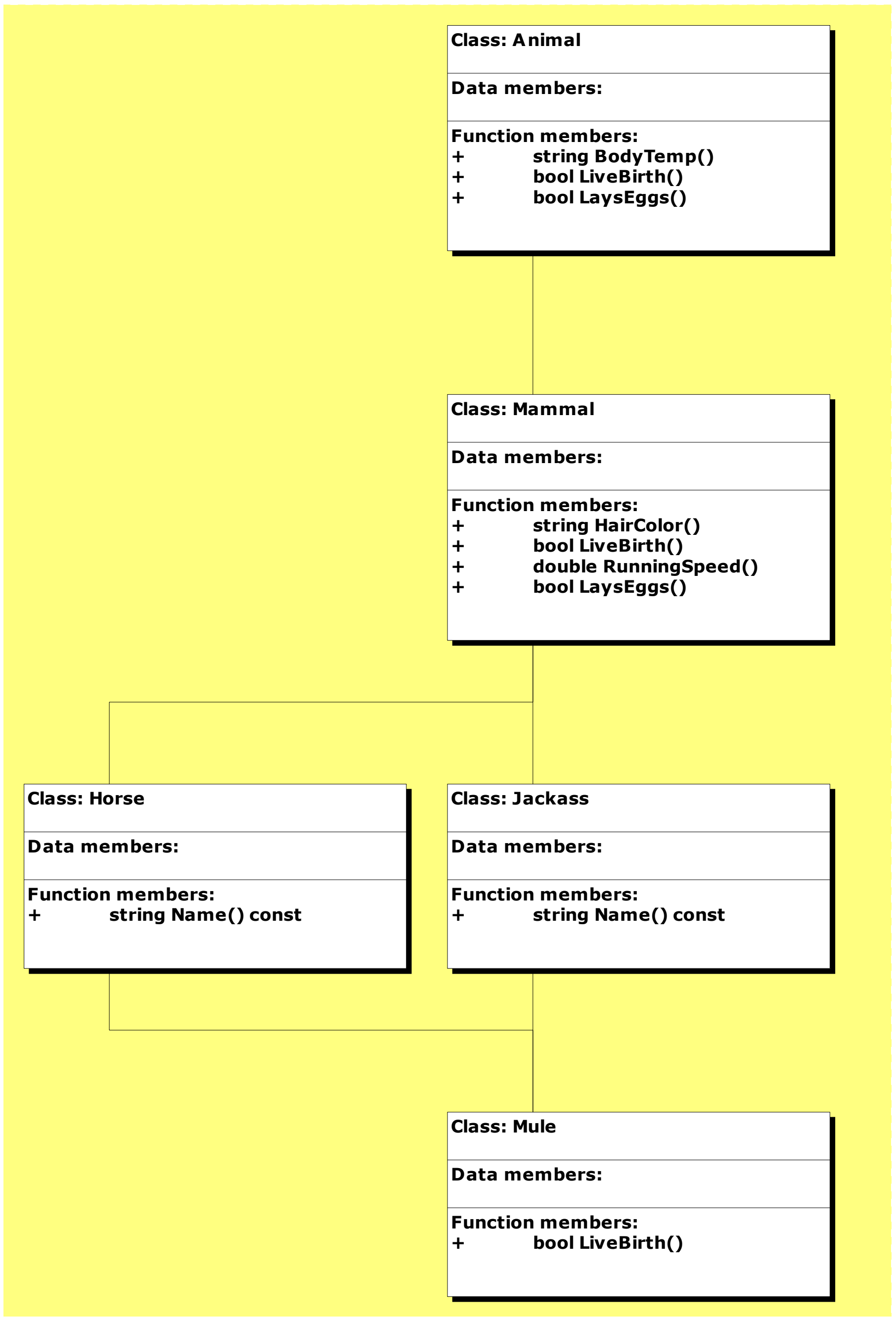


Figure 10.8: Multiple Inheritance of a Mule

Although this approach seems straightforward enough at first, it actually introduces a number of subtleties. For example:

- Both the Horse and the Jackass objects have a Name(). Which one of these should Mule() inherit as a default? (The same would apply to any member where Horse

and Jackass members return different values, such as the RunningSpeed() member).
- If Horse and Jackass have data members with the same name, which of the data members should take precedence in the Mule object, or should both be included?

Now we could argue that part of the problem here is that we haven't used a very good example. After all, because a Mule is a crossbreed, it is reasonable to complain that neither of the two statements:

```
Mule is a Horse
Mule is a Jackass
```

is strictly true. However these types of problems can surface every time we employ multiple inheritance—which is one of the primary complaints leveled against the technique. Thus, where multiple inheritance is used, it is good to employ design techniques that reduce the severity of these problems. One such technique, the use of interface classes, is presented in Section 10.3.3.

10.3.2: Implementing Multiple Inheritance

Implementing multiple inheritance in C++ is quite straightforward: you simple add comma-separated:

derivation-specifier class-name

items after the first inheritance specification, where *derivation-specifier* can either be **public**, **private** or **protected**. For example, the following declaration would create a Mule class using multiple inheritance from Horse and Jackass:

```
class Mule :
        public Horse , public Jackass
{
public:
        Mule(void){}
        virtual ~Mule(void){};

        bool LiveBirth() const {return false;}
};
```

Ambiguities in Multiple Inheritance
Unfortunately, such multiple inheritance leads immediately to problems. For example, assuming that both Horse and Jackass have Name() members, the code fragment:

```
Mule m;
cout << "The default name for a Mule is " << m.Name() << endl;
```

will produce an error message:

```
error C2385: ambiguous access of 'Name' in 'Mule'
      could be the 'Name' in base 'Horse::Name'
      or the 'Name' in base 'Jackass::Name'
```

This problem could—and should—be solved by overriding the Name() member in Mule. As we have already noted, Name() is not the type of member for which natural defaults apply.

Another way we could eliminate the problem is using the scope resolution operator. For example, the code fragment:

```
Mule m;
cout << "One default name for a Mule is " << m.Horse::Name() << endl;
cout << "One default name for a Mule is " << m.Jackass::Name() << endl;
```

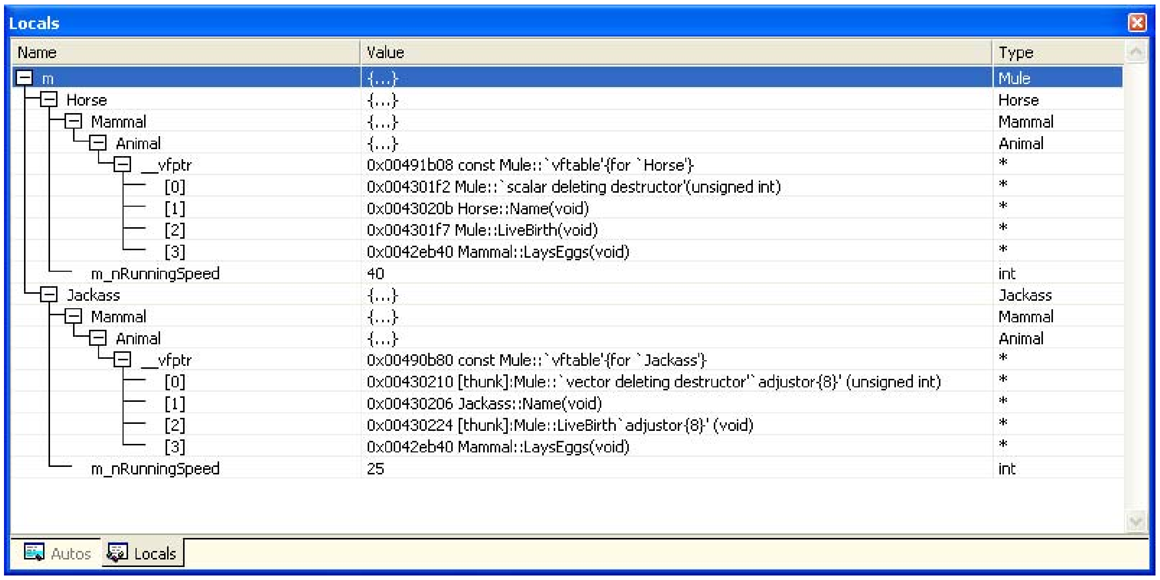
will compile without errors, and will lead to the names "Horse" and "Jackass" being returned, respectively, by the two Name() calls.

The situation can become even more complex when we inherit data. Suppose, for example, we have defined Horse, Jackass and Mule as shown in Example 10.6. Once again, we'll run into ambiguity problems if we call RunningSpeed(), since both Horse and Jackass versions exist. The way we implemented running speed introduced another problem, however. If we examine our Mule class in the debugger, we discover that we now have two data elements called m_nRunningSpeed, as shown in Figure 10.9—one created from Horse and one created from Jackass—that also have different values. (The figure also shows that we also have two parallel sets of thunks, for our virtual functions).

Example 10.6: Horse, Jackass and Mule definitions with data member

```
class Horse :
      public Mammal
{
public:
      Horse(void){m_nRunningSpeed=40;}
      virtual ~Horse(void){};

      virtual RunningSpeed() const {return m_nRunningSpeed;}

      string Name() const {return string("Horse");}
protected:
      int m_nRunningSpeed;

};

class Jackass :
      public Mammal
{
public:
      Jackass(void){m_nRunningSpeed=25;}
      virtual ~Jackass(void){};

      virtual RunningSpeed() const {return m_nRunningSpeed;}

      string Name() const {return string("Jackass");}
protected:
      int m_nRunningSpeed;

};

class Mule :
      public Horse , public Jackass
{
public:
      Mule(void){}
      virtual ~Mule(void){};

      bool LiveBirth() const {return false;}
};
```

The problem here is that unless you specify otherwise, C++ will create complete copies of all objects inherited using multiple inheritance. The practical consequence is that when you inherit casually from classes in the same hierarchy, you can get a lot of duplicate data and other members.



Name	Value	Type
m	{...}	Mule
Horse	{...}	Horse
Mammal	{...}	Mammal
Animal	{...}	Animal
__vfptr	0x00491b08 const Mule::`vftable'{for `Horse'}	*
[0]	0x004301f2 Mule::`scalar deleting destructor'(unsigned int)	*
[1]	0x0043020b Horse::Name(void)	*
[2]	0x004301f7 Mule::LiveBirth(void)	*
[3]	0x0042eb40 Mammal::LaysEggs(void)	*
m_nRunningSpeed	40	int
Jackass	{...}	Jackass
Mammal	{...}	Mammal
Animal	{...}	Animal
__vfptr	0x00490b80 const Mule::`vftable'{for `Jackass'}	*
[0]	0x00430210 [thunk]:Mule::`vector deleting destructor'`adjustor{8}' (unsigned int)	*
[1]	0x00430206 Jackass::Name(void)	*
[2]	0x00430224 [thunk]:Mule::LiveBirth`adjustor{8}' (void)	*
[3]	0x0042eb40 Mammal::LaysEggs(void)	*
m_nRunningSpeed	25	int

Figure 10.9: A Mule object (m) shown in the debugger

virtual Inheritance

C++ does provide us with a mechanism for avoiding the duplication of members and data where multiple inheritance is used with classes from the same hierarchy. Typically, the solution involves a mixture of design and the **virtual** keyword. When this keyword precedes the derivation-specifier during inheritance, it causes any of its data/function members that would be duplicated during multiple inheritance to be consolidated. In our hierarchy, for example, it really doesn't make sense to establish independent RunningSpeed() functions and data in our Horse and Jackass classes. We could, instead, create an intermediate class—LandMammal—that introduces RunningSpeed(), then uses the virtual keyword when Horse and Jackass inherit from it. This is shown in Example 10.7.

If we were to run the following code fragment:

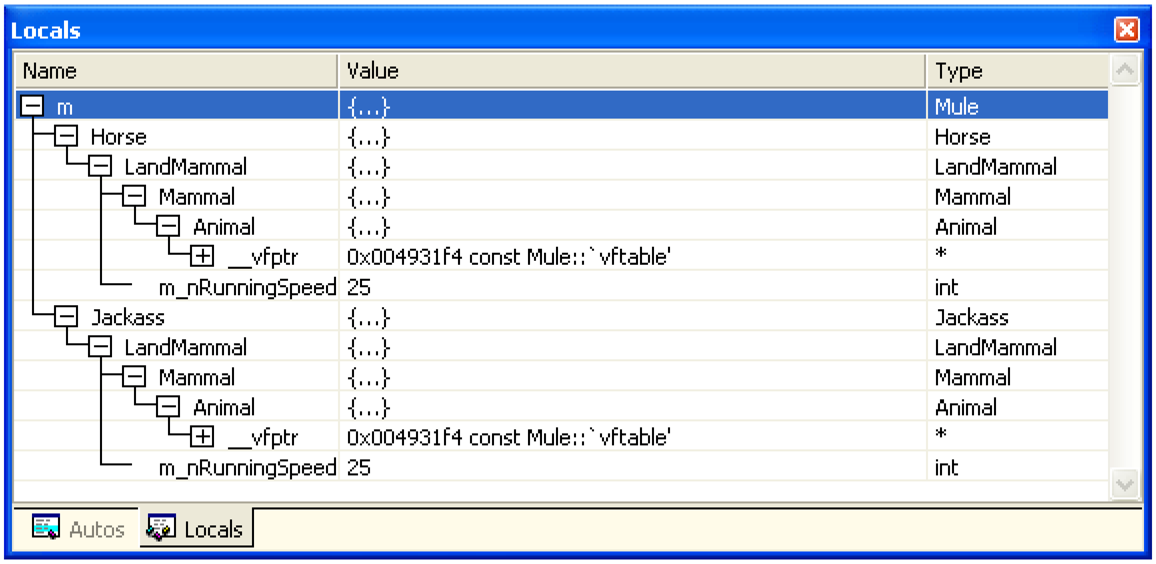
```
Mule m;
cout << "The default speed for a Mule is " << m.RunningSpeed() << endl;
```

the value 25 would now be displayed for running speed. The value is established because when a Mule object is created, the default constructor for a Horse object is called (setting m_nRunningSpeed to 40), after which the default constructor for a Jackass object is called (overwriting the old value by setting m_nRunningSpeed to 25). As a consequence, the 25 value is left when initialization is complete.

Example 10.7: Mule Classes with virtual Inheritance

```
class LandMammal :
      public Mammal
{
public:
      LandMammal(void){}
      virtual ~LandMammal(void){};

      virtual RunningSpeed() const {return m_nRunningSpeed;}

protected:
      int m_nRunningSpeed;
};

class Horse :
      virtual public LandMammal
{
public:
      Horse(void){m_nRunningSpeed=40;}
      virtual ~Horse(void){};

      string Name() const {return string("Horse");}
};

class Jackass :
      virtual public LandMammal
{
public:
      Jackass(void){m_nRunningSpeed=25;}
      virtual ~Jackass(void){};

      string Name() const {return string("Jackass");}
};

class Mule :
      public Horse , public Jackass
{
public:
      Mule(void){}
      virtual ~Mule(void){};

      string Name() const {return string("Mule");}
      bool LiveBirth() const {return false;}
};
```

If we examine the new Mule (m) object in the debugger, as shown in Figure 10.10, we see that the duplicate copies of m_nRunningSpeed are no longer present (and the two vftable arrays, used to store thunks, are at the same address—meaning there is only one array).


Locals

Name	Value	Type
m	{...}	Mule
Horse	{...}	Horse
LandMammal	{...}	LandMammal
Mammal	{...}	Mammal
Animal	{...}	Animal
__vfptr	0x004931f4 const Mule::`vftable'	*
m_nRunningSpeed	25	int
Jackass	{...}	Jackass
LandMammal	{...}	LandMammal
Mammal	{...}	Mammal
Animal	{...}	Animal
__vfptr	0x004931f4 const Mule::`vftable'	*
m_nRunningSpeed	25	int

Figure 10.10: Mule object display after virtual inheritance is implemented

10.3.3: The Interface Style of Multiple Inheritance

Although virtual inheritance solves some of the problems of multiple inheritance, it is not a cure-all. For example, our LandMammal class works fine until you encounter a Hippopotamus—an animal that spends most of its time in water but goes foraging for food at night. All of a sudden, we have to start joining LandMammal and WaterMammal abstract classes. Then we find ourselves wondering, "is swimming speed for a mammal such a different concept from swimming speed for a fish?" and, the next thing you know, we're trying to multiple inherit fish and mammals to get the relevant characteristics.

Problems such as these have led to other C++-derived languages—most notably Java and C#—to eliminate multiple inheritance in favor of interfaces. When a class implements an interface, it is essentially creating a contract (with the compiler) to provide bodies to specified functions. In this way, an interface is very much like a pure virtual function. Furthermore, once a class implements an interface, that fact can be used in the context of a polymorphic collections—which, as we have already seen, is one of the main benefits of using inheritance.

Since C++ supports multiple inheritance, it does not need formal interfaces. Nonetheless, as a matter of programming style, you can choose to limit your use of multiple inheritance to interface-like classes. Such an interface-like class would have a number of characteristics:

- It would not be part of the hierarchy from which you are doing your most significant inheritance (e.g., the animal hierarchy in the example we've been using).

- The "interface class" would consist entirely of pure virtual function members—no implemented functions and no data members.

Writing your code in this way, you gain two benefits:

1. You eliminate the ambiguities associated with multiple inheritance
2. You make your code much easier to rewrite in other languages (e.g., Java, C#) should that ever be necessary.

As an example, in our animal hierarchy we might create some habitat-related interface classes, such as LandDweller and SeaDweller, etc. These classes would then be defined along the lines of the classes in Example 10.8.

Example 10.8: Example "Interface" Classes

```
class LandDweller
{
public:
      LandDweller(void);
      ~LandDweller(void);
      virtual int RunningSpeed() const =0;
      virtual string PreferredTerrain() const =0;
      virtual string FootType() const =0;
};

class SeaDweller
{
public:
      SeaDweller(void);
      ~SeaDweller(void);
      virtual int SwimmingSpeed() const =0;
      virtual string PreferredWaterTemp() const =0;
      virtual string PropulsionMachanism() const =0;
};
```

Because the member functions in these classes are all pure virtual functions, there is never ambiguity regarding which version of a function needs to be called—it will always have to implemented in the class inheriting the interface.

While the interface approach does not lend itself to default values, it does allow for collections to be created based on interface inheritance. As a result, in addition to collections based on our original hierarchy (e.g., collections of mammals, fish, amphibians, etc.), we could also have collections of land-dwellers, sea-dwellers, veldt-dwellers, carnivores, herbivores, and collections based on whatever other interfaces we cared to define.

In the case of our Hippopotamus mammal, we might define the class as inheriting from Mammal, while—at the same time—inheriting from the "interface" classes LandDweller and RiverDweller, from which it would gain members such as RunningSpeed() and

SwimmingSpeed(). While not a perfect approach, use of interface classes both reduces some of the ambiguities resulting from multiple inheritance and increases the potential portability of our code to other languages.

10.4 Section Questions

1. Explain why the Mule example presented in the text is not, from a conceptual standpoint, a perfect case of multiple inheritance.

2. What is the particular challenge presented by multiple inheritance from classes within the same hierarchy?

3. Why do members that serve as natural default values present a more difficult multiple inheritance challenge than variables that tend to be very class specific?

4. What data issue does the virtual keyword tend to eliminate in a multiple inheritance environment?

5. What multiple inheritance issue does the virtual keyword fail to address?

6. If we had written the Mule class definition in Example 10.7 as follows:

   ```
   class Mule : public Jackass, public Horse {…definition…}
   ```

   would that have had any effect on the resulting Mule class?

7. Could you develop a hierarchy of interface classes?

10.4: Keyed Array Demonstration

Walkthrough available in KeyArray.avi

In this demonstration, we're going to show how we can use inheritance to make classes more general. Our specific focus will be on taking the PersonArray, developed in Chapter 8 (Section 8.3.3) and making suitable for storing sorted collections of nearly anything. We will do this in three steps. First, we will create a CKey abstract class that can be inherited by any class that needs to be sorted. Next, we will rework the PersonArray, renaming it KeyArray, to make it suitable for organizing a collection of any object that inherits from the Key class. Finally, we'll create a KeyTable (that inherits from the KeyArray) that can be used to keep an ordered collection of any objects—whether or not they inherit from Key.

10.4.1: The CKey Abstract Class

The CKey abstract class is very simple, as shown in Example 10.9. It consists of 5 pure virtual functions:

- Key() is intended to return a string that can be used as the key
- NewCopy() is intended to return a copy of the object to which it is applied
- Load() is designed to load the object to which it is applied to a binary file
- Save() is intended to save the object to which it is applied to a binary file
- Display() is intended to display the object to standard output

Example 10.9: The CKey Class

```
class CKey
{
public:

	CKey(void){}
	virtual ~CKey(void){}
	virtual string Key() const = 0;
	virtual CKey *NewCopy() const = 0;
	virtual void Load(fstream &strm) = 0;
	virtual void Save(fstream &strm) const = 0;
	virtual void Display() const = 0;

};
```

Why define such a trivial class? The reason is simple. When we created the PersonArray class (Chapter 8, Section 8.3.3), we were maintaining a sorted vector<> of object pointers to Person elements. But was there really anything special about Person objects used in that array? In fact, all that mattered was that a Key() member be available, and that the objects know how to load, save and display themseves.

In theory, then, any class that inherits from CKey—either single or multiple inheritance—could be handled by the PersonArray. That will lead us to our next step—transforming the PersonArray to a KeyArray. As a prelude to that, we will change our Person class so that it inherits from CKey, which requires only adding the : public CKey to the class declaration, and add the NewCopy() member, i.e.:

```
#include "Key.h"
class Person : public CKey
{
	Person *NewCopy() const {return new Person(*this);}
	// Rest of the class remains unchanged
	};
```

The reason that this change is so simple is that we already have the Key(), Load(), Save() and Display() functions implemented in the Person class—which is the only requirement added by inheriting from the abstract Key class.

In the case of our Employee class (Example 10.1), the class will work without *any* changes—since it already has an is-a relationship with CKey through the Person base class. However, for an Employee object, the employee ID probably a better key than the last name, first name combination we have been using. So, it makes sense to override the Key() member in the employee class. We also would want to override the NewCopy() member. The resulting changes are as follows:

```
	class Employee :	public Person
	{
	public:
		Employee *NewCopy() const {return new Employee(*this);}
		string Key() const {return ID();}
		// Remainder of class unchanged
	};
```

The effect of the Key() override is that when we add Person objects to our KeyArray class—to be created next—they will be stored in alphabetical order by name. Employee objects, on the other hand, will be stored in ID order.

10.4.2: The KeyArray Class

The KeyArray class starts as a modification of our PersonArray class (created in Chapter 8, Section 8.3.3) to work with CKey objects instead of Person objects. A reasonable first pass for this would be to take the PersonArray code and do two search-and-replace steps:

- Replace PersonArray with KeyArray
- Replace Person with CKey

When we do this, we get a pretty clean compile, with errors on only a few lines. One error we can fix relatively quickly. The other, unfortunately, requires some serious design rethinking.

The first error we encounter is in the memory allocation line (first line) of the Add() member:

```
	int KeyArray::Add(const CKey &pers) {
		CKey *p=new CKey(pers);
```

```
        int nRet=Add(p);
        if (nRet<0) delete p;
        return nRet;
}
```

The message reads:

```
error C2259: 'CKey' : cannot instantiate abstract class
    due to following members:
    'std::string CKey::Key(void) const' : pure virtual function was not defined
```

When we think about it, this makes sense—we designed CKey to be an abstract class. As a consequence, it stands to reason that we can't apply the **new** operator to it directly.

Fortunately, we've already built in the solution for this problem. Since we are copying an object as part of the Add() routine, and because we have implemented a NewCopy() member as part of our CKey interface, we just replace the first line in the Add() function with:

```
CKey *p=pers.NewCopy();
```

The polymorphic NewCopy() function will then call the proper constructor for the type of object being passed in.

We get essentially the same message on another line, this time on the memory allocation within the loop in the Load() function:

```
void KeyArray::Load(stream &strm)
{
        RemoveAll();
        int nVal=ReadInteger(strm);
        for(int i=0;i<nVal;i++) {
                CKey *pers=new CKey;
                strm >> *pers;
                m_arp.push_back(pers);
        }
}
```

This error, unfortunately, has no easy solution (except to reimplement the entire class as a template class, a process discussed in Chapter 13). The problem is as follows:

- When we were adding an element to our KeyArray, we had a model we could use for creating our object—which could be a Person, an Employee or (as we will find later) a Customer, Film or Rental object. We didn't have to know what type because the NewCopy() function creates the appropriate type of object using polymorphism.

- When we are loading objects, however, we will not know what type of object is being loaded until we look at the data—since there is nothing in our KeyArray class to prevent us from mixing object types. (This was not a problem in our PersonArray class since we knew we'd be loading and saving Person objects.)

There are various different approaches that can be used to solve this problem. One might be to "grow your own" solution. For example, you could:

- Assign a unique code to identify each type of object inheriting from CKey
- When saving the object, save its code first (e.g., the way we saved the number of elements in a collection) then save the object of the data.
- When loading the object, read the code first then, in some static function consisting of a giant case construct, use the code to determine the appropriate class to create, after which the class data could be loaded.

Another approach would be to inherit from predefined class that, essentially, does the above for you. An example of such a class is the CObject MFC class, used as a base class for nearly all MFC objects. Unfortunately, there is no standard C++ class that is available for this purpose.

Yet another approach—and the approach we shall take—is to "punt" and make the Load() member of our KeyArray class a pure virtual function. This creates an abstract class, whose declaration is shown in Example 10.10. The implementation of the functions is not shown, since they are exactly the same as those in Chapter 16, Section 8.3.3 after doing the name substitutions and changing the line in the Add() member.

Example 10.10: KeyArray class declaration

```
class KeyArray
{
public:
	KeyArray(void);
	~KeyArray(void);

	const CKey *operator[](int) const;
	const CKey *operator[](const char *) const;
	const CKey *operator[](const string &szKey) const;
	const CKey *Find(const char *szKey,int &nPos) const;
	const CKey *Find(const char *szKey) const;
	const CKey *Find(const string &szKey) const;

	int Add(CKey *);
	int Add(const CKey &pers);
	void Remove(const char *szKey);
	void RemoveAt(int nVal);
	void Remove(CKey *p);
	int Size() const;
	void RemoveAll();

	virtual void Load(fstream &strm)=0;
	virtual void Save(fstream &strm) const;
	virtual void Display() const;

protected:
	vector<CKey*> m_arp;
	int FindPos(const string &szKey) const;
	int FindPos(const char *szKey) const;
};
```

What this means it that whenever we inherit from the KeyArray object, we'll need to define a Load() function.

10.4.3: The EmployeeArray Class (Revised)

The EmployeeArray class inherits from KeyArray and adds serialization of objects—which we know will be Employee objects—along with a Display() member that shows its contents and various << and >> overloads. The class declaration is shown in Example 10.11.

Example 10.11: EmployeeArray Declaration

```
class EmployeeArray :   public KeyArray
{
public:
      EmployeeArray(void);
      virtual ~EmployeeArray(void);

      int Add(const Employee &emp){return
KeyArray::Add(emp.NewCopy());}
      void Load(fstream &strm);
};
```

Besides adding Load() to the EmployeeArray class, there was one other major change: an override of the two Add() members to take Employee objects as arguments, instead of the more general CKey objects. Since the Add() member is the only way to get objects into the collection, this ensures we'll only have Employee objects in the vector<CKey*> collection m_arp. This is critical, because our serialization operations will crash and burn if non-Employee objects are present.

> *Test your understanding:* Why didn't we override all the remaining CKey related member functions the way we overloaded Add()?

The member Load() member is nearly identical to those that we developed for the original PersonArray class. It is presented in Example 10.12.

Example 10.12: EmployeeArray I/O Members

```
void EmployeeArray::Load(fstream &strm)
{
      RemoveAll();
      int nVal;
      strm.read((char *)&nVal,sizeof(int));
      for(int i=0;i<nVal;i++) {
            Employee *pers=new Employee;
            pers->Load(strm);
            m_arp.push_back(pers);
      }
}
```

The only real change we needed to make to the original PersonArray version was to create a new Employee object (instead of a Person object) when we are loading.

> We now have a specialized array for storing Employee data that is easily modifiable for use with other classes.

10.4.4: The KeyTable Class

Our KeyArray class has two areas of inflexibility that place significant limitations on its use:

1. Only objects inheriting from CKey can be placed in it.
2. Only one key can be specified per object (e.g., once we choose ID as a key for the employee class, we can no longer do an alphabetical sort by name).

To address these limitations, we now create a KeyTable class that can be used to create a sorted array of any objects.

KeyValPair class
At the heart of our KeyTable is a small helper class, the KeyValPair class, that we use associate keys with pointers to anything. The implementation of this class is extremely straightforward, and is presented in Example 10.13.

Example 10.13: KeyValPair helper class

```
class KeyValPair :
      public CKey
{
public:
      KeyValPair(void){m_pVal=0;}
      KeyValPair(const char *szKey,void *pVal){
            m_szKey=szKey;
            m_pVal=pVal;
      }
      KeyValPair(const string &szKey,void *pVal){
            m_szKey=szKey;
            m_pVal=pVal;
      }
      KeyValPair(const KeyValPair &pair){Copy(pair);}
      virtual ~KeyValPair(void){}
      const KeyValPair &operator=(const KeyValPair &pair) {
            if (&pair==this) return *this;
            Copy(pair);
            return *this;
      }
      void Copy(const KeyValPair &pair) {
            m_szKey=pair.m_szKey;
            m_pVal=pair.m_pVal;
      }
      void *Value() const {return m_pVal;}
      // Required members
      KeyValPair *NewCopy() const {return new KeyValPair(*this);}
      string Key() const {return m_szKey;}
protected:
      // These are protected to prevent calls
      void Load(fstream &strm){}
      void Save(fstream &strm) const {}
      void Display() const {}
protected:
      string m_szKey;
      void *m_pVal;
};
```

Almost every function in the KeyValPair class is an accessor function or constructor/copy related function. One interesting thing to notice in the design of the class is the three protected member functions that don't do anything, i.e.:

```
protected:
      // These are protected to prevent calls
      void Load(fstream &strm){}
      void Save(fstream &strm) const {}
      void Display() const {}
```

We needed to override these functions to key the class from being abstract (since it inherits from CKey). On the other hand, we don't need to support loading or saving—

especially since we've got know idea what m_pVal points to. By making the functions protected, we ensure that a class user doesn’t call them.

KeyTable Declaration

Because the KeyValPair class inherits from CKey, it can be used directly in a KeyArray collection. For our purposes, however, we choose to hide the KeyValPair implementation. To do this, we create a new KeyTable class that inherits from KeyArray, changing the members so that keys and associated data pointers are added and retrieved directly, and KeyValPair objects are used for internal purposes only. The class declaration that does this is shown in Example 10.14.

Example 10.14: KeyTable class declaration

```
class KeyTable :
      public KeyArray
{
public:
      KeyTable(void);
      virtual ~KeyTable(void);


      void *operator[](int) const;
      void *operator[](const char *) const;
      void *operator[](const string &szKey) const;
      void *Find(const char *szKey,int &nPos) const;
      void *Find(const char *szKey) const;
      void *Find(const string &szKey) const;

      void *Value(int i) const;
      string Key(int i) const;

      int Add(const string &szKey,void *val);
      int Add(const char *szKey,void *val);

      // Required members protected so they don't get called
protected:
      virtual void Load(fstream &strm){}
      virtual void Save(fstream &strm) const{}
      virtual void Display() const{}
};
```

Once again, we hide the Load(), Save() and Display() functions (required for KeyArray classes) by making them protected. Because doing this suggests that our class design might be a bit sloppy, and end-of –chapter question addresses how we might clean it up, so we didn’t have to protect these members.

Interface
The highlights of the changes to the interface are as follows:

- Our Find() members and [] overloads now return the data pointer associated with a given key string or position.
- Our Add() function takes a key and the data we want to associate with it as arguments.
- The Value() and Key() member functions have been added to allow us to access the key string and pointer value of each element.

Suppose, for example, we wanted to get an alphabetical list of our employees. Unfortunately, the Employee class uses ID as a key, so just displaying the contents of an EmployeeArray won't help us. On the other hand, we can create a KeyTable and add the Employee objects to that table using the Person::Key() version of the member—which will return the last name, first name string. This is illustrated in the code fragment below:

```
int i;
EmployeeArray ar;
// Add some data to the above array (omitted)
KeyTable tab;
for(i=0;i<ar.Size();i++) {
        Employee *pEmp=(Employee *)ar[i];
        tab.Add(pEmp->Person::Key(),pEmp);
}
for(i=0;i<tab.Size();i++) {
        Employee *pEmp=(Employee *)tab[i];
        pEmp->Display();
}
```

The nice thing about KeyTable is that you're not even limited to single member functions, you can make up your own. For example, if you wanted a sort by department, and by name within each department, if your Employee has a Department() member that returns the department name, you could put the elements in the table as follows:

```
for(i=0;i<ar.Size();i++) {
        Employee *pEmp=(Employee *)ar[i];
        tab.Add(pEmp->Department()+pEmp->Person::Key(),pEmp);
}
```

One problem that could occur using the KeyTable is duplication—while we know we'll never have two employees with the same ID, its possible they could have the same first and last names. To deal with this, you can simply append a value to the string that you know is going to be unique, for example:

```
    for(i=0;i<ar.Size();i++) {
            Employee *pEmp=(Employee *)ar[i];
            tab.Add(pEmp->Person::Key()+pEmp->Key(),pEmp);
    }
```

You could even use the sequence number itself, e.g.:

```
    for(i=0;i<ar.Size();i++) {
            Employee *pEmp=(Employee *)ar[i];
        char buf[80]={0};
        sprintf(buf,"%i",i);
            tab.Add(pEmp->Person::Key()+buf,pEmp);
    }
```

Implementation
Because of the similarity between a KeyArray and a KeyTable, the overrides required are relatively modest. The most substantial change is to the Add() members, which construct local KeyValPair objects prior to calling the base class version of Add(), as shown in Example 10.15.

Example 10.15: KeyTable::Add() member functions

```
int KeyTable::Add(const string &szKey,void *val)
{
      KeyValPair pair(szKey,val);
      return KeyArray::Add(pair);
}
int KeyTable::Add(const char *szKey,void *val)
{
      KeyValPair pair(szKey,val);
      return KeyArray::Add(pair);
}
```

The two new members, Key() and Value(), make a call to the base class [] to access the associated KeyValPair object stored in the array, then return the Key() or Value() member from the object. The main thing of interest in these functions, shown in Example 10.16, is how they call the base class version of the operator using its functional form.

Example 10.16: Key() and Value() member functions

```
void *KeyTable::Value(int i) const {
      KeyValPair *pVal=(KeyValPair *)KeyArray::operator [](i);
      return pVal->Value();
}
string KeyTable::Key(int i) const {
      KeyValPair *pVal=(KeyValPair *)KeyArray::operator [](i);
      return pVal->Key();
}
```

The remaining overloads, provided purely for user convenience, all ultimately end up calling the Value() function if the key is found (since the void * is what is of interest, not the KeyValPair pointer which is used for internal purposes only). These are presented in Example 10.17.

Example 10.17: Find() and [] Overload Members of KeyTable

```
void *KeyTable::operator[](int i) const {
      return Value(i);
}
void *KeyTable::operator[](const char *szKey) const {
      return Find(szKey);
}
void *KeyTable::operator[](const string &szKey) const {
      return Find(szKey);
}
void *KeyTable::Find(const char *szKey,int &nPos) const {
      if (!KeyArray::Find(szKey,nPos)) return 0;
      return Value(nPos);
}
void *KeyTable::Find(const char *szKey) const {
      int nPos;
      if (!Find(szKey,nPos)) return 0;
      return Value(nPos);
}
void *KeyTable::Find(const string &szKey) const {
      int nPos;
      if (!Find(szKey.c_str(),nPos)) return 0;
      return Value(nPos);
}
```

10.5: Video Store Lab Exercise

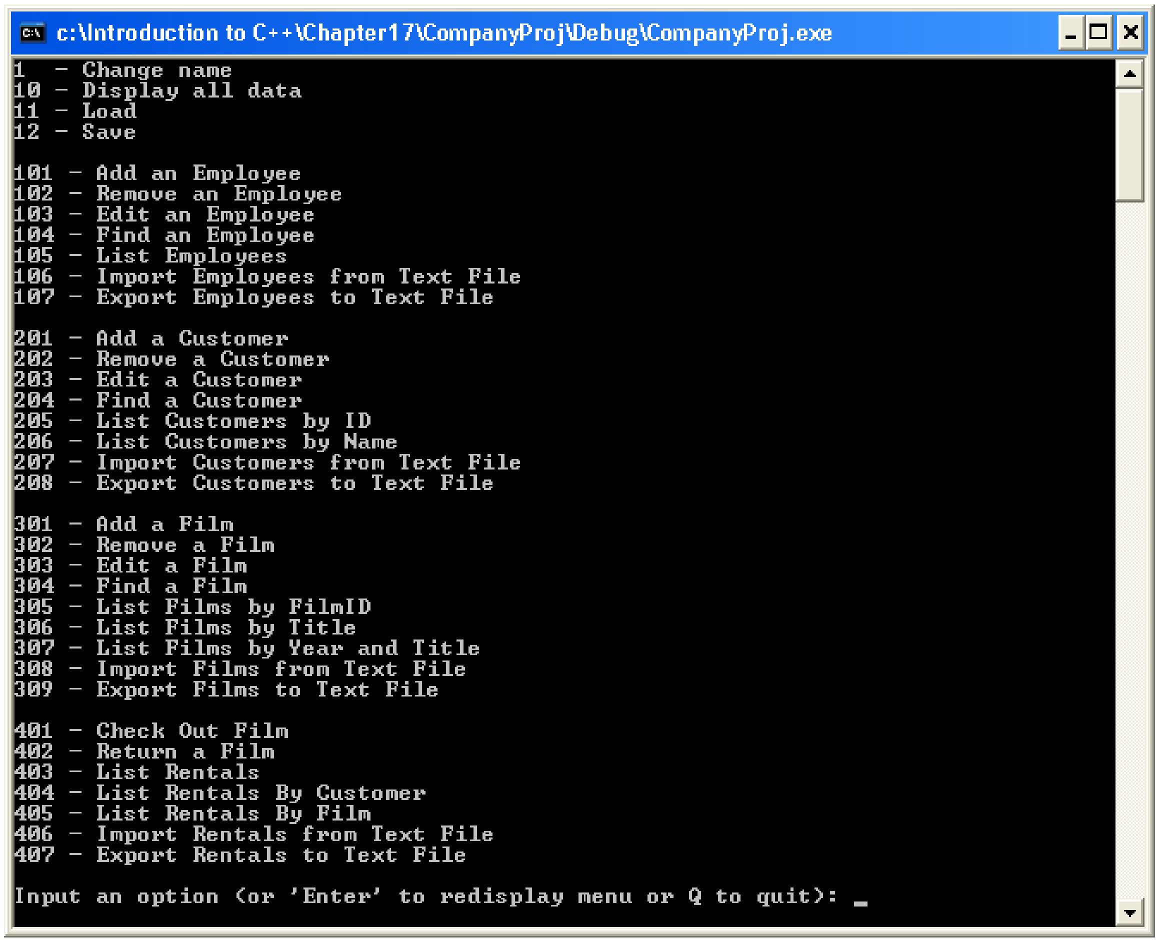
Walkthrough available in VideoStore.avi

The Video Store application is designed to demonstrate the construction of an application that makes extensive use of inheritance and collections of objects. It builds upon the Company and Store classes, developed in a Chapter 8 lab exercise (Section 8.5) and a Chapter 9 lab exercise (Section 9.4).

10.5.1: Overview

In this exercise, our goal is to complete the Video Store project, implementing the full menu shown in Figure 10.13. To accomplish this, we'll need to perform a number of steps, which include:

- Modify the underlying implementation of our collection unit classes (e.g., Employee, Customer) and collections to take advantage of the KeyArray developed in Section 10.4.
- Add a ListCustomersByName() member to the Store class (created in Section 9.4) that uses a KeyTable to create a sorted list.
- Design and program Rental and Film classes needed to keep track of our inventory, plus collection arrays to manage them (e.g., FilmArray, RentalArray)
- Create the VideoStore class, inheriting from Store, and implement its menu options.


c:\Introduction to C++\Chapter17\CompanyProj\Debug\CompanyProj.exe

```
1   - Change name
10  - Display all data
11  - Load
12  - Save

101  - Add an Employee
102  - Remove an Employee
103  - Edit an Employee
104  - Find an Employee
105  - List Employees
106  - Import Employees from Text File
107  - Export Employees to Text File

201  - Add a Customer
202  - Remove a Customer
203  - Edit a Customer
204  - Find a Customer
205  - List Customers by ID
206  - List Customers by Name
207  - Import Customers from Text File
208  - Export Customers to Text File

301  - Add a Film
302  - Remove a Film
303  - Edit a Film
304  - Find a Film
305  - List Films by FilmID
306  - List Films by Title
307  - List Films by Year and Title
308  - Import Films from Text File
309  - Export Films to Text File

401  - Check Out Film
402  - Return a Film
403  - List Rentals
404  - List Rentals By Customer
405  - List Rentals By Film
406  - Import Rentals from Text File
407  - Export Rentals to Text File

Input an option (or 'Enter' to redisplay menu or Q to quit): _
```

Figure 10.13: Interface for VideoStore object

10.5.2: Specifications

The complete Video Store application consists of 3 major classes (Company, Store and VideoStore), four unit classes (Employee, Customer, Film and Rental) and four unit-collection classes (EmployeeArray, CustomerArray, FilmArray and RentalArray). We begin by discussing the unit classes, which are the simplest.

Unit Classes
There are four fundamental object types that are managed within the complete Video Store application. These are Employee, Customer, Film and Rental. The Employee and Customer classes were developed in the previous chapter. All that remains for them is to change Person so that it inherits from CKey.

Like the already constructed Employee and Customer classes, each of the remaining unit classes support a common set of interface members that are listed in Table 10.2.

Members Supported by Unit Classes
string Key() const Returns a unique key for maintaining the object in a KeyArray-based collection. The nature of the key depends on the object type.
void Input() Prompts the user for input for each member of the class
void Display() const Displays the data values for the object.
string FixedString() const Returns a fixed field-width representation of the object's data, suitable for export to a text file.
void FixedString(const string &str) Retrieves data from a string containing a fixed-width representation of the object, consistent with what was produced by FixedString().
void Load(STREAM &strm) Loads a unit object stored in a binary file.
void Save(STREAM &strm) const Saves a unit object to a binary file

Table 10.2: Common properties of all unit classes

All the unit classes ultimately inherit from CKey—making it possible to hold a collection of them in a KeyArray object. The inheritance network is presented in Figure 10.14, with only selected members being shown.

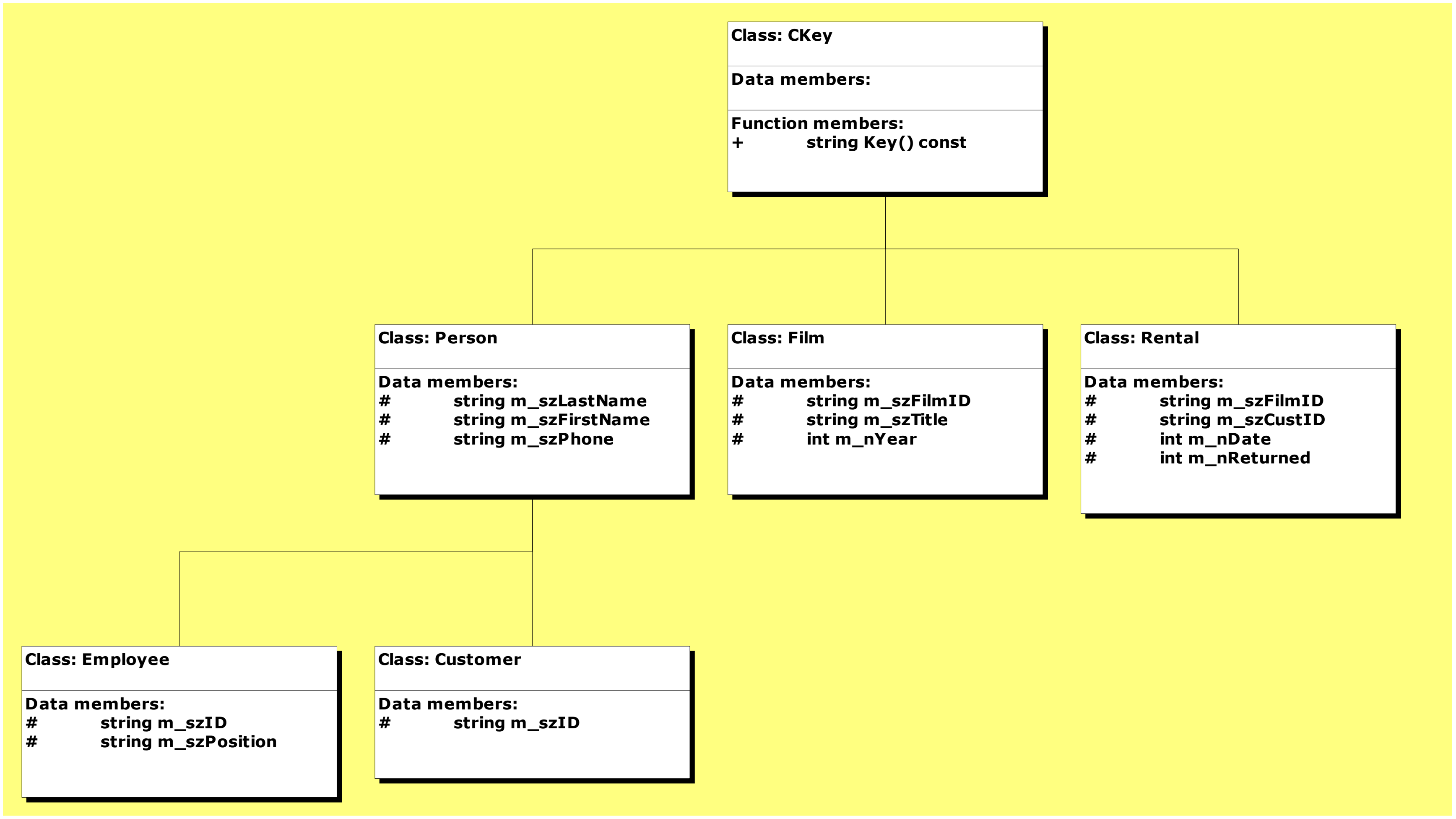


Figure 10.11: Inheritance Relationships Among the Unit Classes

The unit classes were intentionally designed to be simple, and relatively self-explanatory. Some comments, however, are required for each of the classes that need to be implemented:

- **Film:** Each film has a unique film ID code (you may assume no more than 10 characters for fixed width processing), a title and the year released. Only the latter requires any particular thought, since the previous classes only had string data. The utils.h file included with the project contains a Int2String() function that can be useful (particularly for preparing fixed string output).
- **Rental:** The Rental object requires a bit more handling. To begin with, it doesn't have its own built-in key. Instead, you may assume that the combination of date, film ID and customer ID are—together—unique. Also, the m_nDate (check out date) and m_nReturned (returned date) are integers stored in the format YYYYMMDD (e.g, July 18th, 2004 would be 20040718). For convenience, simple (and not very accurate) Date2String() and InputDate() functions are provided in utils.h. The reader is encouraged, however, to implement date classes such as those in the end-of-chapter exercises in Chapter 6, should time permit.

For each class, you should implement not only the functions listed in Table 10.2, but also appropriate accessor functions for getting and setting data member values.

Unit-Collection Classes
For each of our unit classes, we create a dedicated collection class—essentially identical in structure and interface to the revised EmployeeArray (Section 10.4.3). These classes can, essentially, be created with some creative search-and-replace of the EmployeeArray class. The alternative to this is to create a KeyArray template class (presented in Chapter 13, Section 13.6, as a lab exercise) or to use an STL template class such as **map<>** (Chapter 14).

Main Classes
As noted in the overview, the main classes are Company, Store and VideoStore. Most of the members of these classes relate to the collections they manage. The Company class, originally developed in Chapter 8, Section 8.5, requires only minimal modifications from its original form. Its top-level interface is unchanged. The only significant change to the implementation that is required is to make those functions that we intend to override virtual. These include:

```
virtual void RemoveAll();
virtual void DisplayAll() const;
virtual void Load(STREAM &strm);
virtual void Save(STREAM &strm) const;
virtual void Menu() const;
virtual bool Option(int nOption);
```

By inheriting these functions in each class, we do not have to change the top level functions SaveToFile(), LoadFromFile() and Interface().

The Store class requires only the addition of a single member function, ListCustomersByName(), which becomes option 206. The function should display all customers in alphabetical order (as opposed to being ordered by customer ID). It can be implemented using a KeyTable (Section 10.4.4).

The VideoStore class manages two collections—Film and Rental objects—and therefore has more top-level functions, as shown in Table 10.3.

Menu Option No.	Top Level VideoStore Functions
301	**void AddFilm()** Prompts the user for an Film ID and, if the ID is valid (i.e., not already present), allows the user to edit and add the Film to the FilmArray collection.
302	**void RemoveFilm()** Calls FindFilm(true) to access the Film object, displaying the data. Prompts the user for a confirmation and, if the user confirms, removes the Film object from the EmployeeArray collection.
303	**void EditFilm()** Calls FindFilm(false) to access the Film object, then allows the user to edit the Film.
304	**Film *FindFilm(bool bDisplay=true) const** Prompts the user for a Film ID and, if the ID is valid returns a pointer to the film. If bDisplay is true, displays the Film object's data.
305	**void ListFilms() const** Displays the data for all films in the FilmArray, one film per line.
306	**void ListFilmsByTitle() const** Displays the data for all films in the FilmArray in alphabetical order by title, one film per line. A good way to implement this is using a KeyTable object, as discussed in Section 10.5.4.
307	**void ListFilmsByYearAndTitle() const** Displays the data for all films in the FilmArray in alphabetical order by title, one film per line. A good way to implement this is using a KeyTable object, as discussed in Section 10.5.4.
308	**void ImportFilms()** Prompts the user for a file name (presumably a text file), then loads the fixed format film records into the FilmArray.
309	**void Export Films() const** Prompts the user for a file name (presumably a text file), then saves the film records from the FilmArray in a fixed text format.
401	**void CheckOutFilm()** Prompts the user for a Film ID and a Customer ID. If both are found, it prompts the user for a check-out date. If the checkout date is valid (you can use the InputDate() function in utils.h as a starting point), a Rental object is created and added to the RentalArray.
402	**void ReturnFilm()** Prompts the user for a FilmID, then checks for a Rental object with a matching film ID and a Return date of 0 (i.e., has not been returned). If it finds such a Rental object, it prompts the user for the return date and updates the m_nReturned member to that date.
403	**void ListRentals() const** Lists all Rental objects in the RentalArray, one per line. The default order should be checkout-date + film ID + customer ID (assuming the Rental Key() member is properly implemented).
404	**void ListRentalsByCustomer() const** Provides a listing of film ID and title, grouped by customer ID. The KeyTable will be a definite help in implementing this—which is reminiscent of grouping in a database report.
405	**void ListRentalsByFilm() const** Provides a listing of customer ID and name, grouped by film ID.
406	**void ImportRentals()** Prompts the user for a file name (presumably a text file), then loads the fixed format rental records into the RentalArray.
407	**void ExportRentals() const** Prompts the user for a file name (presumably a text file), then saves the rental records from the RentalArray in a fixed text format.

Table 10.3: Top Level Menu Options for VideoStore Class

The declaration of the VideoStore class used in the sample program is presented in Example 10.18.

Example 10.18: VideoStore Class Declaration

```
class VideoStore :
      public Store
{
public:
      VideoStore(void);
      virtual ~VideoStore(void);
      virtual void RemoveAll();
      // Top Level Functions
      // All Store functions plus
      virtual void DisplayAll() const;
      void AddFilm();
      void RemoveFilm();
      void EditFilm();
      Film *FindFilm(bool bDisplay=true) const;
      void ListFilms() const;
      void ListFilmsByTitle() const;
      void ListFilmsByYearAndTitle() const;
      void ImportFilms();
      void ExportFilms() const;
      void CheckOutFilm();
      void ReturnFilm();
      void ListRentals() const;
      void ListRentalsByCustomer() const;
      void ListRentalsByFilm() const;
      void ImportRentals();
      void ExportRentals() const;
      // Helpful functions
      bool AddFilm(const Film &pers);
      void RemoveFilm(const char *szKey);
      void RemoveFilm(const string &szKey);
      Film *FindFilm(const char *szKey) const;
      Film *FindFilm(const string &szKey) const;
      virtual void Load(STREAM &strm);
      virtual void Save(STREAM &strm) const;
      Film *GetFilm(int i) const;
      int FilmCount() const;
      Rental *GetRental(int i) const;
      int RentalCount() const;
      virtual void Menu() const;
      virtual bool Option(int nOption);
      void ImportFilms(INPUT &in);
      void ExportFilms(OUTPUT &out) const;
      void ImportRentals(INPUT &in);
      void ExportRentals(OUTPUT &out) const;
protected:
      FilmArray m_arFilm;
      RentalArray m_arRental;
};
```

10.5.3: Instructions

Although the number of functions to be implemented in the lab exercise is large, few of the individual functions are particularly complex. Indeed, the longest functions are those that do mechanical activities such as prompting the user for object information, displaying a menu and routing menu selections.

A good general approach to the exercise is the following:

- Modify the Store class. Add the ListCustomersByName() using the example presented in Section 10.4.4 as a guide.
- Implement the Film class. The only trick associated with this class is the fact that it has a numeric member (Year). There is a function in Utils.h, however, for converting an integer to a string object, which simplifies the process.
- Implement the Rental class. The trick here is the two date members. There are two functions in Utils.h, however, for converting back and forth between user input and the internal date format (YYYYMMDD). You will also need to convert that format to a string object (e.g., "20040718") as part of the key function. The advantage of the integer format is that it sorts in a sensible way.
- Implement the VideoStore class. A good way to postpone the more conceptually difficult portion of this is to do a first pass implementation that just deals with Film objects (options 301-309) and test that first. Once that is working, you can implement the Rental object management and reports (options 401-407).

Upon completion and testing of the VideoStore class, the lab is complete.

10.7: Review and Questions

10.7.1: Review

A particularly important type of behavior available in inheritance situations is polymorphism. Most commonly used where collections of object pointers or references are present, polymorphic member functions are determined by the nature of the object they are applied to. For example, if LiveBirth() is a polymorphic function:

```
Mammal *p1=new Lion;
Mammal *p2=new Platypus;
p1->LiveBirth(); // returns true
p2->LiveBirth(); // returns false
p2->Mammal::LiveBirth(); // returns true
```

In the first p2 example, even though Mammal returns **true** for LiveBirth(), the platypus pointer returns **false** because the function is overridden (since a platypus lays eggs). The second p2 example returns **true** because we've forced the base class version to be called with the scope resolution operator.

To declare a function polymorphic, the **virtual** keyword is placed in front of its prototype in the class declaration. In our platypus example, if LiveBirth() were *not* declared **virtual**, the following calls would return different values:

```
Platypus plat;
Mammal *p1=&plat;
plat.LiveBirth(); // returns false, the Platypus version
p1->LiveBirth(); // returns true, the Mammal version
```

A particularly powerful way to organize polymorphic objects is into inheritance hierarchies. Often, classes that are never intended to be standalone objects are used to organize the frameworks (e.g., a taxonomy used to create animal objects, where levels such as genus, species, etc. are used to classify and organize the hierarchy). Classes never intended to be objects are called abstract classes. In C++, declaring one or more pure virtual functions makes a class abstract. Such functions are implemented by placing = 0; after the function prototype in the class declaration, e.g.,

```
class CKey {
        string Key() const = 0;
        // etc.
};
```

Until a child of the abstract class provides a body for *all* pure virtual functions, the child class remains an abstract class as well.

In deciding whether or not to make a function polymorphic, the general rule is that functions should be virtual if:

- They are likely to be overridden at some future time
- They are functions you are likely to want to use in a collection of objects based upon some abstract class
- If it is a destructor function, unless you know the object will never be used as part of a collection and doesn't manage any memory internally.

Any member function that doesn't meet any of these three criteria can—and probably should—be left as a non-virtual function for reasons of efficiency and clarity.

Unlike many popular languages (e.g., Java and C#), C++ provides for multiple inheritance, allowing more than one base-class to be specified. This is accomplished by adding comma separated inheritance specifications in the class declaration, e.g.,

```
class Mule :
        public Horse , public Jackass
{
   // etc…
};
```

where Mule inherits from Horse and Jackass. Multiple inheritance tends to lead to a number of problems, including:

- Ambiguity when members with the same name are present in two or more base classes
- Duplication of data, when data members with the same name are inherited
- Conceptual issues (regarding whether or not two exclusive is-a relationships actually exist).

The **virtual** keyword, added to the inheritance specification can eliminate some duplication when two or more base classes come out of the same hierarchy. Another approach is to create interface classes—consisting solely of pure virtual functions—that can establish the presence of the member functions desirable for polymorphism, but force the programmer to define the functions in the inherited class (eliminating ambiguity). Using interface classes as a design principle is also likely to make porting code to other languages more straightforward.

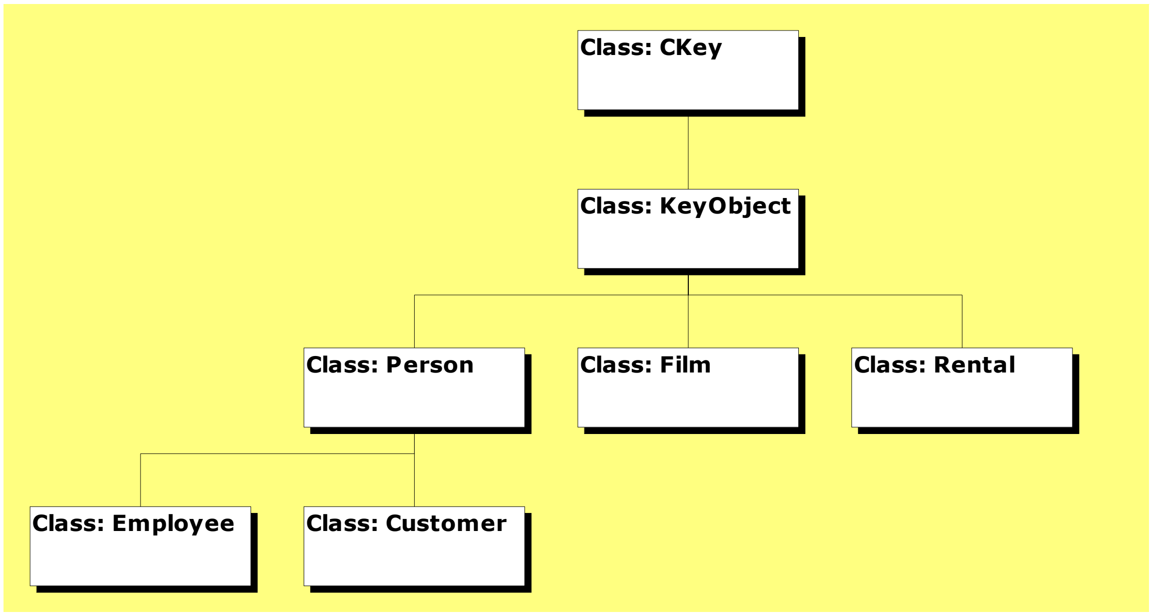
10.7.2: Glossary

Abstract class – A class which is used to organize an inheritance network but cannot be the basis of an actual object because it has one or more pure virtual functions
Base class – A class whose properties are being inherited in an inheritance relationship (same as parent class).
Child class – A class that inherits properties from one or more parent classes.
dynamic_cast<> – A C++ templated operator used as a substitute for the C typecast syntax. A dynamic_cast<> operation does runtime checking to ensure the cast is valid, returning a 0 (pointer) or an exception (reference) if it is not. It is particularly valuable for polymorphic objects.
Hierarchy – A collection of classes parent-child relationships.
Interface Class – An abstract class used that can in multiple inheritance situations that consists solely of pure virtual functions.
Multiple inheritance – The ability of an object to inherit directly from more than one base class.
Override – Redefining a base class member function in a child class. Once a given function name has been overridden, all other overloads that may exist in the base class are hidden.
Parent class – A class whose properties are being inherited in an inheritance relationship (same as base class).
Polymorphism – The ability to choose what member function to call at runtime when an object pointer or reference is present. Polymorphism allows collections of objects in the same hierarchy to perform different tasks when the same member function is used on each object.
Private inheritance – A form of inheritance in which all public and protected members of the base class become private in the child class.
Protected inheritance – A form of inheritance in which all public and protected members of the base class become/remain protected in the child class.
Public inheritance – A form of inheritance in which all public members of the base class remain public in the child class. Private members of the base class are always inaccessible in the child class.
Pure virtual function – A function declared without a body that must be defined in a child class.
thunk – A term used for a virtual function pointer embedded in a polymorphic object.
vftable[] – A term sometimes used to describe the table of function pointers created for virtual functions.
virtual Function – A member function declared to be polymorphic in nature. Pointers to these functions are normally implemented as hidden members of objects.
virtual Inheritance – A form of multiple inheritance that combines duplicated member data and functions when two classes in the same hierarchy are both use as base classes.

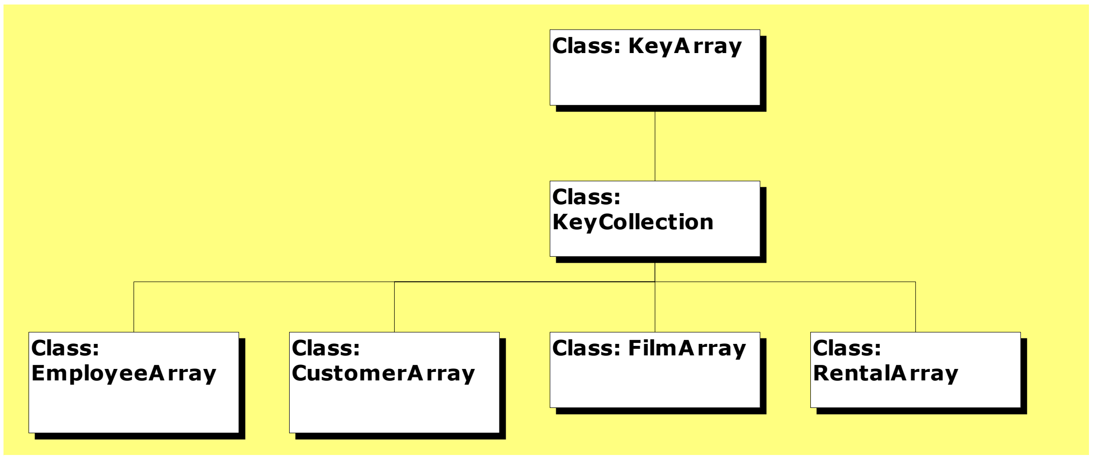
10.7.3: Questions

10.2: Employee and Contractor Hierarchy. Create a hierarchy of employee and independent contractor classes, focusing on how pay is calculated (e.g., W2 vs. 1099, exempt vs. non-exempt, hours*wage, salary, commission*sales, etc.). Identify some categories that could benefit from multiple inheritance (e.g., sales manager).

10.3 Revised unit object hierarchy. In the Section 10.6 lab exercise, suppose we decided to add another abstract class to our hierarchy, KeyObject, as shown below. What pure virtual member functions would it make sense to place in KeyObject? Would this modification offer any benefits?



10.4 Revised unit-array object hierarchy. In the Section 10.6 lab exercise, suppose we decided to add another abstract class to our hierarchy, KeyCollection, as shown below. What member functions would it make sense to place in KeyCollection? Would this modification offer any benefits? (*Hint:* think about how you needed to modify the EmployeeArray code for each different collection. Properly implemented, the KeyCollection class could be constructed with only 1 pure virtual function needing to be overridden in each child class).



10.5: Base class for Company. Would there be any reason to inherit the Company class from an abstract base class, instead of starting it from scratch? What member functions would be placed in the class? Can you think of circumstances where it might make sense to define such a base class?

10.6: Customer rental records. Suppose we wanted to keep track of all the films each customer had rented as part of the Customer class. How would we need to modify the Customer class, and would inheritance be a good way of doing this? How would you need to modify the Video Store application to permit this information to be collected?

10.8: Sorted array allowing duplicate values. Create a SortedArray class, that inherits from KeyArray, that will allow duplicate key values.

In Depth Questions

10.13: Multiple film copies. How would you need to modify the Video Store application if our store had multiple copies of some of its films? Consider whether inheritance would be the best technique for handling this modification.

10.14: Multiple cassette rental transactions. How would you need to modify the Video Store application to allow for a customer to rent multiple videos in the same transaction? Consider what classes you'd want to add and what, if any, use you could make of inheritance.

10.15: Video sales. How would you need to modify the Video Store application to allow for sales of videos, in addition to rentals? Consider what classes you'd want to add and what, if any, use you could make of inheritance.

Part III:

Collections

Chapter 13

Collections and Templates

Executive Summary

Chapter 13 considers the general programming concept of a collection shape, then examines preferred means of implementing such collections in C++: template classes. Understanding collections is crucial to effective OOP, and is the central focus of Part III of this book. Templates, in turn, are a relatively new addition to C++ and one of the most powerful capabilities the language possesses. They are also a capability that is absent from many other important modern languages, such as Java and C#.

The chapter begins with an introduction collections, introducing the notion of a collection shape. It then surveys a variety of common collection shapes, many which will be covered in subsequent chapters. It then proceeds with some background on templates and the STL, then discusses how to construct template functions and classes. Two general classes are then developed in walkthroughs: a Point class that allow coordinate classes of various types to be defined and the GArray template class, modeled after the MFC CArray template class. The chapter concludes with a lab exercise that implements a sorted vector collection, creating a template version of Chapter 10's KeyArray class.

Learning Objectives

Upon completing this chapter, you should be able to:

- Explain the origins and purpose of the STL
- Create objects based on a C++ template class
- Describe the basic approach to creating a C++ template
- Explain what is meant by collection shapes
- Describe the properties of an iterator
- Create template functions
- Create template classes

13.1: Introduction to Collections

When you think of objects, in the context of object-oriented programming, it is natural to think of self-contained objects like employees or your typical row (record) in a database table. While these types of objects are certainly important, as your programs grow more and more complex, less and less of your time will be spent on the objects themselves. What will really challenge you is managing collections of objects.

Take, for example, program like FlowC. The "flash" of the program like that is the graphics, the shadowing, the fonts. As a practical matter, however, building in these capabilities took about 1% of the development time—if that. The challenging programming involved doing things liked developing a data structure for managing the various elements that go into a function, then for managing the functions attached to a class, then for managing the classes attached to a project. Every time the user touches an element, the collection class managing it had to ensure that nothing was done that impacted its integrity—such as changing a function name to one that was already in use. Then there is the matter of layout—each construct needs to manage a collection of its own drawing objects, plus the collection of drawing objects for embedded constructs. To add to the overall challenge, each graphic Window needed to maintain a collection of layout elements telling it where each box, diamond, oval, line and text element needs to be placed.

In this section, we provide a discussion of collections. In the remaining chapter sections we examine the mechanics of creating such collections: the C++ template.

13.1.1: Collection Shapes

Excluding the class itself—which can be viewed as a collection of data elements of different types—we have only really used one collection so far in this text: the array. Now the array is certainly a very powerful collection, especially if we encapsulate it and template it to make it type safe (as we do when we create a **vector<>** object). But an array is only one type of collection, and there are many more. Indeed, aside from I/O, collections are the most important part of the STL. In this section, we turn to the question of what makes a particular collection appropriate.

A collection is defined by its *shape*. By shape, we refer to a set of characteristics that describe capabilities available for accessing and modifying the collection itself and the elements within. What are generally of greatest interest when we look at shape characteristics are two issues: 1) the time it takes to apply the particular capability on a given collection, and 2) how that time changes as the collection grows.

As a general premise, nearly all collections support some form a linear access (i.e., accessing one element at a time, in an order determined by the collection). Other characteristics that help define a collection's shape include:

- *Random access, by position.* An array allows you to move directly to any element in the collection (if you know the position of the element)—like moving directly to a track on a CD. Other collections (e.g., the linked list) do not—to get to an element in the middle you have to trace through some or all the intervening elements.
- *Random access, by value (keyed access)*: How long will it take to find a given element based upon some key value of that element—usually expressed as a function of collection size. Lookups can be further broken down into time to find exact matches (e.g., looking up the name associated with a social security number, so nearby SSNs are irrelevant) and approximate matches (e.g., looking up how to spell a word in a dictionary), which is very different
- *Insertion time:* how long does it take to insert an element into the collection. This is also, sometimes, broken into insertion at front, insertion at back and insertion in the middle.
- *Deletion time:* how long does it take to remove an element from the collection.

13.1.2: Common Shapes

There are a number of shapes commonly used in programming. Some of these are supported directly by the STL, while the others need to be implemented by the programmer. In this section we'll review some of the most common collection types. Understanding how these shapes can be implemented, and how they can impact program performance, is the central focus of Part III of the book.

Array-based collections: vector, deque and sorted array
The array-based collections are comparable to the **vector<>** that we have already discussed. These collections provide very efficient access of elements by position. When we need to insert/delete elements anywhere but the end, however, we need to move all the elements following the insertion/deletion. This requires a loop that will be proportional to array size in time. For example, if the array has N elements, and insertion in the middle will require N/2 elements to be moved—which is proportional to end. As a consequence, arrays can become inefficient when data sets grow very large unless we're always adding elements to the end.

Access by content in arrays tends to be linear (i.e., proportional to the number of elements in the array). If the array has been sorted, however, search techniques can reduce this time dramatically. To get an idea as to how such reduction is possible, let us revisit binary search, originally presented as an example in Chapter 8 (Section 8.3.3, Example 7.11), and repeated here as Example 13.1, which looked for a match in an array of objects sorted by their Key() value. The algorithm conducted a search of a sorted array in much the same way that you or I would search a phone book for a name: it takes a look in the middle to figure out which half of the book the name in in, then kept repeats the process on the selected half.

Example 13.1: Example of a Binary Search Algorithm

```
// Implements binary search
int PersonArray::FindPos(const char *szKey) const {
      int nLower=0,nUpper=(int)m_arp.size();
      if (m_arp.size()==0 || m_arp[0]->Key()>=szKey) return 0;
      while(nUpper-nLower>1) {
            int nTest=(nUpper+nLower)/2;
            if (m_arp[nTest]->Key()<szKey) nLower=nTest;
            else if (m_arp[nTest]->Key()>szKey) nUpper=nTest;
            else return nTest;
      }
      return nUpper;
}
```

Repeating the earlier description, the algorithm—which can be adapted to any *sorted* array—works as follows:

- Two variables, nLower and nUpper are defined to hold the range being searched. nLower starts at 0 (the "lowest" element in the array), nUpper starts at the size of the array—which will be 1 higher than the largest legal coefficient (since all arrays in C++ are zero-based).
- We check the key we are looking for with the lowest element. If it is less than or equal to that element—or if the array size is 0—we return 0.
- We then enter a loop which does the following:
    - We create a test position half-way between the upper and lower bounds
    - We check the key at that test position.
    - If what we are looking for is greater than the test key, the test position becomes our new lower bound (since we know the key we're looking for can't be in the bottom half or our range).
    - If what we are looking for is less than the test key, the test position becomes our new upper bound.
    - If the test key matches what we are looking for, we return the test position.
- Once we have reset our bounds within the loop, we repeat the process with our new upper or lower bound. Each time the loop repeats, the area we search becomes half as large as the previous range.
- The loop ends when either: 1) we return because a match was found, or 2) the upper and lower bound difference is 1—meaning that the item is not there. If this is the case, we return the upper bound.

The significance of its return value, no matter how the function ends, is as follows:

- If it finds the key, it returns the position of the key

- If it doesn't find the key, it returns the position where the key would be located if we were to insert it.

If we look at the time it would take to perform such an algorithm, we notice that each time we iterate, we cut the size of the array we are searching in half. So, if we search *k* times our size will be $N*(1/2)^k$. We also know that the loop stops when the array size is 1. As a result, our maximum number of iterations will be given be the formula:

$$N*(1/2)^k = 1$$

or, doing some simple algebra:

$$N = 2^k$$

Using some slightly more advanced algebra, we can use the fact that $k = \log_2(2^k)$, which means that k—our maximum number of passes—is given by the formula:

$$k \propto \log(N)$$

In other words, search time is proportional to the logarithm of the number of elements being searched (since all logarithms—whatever their base—are proportional to each other).

What are the practical implications of this? Suppose then, we have a computer where it takes us 1 unit of time, using binary search, to search through 10 elements. To give linear search a head start, let's also assume it takes linear search the same amount of time to search through 10 elements. What happens to relative search times as the number of elements in our search space grows? This is illustrated in Table 13.1.

Number of elements	*Linear search time*	*log(elements)—binary search time*
10	1 (given)	1 (given)
100	10	2
1,000	100	3
10,000	1,000	4
100,000	10,000	5
1,000,000	100,000	6
10,000,000	1,000,000	7
100,000,000	10,000,000	8

Table 13.1: Growth in search times

Naturally, the time it takes to search an array-based collection needs to be weighed against the time it takes to sort it (a topic that will be considered in Chapter 15). For some collections, we shall find it is not worth the effort to sort. Nonetheless, the speed of keyed access is a very important shape property.

Linked list: list
The linked list collection shape typically ties its elements together with pointers, as illustrated in Figure 13.1.

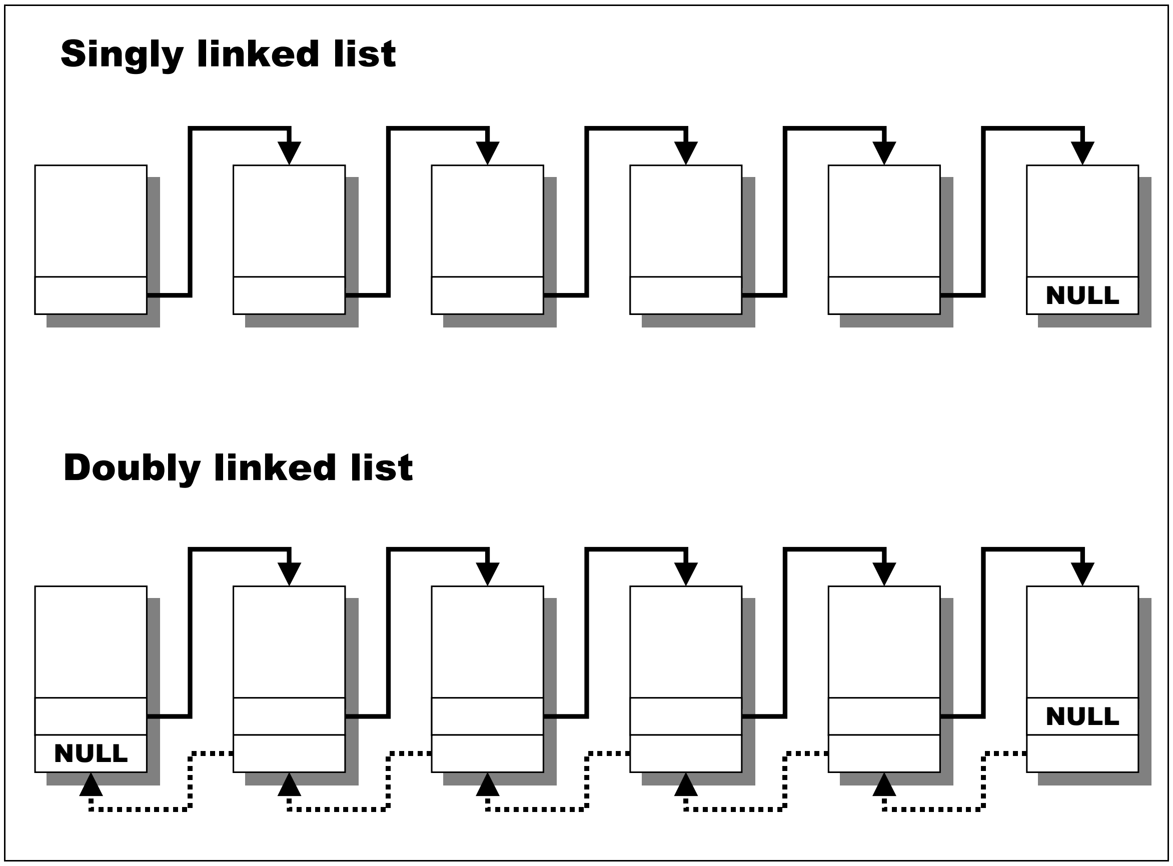


Figure 13.1: Singly and doubly linked lists

The shape makes insertion/deletion of elements anywhere in the list relatively fast, since we don't have to move the existing elements, just splice the pointers to add the new element to the chain or remove an existing element.

Conceptually, a list is like a tape cassette, as opposed to an array—which is like a CD. What this means is that there is no random access: to access a list element by position you need to start at the beginning (or and, if the list is doubly linked) and jump from link to link until you reach the desired content—making keyed access linear (which is as bad as it gets). As a result, lists are typically used for sequential processing or as an underlying shape for implementing queues or stacks.

Trees
The tree shape uses element pointers, the way a list does. Unlike a list, however, each tree element—often referred to as a node—has at least two pointers that allow the tree to branch out, as shown in Figure 13.2 for a binary tree (i.e., a tree where each node has two branches).

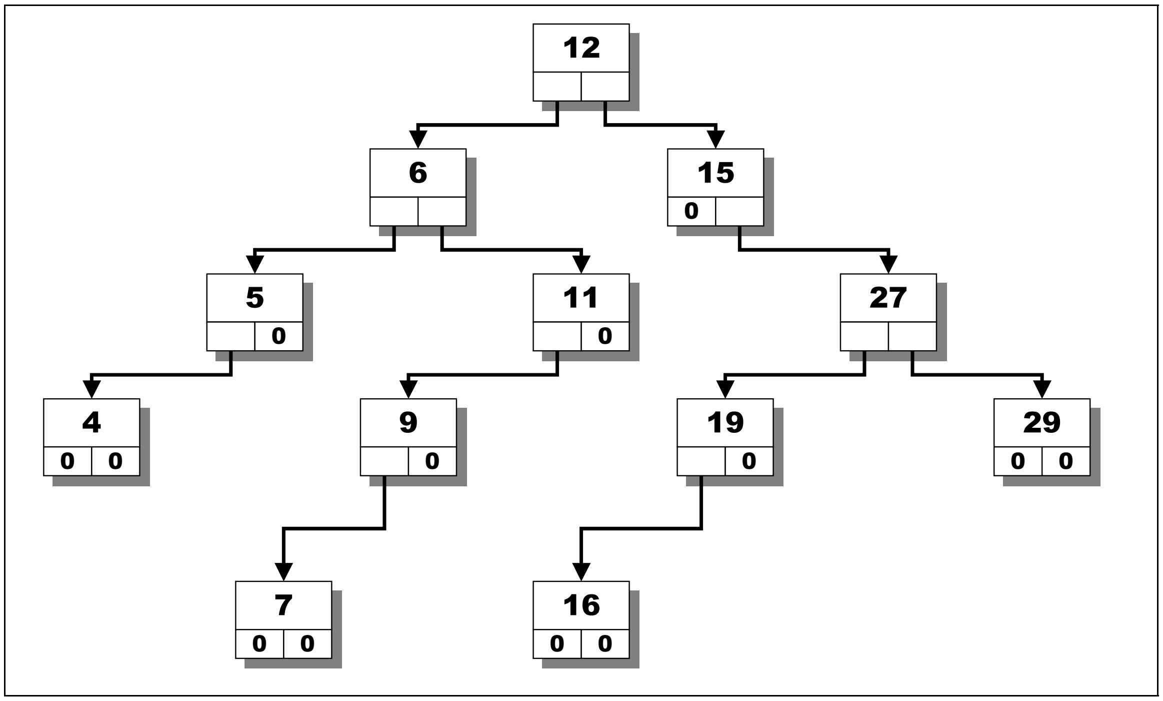


Figure 13.2: A sorted binary tree

An important subcategory of trees is sorted trees. In Figure 13.2, for example, the elements are arranged such that for every node, the left hand pointer points only to nodes with values less than of equal to the node's value, while the right hand pointer points only to nodes of greater value. Arranged in this way, access by content becomes roughly logarithmic. Although, like a list, random access by position is not available, it is also possible to make access by position logarithmic.

There are many ways to construct a tree, so there is no general STL collection template called "tree" (although the underlying implementation of a **map<>** is often tree-based). Sorted trees, however, are a very useful shape in situations where keyed elements are frequently added and removed—as such insertions/deletions can occur in logarithmic time (i.e., the time it takes to find them), as opposed to the linear time required for array insertion/deletions. For this reason, trees are often used as the underlying shape for holding information such as database index tables.

Lookup table: hash_map
The primary characteristic of the lookup table shape is that it allows for keyed access to elements that is virtually independent of collection size. On the down side, however, such speed only applies to exact matches. Whereas a sorted array or tree stores elements with similar keys nearby (e.g., their key may begin with the same few letters), in a typical

lookup table implementation, similar elements tend to be randomly scattered throughout the table, as illustrated by the keys array in Figure 13.3. This makes the shape a good choice where exact matches are required (e.g., accessing information by customer number or SSN), and a very bad choice in situations where adjacent elements are of interest (e.g., a dictionary that will be used to lookup the spelling of words).

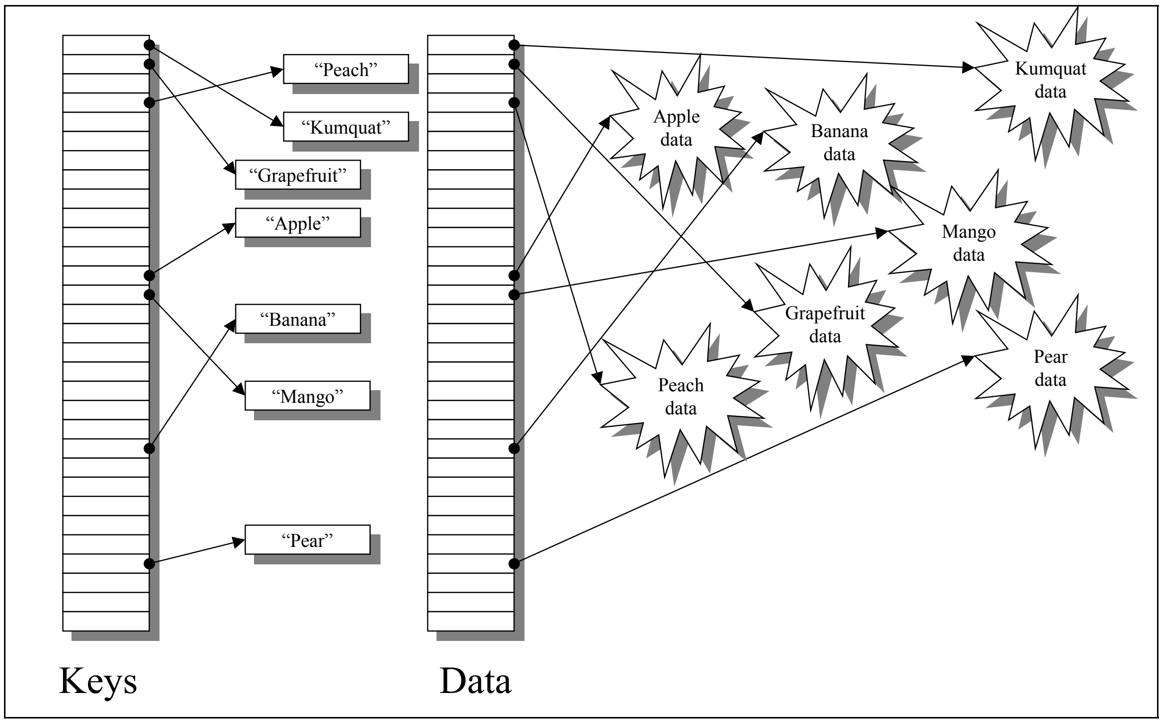


Figure 13.3: Typical parallel array lookup table organization

A typical lookup table will use a technique called hashing. One way to implement a hashing algorithm, illustrated in Figure 13.3, is to have to parallel arrays—one containing pointers to key values and the other containing pointers to the actual data associated with the key. A function that takes the key as an argument then generates a "hash code" used to determine the position of the key and data pointers in the parallel arrays. The nature of the hash code function tends to lead to the apparent randomness in key distribution—which accounts for the shape's inability to access data keyed with similar—but not identical—keys.

Summary

In Table 13.2, we present some common collection types, the associated STL objects, and their characteristics. Of these collections, the STL **vector**, **list** and **map** are the most commonly used.

STL object	Collection Description	Shape characteristics
deque **vector**	Dynamic array type objects that provide random access and can grow efficiently when elements are added to the end. **deque** also allows fast insertion at the beginning.	Random access: yes Keyed access: no Insert/Delete at beginning: linear (**vector**) Insert/Delete at beginning: fast (**deque**) Insert/Delete in middle: linear Insert/Delete at end: fast
(User implemented ***sorted array***)	Sorted array, using vector or deque objects.	Random access: yes Exact keyed access: logarithmic Approximate keyed access: logarithmic Insert/Delete: linear+
list	A collection of objects linked together in a conceptual chain that permits efficient element insertion and deletion.	Random access: no Keyed access: linear Insert/Delete: fast
queue	A collection, normally used for processing elements in sequence, that only allows elements at both ends to be accessed. Implemented as an adapter that is attached to some other underlying shape.	Random access: no Keyed access: no Insert/Delete at beginning: fast Insert/Delete in middle: no Insert/Delete at end: fast
stack	A collection, normally used for processing elements, allowing only elements at the end to be accessed, in LIFO (last-in, first out) order. Implemented as an adapter that is attached to some other underlying shape.	Random access: no Keyed access: no Insert/Delete at beginning: no Insert/Delete in middle: no Insert/Delete at end: fast
(User implemented ***sorted tree***)	A collection of objects linked together to permit efficient searching, addition and deletion. STL **map** template class stores objects in sorted order, and allows for binary searching, but does not support random access.	Random access: no Exact keyed access: logarithmic Approximate keyed access: logarithmic Insert/Delete: logarithmic
hash_map	A lookup table, normally implemented with a technique called hashing, that allows keyed elements to accessed rapidly, and independent of the table size	Random access: no Exact keyed access: size-independent Approximate keyed access: no Insert/Delete: fast

Table 13.2: Common collection types and their shape characteristics

13.1 Section Questions

1. Is a collection's shape best viewed as an interface or implementation property?
2. As elements are added to the end of a dynamic array (e.g., a **vector<>**) we periodically need to resize it. Would the same be true of a list shape?
3. Why does it make sense to implement stacks and queues on top of other collections?
4. Why does the STL fail to provide a standard tree collection?
5. What are the principal advantages of creating an iterator data type with each template class?
6. Is there any obvious drawback of the iterator interface?
7. Does the fact that iterators act like pointers indicate that they are pointers?

13.2: Introduction to Templates

In this section, we take a look at the evolution and use of templates—particularly template classes—and consider the nature of iterators. Sections 13.3 and 13.4 then provide a brief survey of the creation to template functions and classes.

13.2.1 Evolution of C++ Templates

Although Bjarne Stoustrup developed the initial version of C++ programming language in the early 1980s, many of the most important aspects of the language did not become commercially available until the mid-1990s, culminating in the adoption of the ISO standard in 1998.

The most important of the changes that transformed the language in the mid-1990s was the incorporation of support for C++ templates and namespaces into C++ compilers. In the Visual Studio family, for example, template support began with version 4.0, during the 1995-1996 time frame. With template capability built into the compiler, it became possible to include the Standard Template Library—originally developed at Hewlett Packard and then placed in the public domain—with the compiler. The new compiler capabilities also made it possible for Microsoft to develop its own alternative proprietary template-based libraries (e.g., the Active Template Library, or ATL) that could be used as an alternative to the MFC for developing MS-Windows applications and components.

One of the things that makes templates in general, and the STL in particular, so important is the way they enhance our ability to create type-safe collections of data. The use of templates to enhance collections is the principal focus of the remainder of the chapter.

Because the STL has, with some modifications, been incorporated into the C++ ISO standard, the original C++ standard library—designed, for the most part, in the early 1980s—is no longer widely used. Programmers need to be aware, however, that one remaining vestige of the old libraries can be found in the include files of some compilers. Specifically, a preprocessor statement such as:

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

ensures the modern (STL) version of the I/O libraries is used. On the other hand, writing:

```
#include <iostream.h>  // note the .h (and no need to include a namespace).
```

may result in the old libraries being used. Since there is no benefit from using the old libraries when writing new code, it makes sense to be careful not to inadvertently place the .h extension in C++ include statements intended to use the STL library.

13.2.2 What is a Template?

A template is, in effect, a factory for creating source code. There are two forms of templates: function templates and class templates. Function templates are factories that generate new functions on demand. Class templates generate new classes.

So what do we mean by a factory? Lets consider the case of a class template, such as the **vector<>**. When you create a normal C++ class, you are creating source code that will be compiled. When you create a C++ class template, on the other hand, you are creating instructions that will cause the compiler to generate appropriate source code when it is needed.

In the case of the vector<>, what this means is that each time we write the declaration:

```
vector<SomeClass> myVector;
```

there is the possibility that new source code will be generated. Whether or not it will will depend upon whether or not the compiler has previously generated the code here for constructing a vector<SomeClass> object. If it has, there's no need to generate the code. If it has not, on the other hand, the new code will be created.

When using templates, its important to understand that different course code is created for each new template instance that we create. In other words:

```
vector<int> ar1;
vector<char> ar2;
vector<SomeClass> ar3;
```

will lead to three entirely different sets of new course code being generated. On the other hand, if we then declared:

```
vector<int> ar4;
```

no new source code would be required. We already generated the necessary code to construct a vector<int> object when we declared ar1.

13.2.3 Using Template Classes

Walkthrough available in PointUse.avi

Although the syntax of template classes can, at times, become a bit alarming (especially as we start to deal with templates with 4 or more parameters, such as the STL **map<>** template). You use template classes just like any other class name, however. That means, for example, if you were to create your own class, MyClass, you could inherit from a specific template type, such as:

```
 class MyClass : public vector<int>
{
        // class definition
}
```

Similarly, if you wanted to override some vector<int> member function within my class, and wanted to call the base class version (as is often the case), you would do so as shown in the following example:

```
void MyClass::clear() {
        vector<int>::clear();
        // other class specific code
}
```

Although the template classes we have encountered so far (**vector<>** and **basic_string<>**) have a single parameter, template classes with more parameters are common. Such parameters will typically be of one of two types: data type/class names or constant values. As an example, suppose that we wanted to create a template class that could be used to represent points (e.g., on a graph, in space, etc.). We might find ourselves with a template defined as follows:

```
Point<class T,class D,int Dim>
```

(FYI, we will define the actual class in Section 13.4). The parameters of the template would defined as follows:

- **T**: The type of data used to store individual coordinate values (normally int or double)
- **D**: The type of data used to return the distance between two points (normally double, since the distance between two points in space is normally a fractional value)
- **Dim**: A constant, indicating the number of coordinates per point. For example, if we were referring to points on a line, Dim would be 1. If we were referring to points on a standard X-Y graph, Dim would be 2. If we were referring to points in space, Dim would be 3 (i.e., X, Y, and Z). And so forth…

As a user of the Point<> class, you would also need to know what members are defined for the template. These are listed in Table 13.3.

Member	Description
Point(void)	Default constructor
Point(const Point<T,D,Dim> &pt)	Copy constructor
const Point<T,D,Dim> &operator=(const Point<T,D,Dim> &pt)	Assignment overload
void Copy(Point<T,D,Dim> &pt)	Copy helper function
T &operator[](int i) T operator[](int i) const	Two [] overloads that return a specific coordinate from the point (e.g., the X coordinate=0, Y coordinate==1, etc.)
int Dimension() const	Returns number of coordinates per point (i.e., value of parameter Dim)
T Coord(int i) const	Returns value of specified coordinate (i.e., same as second [] overload)
D DistanceTo(const Point<T,D,Dim> &pt) const	Returns distance between object and its argument
void Display() const	Displays Point<> object to cout, using (X,Y,…) format

Table 13.3: Members of the Point<T,D,Dim> Template

In addition to the class members, there is one template function and one template operator overload defined:

```
template<class T,class D,int Dim>
D Distance(const Point<T,D,Dim> &pt1,const Point<T,D,Dim> &pt2)
```

and

```
template<class T,class D,int Dim>
ostream &operator<<(ostream &out,const Point<T,D,Dim> &pt)
```

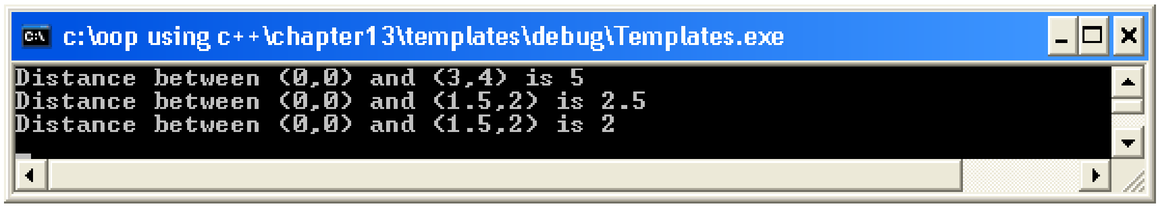
These return the distance between two points and overload the << operator for displaying a point, respectively.

A simple example of the use of the Point<> template class is presented in Example 13.2.

Example 13.2: Demonstration of the Point<> template class

```
void PointDemo()
{
	Point<int,double,2> pt1,pt2;
	pt1[0]=0;
	pt1[1]=0;
	pt2[0]=3;
	pt2[1]=4;
	cout << "Distance between " << pt1 << " and " << pt2 <<
	      " is " << Distance(pt1,pt2) << endl;
	Point<double,double,2> pt3,pt4;
	pt3[0]=0;
	pt3[1]=0;
	pt4[0]=1.5;
	pt4[1]=2.0;
	cout << "Distance between " << pt3 << " and " << pt4 <<
	      " is " << Distance(pt3,pt4) << endl;
	Point<double,int,2> pt5,pt6;
	pt5[0]=0;
	pt5[1]=0;
	pt6[0]=1.5;
	pt6[1]=2.0;
	cout << "Distance between " << pt5 << " and " << pt6 <<
	      " is " << Distance(pt5,pt6) << endl;
}
```

The output from running the demonstration, which creates three pairs of 2-dimensional points and displays the distance between the points in each pair, is presented in Figure 13.4.



```
Distance between (0,0) and (3,4) is 5
Distance between (0,0) and (1.5,2) is 2.5
Distance between (0,0) and (1.5,2) is 2
```

Figure 13.4: Output of PointDemo() function

Looking at the code, it is clear that once a template object is created, we can use it just the way we use any other object. In the first pair we create (Point<int,double,2>), we're creating a 2-dimensional point (3rd parameter) , storing individual coordinate values as integers (1st parameter) and returning distance values as doubles (2nd parameter). Once created, we use the [] overload to assign values to the coordinates, i.e.,

pt2[1]=4;

We then use the output (insertion) overload to display the point values, i.e.,

cout << "Distance between " << pt1 << " and " << pt2 and the call the Distance() template function to compute the distance between the two points, i.e.,

```
Distance(pt1,pt2)
```

which is sent to cout.

> *Test your understanding:* If we had wanted to call the member function Distance() instead of calling the template function, what would the two possible versions of the call look like?

In the second pair of points, pt3 and pt4, we used double instead of integers as our underlying storage for coordinates. Otherwise the template behaves identically. It is important to note, however, that Point<int,double,2> objects are entirely different from Point<double,double,2> objects. Thus, if we tried to assign:

```
pt1=pt3;
```

the compiler would object strenuously.

The third pair of points, pt5 and pt6, are somewhat more problematic. For these points, we've chosen to return distances as integers, and store coordinates as doubles (the reverse of what would be normal). A practical consequence is the truncation of the distance from 2.5 to 2, as is seen in the last line of Figure 13.4.

Another important thing to know about templates is that errors in templates or template are not discovered until the compiler "factory" generates code that is patently unworkable. For example, the following code:

```
Point<string,string,2> pt7,pt8;
pt7[0]="0";
pt7[1]="0";
pt8[0]="1.5";
pt8[1]="2.0";
cout << "Distance between " << pt7 << " and " << pt8 ;
```

compiles and run perfectly, as there is nothing in the template that prevents **string** objects from being used to set coordinates and display them. If, however, we add the line:

```
cout <<      " is " << Distance(pt7,pt8) << endl;
```

the compiler objects with considerable passion, generating 9 errors—all of which refer to code deep within the template itself. The underlying problem here is that the Distance() member simply does not have a clue as to how to translate the mathematical operations involved in computing distance (involving, among other things, taking a square root) when the data passed in is in **string** form. As a user of templates, you need to be aware that when such errors occur, it probably means you've provided parameter values that are inconsistent with how the template is defined. When using collection class templates, a this frequently comes up when you define classes and neglect to create assignment operator and copy constructor overloads.

13.2.4: Iterators

When using collection template classes, you will frequently find that associated data types are automatically defined along with the collection itself. When the **vector<>** template was presented, we saw the most important of these: the iterator.

The design philosophy behind iterators is as follows. Regardless of a collection's shape properties, if a container holds elements in a sequence, is usually desirable to have the ability to go through the set of elements in linear fashion. To do this, STL collection classes automatically define an iterator object type whenever a template class is created. The purpose of the iterator object is to allow you to move through the collection sequentially using a common interface that applies to all collections. Iterator variables can be declared as follows:

> ***template-name***<*parameters*>::iterator *variable-name*;
> ***template-name***<*parameters*>::const_iterator *variable-name*;

In Chapter 8, when we introduced the **vector<>** template, we noted that the interface of the iterator should be very familiar to C++ programmers—it is used exactly like a pointer. In other words, if nPos is declared as an iterator:

> *nPos accesses the collection element at that position
> nPos[offset] accesses collection elements for shapes supporting random access
> *(nPos+offset) also accesses collection elements for shapes supporting random access
> nPos++ moves the iterator to the next element in the collection
> nPos-- moves to the previous element in the collection

We also noted that, two class member functions are defined for use with iterators:

- begin() returns the iterator value at the start of the collection.
- end() returns a special iterator value signifying we have incremented the last element of the collection

As noted in the STL documentation provided by Visual Studio .Net, there are five general categories of iterators provided by the STL:

- *Output iterators:* can only be used to set collection values, and must be incremented after each access (i.e., value setting operation).
- *Input iterators:* can only be used to access collection values. A given value can be accessed any number of times but can only be moved in the forward direction.
- *Forward iterators:* Can be used to set or access values any number of times, but can only be moved in the forward (i.e., ++) direction.
- *Bidirectional iterators:* Can be used to set or access values any number of times, and can be moved in the forward (i.e., ++) or backward (--) direction.
- *Random-access iterators:* Can be used to set or access values any number of times, can only be moved in the forward (i.e., ++) or backward (--) direction, and can also be used to access any element in the collection directly, using the [] or *() notation that is also used in pointers arithmetic.

When looking at the STL documentation for a particular collection class, you can determine what iterators are available. For example, a **vector<>** collection provides the most powerful of all iterators, the random access iterator. The **list<>** template class, on the other hand, only supplies bi-directional iterators. In other words, you can't use [] notation to access elements in a list.

We will continue to consider iterators as we introduce new collection shapes in future chapters.

13.2.5: Templates Built on Other Shapes

It is not unusual to find template classes that are actually constructed using other shapes. (An end-of-chapter question, for example, presented the problem of building our Point<> template using a **vector<>** to store the underlying coordinates). Understanding how this process works can be critical if you are using template collections in your applications.

A good example of this process can be found in the STL **stack<>** template. A stack is a commonly used shape that implements last-in, first-out (FIFO) access. The STL **stack<>** template class implements small set of member functions, the most critical of which are:

- **void push(***data-type***):** Places an element on the top of the stack—making it the next element to be processed.
- **void pop():** Removes the element on the top of the stack
- *data-type* **&top():** Provides access to the data element on the top of the stack (both const and non-const versions of the function are provided)
- **bool empty() const:** Returns **true** if the stack is empty
- **int size() const:** Returns the number of objects on the stack

Unlike the **vector<>**, **stack<>** access is specifically intended to be limited to the top of the collection, therefore no iterator is defined.

Another characteristic that is rather unique about the **stack<>** is that it is always constructed on top of another container, which it uses to hold the elements. The default container it uses is a **deque<>** (which is almost identical to a **vector<>** except it allows fast insertion at the front, as well as the back), but any container that supports push_back() and pop_back() is acceptable (including **vector<>** and **list<>**).
The practical implication of this is that when you construct a stack, you may add a second parameter to your template specification. For example:

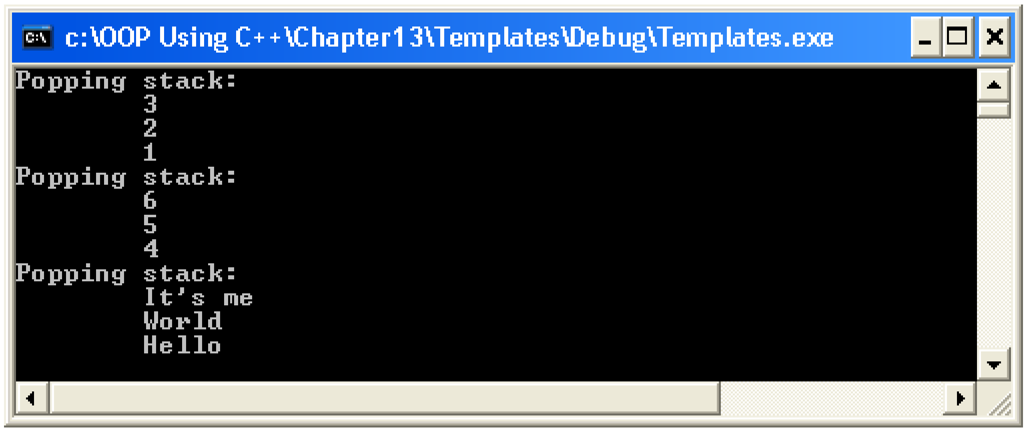
```
stack<int> s1; // s1 is a stack that uses a deque<> for underlying implementation
stack<int,vector<int> > s2;    // s2 is a stack that uses a vector<>
stack<int,list<int> > s3;                    // s3 is a stack that uses a list<>
```

As far as the programmer is concerned, the collection on which a **stack<>** is implemented is virtually irrelevant. The only impact will be on performance (and such performance impacts are likely to be small). The interface will be unchanged. This can be seen in Example 13.3, where three **stack<>** objects are created, each using a different underlying collection.

Example 13.3: Alternative stack<> Implementations

```
void StackTest()
{
	// Note: space > > can be significant when using template
	stack<int,vector<int> > st1;
	st1.push(1);
	st1.push(2);
	st1.push(3);
	cout << "Popping stack:" <<endl;
	while(!st1.empty())
	{
		cout << "\t" << st1.top() << endl;
		st1.pop();
	}
	stack<int,list<int> > st2;
	st2.push(4);
	st2.push(5);
	st2.push(6);
	cout << "Popping stack:" <<endl;
	while(!st2.empty())
	{
		cout << "\t" << st2.top() << endl;
		st2.pop();
	}
	// Default shape is deque
	stack<string> st3;
	st3.push("Hello");
	st3.push("World");
	st3.push("It\'s me");
	cout << "Popping stack:" <<endl;
	while(!st3.empty())
	{
		cout << "\t" << st3.top() << endl;
		st3.pop();
	}
}
```

The output from running StackTest() is presented in Figure 13.5. The LIFO nature of the stack can be readily observed—with the elements being popped off in the reverse order that they were pushed on.



```
Popping stack:
        3
        2
        1
Popping stack:
        6
        5
        4
Popping stack:
        It's me
        World
        Hello
```

Figure 13.5: Output of StackTest() function

There are a number of STL templates that are based on other shapes, such as **stack<>**, **queue<>** (like a stack, but first-in, first-out), and **set<>** (which manages a collection of keyed items). Templates based on other shapes are also found in other important template libraries, such as the one provided by the MFC.

13.2 Section Questions

1. What is the relationship of the STL to the original C++ standard library?
2. Why is it sometimes hard to tell if an object has been created from a template class?
3. Why are templates sometimes described as factories?
4. Do you need to know how to create templates in order to use them?
5. Why does it sometimes make sense to separate type and argument specifications in a template definition?
6. Why don't you need to know how a list<> template works when using a stack<> template based on a list<>?

13.3: Function Templates

Walkthrough available in TempFunc.avi

Function templates are easier to define and use than class templates, but they tend to be less powerful in terms of what they can accomplish. For this reason, our treatment of them will be brief.

13.3.1: Defining a Function Template

The mechanism for defining any template, whether it be function or class, is to place a **template<>** specifier immediately before the definition. For example, suppose we had a function, LessThan() that was defined for some class A and B. Its definition might look something like the following:

```
bool LessThan(A &v1,B &v2)
{
        return v1<(A)v2;
}
```

This function works by typecasting its second argument to the first type and will—of course—only work if a typecast is available from B to A.

If we wanted to turn this into a template function, we would just add the template<> keyword to the top line, leading to the following:

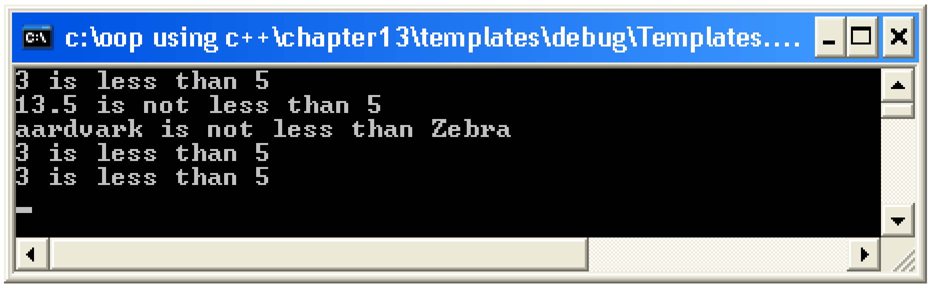
```
template<class A,class B>
bool LessThan(A &v1,B &v2)
{
        return v1<(A)v2;
}
```

Defined in this way, all of the following calls shown in Example 13.3 are legal and automatically call our function. Even string objects can be used (a5 and a6), since the < operator is overloaded for the STL **string** object.

Example 13.3: Using the LessThan() function

```
void TemplTest()
{
	int a1=3,a2=5;
	cout << a1 << ((LessThan(a1,a2)) ? " is " : " is not ")
		<< "less than " << a2 << endl;
	double a3=13.5,a4=5;
	cout << a3 << ((LessThan(a3,a4)) ? " is " : " is not ")
		<< "less than " << a4 << endl;
	string a5="aardvark",a6="Zebra";
	cout << a5 << ((LessThan(a5,a6)) ? " is " : " is not ")
		<< "less than " << a6 << endl;
	cout << a1 << ((LessThan(a1,a4)) ? " is " : " is not ")
		<< "less than " << a4 << endl;
	unsigned int a7=5;
	cout << a1 << ((LessThan(a1,a7)) ? " is " : " is not ")
		<< "less than " << a7 << endl;
}
```

The output from running the function is presented in Figure 13.5.



```
3 is less than 5
13.5 is not less than 5
aardvark is not less than Zebra
3 is less than 5
3 is less than 5
```

Example 13.5: Output from TemplDemo function

Of particular interest in the example are the last two calls, where we mix data types. Thus, when the compiler sees:

```
	LessThan(a1,a4)
```

it will generate new source code:

```
	bool LessThan(int &a1,double &a2)
	{
		return a1<(int)a2;
	}
```

When it sees:

```
	LessThan(a1,a7)
```

it will generate new source code:

```
bool LessThan(int &a1,unsigned &a2)
{
        return a1<(int)a2;
}
```

Neither one of these produces a warning—the reason being that in our function we typecast the second parameter to the first parameter, which tells the compiler we don't want to hear from it.

Operator overloads can also be accomplished with templates. For example, in our Point<> template, we overloaded output as follows:

```
template<class T,class D,int Dim>
ostream &operator<<(ostream &out,const Point<T,D,Dim> &pt) {
    pt.Display();
    return out;
}
```

This basically states that any time we see a three parameter Point<> object being sent to cout, we should try calling its Display() member function.

Template functions, like template classes, should normally be defined in header files. The reason for this is that the compiler needs to have access to the template definition in order to create the function. The one exception we sometimes encounter is in the case of explicit template instantiations.

13.3.2: In Depth: Instantiating Function Templates

Defining and using function templates is straightforward. However, sometimes a general templated doesn't do exactly what we'd like. In example 13.5, for example, we see that:

```
aardvark is not less that Zebra
```

Since C++ does everything case sensitive, that is literally true. Suppose, however, we prefer case-insensitive comparisons when calling our LessThan() function? This can be accomplished by creating an instantiated template function. This is accomplished by defining a specific function as follows:

```
template<>
bool LessThan(string a1,string a2)
{
                return (stricmp(a1.c_str(),a2.c_str())<0);
}
```

This function will then be called—in preference to our general version—whenever two string objects are supplied as arguments to our LessThan() function.

When instantiated functions are used, a prototype, such as:

```
template<>
bool LessThan(string &a1,string &a2);
```

can be placed in the header (.h) file, and the function itself can be defined in the .cpp file. There are some code deployment advantages to doing do, but nothing we need to be particularly concerned about at this point.

13.3 Section Questions

1. What it the relationship between function overloading and function templates?

2. Why are templates normally contained entirely in the header file?

3. What would happen if we did not typecast the B argument to the A type in our LessThan() function?

4. Does it matter if a template function definition matches an overload of an existing (non-template) function?

5. Why don't we need the <> delimiters when calling a template function (the way we do when we're using a template class)?

13.4: A Simple Template Class

Walkthrough available in Point.avi

The Point<> template class, introduced in Section 13.2.3, provides a useful illustration of how templates can be created. We will look at the basic design of the class, then examine how to create a special case.

13.4.1: Defining a Point<> Class Template

Defining a template class is normally a three-step process:

- Use or define a working class that you want to turn into a template class
- Identify the key elements you want to turn into parameters
- Use the template<> construct to create the definition.

In the case of our Point<> template, the 2nd decision has already been made, but we'll walk through the process anyway.

We begin with a simple class that manages a 2-coordinate (i.e., x,y) point, such as might appear on a 2-D graph (or on a computer screen, when you're using drawing functions). The class, PointInt, is very simple and manages a 2-integer array. Its declaration is presented in Example 13.4.

Example 13.4: PointInt class

```
class PointInt
{
public:
      PointInt(void) {}
      PointInt(const PointInt &pt) {
            Copy(pt);
      }
      ~PointInt(void) {
      }
      const PointInt &operator=(const PointInt &pt) {
            if (this==&pt) return *this;
            Copy(pt);
            return *this;
      }
      void Copy(const PointInt &pt) {
            int i;
            for(i=0;i<2;i++) {
                  (*this)[i]=pt[i];
            }
      }
      int &operator[](int i){
            assert(i>=0 && i<2);
            return coord[i];
      }
      int operator[](int i) const {
            assert(i>=0 && i<2);
            return coord[i];
      }
      int Dimension() const {return 2;}
      int Coord(int i) const {
            return (*this)[i];
      }
      double DistanceTo(const PointInt &pt) const;
      void Display() const;
protected:
      int coord[2];
};
```

The class, which is implemented almost entirely with inline member functions, provides assignment and bracket operator overloads, a copy constructor and a variety of accessor functions. The two non-inline member functions, DistanceTo() and Display() are presented in Example 13.5. The DistanceTo() function uses the Pythagorean theorm to compute the distance between the points, i.e.:

Distance between (x1,y1) and (x2,y2) is: sqrt($(x2-x1)^2 + (y2-y1)^2$)

Example 13.5: PointInt DistanceTo() and Display() members

```
double PointInt::DistanceTo(const PointInt &pt) const {
	double dTotal=0;
	int i;
	for(i=0;i<2;i++) {
		dTotal+=(double)((pt[i]-Coord(i))*(pt[i]-Coord(i)));
	}
	return (double)sqrt((double)dTotal);
}
void PointInt::Display() const
{
	cout << "(";
	for(int i=0;i<2;i++) {
		cout << Coord(i);
		if (i<2-1) cout << ",";
	}
	cout << ")";
}
```

Once we verify out prototype class is working, we go about thinking about parameters we may want to change. Obviously, the way we store individual coordinates is one—we'll call it T (for data type). Another type that crops up in our analysis is the double that is returned by the distance function. While we certainly could return an integer for this value, it might make sense to give the user/programmer control over that type—we'll call it D. Finally, we want to be able to control the number of coordinate values per point. In our example there are 2, but we might end up wanting to deal in 3-D. We might also decide we wanted to use this class to track the stock prices of 112 companies at various "points" in time, in which case we'd need 112. For this parameter, however, we don't need to be able to change the data type. Instead, we need to be able to specify a constant value—an integer—that specifies the number of dimensions in our point. So we'll call it Dim.

Having performed this analysis, we know our template specifier will appear as follows:

	template<class T,class D,int Dim>

The class signifies that T will be class name or data type, as is the case for D. The int specifies that Dim will be an integer constant. In making such a definition, you should also be aware that	Point<int,double,9> and Point<int,double,10> are as different as night and day as far as the compiler in concerned. The slightest difference in the way the template is specified causes an entirely new class to be created.

Once we've done the background work, the actual creation of the template class is fairly mechanical. It proceeds as follows:

- You place the template specifier before the class name

- You go through the class and all member function bodies and replace any references to data type with T
- You go through the class and all member function bodies and replace any references to distance values (in this case, double values) with D
- You look for anywhere that array dimensions are specified (in this case, you look for the number 2) and replace them with Dim.
- You move the member functions into the header file, placing the template specifier above each (the same as we did for functions).
- Anywhere you see the original class name (except the constructor and destructor names), replace it with the templated name. In other words, PointInt is transformed to Point<T,D,Dim>.

The results of this are presented in Example 13.6.

Example 13.6: Point<> Template Class

```
template<class T,class D,int Dim>
class Point
{
public:
        Point(void) {}
        Point(const Point<T,D,Dim> &pt) {
                Copy(pt);
        }
        ~Point(void) {
        }
        const Point<T,D,Dim> &operator=(const Point<T,D,Dim> &pt) {
                if (this==&pt) return *this;
                Copy(pt);
                return *this;
        }
        void Copy(const Point<T,D,Dim> &pt) {
                int i;
                for(i=0;i<Dim;i++) {
                        (*this)[i]=pt[i];
                }
        }
        T &operator[](int i){
                assert(i>=0 && i<Dim);
                return coord[i];
        }
        T operator[](int i) const {
                assert(i>=0 && i<Dim);
                return coord[i];
        }
        int Dimension() const {return Dim;}
        T Coord(int i) const {
                return (*this)[i];
        }
        D DistanceTo(const Point<T,D,Dim> &pt) const;
        void Display() const;
protected:
        T coord[Dim];
};

// Member functions moved in from .cpp file
template<class T,class D,int Dim>
D Point<T,D,Dim>::DistanceTo(const Point<T,D,Dim> &pt) const {
        D dTotal=0;
        int i;
        for(i=0;i<Dim;i++) {
                dTotal+=(D)((pt[i]-Coord(i))*(pt[i]-Coord(i)));
        }
        return (D)sqrt((double)dTotal);
}
template<class T,class D,int Dim>
void Point<T,D,Dim>::Display() const
{
        cout << "(";
        for(int i=0;i<Dim;i++) {
                cout << Coord(i);
                if (i<Dim-1) cout << ",";
        }
        cout << ")";
}
```

13.4.2: In Depth: Instantiating a Class Template or Member Function

As was the case with functions, we can create specific instances of class templates or member functions that handle unusual cases differently. For example, if we wanted to handle the peculiar case of coordinates presented as **string** values for a two-coordinate point (that led to a well-deserved compiler error in Section 13.2.3), we could define a new class:

```
template<>
class Point<string,string,2> {
        // class definition
}
```

In this class, we could choose exactly how to handle **string** objects. The obvious drawback of this approach is that we need to redefine the entire class. An alternative approach is to specialize a single member function. An example of how this can be done is shown in Example 13.7. This particular function converts the two points to double values (using atof), computes the distance, and then uses sprintf to return to text display. A string is then created to return the text.

Example 13.7: Specializing the Distance Member Function

```
string Point<string,string,2>::DistanceTo(const Point<string,string,2> &pt) const
{
	double dv1,dv2,dTotal=0;
	dv1=atof(Coord(0).c_str())-atof(pt[0].c_str());
	dv1=dv1*dv1;
	dv2=atof(Coord(1).c_str())-atof(pt[1].c_str());
	dv2=dv2*dv2;
	dTotal=sqrt(dv1+dv2);
	char buf[80];
	sprintf(buf,"%.5lf",dTotal);
	return string(buf);
}
```

13.5 GArray Walkthrough

Walkthrough available in GArray.avi

Even if you plan to start out by leaving template creation to "the professionals", it is useful to walk through the process of creating a class template. There are many subtleties to "profession-grade" template definition (e.g., defining embedded iterator classes), all of which are far beyond the scope of this textbook. Nonetheless, it can be relatively easy to create a simple and powerful template if you have a working class to start from.

We will begin with the PtrArray class (Chapter 7, Section 8.4.2), which is a fairly ideal candidate for a template class, for three reasons:

- Collections, such as arrays, are the type of object most commonly templated.
- We already know, from the Chapter 7 end-of-chapter questions (7.3 and 7.4) that it is easy to modify the PtrArray class to hold other types of data.
- Its interface, being similar to the **vector<>**, requires little introduction.

13.5.1: Template Design

The first decision associated with designing a template is the choice of parameters for the template. Naturally, one thing we want to control is the type of data being stored in the array—so that will have to be a parameter (and, for the vector template class, it is the only parameter). Often, it is useful to specify a second parameter which identifies how data will passed into certain functions, such as Add(), SetAt(), and InsertAt(). As we know, when you pass objects into a function, a copy is made (unless you pass it as a pointer or a reference). Making such copies, however, might add greatly to the overhead associated with using the class when large objects are stored. For this reason, it is often nice to be able to specify that certain functions take references as values. We'll return to this shortly, when we look at the actual code.

Based on this, we'll call our template class GArray<T,A>, where T is the data type being stored, and A is how it is passed into functions as an argument. (Not coincidentally, this will make out resulting template quite similar to the CArray<T,A> template provided with the MFC).

Our GArray<T,A> template will allow us to create a wide range of type-safe arrays. For example:

```
GArray<int,int> ar1;
```

would declare an array of integers named a1.

```
GArray<void*,void*> ar2;
```

would create an array void pointers—effectively identical to the PtrArray class.

```
GArray<employee,employee&> ar3;
```

would create an array of employee objects. In this template, where employee values were passed into key functions, they'd be passed in as references. Finally:

```
GArray<GString,const char*> ar4;
```

would create an array of GString objects. When we passed GString objects into function, such as Add(), however, the argument would be a const char *, allowing us write code such as:

```
ar4.Add("Hello, World!");
```

This mixing of T and A data types is made possible by virtue of the fact that our GString class has build in typecast and constructor functions that provide for const char * arguments. On the other hand, if we made the following declaration:

```
GArray<GString,int> ar5;
ar5.Add("Hello");
```

we would get the following error (on the Add() line) when we compiled:

```
C2664: 'GArray<class GString,int>::Add' : cannot convert parameter 1 from 'char [6]'
to 'int'
    This conversion requires a reinterpret_cast, a C-style cast or function-style cast
```

This tells us that it cannot figure out how to turn the string into an argument. On the other hand, if we called it:

```
ar5.Add(17);
```

we'd get something along the lines of the following lengthy error message:

```
C2679: binary '=' : no operator found which takes a right-hand operand of type
'int' (or there is no acceptable conversion)
      c:\Program Files\Microsoft Visual Studio .NET\Vc7\include\xlocale(582) :
while compiling class-template member function 'void
GArray<T,A>::SetAt(int,int)'
      with
      [
         T=GString,
         A=int
      ]
      c:\Program Files\Microsoft Visual Studio .NET\Vc7\include\xmemory(111) :
while compiling class-template member function 'GArray<T,A>::GArray(void)'
      with
      [
         T=GString,
         A=int
      ]
```

```
        c:\Introduction to C++\Chapter16\GArray\GArray.cpp(67) : see reference to
class template instantiation 'GArray<T,A>' being compiled
        with
        [
            T=GString,
            A=int
        ]
```

What this tells us that it can't figure out how to convert between GString and int within the SetAt() function (which was called within the Add() function). This demonstrates an important fact about templates: the parameters you supply *must* make sense.

13.5.2 GArray<> Construction

If you have a working class as a model for your template, constructing the template version can be relatively straightforward. There are three basic steps:

- You place a template declaration in front of your class declaration and every member function in the .cpp file. This declaration is of the form template<*parameter-type-list*>. For example, in our illustration:

  ```
              template<class T,class A>
  ```

  which specifies that both the parameters T and A are going to be type names (they don't have to be actual class names).
- Everywhere that you have you old class name inside the class definition itself, replace it with the *template-name*. Outside of the definition (including member function definitions in the .cpp file), replace it with the full template specifier *template-name*<*parameter-list*>. In out example, this would be:

  ```
  GArray<A,T>
  ```

- Go through your class and member function definitions and replace data types with the appropriate parameter name. In our example, you could do a global search and replace of "void *" with T (since it will be holding T type objects instead of the original void * objects), then make selective replacements of function arguments with A.
- Move all member functions in the .cpp file into the .h file, under the class definition. This is required because the compiler needs to access the full function definitions every time a template member is called.

At this point, you can start testing your template by declaring some objects. Normally, this will surface some modifications of things you didn't think about. (The GArray<>

template didn't require any, but this is pretty unusual). The declaration and a few are presented in Example 13.8.

Example 13.8: GArray<> declaration and selected members

```
template<class T,class A>
class GArray
{
public:
      GArray();
      GArray(const GArray &ar);
      ~GArray();
      const GArray &operator=(const GArray &ar);
      void Copy(const GArray &ar);
      T &operator[](int);
      T &operator[](int) const;
      T GetAt(int i) const;
      void SetAt(int i,A pData);
      void Add(A pData);
      void InsertAt(int i,A pData);
      void SetSize(int i);
      int GetSize() const;
      void RemoveAt(int i);
      void RemoveAll();
protected:
      int m_nCount;
      int m_nSize;
      T *m_pData;
      void ReallocArray(int nNewSize);
};


template<class T,class A>
GArray<T,A>::~GArray() {
      RemoveAll();
}
// Returns the pointer value at position i, asserting i is legal
template<class T,class A>
T GArray<T,A>::GetAt(int i) const
{
//    return m_pData[i];
      return (*this)[i];
}
// Sets the pointer value at position i to pData, asserting i is legal
template<class T,class A>
void GArray<T,A>::SetAt(int i,A pData)
{
      (*this)[i]=pData;
}

template<class T,class A>
const GArray<T,A> &GArray<T,A>::operator=(const GArray &ar) {
      if (&ar==this) return *this;
      Copy(ar);
      return *this;
}
```

These definitions are identical to those in the PtrArray class, except for the changes we discussed. We also see the A parameter was used in the three functions used to add data to the array (the Add(), SetAt() and GetAt() functions), while the T parameter was used everywhere else that void * has originally occurred.

Examining the template, we can conclude an additional important fact. Any object we wanted to use this template for would need to have:

- An assignment overload, since the SetAt() function uses assignment to set array elements
- A copy constructor, since GetAt() member returns a copy of the object.

It the classes being used in the template do not meet the requirements of the template, compiler error messages (such as the ones earlier in the section) will be generated when template members are called. The compiler will not generate error messages on functions that aren't used.

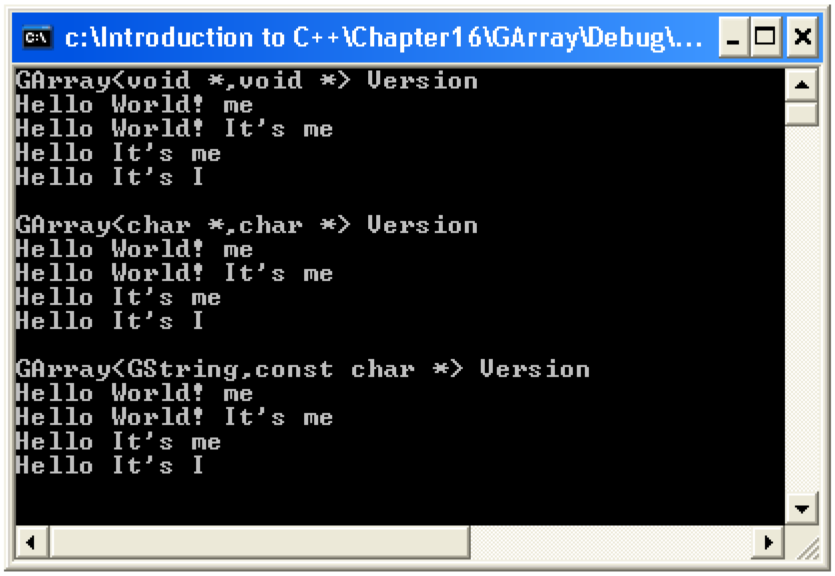
To demonstrate the GArray<> template, we used three different versions of the same code used to demonstrate the PtrArray class. The were identical except that each version used a different template:

```
GArray<void*,void*>
GArray<char*,char*>
GArray<GString,const char *>
```

The first function is presented in Example 13.9. The results of running all three functions are shown in Figure 13.2.

Example 13.9: GArray<> demonstration function

```
void GArrayDemo1()
{
	int i;
	GArray<void*,void*> ar;
	ar.Add("Hello");
	ar.Add("World!");
	ar.Add("me");
	for(i=0;i<ar.GetSize();i++) {
		cout << (const char *)ar[i] << " ";
	}
	cout << endl;
	ar.InsertAt(2,"It\'s");
	for(i=0;i<ar.GetSize();i++) {
		cout << (const char *)ar[i] << " ";
	}
	cout << endl;
	ar.RemoveAt(1);
	for(i=0;i<ar.GetSize();i++) {
		cout << (const char *)ar[i] << " ";
	}
	cout << endl;
	ar.SetAt(2,"I");
	for(i=0;i<ar.GetSize();i++) {
		cout << (const char *)ar[i] << " ";
	}
	cout << endl;
}
```



```
GArray<void *,void *> Version
Hello World! me
Hello World! It's me
Hello It's me
Hello It's I

GArray<char *,char *> Version
Hello World! me
Hello World! It's me
Hello It's me
Hello It's I

GArray<GString,const char *> Version
Hello World! me
Hello World! It's me
Hello It's me
Hello It's I
```

Figure 13.1: Output from three versions of GArrayDemo() function

13.6: Template Lab Exercises

In this section, we redevelop the KeyArray class, originally developed in Chapter 10, Section 10.4.2, as a template. The resulting class is dramatically superior to the original version.

13.6.1: Objectives

When the KeyArray class was developed in Chapter 10, it provided us with a way to manage sorted arrays of different types of objects. Because of problems with serialization, however, you needed to inherit from the class in order to use it. You also needed an unusual function, NewCopy(), to be implemented in the class you were collecting (e.g., in Employee objects for an EmployeeArray).

So as not to change the class entirely, you may assume that the objects being collected:

- Have a member function Key() that returns a string that will be unique in the collection.
- Have Load(), Save() and Display() members implemented (just like the unit classes in the Chapter 10 lab exercise).
- Have an assignment operator and copy constructor overload

You should not assume a NewCopy() function exists for the objects in the collection, or that those objects necessarily inherit from CKey.

13.6.2: Specifications

Your KeyArray<> should support the same members as the original KeyArray, including:

- operator[]: integer and string versions (for lookup)
- Find() const: integer and string versions
- int Add(): One or more versions, with sensible arguments
- void Remove():One or more versions, with sensible arguments
- void RemoveAt(int nVal)
- int Size() const
- void RemoveAll()
- void Load(fstream &strm)
- void Save(fstream &strm) const
- void Display() const

Your template should be of the form:

KeyArray<class T,class A>

where T is the data type being used to store data (which should no longer have to be a pointer) and A is the data type used to pass in arguments.

13.6.3: Instructions

You will be performing the same basic steps completed for the Point<> template (Section 13.4) and for the GArray class (Section 13.5). What you will need to do is:

- Get a copy of the KeyArray class and study it
- Replace references to the CKey object with your T parameter
- Replace appropriate member function arguments with the A parameter
- Write a main() function to test the class, making sure you try it with a number of objects—some inheriting from CKey and some not inheriting. Remember, you don't always discover template collection problems with a single substitution

16.7: Review and Questions

16.7.1: Review

Collections are a critical part of OOP. To describe a collection, we refer to its "shape", which is a set of properties that relate to its performance across a variety of collection access and modification activities. The STL supports a variety of collection shapes, each of which has its own performance properties. These include:

- ***Array*** shapes, such as **vector<>**, which provide random access but relatively slow insertion/deletion except at the end.
- ***Linked List*** shapes, such as **list<>**, that are efficient only for sequential access but which provide for highly efficient insertion/deletion anywhere in the collection.
- ***Lookup Table*** shapes, such as **map<>**, that provide very efficient content-based lookup for keyed data (exact matches only) in a collection.
- ***Stack and Queue*** shapes, such as the **stack<>** and **queue<>**, that limit collection access to LIFO (last-in first-out) and FIFO (first-in, first-out) respectively, and are normally implemented using another collection shape, such as a **list<>** or **vector<>**.

In addition, some common collection shapes tend to be user-implemented, such as the ***sorted array*** and ***tree***.

The Standard Template Library (STL)—which began to be widely supported by compilers in the mid-1990s and became part of the ISO standard in 1998—has largely replaced the original C++ class libraries developed in the early 1980s. When STL classes are included, the .h in the #include statement is typically omitted. This is important to be aware of, as adding the .h can lead to the wrong header being accessed. For example:

```
#include <string>
```

accesses the STL **string** implementation, whereas:

```
#include <string.h>
```

accesses the headers that support the standard C NUL terminated string functions.

As the name suggests, the STL is based around the C++ template feature. A template is language construction that allows code for classes and functions to be generated automatically, based on supplied parameters. Template parameters are provided as a comma separated list between < and > symbols, and may be either data types or constants. The serve to specify the way the templated code is generated. For example, the declaration:

```
vector<int> ar1;
```

serves to declare a dynamic array of integers, generating appropriate source code. On the other hand:

```
vector<string> ar2;
```

declares an array of string objects, and would then generate a different set of source code. As part of compilation and linking, duplicated code generated by template usage is removed—one of the major challenges of implementing development tools that support templates.

Template parameters can be of two types: unspecified data types (indicated with the class keyword) and specified constants, indicated with a data type. For example:

```
Point<class T,class D,int Dim>
```

identifies a template with two class/data-types and an integer constant. Thus, we could use the template to create an object such as:

```
Point<int,double,3> pt;
```

It should be noted that any variation in template parameters renders objects entirely different data types. In other words, a Point<int,double,3> object is entirely distinct from a Point<int,double,2> object.

Some template collections, such as the STL **stack<>** template, are built on top of other collections. The collection to be used can then be specified as a template parameter. For example:

```
stack<int,vector<int> >
```

implements an integer stack using a vector, whereas:

```
stack<int,list<int> >
```

implements an integer stack using a list. As a general rule, the underlying collection is chosen mainly for performance reasons, and does not change the template class interface.

Each STL collection holding a sequence of objects supports a template-specific iterator data type. Once an iterator has been declared, it supports sequential access to collection elements using an interface closely reminiscent of standard pointer arithmetic. For example:

*nPos accesses the collection element at that position
nPos[offset] accesses collection elements for shapes supporting random access
*(nPos+offset) also accesses collection elements for shapes supporting random access
nPos++ moves the iterator to the next element in the collection
nPos-- moves to the previous element in the collection

In addition, two class member functions are defined for use with iterators:

- begin() returns the iterator value at the start of the collection.
- end() returns a special iterator value signifying we have incremented the last element of the collection

The two primary advantages of using iterators are: 1) the programmer can use the same approach to gain access to elements in different shapes, and 2) using iterators can rfeduce the number of code changes required if the programmer changes an object's underlying collection shape.

Either functions or classes may be templated. To define a template, you place the template<> specifier in front of your definition. For example:

```
template<parameter-list>
rettype your-function-name(arg-list) {
    your-function-body
}
```

would define a function template. Similarly:

```
template<parameter-list>
class class-name {
    member-definitions
};
```

would define a class template.

During the compilation process, template inconsistencies are detected only as template class objects are declared and used—and the types of errors generated can vary based on the parameters supplied. For example, a template may only work with classes that support assignment overload.

Occasionally, and as a rather advanced programming activity, it is desirable to obtain special behaviors for specific parameter combinations in a template. A template using strings, for example, might want to implement case-insensitive comparison. Such behaviors are generally accomplished by specifying/instantiating a function, class or member function. Such instantiation uses the **template<>** specifier with no arguments, then places the specific classes/constants in the template argument list. For example:

```
template<>
string Point<string,string,2>::DistanceTo(const Point<string,string,2> &pt) const
{
    // function body
}
```

would override the general DistanceTo() member of the Point<> template class when the particular parameter combination Point<string,string,2> was encountered.

16.7.2: Glossary

Active Template Library (ATL) – A collection of template-based classes, developed by Microsoft, primarily aimed for developing components and applications in a Windows-based environment..
Bounds Checking – Ensuring that array positions are not accessed that are outside of the valid coefficients of the array.
Collection – A set of objects that is managed by some other object, the collection object.
Collection shape – The set of performance properties for collection access and modification that determine a collection's most appropriate uses.
deque<> – An STL collection template with properties similar to that of a vector<> except that fast insertions/deletions at the front, and not just the end, are supported.
Doubly Linked List – A linked list shape that supports processing from front-to-back or back-to-front.
Dynamic Resizing – The ability of a collection to change its size, as necessary, to accommodate the addition and removal of elements.
First-In, First-Out (FIFO) – Collection access in which elements are processed in the order in which they were added (see **queue<>**).
Hashing – An algorithm supporting keyed access to data that uses a coding function to position elements for fast access.
Iterator – An object that can be uses to traverse the elements of a collection using a pointer-style interface. Each STL collection template defines an iterator data type as part of its implementation, and a common interface is used for all iterators.
Last-In, First-Out (LIFO) – Collection access in which elements are added most recently are processed before previously added elements (see **stack<>**).
list<> – An STL template class that supports a linked list.
Linked List – A collection shape that supports sequential processing and rapid insertion and deletion of internal elements. Conceptually, it can be viewed as a chain of elements linked together by pointers..
Lookup table – A common term used to describe a map, normally implemented using a hashing algorithm.

map<> – An STL template class that supports keyed lookup (i.e., a lookup table shape) that is collection size independent.
Node – A term commonly used to refer to an element contained in a tree shape.
queue<> – An STL template class that implements waiting line (FIFO) access to collection members, built on top of some other underlying shape.
Random access – The ability to access elements in a collection without traversing other elements, often implemented with the [] operator.
Serialization – A term frequently used to refer to loading and saving objects in binary form.
set<> – An STL collection that manages a collection of unique keyed objects, built on top of another underlying shape.
Shape – see collection shape.
Singly Linked List – A linked list shape that supports processing from front to back only.
stack<> – An STL template class that implements stack (LIFO) access to collection members, built on top of some other underlying shape.
Standard Template Library (STL) – A library of template-based objects that began to replace the original C++ standard libraries in the mid-1990s and became part of the ISO standard in 1998.
string – An STL template class that implements an encapsulated string object.
Template – A parameterized way to declare a class or function in C++ that causes source code to be generated when a template object/function is created with specified parameters. Templates are sometimes likened to code factories.
Tree – A collection shape that supports rapid insertion and deletion of internal elements and access by content and position that is slightly more efficient than that afforded by a linked list. Conceptually, it can be viewed as an "organization chart" of elements linked together by pointers, with each parent node potentially pointer to two or more child nodes. Trees may be sorted or unsorted.
vector<> – An STL template class that implements a dynamic array. It takes a single parameter, the data type of the array elements.

13.7.3: Questions

13.1: PersonArray collection shape. Although the PersonArray class was implemented as a sorted vector<>, there is another shape that would have been even more appropriate. Using the general shape descriptions, identify a shape that would have allowed even faster access. Are there any foreseeable circumstances under which that alternative shape would have been less attractive.

13.2: Tree performance. Justify why access by content in a sorted tree is logarithmic. (Hint: contrast what would happen when searching a sorted tree with what happens in binary search of an array.)

13.3: The set<> template: The STL **set<>** template, which manages a collection of unique keyed elements, was omitted from Table 13.2 because nearly all of its shape properties are determined by the underlying collection. Explain why a **set<>** collection implemented using sorted **vector<>** collection might have different performance from one implemented using a **map<>** collection.

13.4: Reimplementing the Point<> Template Class. Create a revised version of the Point template class (Section 13.4.1) that uses a vector<> template internally, instead of an array.

Questions 13.6-13.10 involve enhancing the employee and EmpData classes as originally presented in Chapter 14, Sections 14.6.1 and 14.6.2. These modifications parallel the modifications made in the Chapter 15 end-of-chapter questions 15.8-15.10.

13.6. Replace employee embedded name string arrays with string objects. Change the employee class implementation so that the last name and first names are stored as STL string objects. The only interface changes you should make is to have the LastName() and FirstName() functions return string objects, i.e.:

```
string LastName() const;
string FirstName() const;
```

Note: as long as the employee interface is maintained, you should not have to change EmpData.

13.7. Using a vector<employee> template array. Replace the embedded array of employees in the EmpData class with a *vector<employee>* member. Implement the change so that the EmpData interface does not change.

13.8. Overloading EmpData operators. After making the modification in 13.7, add a copy constructor, assignment operator and insertion/extraction (<< and >>) operator overloads to the EmpData class, making its interface consistent with the employee class. (*Hint:* you'll also want to define a Copy() member, naturally).

13.9. Using a vector<employee> template array.* Replace the embedded array of employees in the EmpData class with a *vector<employee*>* member. Implement the change so that the EmpData interface does not change. How does implementing the internal collection using pointers differ from using objects?

13.10. Overloading EmpData operators (when employee objects are managed as pointers). After making the modification in 13.9, add a copy constructor, assignment operator and insertion/extraction (<< and >>) operator overloads to the EmpData class, making its interface consistent with the employee class. (*Hint*: this will be very similar to 13.3)

Chapter 14

Lookup and Hashing

Executive Summary

Chapter 14 introduces an important collection shape, the hash table. Using an algorithm referred to as hashing, this collection can provided keyed lookup that is independent of collection size. This chapter focuses on both the underlying implementation of the collection and two related STL classes—the **map<>** and **hash_map<>** template classes (the second of which implements hashing).

The chapter begins by introducing the concept of a hash table and exploring a simple algorithm that can be used to implement hashing. This leads to a discussion of the STL **map<>** and **hash_map<>** classes. The interface of these classes and issues related to customizing the classes are explored in some detail. The issue of case-insensitive string-based lookup is investigated in particular detail. The concept of non-unique hashing is then presented. A walkthrough of a hash table collection implementation—modeled after the MFC CMapStringToPtr class—is then conducted. A second walkthrough extends the Tokenizer class, introduced in Chapter 8, to create a list of typed tokens. Finally, a lab exercise for counting the frequencies of words in a document is presented.

Learning Objectives

Upon completing this chapter, you should be able to:

- Explain the basic algorithm used in hashing
- Explain the problem of collisions in a hash table
- Use the STL map<> template class to implement a lookup table
- Use the STL hash_map<> class to implement a hash table
- Discuss how template functions can be used to customize lookup performance
- Discuss how string-based hashing can be implemented using the hash_map template
- Implement a basic hash table class, modeled after the MFC version
- Extend a tokenizing class to implement typed tokenizing
- Using hashing to maintain variable-value relationships
- Use a list and a hash table to implement a unique word counter

14.1: Introduction to Hashing

Walkthrough available in Hashing.avi

Hashing is an elegant technique for looking up values in a table. Of all the collection shapes available, it is generally the most efficient at looking up exact values (e.g., ID codes, SSNs, phone numbers, user names). In this section, we present an overview of a simple version of the technique.

14.1.1: Hash Code

As a prelude to our discussion of the hashing technique, we will introduce a simple—and apparently useless—function called HashCode(). The function is prototyped as follows:

    unsigned int HashCode(const char *szKey,unsigned int nSize);

The function, presented in Example 14.1, does the following:

- Takes a string (szKey) and keeps a running total of the value of each character multiplied by 37.
- Upon completion, returns the remainder of the total divided by nSize

Example 14.1: HashCode() function

```
unsigned int HashCode(const char *szKey,unsigned int nSize)
{
        unsigned int nTotal=0;
        int i;
        assert(nSize>0);
        for(i=0;szKey[i]!=0;i++) {
                nTotal+=(szKey[i]*37);
        }
        return nTotal % nSize;
}
```

It is hard to imagine what a function such as this one could be useful for. Throwing a set of test data at it doesn't seem to clarify matters. For example, in Table 14.1, we see the results of applying the function to a series of fruit names, with a nSize value of 31.

Code	*Key*
12	Apple
21	Banana
1	Grapefruit
0	Kumquat
3	Peach
27	Pear
12	Mango
25	Grape
1	Grapefruit
28	Orange
4	Lemon
21	Lime
6	Cherry
28	Papaya
1	Grapefruit
24	Guava

Table 14.1: Example Code and Key Values

Scanning these codes and keys for a while, we can come to three conclusions:

- *The codes appear to be assigned with no rhyme or reason.* Names that should be close together, such as Peach (3) and Pear (27) are often far apart. Given the nature of the HashCode() function, this is not surprising.
- *Whenever a key is repeated (e.g., Grapefruit), it always has the same code.* Again, this is to be expected because the codes aren't random. Multiplying the sum of all the letters by 37 will always produce the same result.
- *Our codes are not necessarily unique.* For example, Banana and Lime both have the code 21. When you think about it, this would also have to be the case. As mentioned earlier, we used an nSize of 31 for this particular function. Since the function returns the remainder of our total divided by nSize, the code has to be between 0 and 30. And, with a little thought, we could certainly come up with more than 30 different fruit names—not to mention words in general. Thus, duplication of codes is to be expected.

14.1.2: Hash Arrays

How this is going to help us implement a lookup table is still, probably, less than obvious. Let us suppose, however, that we have two parallel arrays, both of which just happen to have a base size of 31. The first array we'll call the key array. It will contain an character pointers. The second array we'll call the data array. It will contain void pointers. The two arrays could be declared as follows:

```
    #define OVERFLOW 9
char *arKeys[31+OVERFLOW];
void *arData[31+OVERFLOW];
```

"What's this OVERFLOW", you may wonder, "and why is it set to 9?". The first question we'll get to shortly. The 9, on the other hand, is easy to explain. It came from the fact that there happened to be 9 gray cells in the diagrams the author had already drawn. In other words, it's arbitrary (much like the 31).

So, returning to the parallel arrays, what we we'll use them for is the following:

- The arKeys array will be used to hold unique lookup keys (e.g., serial numbers, employee IDs, user names) that identify a particular object
- The arData array will hold pointers to the data structures containing the information associated with the key. For example, if employee ID were a key, the associated data pointer might point to an employee structure object, such as the one we've used as an example in this chapter.

Suppose, in our particular application, that we are using fruit names as unique key. Perhaps they are code names for computer components. Or maybe we're just a fruit distributor. Whichever is the case, we can uniquely identify complex data structure objects we want to be able to access rapidly with a fruit name.

Adding Elements
Now, we can return to the codes we generated. The first object we want to insert in our arrays is code named "Apple". Where do we put it? At position 12, the code our function generated for "Apple". As illustrated in Figure 14.1, we place a pointer to the string "Apple" at position 12 in our code array. Then, at position 12 in our data array, we set the pointer to our complex data object.

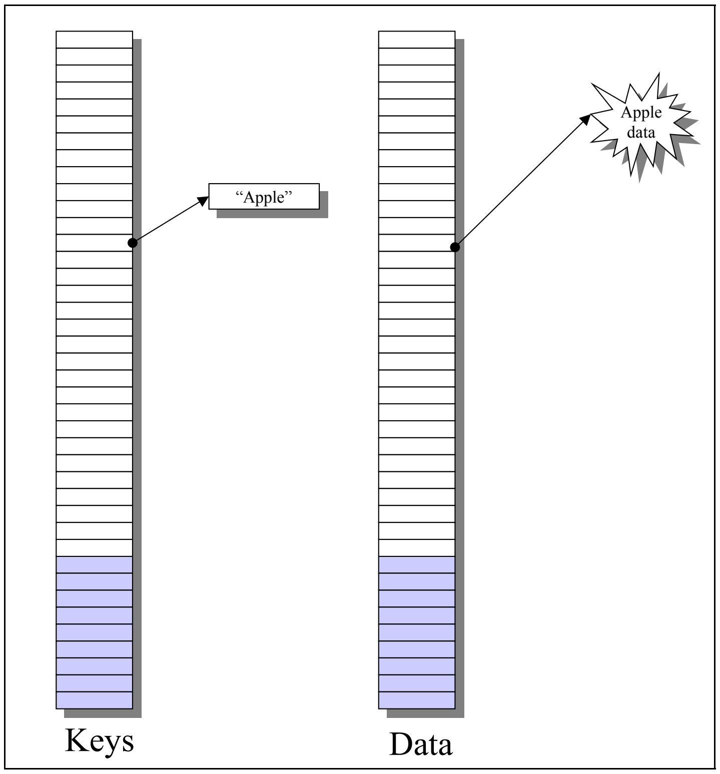


Figure 14.1: Adding "Apple" element to arrays

We can, of course, continue the process of adding elements using their hash codes. By the time we have added six elements, for example, our table appears as in Figure 14.2.

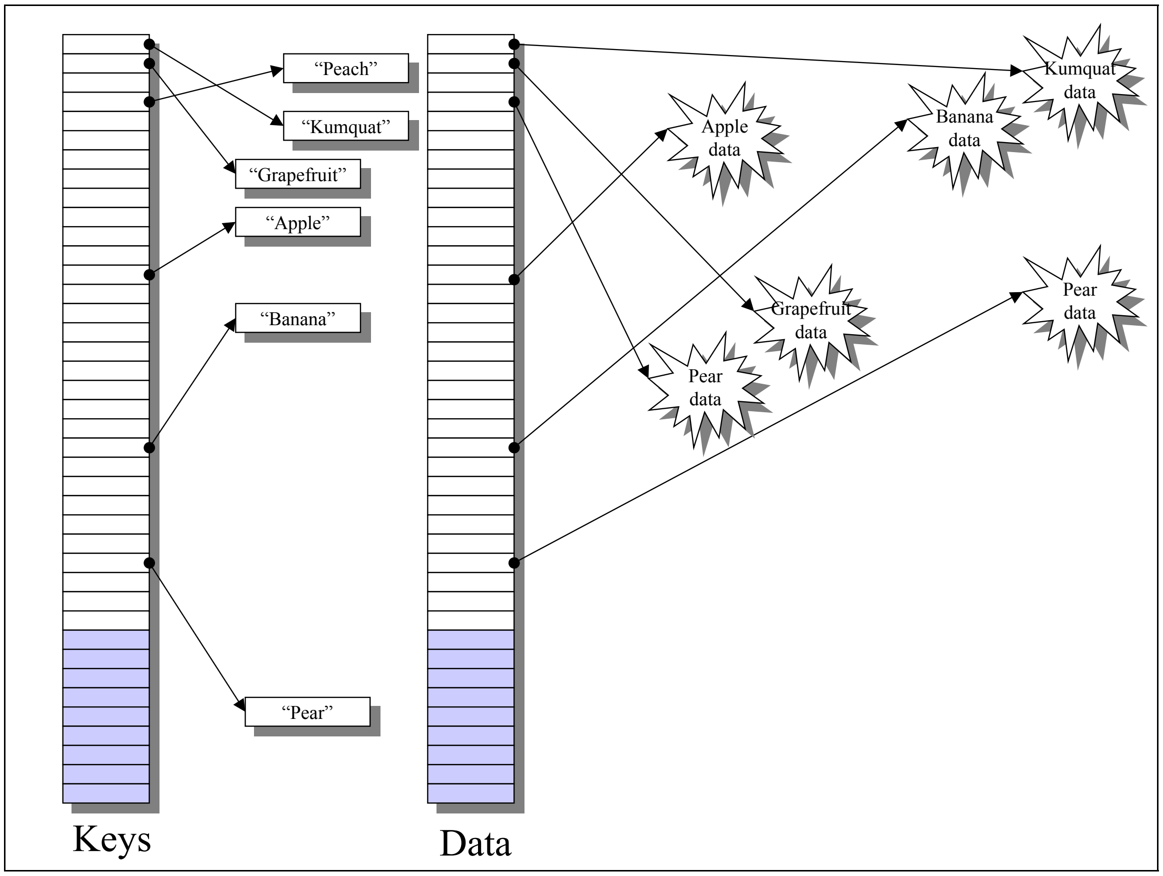


Figure 14.2: Parallel arrays after six additions

What does this process buy us? Well, suppose you wanted to find the data coded by the key "Pear". Where would you start looking? At the beginning of the array? Not likely. To find "Pear" you'd find its code, using the HashCode() function, then jump directly to element 27 (the return value for "Pear") to start your search. And what do you know? You've found it!

This is impressive performance, when you think of it. Our binary search routine (Chapter 9, Section 9.4) only gave us Log_2 performance (around 5 probes in a 31 element array). Here we've done it in 1!

Of course, hashing isn't perfect. We still have the problem of duplicate code values (referred to as "collisions"). When we add our seventh element to the table, for example, we find that "Mango" has the same code value as apple (12). So, what do we do?

A lot of doctoral theses have been written on that particular subject. But for our purposes, there's an easy solution. Just place the pointers to the "Mango" key and associated data in the next location available in the table—13. This is illustrated in Figure 14.3.

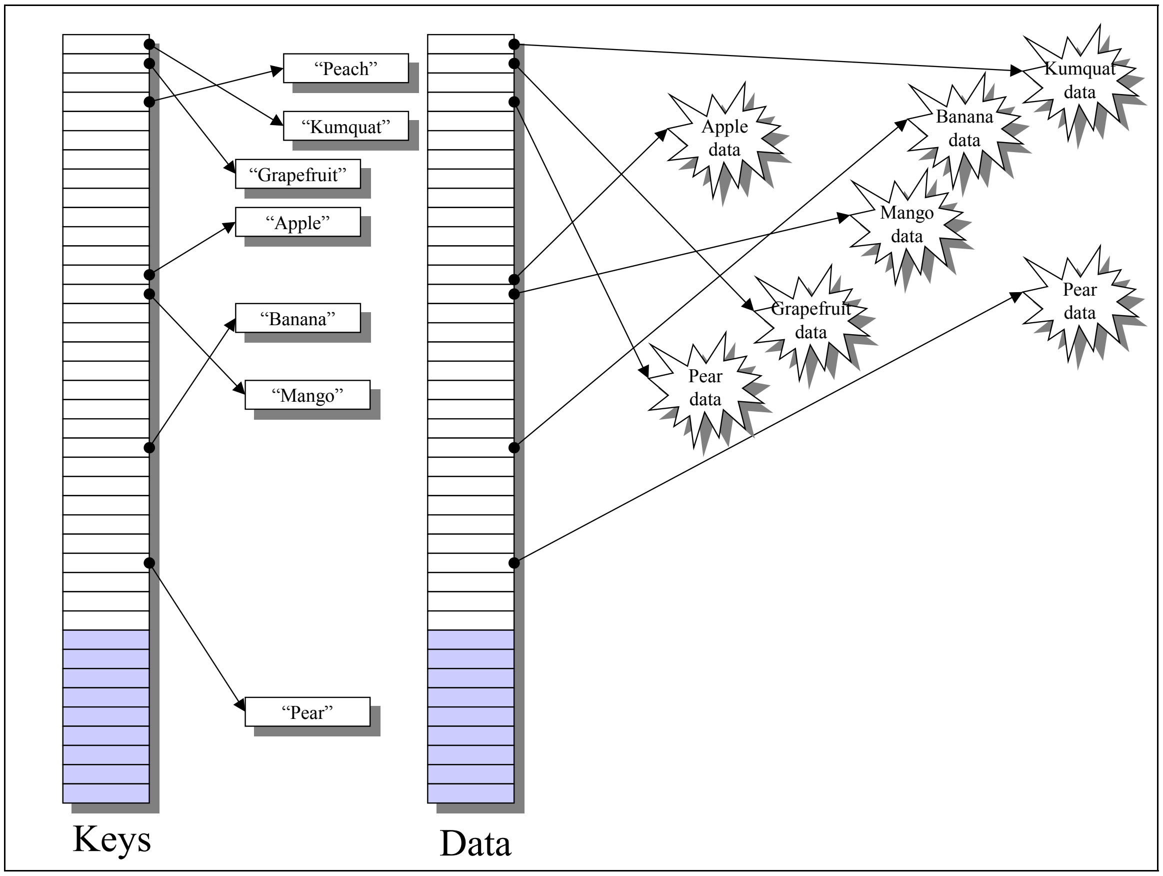


Figure 14.3: Arrays after "Mango" collides with "Apple"

The possibility of collisions does mean that we won't always find what we're looking for on the first probe. What we need to do is as follows:

- Check the key at our key code, make sure it matches. If it does we're done.
- If our key at the probe coefficient does not match, we simply keep moving down until either:
    - We find a matching key—meaning we're done, or
    - We find an empty element—meaning the key is not in the table

Overflow Area

The gray area at the bottom of Figures 14.8-14.10 represents an overflow area. This area is specifically set aside so that collisions at the bottom of the table have somewhere to go. There are many other ways to implement this (e.g., wrap around to the top, have a separate overflow table), but this approach is the simplest. In our example, the overflow area never gets used (we get as high as 29, as a result of a collision between Orange and Papaya). But you need to do something about collisions at the bottom.

Hash Table Size
The performance of a hash table is pretty independent of the number of elements in it—as long as it is big enough. If the hash table gets overly full, however, then there will be lots of collisions and, ultimately, performance will start to degrade towards that of linear search (particularly when looking for elements that are not there). A general rule of thumb is that if 50% of your elements are empty, you'll get near optimum performance. Sophisticated commercial tables often go as high as 70% full.

The practical implication of this is that if you are creating a general hash table implementation, you probably want to be able to specify the table size. Or—even better—dynamically resize it as it grows larger.

14.1 Section Questions

1. What are the most important characteristics of a hash code function?

2. What is the principal advantage of using wraparound for collisions at the bottom of a hash table, as opposed to using an overflow area?

3. What would be the advantages and disadvantages of using hashing in a spell-checker?

4. Why is hash table search size-independent?

5. Would the "optimal" maximum allowable percentage full for a hash table depend on the percentage of searches for missing items you conducted?

6. Suppose, by accident, you wrote a hash code algorithm that only generated even numbers. What impact would that have on hash table performance?

14.2: STL Lookup Classes

The STL provides a number of classes that can be used for lookup. In this section we will examine two of these: the **map<>** class, providing the lookup properties of an ordered array or sorted tree, and the **hash_map<>** class, providing an implementation of a hash table. The section concludes by summarizing STL classes that can be used when duplicate key values may be present.

14.2.1: The STL map<> Template Class

Walkthrough available in STLmap.avi

The STL **map<>** template class offers rapid (logarithmic) lookup along with a standard bi-directional iterator interface that accesses elements in sorted order. The template takes two parameters:

- K: the data type of the key
- T: the data type of the associated data

For example:

```
map<string,employee> mapDemo;
```

would create an associative mapping between a string and some employee object.

pair<> Template Class
The **map<>** template class uses special pair<K,T> structure to hold the key and associate values. The two elements of this structure are:

- first: accesses the key value
- second: accesses the data value

The iterator defined for the **map<>** class—which is bi-directional, like the **list<>** template—returns a **pair<>** object when dereferenced. Thus, extraction of key and value pairs involves taking the .first and .second members of the iterator, as illustrated by the following code fragment:

```
map<string,const char *> mapTest;
// Initialization omitted
map<string,const char *>::iterator iPos;
iPos=mapTest.begin();
while(iPos!=mapTest.end()) {
        string szKey=(*iPos).first;
        string szValue=iPos->second;
        cout << szKey << " is associated with the color " <<
                szValue << endl;
        iPos++;
}
```

The key associative elements of the class are as follows:

- insert(pair<K,T>(key,value)): Inserts a key-value pair into the hash table.
- T &operator[](key)=value: Used to set an association.
- iterator find(key): Returns an iterator to the key-value pair, or end() if no association is present
- T operator[key]: Returns value portion

Map Demonstration
The **map<>** interface is demonstrated in Example 14.2.

Example 14.2: map<> Class Interface Demonstration

```
void STLMapTest()
{
	char *arFruit[12] = {"Orange", "Apple", "Pear", "banana",
		"Tangerine", "Kumquat",
		"Lemon", "Pineapple", "Grapefruit", "Grape", "Cherry", "Mango"};
	char *arColor[12] = {"Orange", "Red", "Yellow/Green", "Yellow",
		"Orange", "No Clue",
		"Yellow", "Tan", "Yellow", "Blue", "Red", "Red/Green"};
	map<string,const char *> mapTest;
	int i;
	for(i=0;i<12;i++) {
		mapTest.insert(pair<string,const char *>(arFruit[i],arColor[i]));
	}
	mapTest["Strawberry"]="Red";
	map<string,const char *>::iterator iPos=mapTest.find("Grape");
	if (iPos==mapTest.end())
		cout << "STL map<> Error  -- Couldn't find appropriate object!\n";
	else
		cout << "STL map<> Test -- \"Grape\" should be: Blue " << " IS "
<<
			(*iPos).second << endl;
	iPos=mapTest.find("Tangelo");
	if (iPos==mapTest.end())
		cout << endl
		<< "Could not find Tangelo! STL map<> worked on missing item."
		<< endl;
	else cout << endl
		<< "STL map<> thought it found Tangelo. Failed on missing item."
		<< endl;
	const char *pVal=mapTest["Tangerine"];
	if (pVal!=0) {
		cout << endl << "Accessed \"Tangerine\" using [] to find color "
			<<	pVal <<endl;
	}
	cout << endl << "Table Display:" << endl;
	iPos=mapTest.begin();
	while(iPos!=mapTest.end()) {
		string szKey=(*iPos).first;
		string szValue=iPos->second;
		cout << szKey << " is associated with the color " <<
			szValue << endl;
		iPos++;
	}
	return;
}
```

Two types of insertion are demonstrated, using the insert() member and the [] overload. Specifically:

mapTest.insert(pair<string,const char *>(arFruit[i],arColor[i]));

and

```
mapTest["Strawberry"]="Red";
```

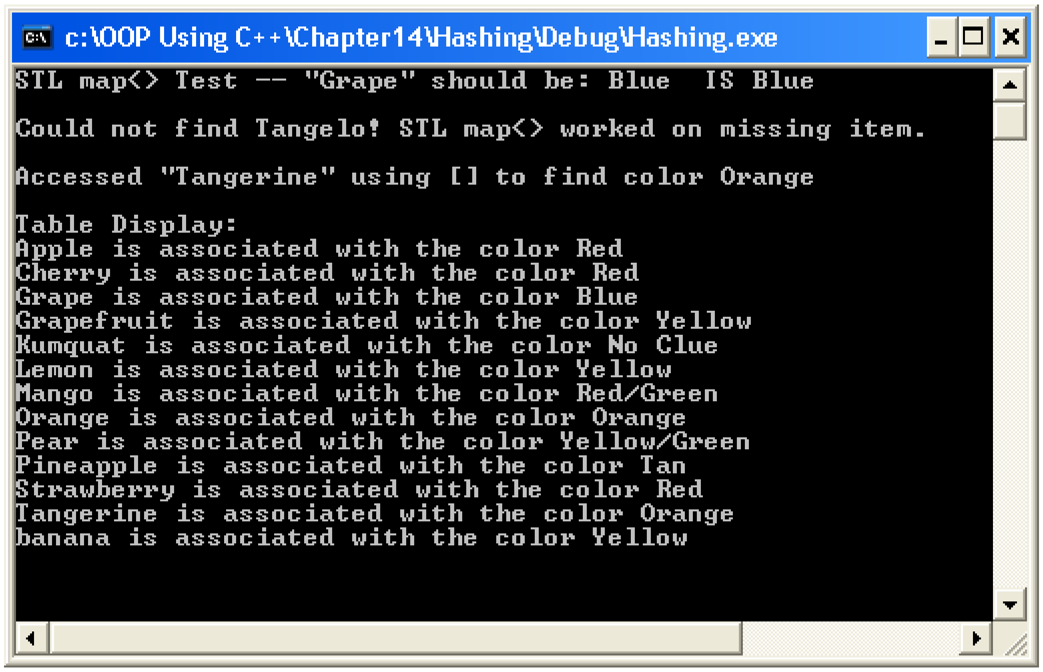
The two types of lookup, using the find() member (which returns an iterator) and the [] operator (which returns the associated value) are also presented. Specifically:

```
map<string,const char *>::iterator iPos=mapTest.find("Grape");
```

and

```
const char *pVal=mapTest["Tangerine"];
```

The output from running the STLMapTest() function is presented in Figure 14.4.


c:\OOP Using C++\Chapter14\Hashing\Debug\Hashing.exe

```
STL map<> Test -- "Grape" should be: Blue  IS Blue

Could not find Tangelo! STL map<> worked on missing item.

Accessed "Tangerine" using [] to find color Orange

Table Display:
Apple is associated with the color Red
Cherry is associated with the color Red
Grape is associated with the color Blue
Grapefruit is associated with the color Yellow
Kumquat is associated with the color No Clue
Lemon is associated with the color Yellow
Mango is associated with the color Red/Green
Orange is associated with the color Orange
Pear is associated with the color Yellow/Green
Pineapple is associated with the color Tan
Strawberry is associated with the color Red
Tangerine is associated with the color Orange
banana is associated with the color Yellow
```

Figure 14.4: Output from running STLMapTest() function

map<> Members
Commonly used **map<>** members are summarized in Table 14.2.

Operator	Arguments	Purpose
begin rbegin	None	Returns an iterator that can be used for bi-directional access referring to the first element of the list. rbegin returns iterator to object at the end of the map, used for reverse iteration.
clear erase	None (clear) 1. iterator (erase) 2. iterator=end() (erase) OR 1. Key data type (erase)	Empties the contents of a list Erase versions 1) allow an optional specified range of the list to be emptied, or 2) allows a key to be specified, erasing a particular association
count	1. Key data type	Returns 1 if matching key is found, 0 otherwise
end rend	None	Returns an iterator representing the position beyond the data in the list. rend is used to find the end of a reverse iteration through the map<>.
find	1. Key data type	Returns an iterator referencing a pair<> object. The iterator points to end() if key is not found.
insert	1. pair<K,T> element	Inserts a pair<> object into the map<> (*Note*: other overloads also exist). Returns a pair<iterator,bool> object where the iterator refers to the inserted position and bool is **true** if the key is a new key, or **false** if the key previously existed.
size	None	Returns number of elements in the map

Table 14.2: Selected map<> Member functions, compiled from Visual Studio .Net documentation

14.2.2: The hash_map<> Template Class

Walkthrough available in hashmap.avi

The **hash_map<>** class implements a hash table. Although not part of the ISO standard, the class was included in the Microsoft STL library prior to Visual Studio .Net Version 2003. At that time, it was shifted to the **stdext** namespace, which holds Microsoft extensions. As a result of this change, programs that use the hash_map class need to include it with the statements:

#include <hash_map>
using namespace stdext;

The STL **hash_map<>** template class provides an interface that is nearly identical to the **map<>** class. What is different between the two classes, however, is that while **map<>** uses a sorted collection as its underlying shape, the **hash_map<>** uses a hash table. The difference in underlying shape has two practical implications for the programmer:

- When iterating through a **hash_map<>**, the elements will not appear to be sorted.
- A function must be available for any object used as a key in a **hash_map<>** class that generates a hash code—which will be a value of type **size_t**.

The first of these implications is what we'd expect whenever we create a hash table. The second means that we have the ability to customize our own keys. One way this can be done is to declare a function, hash_value, for the specific class we want to use as our key. This must be done in the **stdext** namespace, however, to avoid a warning (having to do with argument-dependent lookup, an advanced feature of the .Net compiler that searches across namespaces for matching functions). For example, if you were to define a case-insensitive string class IString, you could define its key to be a simple sum of the upper case letter ASCII values using the definition in Example 14.3.

Example 14.3: The hash_value Overload for the IString class

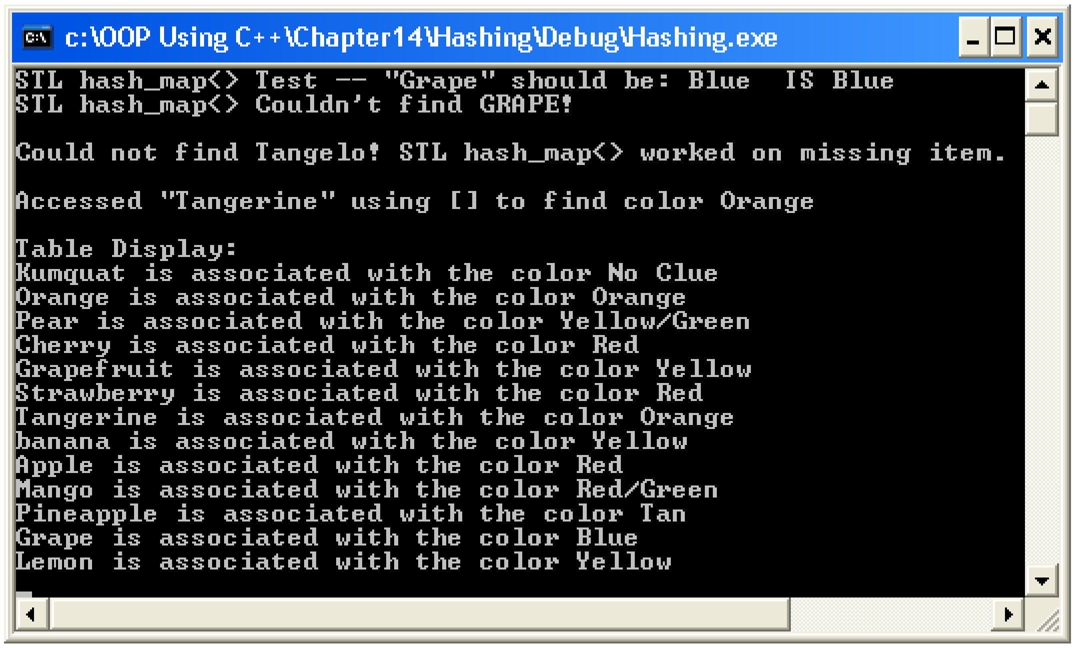
```
namespace stdext
{
   inline size_t hash_value(const IString &str) {
            size_t nTotal=0;
            for(int i=0;i<(int)str.size();i++){
                  nTotal+=toupper(str.at(i));
            }
            return nTotal;
      }
};
```

In .Net Version 2003, the hash_map<> class has built-in overloads for string keys. As a result, it can be used in a manner that is identical to the way we used our map<> object when case-sensitive lookup is desired, as illustrated by the sample code in Example 14.4.

Example 14.4: hash_map<> Class Interface Demonstration

```
void STLHashTest()
{
	char *arFruit[12] = {"Orange", "Apple", "Pear", "banana",
		"Tangerine", "Kumquat",
		"Lemon", "Pineapple", "Grapefruit", "Grape", "Cherry", "Mango"};
	char *arColor[12] = {"Orange", "Red", "Yellow/Green", "Yellow",
		"Orange", "No Clue",
		"Yellow", "Tan", "Yellow", "Blue", "Red", "Red/Green"};
	hash_map<string,const char *> mapTest;
	int i;
	for(i=0;i<12;i++) {
		mapTest.insert(pair<string,const char *>(arFruit[i],arColor[i]));
	}
	// Insertion using [] overload
	mapTest["Strawberry"]="Red";
	hash_map<string,const char *>::iterator iPos=mapTest.find("Grape");
	if (iPos==mapTest.end())
		cout << "STL hash_map<> Error  -- Couldn't find appropriate
object!\n";
	else
		cout << "STL hash_map<> Test -- \"Grape\" should be: Blue " << "
IS " <<
			(*iPos).second << endl;
	iPos=mapTest.find("GRAPE");
	if (iPos==mapTest.end())
		cout << "STL hash_map<> Couldn't find GRAPE!\n";
	else
		cout << "STL hash_map<> Test -- \"GRAPE\" should be: Blue " << "
IS " <<
			(*iPos).second << endl;
	iPos=mapTest.find("Tangelo");
	if (iPos==mapTest.end())
		cout << endl
		<< "Could not find Tangelo! STL hash_map<> worked on missing
item."
		<< endl;
	else cout << endl
		<< "STL hash_map<> thought it found Tangelo. Failed on missing
item."
		<< endl;
	const char *pVal=mapTest["Tangerine"];
	if (pVal!=0) {
		cout << endl << "Accessed \"Tangerine\" using [] to find color "
			<<	pVal <<endl;
	}
	cout << endl << "Table Display:" << endl;
	iPos=mapTest.begin();
	while(iPos!=mapTest.end()) {
		string szKey=(*iPos).first;
		string szValue=iPos->second;
		cout << szKey << " is associated with the color " <<
			szValue << endl;
		iPos++;
	}
	return;
}
```

The output from running the code presented in Example 14.4 is shown in Figure 14.5.



```
STL hash_map<> Test -- "Grape" should be: Blue  IS Blue
STL hash_map<> Couldn't find GRAPE!

Could not find Tangelo! STL hash_map<> worked on missing item.

Accessed "Tangerine" using [] to find color Orange

Table Display:
Kumquat is associated with the color No Clue
Orange is associated with the color Orange
Pear is associated with the color Yellow/Green
Cherry is associated with the color Red
Grapefruit is associated with the color Yellow
Strawberry is associated with the color Red
Tangerine is associated with the color Orange
banana is associated with the color Yellow
Apple is associated with the color Red
Mango is associated with the color Red/Green
Pineapple is associated with the color Tan
Grape is associated with the color Blue
Lemon is associated with the color Yellow
```

Figure 14.5: Output from running STLHashTest()

14.2.3: In Depth: STL Insensitive Lookup

Walkthrough available Insensitive.avi

When working with lookup keys, you frequently find yourself in a situation where you would like you key to be case insensitive. As we know, however, this is not the "native" behavior of C++. Fortunately, the STL gives us some relatively simple tools for changing the nature of our lookups, applicable both for the **map<>** and **hash_map<>** templates.

The first thing we need to be aware of when implementing case-insensitive lookup is that the two templates actually have more than 2 parameters—actually 4, all told. The last two arguments are:

- A comparison class (defaulting to the STL **less<Type>** template class)
- An allocator class (which defaults to allocating the **pair<Key,Type>** template class that we have already discussed).

Because you can go for years programming in C++ without needing to override the default allocator, we'll focus our attention to the comparison class.

To be a valid comparison class for a **map<>** or **hash_map<>** template, the class must have an overloaded parenthesis (function) operator that takes two key objects as arguments and returns **true** if the first is less than the second, **false** otherwise. The default class, the **less<Key>** template class, simply returns the result of applying the < operator on the two objects. This means we can change the behavior in two ways: creating our own comparison class (and specifying it as the third template parameter) or by specializing the less<Key> function overload for our particular key class.

Creating a Comparison Class
Suppose we want to use a **map<>** template to establish case insensitive associations. The easiest way to do this is to establish a trivial class with a single member, a () overload (see Chapter 7, Section 7.1.3 for a discussion of () overloads). An example of such an overload is presented in Example 14.5, which returns a case insensitive comparison.

Example 14.5: MyLess class with overloaded () operator

```
class MyLess {
public:
      bool operator () (const string &s1,const string &s2) const {
            return (stricmp(s1.c_str(),s2.c_str())<0);
      }
};
```

We can then use this overload create the template map<> objects. Specifically, we replace the map<string,const char *> template in Example 14.5 with:

map<string,const char *,MyLess>

This simple change will cause the MyLess () overload to be called whenever two keys are compared. The practical effect will be:

- The order will be case-insensitive when iterating through the **map<>**
- Any find() operations will be case insensitive

Specializing the less<> Template () Overload
The alternative approach to implementing case-insensitive comparisons is to create your own key class (as was done in Example 14.4) and then specialize the STL **less<>** template class for that object type. Example 14.6 illustrates how this could be accomplished if we wished to implement case-insensitive hashing.

Example 14.6: Case-Insensitive Key Class and less<> Overload

```
class IString : public string
{
public:
	IString(){}
	IString(const char *str) {
		assign(str);
	}
};

template<>
inline bool less<IString>::operator ()
	(const IString &s1,const IString &s2) const
{
	return (stricmp(s1.c_str(),s2.c_str())<0);
}

namespace stdext
{
   inline size_t hash_value(const IString &str) {
		size_t nTotal=0;
		for(int i=0;i<(int)str.size();i++){
			nTotal+=toupper(str.at(i));
		}
		return nTotal;
	}
};
```

The IString class incorporates the previously defined hash_value overload (Example 14.3). The second part of the modification states that when the **less<IString>** class is constructed, instead of the using default ()operator overload (i.e., which calls the < operator), our own version of the override—performing case-insensitive comparison—will be called.

Because we've changed the **less<IString>** class with our specialization, when we create a **map<>** or **hash_map<>** using IString objects, we don't need to specify a third parameter. In other words, the default **less<>** parameter will work now that we've changed its behavior for the IString object. Thus:

```
map<IString,const char *> m1;
hash_map<IString,const char *> m2;
```

will both give us case-insensitive mappings.

14.2.4: STL Lookup With Duplicate Keys

The **map<>** and **hash_map<>** template classes both have one trait that may, or may not, be desirable: they do not allow any duplicate key values. For many types of collections (e.g., employee ID to employee object mappings), this may be exactly the type of behavior that is desired. There are, however, some situations where we want to associate more than one value with a given key (e.g., employee ID to dependent name mappings). The STL also provides template classes for managing such collections. Two that parallel the collections we've already discussed are the **multimap<>** and **hash_multimap<>** classes. The interfaces of these classes are nearly identical to the corresponding map<> and hash_map<> interfaces, with a few subtle differences. These are noted in Table 14.3.

Operator	Arguments	Purpose
count	1. Key data type	Returns number of elements matching the specified key
erase	1. Key data type	Erases all associations with a particular key, returning the number of elements removed.
find	1. Key data type	Returns an iterator referencing a pair<> object referring to the first association matching the key. Iteration can then be used to find additional associations matching the key. The iterator points to end() if key is not found.

Table 14.3: Selected multimap<> and hash_multimap<> Member functions that differ from map<> and hash_map<>

14.2 Section Questions

1. How do we know that the underlying shape of a map<> template object is not a hash table?
2. In what key way is a map<> object different from a sorted array?
3. Why is a hash_map<> object harder to set up than a map<> object?
4. Why is hash_map<> class particularly well suited to integer and pointer keys?
5. How does the function overload relate to map<> and hash_map<> classes?
6. What alternative to using iterators is available for map<> and hash_map<> lookup and setting?

14.3: GMapStringToPtr Class

Walkthrough available in GMapStringToPtr.avi

The GMapStringToPtr class is a simple has table that associates a string with a void * data object. Its interface is modeled after the MFC CMapStringToPtr class (although its implementation is quite different).

14.3.1 Class Overview

The GMapStringToPtr class is designed to allow rapid, case sensitive lookups of any data that can be associated with a string lookup key. The interface, which is quite simple, is specifically designed to mimic the MFC CMapStringToPtr class, along with its other lookup table classes.

The class declaration is presented in Example 14.7. Only the public members of the class match those provided with the MFC version. The internal implementation is quite different.

Example 14.7: CMapStringToPtr Class Declaration

```
class GMapStringToPtr
{
public:
	GMapStringToPtr(){Rehash(HASHBLOCK+OVERFLOW);}
	~GMapStringToPtr(){RemoveAll();}
	void SetAt(const char *key,void *p);
	bool Lookup(const char *key,void* &p) const;
	void RemoveKey(const char *key);
	void RemoveAll();
	int GetCount() const {return m_lstData.GetCount();}
	POSITION GetStartPosition()const {
		return m_lstData.GetHeadPosition();
	   };
	void GetNextAssoc(POSITION &rpos,string &key,void * &ptr) const;

protected:
	vector<POSITION> m_arData;          // hash table
	GPtrList m_lstData;
	void Rehash(int i=-1);
	int HashCode(const char *szKey) const;
	int FindPosition(const char *szKey) const;
	GPair *Pair(int i) const {
		if (m_arData[i]==0) return 0;
		return (GPair *)(m_arData[i]->val);
	}
};
```

Three functions are central to the GMapStringToPtr interface:

- *Lookup(const char *key,void *&p)*: This function is used to retrieve values from the table. If a key is found in the table it returns **true**, and the reference argument p is set to the value that corresponds to the key. If the key is not found, the function returns **false**.
- *SetAt(const char *key,void *p)*: Creates a new association in the hash table, or replaces the pointer in an existing association.
- *RemoveKey(const char *key)*: Removes a name-pointer association in the hash table.

In addition, a GPtrList-style interface is provided for iterating through the elements of the table:

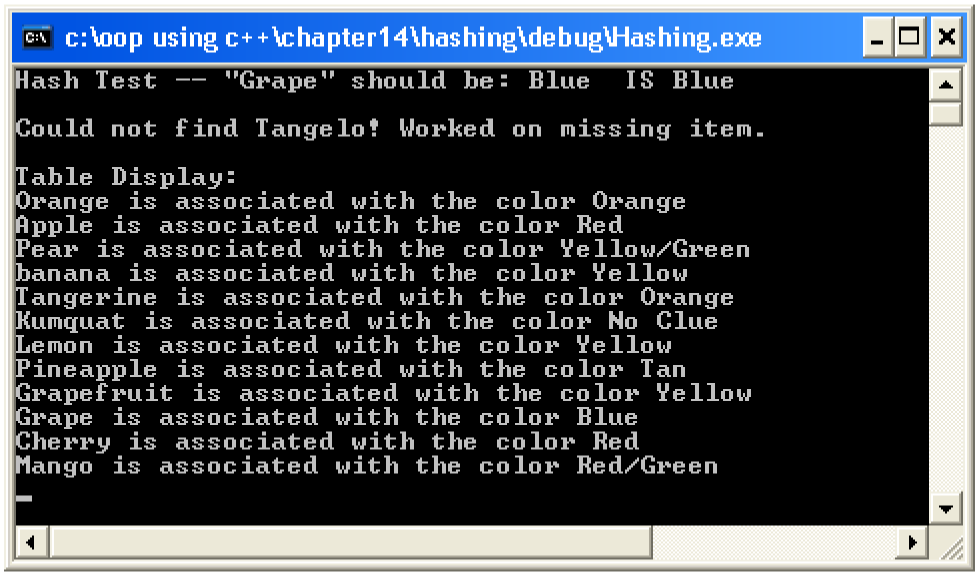
- *GetStartPosition()*: returns a position pointer that—effectively—corresponds to the start of a list of the key-value associations in the table.
- *GetNextAssoc(POSITION &rpos,string &key,void *&ptr)*: Acts like a hash table version of GPtrList::GetNext(), getting the key and pointer associated with the argument's association (placing them in the reference arguments *key* and *ptr)*, then moving the *pos* argument to the next association.

A demonstration of the GMapStringToPtr interface is presented in Example 14.8.

Example 14.8: GMapStringToPtr Demonstration

```
void HashTest()
{
      char *arFruit[12] = {"Orange", "Apple", "Pear", "banana",
            "Tangerine", "Kumquat",
            "Lemon", "Pineapple", "Grapefruit", "Grape", "Cherry",
"Mango"};
      char *arColor[12] = {"Orange", "Red", "Yellow/Green", "Yellow",
            "Orange", "No Clue",
            "Yellow", "Tan", "Yellow", "Blue", "Red", "Red/Green"};
      GMapStringToPtr mapTest;
      int i;
      for(i=0;i<12;i++) {
            mapTest.SetAt(arFruit[i],arColor[i]);
      }

      void *pVal;
      if (! mapTest.Lookup("Grape",pVal))
            cout << "Error  -- Couldn't find appropriate object!\n";
      else
            cout << "Hash Test -- \"Grape\" should be: Blue " << " IS "
<<
                  (const char *)pVal << endl;
      if (! mapTest.Lookup("Tangelo",pVal))
            cout << endl << "Could not find Tangelo! Worked on missing
item."
            << endl;
      else cout << endl <<
         "Thought it found Tangelo. Failed on missing item." << endl;
      cout << endl << "Table Display:" << endl;
      POSITION pos=mapTest.GetStartPosition();
      while(pos!=0) {
            string szKey;
            mapTest.GetNextAssoc(pos,szKey,pVal);
            cout << szKey << " is associated with the color " <<
                  (const char *)pVal << endl;
      }
      return;
}
```

The test code takes an array of fruit names and uses the GMapStringToPtr class to associated the fruit with color values (put in a parallel array for convenience in initializing the hash table object). A loop performs a series of SetAt() operations to establish the associations, then lookup tests on "Grape" (should be there) and "Tangelo" (should not be there) are performed. Finally, it iterates through the associations, demonstrating the similarity between iterating through a the GMapStringToPtr and GPtrList classes. The output resulting from running the demonstration function is presented in Figure 14.6.



```
Hash Test -- "Grape" should be: Blue  IS Blue

Could not find Tangelo! Worked on missing item.

Table Display:
Orange is associated with the color Orange
Apple is associated with the color Red
Pear is associated with the color Yellow/Green
banana is associated with the color Yellow
Tangerine is associated with the color Orange
Kumquat is associated with the color No Clue
Lemon is associated with the color Yellow
Pineapple is associated with the color Tan
Grapefruit is associated with the color Yellow
Grape is associated with the color Blue
Cherry is associated with the color Red
Mango is associated with the color Red/Green
_
```

Figure 14.6: Output from running HashTest() function

14.3.2 Implementation

In implementing the GMapStringToPtr, a single table hash table was chosen. To make this convenient, a pure structure, GPair, was defined to allow the key and the associated pointer to be stored in a single element. The GPair implementation is shown in Example 14.9.

Example 14.9: GPair Structure

```
struct GPair
{
	GPair(const char *szKey,void *pData) {
		key=szKey;
		data=pData;
	}
	string key;
	void *data;
};
```

In addition, to facilitate the simulation of the MFC CMapStringToPtr interface, it was decided that the GPair elements would be stored in a GPtrList collection and that the hash table itself would be implemented as a vector<POSITION> collection. The two collections were defined as follows:

```
	vector<POSITION> m_arData;
	GPtrList m_lstData;
```

Protected Function Summary
In addition to the interface functions, already discussed, the protected, implementation-specific functions in the GMapStringToPtr class are summarized in Table 14.4.

Function	*Description*
Rehash	***void Rehash(int nSize);*** Regenerates the hash table by resetting the m_arData arrays, then iterating through the m_lstData list and placing each GPair() object back in the table. This function needs to be called whenever the hash table is resized—since resizing scrambles up the position of all the keys. It is also called when a key is removed, although a more efficient version is possible (see end-of-chapter exercises)
FindPosition	***int FindPosition(const char *szKey) const;*** Performs a lookup of a particular key (szKey) within the GMapStringToPtr object. Its return value is an array coefficient referring to a position in the m_arData array. It can mean one of three things: 1) if a match is found, the coefficient of the matching key is returned, 2) if no match is found, the coefficient refers to the first 0 pointer in m_arData that it found (which would be the location where the key would be inserted), 3) if no key and no empty element is found, this means the hash table is too full to continue. The function returns the value –1 to signify this.
HashCode	***int HashCode(const char *key) const;*** Creates and initializes a local SimpleHash object. Takes a series of strings entered by the user and generates a hash table, testing adding (SetAt), finding (Lookup) and removing (RemoveKey) functions.
Pair	***GPair *Pair(int i) const;*** Returns the GPair pointer (extracted from the list_position structure) at position i in the hash table (i.e., m_arData). If the table element is unfilled, it returns 0.

Table 14.4: Protected Functions in SimpleHash

Constructing the Table
When creating the GMapStringToPtr object, we need to supply a hash table even though there aren't any associations yet. This is done with a call to Rehash() in the constructor function, as shown in Example 14.7. The initial size supplied is HASHBLOCK (our default value for increasing the table size) + OVERFLOW (our value for the overflow region size, which is independent of hash table size).

Finding Keys and Data in the Table
The process of finding data in the table involves two functions, presented in Example 14.10, and prototyped as follows:

```
bool Lookup(const char *szKey,void *&p) const;
int FindPosition(const char *szKey);
```

The first function, Lookup(), is implemented as a wrapper around FindPosition(), which does the actual searching of the table.

Example 14.10: Lookup functions

```
bool GMapStringToPtr::Lookup(const char *key,void* &p) const
{
	int nCode=FindPosition(key);
	if (nCode<0 || Pair(nCode)==0) return false;
	p=Pair(nCode)->data;
	return true;
}
int GMapStringToPtr::FindPosition(const char *key) const
{
	int nCode=HashCode(key);
	if (nCode<0) return -1; // Error:  allow for illegal code
	while (Pair(nCode)!=0 &&
			key!=Pair(nCode)->key &&
			nCode < (int)m_arData.size()) nCode++;
	if(nCode==(int)m_arData.size()) return -1;  // throw exception
here
	return nCode;
}
```

The LookupPos() function calls HashCode() to find the appropriate table position, then loops through the hash table, stopping only when a match (case sensitive), an empty element, or the end of the overflow region is found. When it stops, for whatever reason, it returns the position where the loop ended. If it stops because it ran out of elements, it returns –1. Otherwise, it returns the coefficient.

Adding Elements

Adding elements to the hash table is accomplished using the SetAt() function, presented in Example 14.11.

Example 14.11: SetAt() function

```
void GMapStringToPtr::SetAt(const char *key,void *p)
{
      int nCode=FindPosition(key);
      if (nCode<0) return;
      if (Pair(nCode)!=0) Pair(nCode)->data=p;
      else
      {
            GPair *pPair=new GPair(key,p);
            POSITION pos=m_lstData.AddTail(pPair);
            m_arData[nCode]=pos;
            if (m_arData.size()/m_lstData.GetCount() < 2)
                  Rehash((int)m_arData.size()+HASHBLOCK);
      }
}
```

The function is very simple. It calls FindPosition() to find the appropriate insertion point (either the location where the key already exists or a blank key pointer). After verifying a valid nCode return value, it checks to see if the return position is empty. If not, it just replaces the pointer in the existing GPair structure with its p argument. If, on the other hand, the hash table position is empty (i.e., a new key is being added), it:

- Creates a new GPair element, pPair, holding the key and associated pointer
- Adds the pPair element to the end of the m_lstData collection.
- Takes the POSITION pointer that is generated by that addition and places it in the m_arData hash table at the assigned position.

Removing Elements

The RemoveKey() function is presented in Example 14.12. The process of removing elements from the hash table is slightly more complicated than that of adding elements. While it is obvious that the key and associated data elements need to be removed, what is less obvious is that doing so can create a spaced between elements that collided—possibly rendering one or more elements below the deleted item invisible.

The easiest solution to this problem of missing elements is just to regenerate the entire table with the remaining Name-Value pairs, called rehashing the table. While there are more elegant (and computationally less-wasteful) ways of handling the problem, the rehashing approach is definitely the simplest.

Example 14.12: RemoveKey() function

```
void GMapStringToPtr::RemoveKey(const char *key)
{
      int nCode=FindPosition(key);
      if (nCode<0 || Pair(nCode)==0) return;
      delete Pair(nCode);
      POSITION pos=m_arData[nCode];
      m_lstData.RemoveAt(pos);
      m_arData[nCode]=0;
      // Force rehash without resizing
      // See end of chapter questions for more efficient approach
      Rehash();
      return;
}
void GMapStringToPtr::RemoveAll()
{
      POSITION pos=m_lstData.GetHeadPosition();
      while(pos!=0) {
            GPair *pPair=(GPair *)m_lstData.GetNext(pos);
            delete pPair;
      }
      m_lstData.RemoveAll();
      m_arData.clear();
      return;
}
```

Removing all elements is simpler, since we don't have to worry about any remaining positions. The RemoveAll() function begins by iterating through the list and deleting all the GPair objects (created when SetAt() was called with a new key). The RemoveAll() member is then called on the m_lstData object, and its STL equivalent—clear()—is called on m_arData.

Rehashing Elements

The rehashing process can occur for two reasons:

1. Our hash table has become more than half full, and we need to resize it
2. We have removed a key, and we need to make sure we didn't create any "holes" in the table that hide keys that were originally below our removed key only as a result of collisions.

The Rehash() function, shown in example 14.13, serves both purposes.

Example 14.13: Rehash() function

```
// nSz is new target size, -1 signals rehash without resizing
void GMapStringToPtr::Rehash(int nSz)
{
      if (nSz>0) {
            nSz=nSz-OVERFLOW;
            nSz=(1+((nSz-1)/HASHBLOCK))*HASHBLOCK;
            nSz=nSz+OVERFLOW;
      m_arData.resize(nSz);
      }
      for(int i=0;i<(int)m_arData.size();i++) m_arData[i]=0;
      POSITION pos=m_lstData.GetHeadPosition();
      while(pos!=0) {
            GPair *pPair=(GPair *)m_lstData.GetAt(pos);
            int nCode=FindPosition(pPair->key.c_str());
            if (nCode<0) continue;
            m_arData[nCode]=pos;
            m_lstData.GetNext(pos);
      }
      return;
}
```

The Rehash() function operates as follows:

- If a negative size is sent in, we don't bother resizing the table.
- If a positive size is sent it, we adjust the size so that the new m_arData size will be an even multiple of HASHBLOCK units, plus the OVERFLOW size. This is done to prevent countless unnecessary resizings.
- We go through the m_arData array an initialize all elements to 0
- We loop through our m_lstData list and reinsert each element in the table. Rather than using SetAt() (which would create a new GPair element, meaning we'd have to delete the old one), we just :
    - Extract the key from the existing pair element
    - Call FindPosition() to determine where it should be inserted
    - Place the list_position pointer at m_arData at that position
- GetNext() is called at the end of the loop because we need to add pos before we go on to the next position.

Iterating Through Elements

The manner in which the GMapStringToPtr table is implemented greatly simplifies the process of iterating through it. The GetStartPosition() function simply returns m_lstData.GetHeadPosition(). The GetNextAssoc() function, shown in Example 14.14, calls m_arData.GetNext() to move to the next list element and to get a pointer to the current GPair object—from which the key and data are extracted.

Example 14.14: GetNextAssoc()

```
void GMapStringToPtr::GetNextAssoc(POSITION &rpos,
                                                 string &key,void
* &ptr) const
{
      GPair *pPair=(GPair *)m_lstData.GetNext(rpos);
      key=pPair->key;
      ptr=pPair->data;
      return;
}
```

14.4:Variables Class

Walkthrough available in Variables.avi

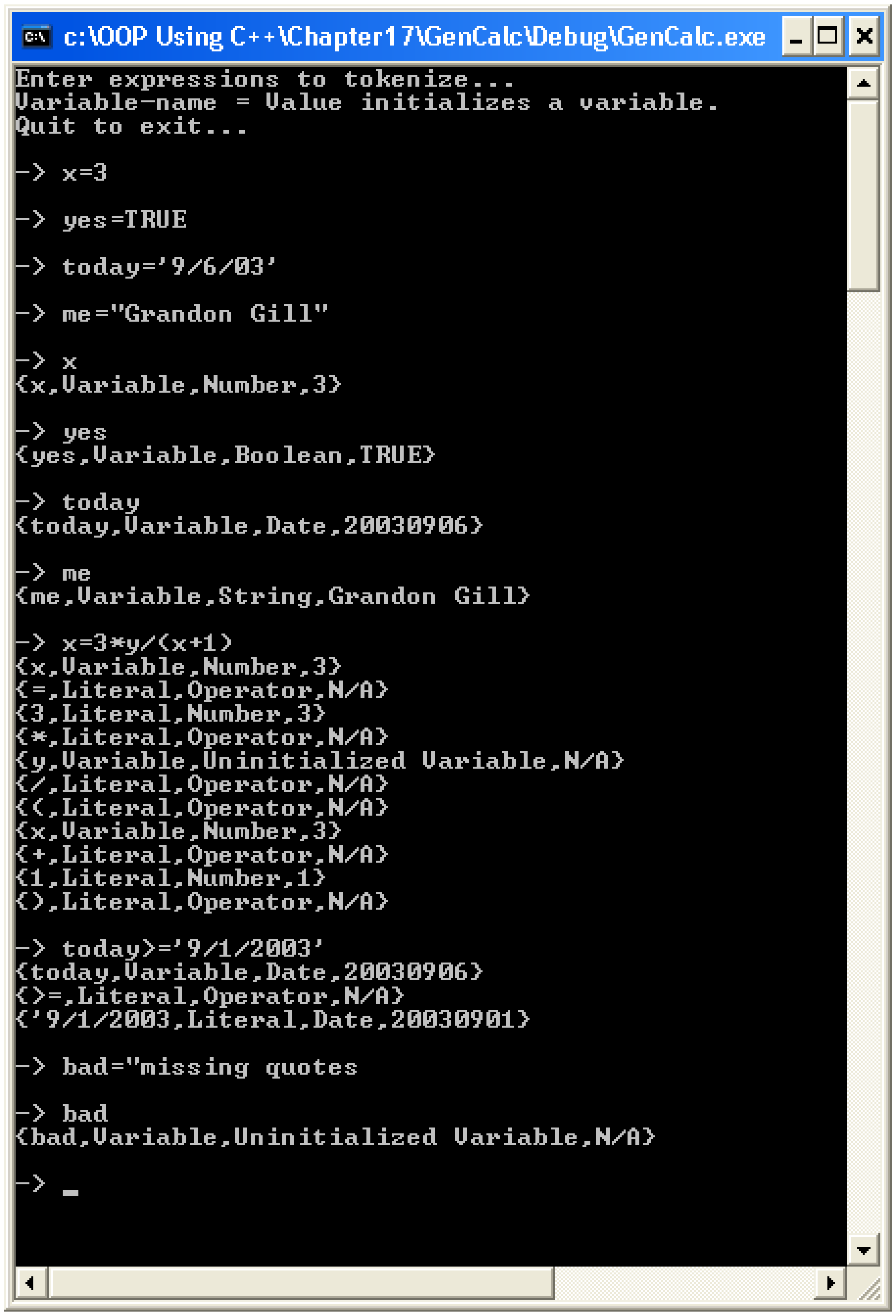
In Chapter 8, Section 8.4, a Tokenize class was presented that could be used to break a string up into a vector<string> collection of tokens. In this section, we enhance that class in two ways:

- *Token typing:* Instead on just returning a collection of string objects, a Token object is created that identifies the type of token.
- *Variable-value relationships:* The class allows us to assign values to non-literal tokens (e.g., numbers, "quoted strings") and retrieve those values later—a capability implemented using a hash_map<> object.

The interface of the Variables class is illustrated in Figure 14.7. Instead of just displaying a list of string tokens, it displays the attributes and values of each token, including:

- Name
- Source (variable or literal)
- Data type (e.g., operator, string, number, Boolean)
- Value (variables and non-operator literals only)

In addition to being a useful demonstration of hashing, the Variables class also plays a central role in the development of the symbolic calculator (Chapter 17, Section 17.3) and the dBase interpreter project (Chapter 22).



c:\OOP Using C++\Chapter17\GenCalc\Debug\GenCalc.exe

```
Enter expressions to tokenize...
Variable-name = Value initializes a variable.
Quit to exit...

-> x=3

-> yes=TRUE

-> today='9/6/03'

-> me="Grandon Gill"

-> x
{x,Variable,Number,3}

-> yes
{yes,Variable,Boolean,TRUE}

-> today
{today,Variable,Date,20030906}

-> me
{me,Variable,String,Grandon Gill}

-> x=3*y/(x+1)
{x,Variable,Number,3}
{=,Literal,Operator,N/A}
{3,Literal,Number,3}
{*,Literal,Operator,N/A}
{y,Variable,Uninitialized Variable,N/A}
{/,Literal,Operator,N/A}
{(,Literal,Operator,N/A}
{x,Variable,Number,3}
{+,Literal,Operator,N/A}
{1,Literal,Number,1}
{),Literal,Operator,N/A}

-> today>='9/1/2003'
{today,Variable,Date,20030906}
{>=,Literal,Operator,N/A}
{'9/1/2003,Literal,Date,20030901}

-> bad="missing quotes

-> bad
{bad,Variable,Uninitialized Variable,N/A}

-> _
```

Figure 14.7: Variables interface

14.4.1: Variables Class Design

The Variables class inherits from the Tokenize class (Chapter 8, Section 8.4) and involves three inter-related classes:

- **IString:** Implements a string object suitable for hashing and case-insensitive comparision.
- **Token:** Inherits from IString, the Token adds three additional properties: value, data type (e.g., number, string, Boolean) and data source (e.g., variable, literal) properties.
- **Variables:** Inherits from Tokenize and maintains a hash_map<IString,Token> collection to permit case-insensitive lookup.

The IString class extends the skeletal version of the class (presented in Example 14.6) by adding:

- Typecast to const char * (making it easier to use)
- Overloads of all comparison operators
- A variety of convenient constructors and assignment overloads

These additions, shown in Example 14.15, make it easy to use where case-insensitivity is desirable. In addition, the overload of the hash_value function, shown previously in Example 14.3, is included in the header file.

Example 14.15: IString Class

```
class IString :        public string
{
public:
        IString(void){}
        IString(const char *str) {Copy(str);}
        IString(const IString &str) {Copy(str.c_str());}
        IString(const string &str) {Copy(str.c_str());}
        ~IString(void){}
        const IString &operator=(const string &str) {
                if (str.c_str()!=c_str()) Copy(str.c_str());
                return *this;
        }
        const IString &operator=(const IString &str) {
                if (&str!=this) Copy(str.c_str());
                return *this;
        }
        const IString &operator=(const char *str) {
                if (str!=c_str()) Copy(str);
                return *this;
        }
        void Copy(const char *str) {
                assign(str);
        }
        operator const char *() const {return c_str();}
        friend bool operator==(const IString &s1,const IString &s2) {
                return (stricmp(s1.c_str(),s2.c_str())==0);}
        friend bool operator==(const IString &s1,const char *s2) {
                return (stricmp(s1.c_str(),s2)==0);}
        friend bool operator==(const char *s1,const IString &s2) {
                return (stricmp(s1,s2.c_str())==0);}
        friend bool operator!=(const IString &s1,const IString &s2) {
                return (stricmp(s1.c_str(),s2.c_str())!=0);}
        friend bool operator!=(const IString &s1,const char *s2) {
                return (stricmp(s1.c_str(),s2)!=0);}
        friend bool operator!=(const char *s1,const IString &s2) {
                return (stricmp(s1,s2.c_str())!=0);}
        friend bool operator<(const IString &s1,const IString &s2) {
                return (stricmp(s1.c_str(),s2.c_str())<0);}
        friend bool operator<(const IString &s1,const char *s2) {
                return (stricmp(s1.c_str(),s2)<0);}
        friend bool operator<(const char *s1,const IString &s2) {
                return (stricmp(s1,s2.c_str())<0);}
        friend bool operator<=(const IString &s1,const IString &s2) {
                return (stricmp(s1.c_str(),s2.c_str())<=0);}
        friend bool operator<=(const IString &s1,const char *s2) {
                return (stricmp(s1.c_str(),s2)<=0);}
        friend bool operator<=(const char *s1,const IString &s2) {
                return (stricmp(s1,s2.c_str())<=0);}
        friend bool operator>(const IString &s1,const IString &s2) {
                return (stricmp(s1.c_str(),s2.c_str())>0);}
        friend bool operator>(const IString &s1,const char *s2) {
                return (stricmp(s1.c_str(),s2)>0);}
        friend bool operator>(const char *s1,const IString &s2) {
                return (stricmp(s1,s2.c_str())>0);}
        friend bool operator>=(const IString &s1,const IString &s2) {
                return (stricmp(s1.c_str(),s2.c_str())>=0);}
        friend bool operator>=(const IString &s1,const char *s2) {
                return (stricmp(s1.c_str(),s2)>=0);}
        friend bool operator>=(const char *s1,const IString &s2) {
                return (stricmp(s1,s2.c_str())>=0);}
};
```

The Token class, which inherits from IString, is presented in Example 14.16. Like the IString class, it is almost entirely implemented in the header file.

Example 14.16: Token Class

```
class Token : public IString
{
public:
	Token(){m_tType=tUnknown;}
	Token(const char *str) {
		Initialize(str,0);
	}
	Token(const Token &tok) {
		Copy(tok);
	}
	~Token();
	const Token &operator=(const Token &tok) {
		if (&tok!=this) Copy(tok);
		return *this;
	}
	void Copy(const Token &tok) {
		Initialize(tok.Name(),tok.Value(),tok.Type(),tok.Source());
	}
	void Initialize(const char *str,const char *val=0,
		enum tType tT=tUnknown,enum tSource tS=tUnspecified) {
		assign(str);
		if (val!=0) Value(val);
		else m_szValue.clear();
		Type(tT);
		Source(tS);
	}
	void Value(const char *str){m_szValue=str;}
	const char *Value() const {return m_szValue.c_str();}
	void ResetValue(){m_szValue.clear();}
	bool ValidValue() const {return !m_szValue.empty();}
	IString Name() const {return c_str();}
	enum tType Type() const {return m_tType;}
	void Type(enum tType t){m_tType=t;}
	enum tSource Source() const {return m_tSource;}
	void Source(enum tSource s){m_tSource=s;}
protected:
	IString m_szValue;
	enum tType m_tType;
	enum tSource m_tSource;

};
```

In addition to the class definitions, there are two sets of enumerations to hold token data type and token source values, as shown below:

```
enum tType {
    tUnknown=0,
    tString = 'C',
    tNumber = 'N',
    tBoolean = 'L',
    tDate = 'D',
    tOperator = 'O',
    // Additional types could be added
};

enum tSource {
    tUnspecified=0,
    tLiteral,
    tValue,
    tVariable,
    // Additional sources, such as databases, could be added
};
```

These were designed to be extensible, and were specifically chosen to meet the needs of Chapter 22 (the dBase Interpreter, using the field type codes in the tType enumeration). The remainder of the class consists of various accessor functions to get at name, value, type and source values.

The Variables class is the most complex, and is declared in Example 14.17.

Example 14.17: Variables Class Declaration

```
class Variables : public Tokenize
{
public:
      Variables(void);
      Variables(const char *szNonB,const char *szDelim,
            const vector<string> &arJn,bool bHide=false)
      {
            Initialize(szNonB,szDelim,arJn,bHide);
      }
      ~Variables(void);
      bool InitFromData(const char *sz,Token &tok) const;
      bool InitValue(Token &tok) const;
      bool SetValue(const char *szName,const char *szVal,
            enum tType t=tUnknown);
      enum tType GetType(const char *szName) const;
      enum tSource GetSource(const char *szName) const;
      bool GetValue(const char *szName,string &val) const;
      bool TokenizeString(const char *buf,vector<Token> &arDest) const;
      void TokenData(const char *str,string &vType,string &dType,
            string &val) const;

      // Data type tests
      bool IsNumber(const char *str) const;
      bool IsBoolean(const char *str) const;
      bool IsDate(const char *str) const;
      bool IsLegalName(const char *str) const;
      bool IsOperator(const char *str) const;
      bool IsString(const char *str) const;
      bool IsVariable(const char *str) const;

      // Conversions Routines
      static double Number(const Token &tok);
      static Token Number(double dVal);
      static unsigned int Date(const Token &tok);
      static Token Date(unsigned int nVal);
      static bool Bool(const Token &tok);
      static Token Bool(bool bVal);
      static string DateToValue(const char *str);
      static bool LegalDate(int nM,int nD,int nY);

protected:
      hash_map<IString,Token> m_mapVariables;
};
```

The class extends the Tokenize class in two key ways: 1) a new ToknizeString() function is provided that generates a vector<Token> array, instead of a vector<string> array, and 2) variable lookup is supported with a hash_map<IString,Token> data member. We now turn to the implementation of the class.

14.4.1: Variables Class Implementation

A number of the Variables member functions are designed to determine the type of token to be created from a string. These functions are presented in Example 14.18.

Example 14.18: Variables Token-Typing Members

```
// Only decimal numbers supported
bool Variables::IsNumber(const char *str) const {
      size_t i,nLen=strlen(str);
      for(i=0;str[i]>='0' && str[i]<='9';i++){}
      if (str[i]=='.') i++;
      while(str[i]>='0' && str[i]<='9') i++;
      if (str[i]!=0) return false;
      return (i>1 || (i==1 && str[0]!='.'));
}
bool Variables::IsBoolean(const char *str) const {
      IString s(str);
      return (s=="TRUE" || s=="FALSE");
}
bool Variables::IsString(const char *str) const {
      return (strchr(Delimiters().c_str(),*str)!=0);
}
bool Variables::IsOperator(const char *str) const {
      return IsBreak(*str);
}
bool Variables::IsLegalName(const char *str) const {
      if (IsBoolean(str)) return false;
      char cFirst=toupper(*str);
      return (cFirst>='A' && cFirst<='Z');
}
bool Variables::IsVariable(const char *str) const {
      hash_map<IString,Token>::const_iterator iter;
      iter=m_mapVariables.find(IString(str));
      return (iter!=m_mapVariables.end());
}
bool Variables::IsDate(const char *str) const {
      if (strchr(Delimiters().c_str(),*str)==0) return false;
      string sz=DateToValue(str+1);
      return (!sz.empty());
}
```

All of these functions are straightforward attempts to deduce a token string's type from its form, excepting the last two. The first, IsVariable(), uses a hash table lookup to determine if the variable is already present in the m_mapVariables hash table. The second, IsDate() attempts to determine if a token is a date, in the form "MM/DD/YYYY". (Because dates must be quoted—since they contain break characters—a leading delimiter is a prerequisite).

Normally, we'd want to use local-specific functions to perform date validation, such as those provided in the <time.h> library or the COleDateTime class in the MFC library.

Otherwise, our program will not adapt to locale changes, such as a move to Europe where month and day are swapped. For our purposes, however, we've create a series of functions that do validation manually. In addition, we have chosen to use an internal format of YYYYMMDD to represent dates within our. program. For example, a token string of '12/23/04' would have "20041223" as its value member. (This particular format has the nice property that it sorts properly, using either its integer or string value). A series of date conversion functions are presented in Example 14.19.

Example 14.19: CVariables Date Conversion Functions

```
string Variables::DateToValue(const char *str)
{
	string szDate;
	int nM,nD,nY;
	if ((sscanf(str,"%d/%d/%d",&nM,&nD,&nY)==3 ||
		sscanf(str,"%d-%d-%d",&nM,&nD,&nY)==3) &&
		LegalDate(nM,nD,nY))
	{
			char buf[20];
			// Introducing some Y2K-style assumptions
			if (nY<100) {
				if (nY>=50) nY+=1900;
				else nY+=2000;
			}
			// Changing date to a YYYYMMDD problem
			sprintf(buf,"%d",nY*10000+nM*100+nD);
			szDate=buf;
	}
	return szDate;
}

bool Variables::LegalDate(int nM,int nD,int nY)
{
	int arM[]={31,28,31,30,31,30,31,31,30,31,30,31};
	if (nM==2 && (nY%4==0)) arM[1]=29;
	// We'll limit dates to 1900 and above, to keep the leap year
valid
	return (nM>0 && nM<=12 && nD>0 && nD<=arM[nM-1] &&
			(nY<100 || nY>1900));
}
```

A series of simple functions for converting back and forth between tokens and their associated values are also presented in Example 14.20. These functions prove useful in Chapter 17 (the calculator lab exercise) and Chapter 22 (the dBase interpretor project).

Example 14.20: Simple Conversion Functions

```
double Variables::Number(const Token &tok)
{
	return atof(tok.Value());
}
Token Variables::Number(double dVal)
{
	char buf[80];
	sprintf(buf,"%lf",dVal);
	Token tok;
	tok.Initialize(buf,buf,tNumber,tValue);
	return tok;
}
unsigned int Variables::Date(const Token &tok)
{
	unsigned int nRet=0;
	return atoi(tok.Value());
}
Token Variables::Date(unsigned int nVal)
{
	Token tok;
	int nY=0,nM=0,nD=0;
	nY=nVal%10000;
	nM=(nVal/100)%100;
	nD=nVal%100;
	if (LegalDate(nM,nD,nY)) {
		char buf[20],szVal[10];
		sprintf(buf,"\'%d/%d/%d",nM,nD,nY);
		sprintf(szVal,"%d",nVal);
		tok.Initialize(buf,szVal,tDate,tValue);
	}
	return tok;
}
bool Variables::Bool(const Token &tok)
{
	return (stricmp(tok.Value(),"TRUE")==0);
}
Token Variables::Bool(bool bVal)
{
	char *szVal=((bVal) ? "TRUE" : "FALSE");
	Token tok;
	tok.Initialize(szVal,szVal,tBoolean,tValue);
	return tok;
}
```

The engine that actually drives token creation is the combination of the TokenizeString() and InitValue() and InitFromData() functions, presented in Example 14.21. TokenizeString() is actually qutie simple. It calls the base class function to create an array of string tokens. It then takes each string, creates a token with it, calls InitValue() to set its type and source, then adds the token to the *arDest* token vector.

Example 14.21: TokenizeString() and InitValue() functions

```
bool Variables::TokenizeString(const char *buf,vector<Token> &arDest) const
{
        vector<string> ar;
        if (!Tokenize::TokenizeString(buf,ar)) return false;
    for(size_t i=0;i<ar.size();i++) {
                Token tok(ar[i].c_str());
                InitValue(tok);
                arDest.push_back(tok);
        }
        return true;
}
bool Variables::InitValue(Token &tok) const
{
        bool bValid=true;
        if (IsLegalName(tok.c_str())) {
                if (IsVariable(tok)) {
                        string szVal;
                        GetValue(tok.c_str(),szVal);
                        tok.Value(szVal.c_str());
                        tok.Source(GetSource(tok.c_str()));
                        tok.Type(GetType(tok.c_str()));
                }
                else {
                        tok.ResetValue();
                        tok.Type(tUnknown);
                        tok.Source(tUnspecified);
                        bValid=false;
                }
        }
        else bValid=InitFromData(tok.c_str(),tok);
        return bValid;
}

bool Variables::InitFromData(const char *sz,Token &tok) const
{
        bool bValid=true;
        if (IsOperator(sz)) tok.Initialize(sz,sz,tOperator,tLiteral);
        else if (IsNumber(sz)) tok.Initialize(sz,sz,tNumber,tLiteral);
        else if (IsDate(sz))
        {
                string szD=DateToValue(sz+1);
                tok.Initialize(sz,szD.c_str(),tDate,tLiteral);
        }
        else if (IsBoolean(sz)) tok.Initialize(sz,sz,tBoolean,tLiteral);
        else if (IsString(sz)) tok.Initialize(sz,sz+1,tString,tLiteral);
        else {
                tok.Initialize(sz,0);
                bValid=false;
        }
        return bValid;
}
```

InitValue() attempts to deduce a token's type. It begins be checking for a legal variable name. If found, it looks up the variable with IsVariable(). If found, it gets the type and source information from the existing variable. If the variable is not already present, type is set to tUnknown and source is set to tUnspecified.

If the token is not a legal variable name, the function calls InitFromData(), which goes through a series of tests to determine type (with source always being set to tLiteral, meaning a constant).

- If the token string proves to be an operator, number or Boolean, the token value is set to the token string itself.
- For delimited strings, the leading character (the delimiter) is removed so the value is the string itself (recall that the Chapter 8, Section 8.4 tokenizer already removed trailing delimiters).
- For date literals (e.g., "9/6/03"), the DateToValue() function is called to translate the value into its internal representation (e.g., "20030906")
- If the token does not match any known pattern, the value is cleared, type is set to tUnknown and source is set to tUnspecified.

The function returns true unless an unidentifiable or unspecified token is encountered.

A variety of functions are provided for accessing token information using the token's name. These are presented in Example 14.22. Each of these functions is constructed so that the relevant information for the token (value, type, source) can be extracted for either variables or literals. For this reason, they all have a structure similar to InitValue(), namely:

- A test is made to see if the token is a legal variable. If so, the information is extracted from the variable token.
- A series of tests are made using different literal types, until the proper one is identified
- The function returns an appropriate value (**false**, tUnknown, tUnspecified) in the event of failure.

In the case of the GetValue() function, a string reference argument is used to hold the value. The other two functions return the type/source value.

Example 14.22: Token Information Lookup Functions

```
bool Variables::GetValue(const char *szName,string &val) const
{
      val.clear();
      bool bValid=true;
      if (IsVariable(szName)) {
            hash_map<IString,Token>::const_iterator iter;
            iter=m_mapVariables.find(IString(szName));
            val=iter->second.Value();
      }
      else if (IsLegalName(szName)) bValid=false;
      else if (IsBoolean(szName) || IsNumber(szName)) val=szName;
      else if (IsDate(szName)) val=DateToValue(szName);
      else if (IsString(szName)) val=szName+1; // Strip delimiter
      else bValid=false;
      return false;
}
enum tSource Variables::GetSource(const char *szName) const
{
      enum tSource t=tUnspecified;
      if (IsVariable(szName)) {
            hash_map<IString,Token>::const_iterator iter;
            iter=m_mapVariables.find(IString(szName));
            t=iter->second.Source();
      }
      else if (IsBoolean(szName) || IsNumber(szName)||
            || IsDate(szName) || IsString(szName))
            t=tLiteral;
      return t;
}
enum tType Variables::GetType(const char *szName) const
{
      enum tType t=tUnknown;
      if (IsVariable(szName)) {
            hash_map<IString,Token>::const_iterator iter;
            iter=m_mapVariables.find(IString(szName));
            t=iter->second.Type();
      }
      else if (IsBoolean(szName)) t=tBoolean;
      else if (IsNumber(szName)) t=tNumber;
      else if (IsDate(szName)) t=tDate;
      else if (IsString(szName)) t=tString;
      return t;
}
```

Variables are added to the hash table using the SetValue() function (Example 14.23). This function takes a name, value and (optional) type argument. After checking that the name is legal, the function builds a token, setting its value to the evaluated value of szVal (either a literal or the value associated with the variable name), its source to tVariable and its type based on either a type argument (defaulting to tUnknown) or whatever type it is induced to be based on the value string. The function returns **true** unless we are trying to set the value of an illegal name.

Example 14.23: SetValue() function

```
bool Variables::SetValue(const char *szName,const char *szVal)
{
      bool bValid=true;
      if (IsLegalName(szName)) {
            Token tok,rhs(szVal);
            tok.Source(tVariable);
            if (!InitValue(rhs)) {
                  tok.Initialize(szName,0);
                  bValid=false;
            }
            else
tok.Initialize(szName,rhs.Value(),rhs.Type(),tVariable);
            m_mapVariables[tok.Name()]=tok;
      }
      else bValid=false;
      return bValid;
}
```

The final function in the class, TokenData(), is provided mainly for display and testing purposes. It takes a string and gathers the information about the associated token (similar to InitValue() in Example 14.19) that are placed in its string arguments. It is presented in Example 14.22.

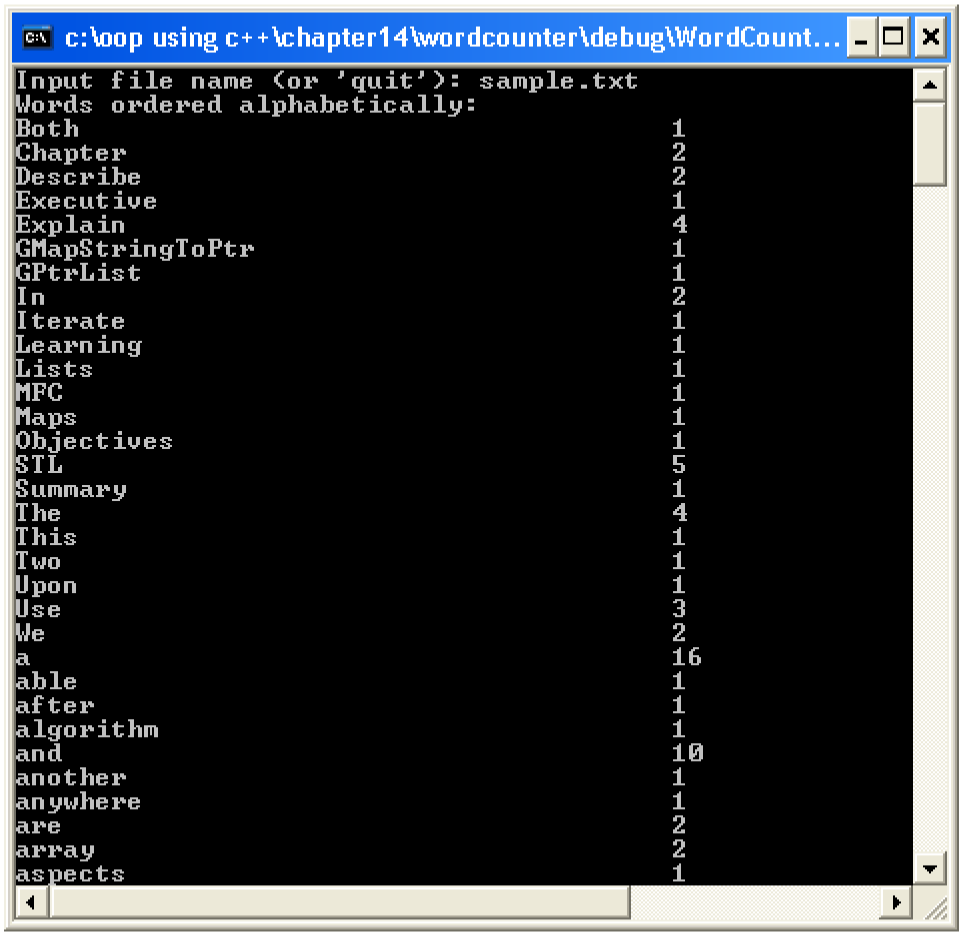
Example 14.22: TokenData() function

```
void Variables::TokenData(const char *str,string &vType,string &dType,
                                      string &val) const
{
      val.clear();
      enum tType t=GetType(str);
      if (IsLegalName(str)) {
            vType="Variable";
            if (IsVariable(str)) {
                  GetValue(str,val);
                  t=GetType(str);
            }
            else {
                  t=tUnknown;
                  val="N/A";
            }
      }
      else if (IsOperator(str)) {
            vType="Literal";
            t=tOperator;
            val="N/A";
      }
      else {
            vType="Literal";
            Token tok;
            InitFromData(str,tok);
            t=tok.Type();
            val=tok.Value();
      }
      if (t==tBoolean) dType="Boolean";
      else if (t==tNumber) dType="Number";
      else if (t==tString) dType="String";
      else if (t==tDate) dType="Date";
      else if (t==tOperator) dType="Operator";
      else if (vType=="Variable" && val=="N/A") dType="Uninitialized
Variable";
      else dType="Unknown";
}
```

14.5: Lab Exercise: Unique Word Counter

Walkthrough available WordCounter.avi

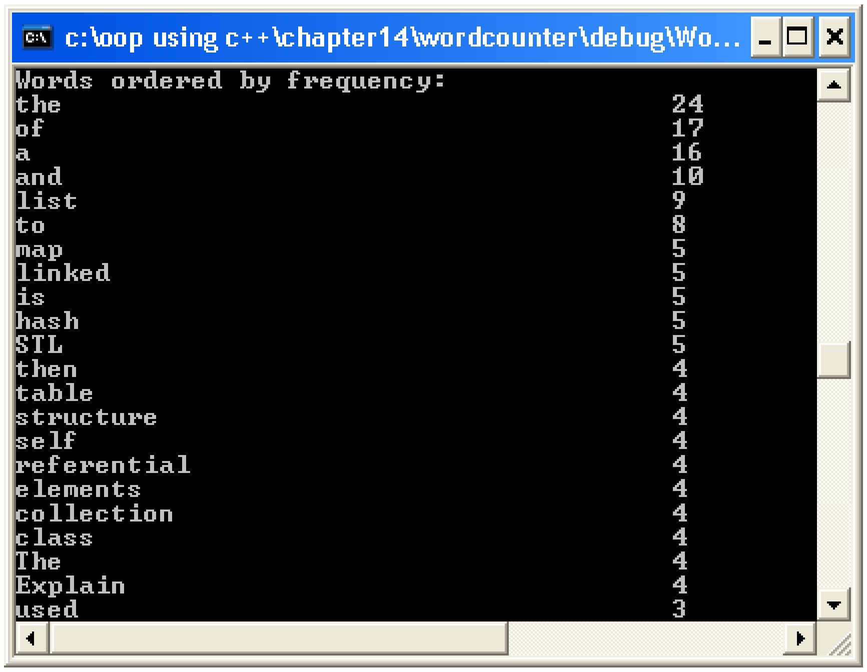
In this exercise, we will construct a program that performs word counts of a text file, and organizes the words both alphabetically and by frequency. An example of an alphabetical sort is presented in Figure 14.8.



```
Input file name (or 'quit'): sample.txt
Words ordered alphabetically:
Both                                        1
Chapter                                     2
Describe                                    2
Executive                                   1
Explain                                     4
GMapStringToPtr                             1
GPtrList                                    1
In                                          2
Iterate                                     1
Learning                                    1
Lists                                       1
MFC                                         1
Maps                                        1
Objectives                                  1
STL                                         5
Summary                                     1
The                                         4
This                                        1
Two                                         1
Upon                                        1
Use                                         3
We                                          2
a                                           16
able                                        1
after                                       1
algorithm                                   1
and                                         10
another                                     1
anywhere                                    1
are                                         2
array                                       2
aspects                                     1
```

Figure 14.8: Case Sensitive Alphabetical Sort

An example of the sort by frequency is presented in Figure 14.9.



```
Words ordered by frequency:
the                                     24
of                                      17
a                                       16
and                                     10
list                                    9
to                                      8
map                                     5
linked                                  5
is                                      5
hash                                    5
STL                                     5
then                                    4
table                                   4
structure                               4
self                                    4
referential                             4
elements                                4
collection                              4
class                                   4
The                                     4
Explain                                 4
used                                    3
```

Figure 14.9: Example Sort by Frequency

14.5.1: Case Sensitive Word Counter

The case sensitive version of the word counter treats words as being different if any letters in the two words differ in case. A WordCounter class should be created, inheriting from the appropriate STL map<> template, and should have at least the members listed in Table 14.5.

Table 14.5: Functions for Word Counter Class

bool Load(const char *filename) Opens a text file, tokenizes each line and places each word in the map, incrementing a count associated with each word.
void DisplayAlpha() Displays the words from a document in alphabetical order, with the frequency of each in a right hand column, as illustrated in Figure 14.8.
void DisplayFreq() Displays the words from a document in descending order of frequency, as shown in Figure 14.9.
void Interface() Prompts the user for a file name (or quit) then calls Load() followed by the two display functions. Should loop, prompting for more files, until the user types quit.

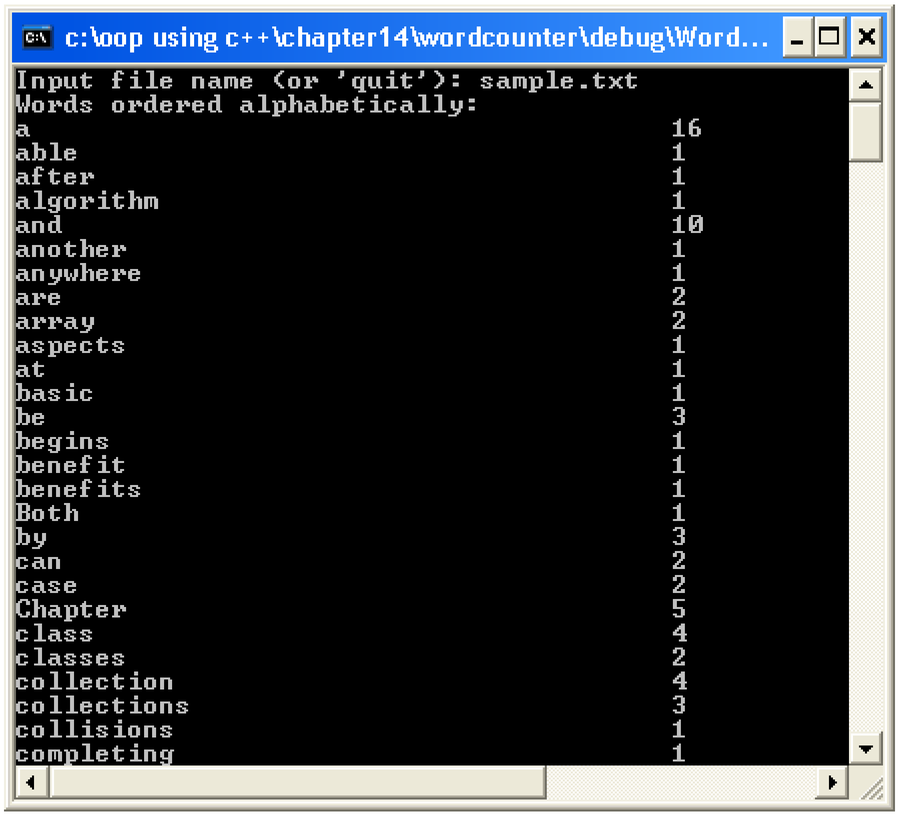
int AddWord(const char *p)
Adds a word to the map, incrementing its count if it is already there.

Of these functions, the DisplayFreq() is the most complex to implement. Some comments may be helpful in this regard:

- Use a separate map<> object to construct the frequency table (you can make it a member of the WordCounter class)
- Since many words are likely will have the same frequency, you can't just use frequency as a organizing key (although you could if you use a **multimap<>** template, as discussed in Section 14.2.4). What you can do, however, is to construct a key with the frequency followed by the word—sprintf() being a very useful function in this construct. Then, each key will be unique but will still be ordered by frequency.
- To get output in descending order, declare a reverse_iterator variable, instead of an iterator. Then use the rbegin() and rend() members to iterate through the collection.

14.5.2: In Depth: Case Insensitive Word Counter

Modify the class so it produces case-insensitive output, as shown in Example 14.10.



```
Input file name (or 'quit'): sample.txt
Words ordered alphabetically:
a                                         16
able                                      1
after                                     1
algorithm                                 1
and                                       10
another                                   1
anywhere                                  1
are                                       2
array                                     2
aspects                                   1
at                                        1
basic                                     1
be                                        3
begins                                    1
benefit                                   1
benefits                                  1
Both                                      1
by                                        3
can                                       2
case                                      2
Chapter                                   5
class                                     4
classes                                   2
collection                                4
collections                               3
collisions                                1
completing                                1
```

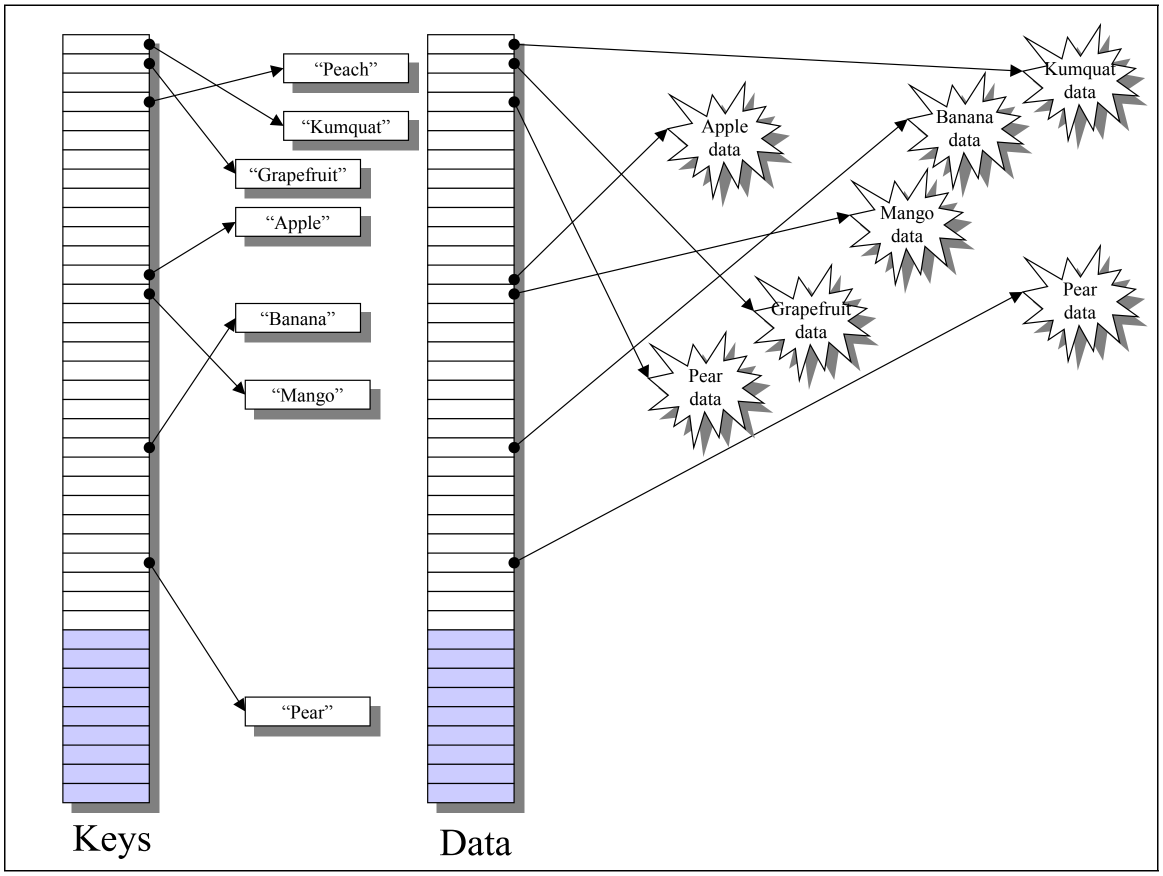
Example 14.10: Insensitive Sort Alphabetical Listing

14.6: Review and Questions

14.6.1: Review

Chapter 14 discusses the lookup table, a shape that supports content-based or keyed lookup. Such a shape can be implemented either using an underlying sorted array or tree shape—in which case lookup time tends grow logarithmically with the number of objects in the collections (consistent with binary search)—or using a hashing algorithm, in which case lookup time can be independent of collection size.

The hashing algorithm, as illustrated below, is frequently implemented using an integer lookup key, known as a hash code, to assign keyed objects to an array. A variety of algorithms can be used to generate such hash codes, typically depending on the data type and distribution characteristics of the data key being used (which could be string, numeric, or some other type). There are also many possible algorithms to deal with collisions, occurring where two keys produce the same hash code. These can be as simple as using the next open element in the array below the collision.



The STL provides a number of templates classes that implement lookup. These templates take a minimum of two parameters, the type of key being used (K) and the type of object being accessed (T). Key and object values are normally added and accessed using **pair<K,T>** template objects. In particular, iterators through STL lookup tables act like pointers to **pair<>** objects. The individual key and data values can of a **pair<>** object can be accessed using the .first and .second data members of the pair<> class.

Two other parameters are also available when creating STL lookup tables, 1) a comparison class that provides a () (function) overload to determine if one key is less than another, and 2) a allocator class that encapsulates how the data is added to the table, defaulting to the **pair<>** template.

STL classes implementing lookup tables include:

- **map<>**: Implements a unique key lookup using a sorted collection (logarithmic lookup)
- **hash_map<>**: Implements a unique key lookup using a hashing algorithm (size-independent lookup)
- **multimap<>**: Implements a keyed lookup with duplicate keys allowed using a sorted collection.
- **hash_multimap<>**: Implements a keyed lookup with duplicate keys allowed using a hashing algorithm.

14.6.2: Glossary

Allocator Class – A class that specifies how memory for a given templated object is to be allocated.
Bi-directional Iterator – An iterator that can be used front to back or back to front but cannot be used for random access
Collisions – A situation that occurs when two or more keys produce the same hash code in a hashing algorithm
Comparison Class – A class that allows the comparison operator to be used to compare lookup keys (or for sorting) to be specified by overloading the () (function) operator.
first Member – The member of a pair<> template object that returns the value of the key.
Function Operator – An alternative name for a () overload within a class, allowing objects of that class to be used as if they were function names. Often overloaded as part of template customization.
Hash Code – A numerical code, derived from a data object's key, that is used tomplace objects within a hash table.
Hashing Algorithm – An algorithm commonly used to implement lookup tables with access times that are size independent. Often implemented by placing objects and keys in an array at positions determined by a hash code.

hash_map<> – STL class template that implements a unique key lookup using a hashing algorithm (size-independent lookup)
hash_multimap<> – STL class template that implements a keyed lookup with duplicate keys allowed using a hashing algorithm.
Lookup Key (or Key) – A string, number or other form of identifier used to organize objects in a collection.
Lookup Table – A collection shape that implements content-based (keyed) lookup of objects.
map<> – STL class template that implements a unique key lookup using a sorted collection (logarithmic lookup)
multimap<> – STL class template that implements a keyed lookup with duplicate keys allowed using a sorted collection.
pair<K,T> – STL class template used for inserting key-value pairs into map-related classes
Reverse Iterator – An iterator that can be used to traverse a collection from back to front. By definition, a bi-directional iterator implies the existence of a reverse iterator.
second Member – The member of a pair<> template object that returns a reference to an abject associated with a given key (accessed using the first member of the pair).
Singly Linked List – A list shape allowing iteration in only one direction (normally forward).

14.6.3: Questions

14.1: Case insensitive GMapIStringToPtr class. Create a case insensitive version of the GMapStringToPtr class, called the GMapIStringToPtr class, using inheritance.

14.2: GMapIntToPtr class. Create version of the GMapStringToPtr class, called the GMapIntToPtr, that uses an integer key, instead of a string. For convenience, you may assume that the last four digits (at least) of each integer are relatively randomly distributed (e.g., as they would be for phone number or SSN-based integer keys).

14.3: GMapStringToPtr class using wraparound. Create version of the GMapStringToPtr class, called the GMapStringToPtrW, that uses wraparound (i.e., looks for empty cells starting at position 0), instead of an overflow area, when collisions occur at the bottom of the table.

14.4: GMapStringToPtr class with duplicate keys. Create version of the GMapStringToPtr class, called the GMultiMapStringToPtr, that allows multiple objects to be associated with the same key. Inherit from GMapStringToPtr and override members as necessary. In addition, add the following members:

```
bool Lookup(const char *key,vector<void*> &ar) const;
void RemoveKey(const char *key,const void *ptr);
```

The new Lookup() member should place all associations with a given key in its second argument (as opposed to placing the first such association). The RemoveKey() should remove the association with a given pointer, as opposed to all associations for a given key. You should also be able to call Lookup(const char *key,void* &p), which returns the *first* association, and the RemoveKey(const char *key), which removes *all* associations.

14.5: hash_map Word Counter. Modify the Word Counter lab (Section 14.8) to use a hash_map class.

14.6: Modified Word Counter frequency sort. The frequency count display (Figure 14.16) in the Word Counter lab (Section 14.8) has the annoying characteristic of displaying words in reverse alphabetical order within each frequency count. Modify the code so it does it correctly. (Hint: think about adding a customized third template argument that performs a more sophisticated less-than comparison).

Chapter 15

Recursion and Sorting

Executive Summary

Chapter 15 introduces the topic of recursion, the process by which a function calls itself. While the benefits of using recursion in programs tend not to be immediately obvious to most people, there are many complex programming tasks—such as evaluating general expressions typed in by a user—that are virtually impossible without recursion.

The chapter begins by introducing the concept of recursion and showing how it makes use of the program stack. We then consider two simple recursive arithmetic series, factorial and Fibonacci, and show how they can be implemented recursively. We then demonstrate how a recursive version of binary search can become the heart of an insertion sort algorithm. The fact that some many functions can be written both iteratively and recursively leads to a brief discussion of performance issues in choosing between recursive and iterative implementation of functions. The focus of the chapter then turns specifically to sorting. First, highly recursive the quicksort algorithm for sorting an array of values is presented in a walkthrough. Then the STL sorting and merging capabilities are introduced. To conclude the chapter, a lab exercise to create a recursive tokenizer is specified.

Learning Objectives

Upon completing this chapter, you should be able to:

- Explain what is meant by direct and indirect recursion
- Explain how recursion is implemented using a program stack
- Define recursive functions for simple mathematical series
- Define a recursive binary search function and use it as the basis of an insertion sort
- Discuss the benefits of recursion vs. iteration
- Use recursion to sort an array of values using Quicksort algorithm
- Use a variety of STL functions to sort standard collections
- Show how recursion can be used to interpret arbitrarily complex arithmetic expressions entered by a user

15.1: What is Recursion?

Whenever a function calls itself, we have recursion. Used effectively, such recursion can prove to be a powerful tool. There are, in fact, many situations where iterative programming--that is, programming that relies heavily on looping as a control structure--is unwieldy and can lead to programs nearly impossible to write and debug. Skilled use of recursion can often greatly simplify the programs in such situations. There are also many situations, such as using a grammar to validate and evaluate a user's freeform input, where the use of recursion is virtually mandated. For these reasons, experienced programmers find that knowledge of recursion is one of the most valuable assets in their programming bag of tricks.

Recursion and be direct or indirect. When a function call to itself exists within the body of a function, we have direct recursion. Often, however, a function will call another function that—eventually—ends up with the original function being called. In such cases, the recursion is indirect.

15.1.1: Recursion and the Program Stack

Walkthrough available in Recursion.avi

What happens when we call a function recursively on the computer? To understand this, we will begin by defining a recursive function Rstrlen() as follows:

```
int Rstrlen(const char *str)
{
if (str==NULL || *str=='\0') return 0;
else return 1+Rstrlen(1+str);
}
```

As is suggested by the name, this function is actually a recursive implementation of the strlen() library function. The way it works is as follows:

- If its argument is the NUL terminator (or a NULL pointer), it returns 0—since the length of an empty string is 0.
- Otherwise, it returns 1+Rstrlen(1+str).

To understand the second point, notice that strlen(str+1) is going to be 1 less than strlen(str) for any non-empty string str. For example, if str points to "Hello", then str+1 will point to "ello", str+2 will point to "llo", and so forth. It would therefore follow that if Rstrlen actually works like strlen(), Rstrlen(str+1) will return 1 less than Rstrlen(str).

What will actually happen is as follows, illustrated for Rstrlen("Hello!") in Figure 15.1. When the function is called with the argument "Hello!", it reaches the expression:

```
        return 1+Rstrlen(1+str);
```

Before it can return, it must evaluate the expression Rstrlen(str+1), leading to the second call. This continues until Rstrlen() is called with an empty string (""). In this case, the function can return 0 (illustrated by the arrows on the left). This allows the expression 1+Rstrlen("") to be evaluated, returning a 1 from Rstrlen("!"). This allows Rstrlen("o!") to return a 2, and so forth. Finally, Rstrlen("Hello!") is able to return a 6 and the recursion has ended.

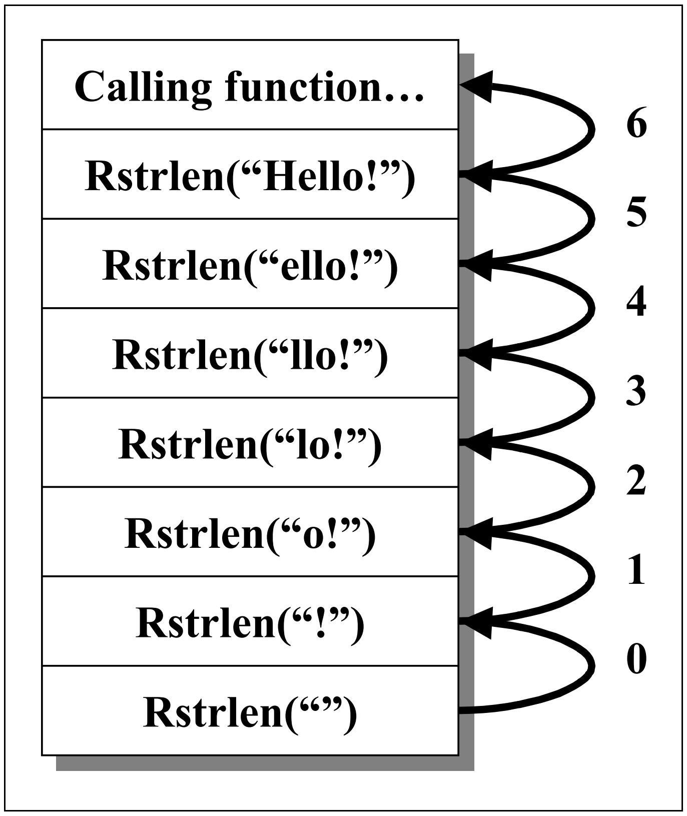


Figure 15.1: Stack during call of Rstrlen()

In the "real world", strlen() would be about the last function you'd ever want to use recursion for. The iterative implementation is small, efficient and easy to understand. Thus, the above definition is for illustrative purposes only.

15.1.2: Design of a Recursive Function

What general design principles apply to creating recursive functions? The major challenge most people experience in understanding recursion is not the inherent difficulty of the technique. Rather, the difficulty seems to stem from the fact that recursive solutions to problems are very different from iterative solutions. Thus, after spending many months honing their iterative skills, a programmer often comes to view problems in terms of a series of procedures that will lead to a solution. For example, if asked to write a function that takes a double and integer argument, X and N, and returns the value of X^N, the natural reaction is to think of a loop that could be used to multiply X by itself N times. Shortly afterwards, a piece of code something like:

```
for(i=0,total=1;i<N;i++) total=X*total;
```

suggests itself. The next step is to declare the proper variables then embed the code in a function.

Unfortunately, looking at the world in procedural terms makes designing recursive functions much more difficult. The recursive solution to a problem is much more like a declarative statement of known facts about the problem, rather than a statement of the procedure that needs to be taken in order to solve the problem. For example, in the power function just discussed, there are three facts that we know to be true about any integer power of a number:

1. If the value of the power (i.e., N) is 0, the result is 1.
2. If the value of the power is greater than 0, the value of function power(X,N) must be equal to the X * power(X,N-1).
3. If the value of the power (N) is less than 0, then the value of the function power(X,N) must be equal to 1/power(X,-1*N).

Taken together, these facts form a pattern that is very common in recursive problem solving. Specifically:

- One or more non-recursive cases is presented: In the example, Statement (1) is an identity—there is nothing recursive about it. In other words, it is a solution to a problem case for which no further computation is necessary.
- Recursive cases are presented which allow us to restate the problem in a way that is simpler (i.e., brings us closer to a non recursive case) than the original problem. Statements (2) and (3) both fall under this heading. Both are recursive cases, because they use the function itself in the definition. Both also move the problem closer to the non-recursive case, as well. For example, N-1 is closer to 0 than N, if N is positive. Similarly, -N is an improvement over N where N is negative because we know how to move a power closer to 0 if it is positive by applying statement 2.

Notice, however, that the three statements don't explicitly describe a procedure for writing the power function. Rather, they simply declare what we know to be true about the function. The power of recursion is that, by simply stating these facts in the form of a function, we can actually solve the problem. Such a solution is presented in the example below:

```
double power(double X, int N)
{
        if (N==0) return 1.0;
        else if (N>0) return X*power(X,N-1);
        else return 1.0/power(X,-1*N);
}
```

15.1 Section Questions

1. Given what you know about static variables (Chapter 6, Section 6.3.3), do you think that each copy of a recursive function on the stack will have its own copies of any locally declared variables?
2. What type of problem will defining a recursive function without any non-recursive cases always cause?
3. Does recursion only apply to OOP situations?
4. Recursive statements of problems are sometimes called declarative. How does this differ from how iterative problems are usually presented?
5. Will it always be obvious whether or not to use recursion in writing a function?

15.2: Simple Recursive Functions

Walkthrough available in SimpleRecurse.avi

We'll start our discussion of recursion by implementing some simple functions where we can envision both recursive and iterative solutions. We'll begin with two nurmeric series that can be evaluated recursively: the factorial function and the Fibonacci series. We'll then see how the binary search algorithm (introduced in Chapter 9, Section 9.3) can be implemented recursively and will use that as the basis for an insertion sort routine.

15.2.1: Factorial

The factorial function, used extensively in probability and statistics provides one of the most intuitive uses of recursion. Normally evaluated only for positive integers, N factorial (written as N!) is defined to be:

N! = N*(N-1)*(N-1)*...3*2*1

For example:

0! == 1 (By convention)
1! == 1
2! == 2*1 == 2
3! == 3*2*1 == 6
4! == 4*3*2*1 == 24
etc.

The above method of defining factorial suggests an iterative procedure for calculating N! involving a loop. We can, however, define the function in another way, using two declarations:

1. 0! = 1
2. N! = N*(N-1)!

These two statements, in contrast to the first definition, very clearly fit the recursive pattern previously discussed. The first statement provides us with a non-recursive case. The second statement is recursive (i.e., uses factorial to define itself) *and* moves us closer to a non-recursive case (i.e., N-1 is always closer to 0 than N for all positive N).

Using the conditional expression operator (? :), a single line recursive factorial() function can be constructed, as shown in the example below.

```
unsigned int Factorial(unsigned int nVal)
{
    return (nVal==0) ? 1 : nVal*Factorial(nVal-1);
}
```

Test your understanding: Rewrite the Factorial function using the if construct instead of the conditional (ternary) operator.

15.2.2: Fibonacci Series

A Fibonacci series is a sequence of numbers, beginning with 1 and 1, where each number in the sequence is the sum of the previous two numbers. For example, the first 10 elements of the series are:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55

with Fib(0)==1, Fib(1)==1, Fib(2)==2, Fib(3)==3, Fib(4)==5, etc.

Test your understanding: What are Fib(10) and Fib(11)?

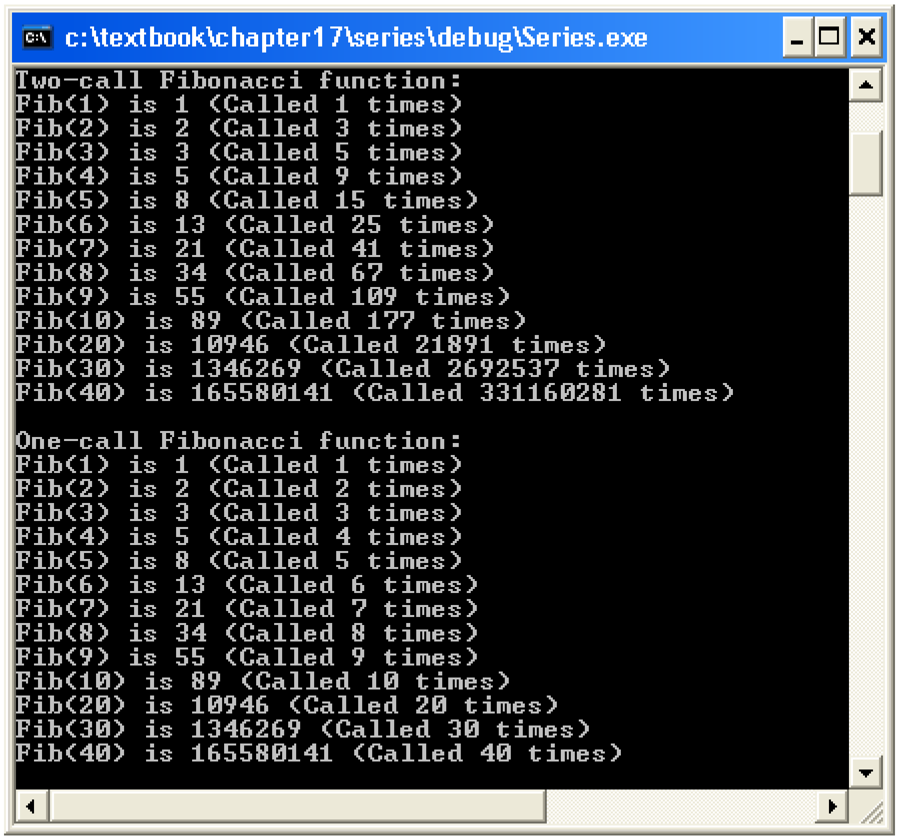
We can also write a declarative description of the Fibonacci series, namely:

1. Fib(i) == 1, for i<2
2. Fib(i) == Fib(i-1)+Fib(i-2), for i>=2

This definition would allow us to write a recursive definition of the function as follows:

```
unsigned int Fib(unsigned int N)
{
return (N<2) ? 1 :  Fib(N-1)+Fib(N-2);
}
```

While this is a very short and clear function—entirely consistent with the definition—it also has a serious weakness. Every time we encounter a recursive case, we call the function twice—in the line Fib(N-1)+Fib(N-2). Intuitively, however, we know that Fib(N-2) must have been calculated in the process of calculating Fib(N-1). Moreover, each time we go down a level—towards the non-recursive case—we spawn two more calls. As a result, the number of unnecessary calls will be staggering for larger values of N. This is illustrated in Figure 15.2, where the 2-call version of the function is shown to generate over 331 million function calls in computing Fib(40).



```
Two-call Fibonacci function:
Fib(1) is 1 (Called 1 times)
Fib(2) is 2 (Called 3 times)
Fib(3) is 3 (Called 5 times)
Fib(4) is 5 (Called 9 times)
Fib(5) is 8 (Called 15 times)
Fib(6) is 13 (Called 25 times)
Fib(7) is 21 (Called 41 times)
Fib(8) is 34 (Called 67 times)
Fib(9) is 55 (Called 109 times)
Fib(10) is 89 (Called 177 times)
Fib(20) is 10946 (Called 21891 times)
Fib(30) is 1346269 (Called 2692537 times)
Fib(40) is 165580141 (Called 331160281 times)

One-call Fibonacci function:
Fib(1) is 1 (Called 1 times)
Fib(2) is 2 (Called 2 times)
Fib(3) is 3 (Called 3 times)
Fib(4) is 5 (Called 4 times)
Fib(5) is 8 (Called 5 times)
Fib(6) is 13 (Called 6 times)
Fib(7) is 21 (Called 7 times)
Fib(8) is 34 (Called 8 times)
Fib(9) is 55 (Called 9 times)
Fib(10) is 89 (Called 10 times)
Fib(20) is 10946 (Called 20 times)
Fib(30) is 1346269 (Called 30 times)
Fib(40) is 165580141 (Called 40 times)
```

Figure 15.2: Fibonacci series and number of function calls

This particular explosion in function calls can be resolved by figuring out a way to bring both the value of Fib(N) and Fib(N-1) out of the function when it is called. This can be accomplished by adding another argument to the recursive function. Since such an argument is inconvenient from the perspective of the programmer using of the function, we can hide it by creating a function with one argument (sometimes called a "driver function") and a second function that actually performs the recursion. This is illustrated by Fib1() and Fib2() in Example 15.1, where Fib1() provides a local integer whose address can be passed into Fib2 as required by the declaration (and is then discarded).

Example 15.1: One-call recursive version of Fibonacci series

```
// Fib1: driver function
unsigned int Fib1(unsigned int N)
{
	int nVal;
	return Fib2(N,&nVal);
}

// Fib2: recursive function, passing out N-1 value in pInt
unsigned int Fib2(unsigned int N,int *pInt)
{
	int nLast;
	if (N==0) {
		*pInt=0;
		return 1;
	}
	else if (N==1) {
		*pInt=1;
		return 1;
	}
	*pInt=Fib2(N-1,&nLast);
	return *pInt+nLast;
}
```

The Fib2() function is somewhat more complicated than our 2-call Fib() function but—once you're used to recursion—is still quite simple. It treats three cases separately:

- N==0. Returns 1 and sets the pInt integer pointer value to 0—although doing so is unnecessary as the only way to get to the 0 case is a call to Fib1(0).
- N==1. Returns 1 and sets the pInt integer pointer value to 1 (which is the value of Fib(0)—the previous element of the sequence).
- N>1. Calls Fib2(N-1,&nLast) to compute the series value for Fib(N-1)—which means the Fib(N-2) value is placed in nLast—and sets the pInt integer pointer value to the return value. Then returns the sum of *pInt (which is the Fib(N-1) value) and nLast (which is the Fib(N-2) value)—since the value of Fib(N) is just Fib(N-1)+Fib(N-2).

As shown in Figure 15.2, the resulting function dramatically reduces the number of calls—since each element of the series only gets computed once. The performance difference is extraordinary.

15.2.3: Insertion Sort

Insertion sort, first introduced in PersonArray class in Chapter 8, Section 8.3.3, is another example of an algorithms that can be implemented either recursively or iteratively. The basic principle behind the algorithm is simple:

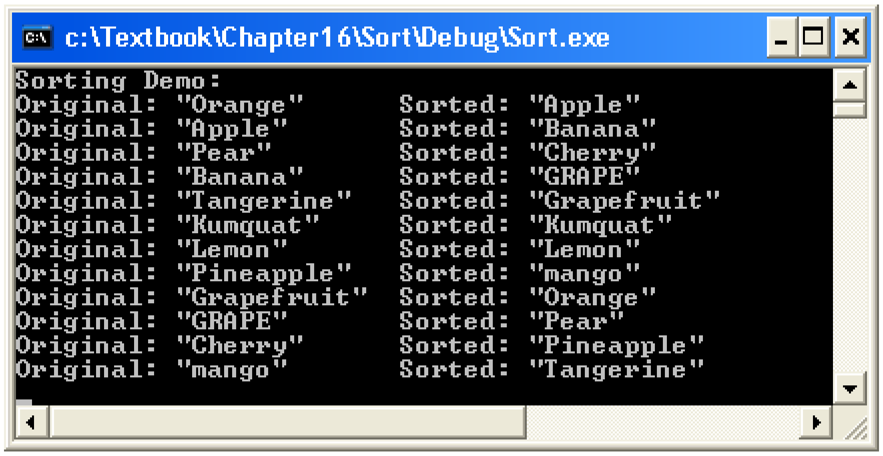
- We have a collection of elements that need to be sorted
- We have a binary search routine that tells us where to place elements in a sorted array
- We take each element from the unsorted collection and add it to an array (which starts out empty), using binary search to tell us where to place the element
- Once we've added all the elements, we're done.

Example 15.2 shows a collection of functions used to illustrate a case-insensitive insertion sort into a vector<string> collection.

Example 15.2: Insertion sort demonstration

```
void InsertDemo()
{
	int i;
	vector<string> arFruit;
	arFruit.push_back("Orange");
	arFruit.push_back("Apple");
	arFruit.push_back("Pear");
	arFruit.push_back("Banana");
	arFruit.push_back("Tangerine");
	arFruit.push_back("Kumquat");
	arFruit.push_back("Lemon");
	arFruit.push_back("Pineapple");
	arFruit.push_back("Grapefruit");
	arFruit.push_back("GRAPE");
	arFruit.push_back("Cherry");
	arFruit.push_back("mango");
	vector<string> arSorted;
	SortArray(arFruit,arSorted);
	cout << "Sorting Demo:" << endl;
	for(i=0;i<12;i++) {
		cout << "Original: \"" + arFruit[i]+ "\"\tSorted: \""
			+ arSorted[i] + "\"" << endl;
	}

}
void SortArray(const vector<string> &arUnsorted,vector<string> &arSorted)
{
	int i;
	for(i=0;i<(int)arUnsorted.size();i++) {
		int nPos=BinarySearchI(arSorted,arUnsorted[i].c_str(),0,i);
		InsertString(arUnsorted[i].c_str(),arSorted,nPos);
	}
}
int BinarySearchR(const vector<string> &arVals,const char *szTarget,
                          int nStart,int nEnd)
{
	int nTest=(nStart+nEnd)/2,nFlag;
	if (nEnd==nStart) return nEnd;
	nFlag=stricmp(arVals[nTest].c_str(),szTarget);
	if (nTest==nStart || nFlag==0)  return (nFlag>=0) ? nTest : nEnd;
	return (nFlag > 0) ?	BinarySearchR(arVals,szTarget,nStart,nTest) :

	BinarySearchR(arVals,szTarget,nTest+1,nEnd);


}
void InsertString(const char *szStr,vector<string> &arVals,int nPos)
{
	arVals.push_back("temp");
	for(int i=arVals.size()-1;i>nPos;i--) arVals[i]=arVals[i-1];
	arVals[nPos]=szStr;
}
```

The results of running the test function, InsertDemo() are shown in Figure 15.3.



```
Sorting Demo:
Original: "Orange"      Sorted: "Apple"
Original: "Apple"       Sorted: "Banana"
Original: "Pear"        Sorted: "Cherry"
Original: "Banana"      Sorted: "GRAPE"
Original: "Tangerine"   Sorted: "Grapefruit"
Original: "Kumquat"     Sorted: "Kumquat"
Original: "Lemon"       Sorted: "Lemon"
Original: "Pineapple"   Sorted: "mango"
Original: "Grapefruit"  Sorted: "Orange"
Original: "GRAPE"       Sorted: "Pear"
Original: "Cherry"      Sorted: "Pineapple"
Original: "mango"       Sorted: "Tangerine"
```

Figure 15.3: Results of running insertion sort demonstration

Three of the functions are straightforward iterative functions. Their purpose and basic operation are as follows:

- *InsertDemo()*: sets up an array of 12 strings (arFruit[], names of fruits), calls SortArray() to cause it to be sorted into arSorted[], then performs a loop to display the results.
- *SortArray(char *arUnsorted[],char *arSorted[],int nCount)*: calls each element of arSorted[], which has nCount elements, and inserts it into arSorted[], using BinarySearchI() to find the proper position and InsertString() to perform the insertion.
- *InsertString(const char *szStr,char *arVals[],int nPos,int nMax)*: Takes a string szStr, and inserts it into an array of string pointers (arVals[], containing nMax elements) at position nPos. If the insertion is not taking place at the end (i.e., where nPos==nMax), it calls memmove to move the elements above the insertion point up a position in the array.

The recursive function, and the key to the entire routine, is the BinarySearchR() function. Its purpose and arguments are identical to those of the iterative version of the function (as presented in Chapter 8, Section 8.3.3) , except that we've chosen to use an array of sorted strings, instead of integers. The function is prototyped as follows:

```
int BinarySearchR(vector<string> &arVals[],const char *szTarget,
                                    int nStart,int nEnd);
```

where arVals is the vector to be searched, szTarget is the string we are looking for, nStart is the lower bound of our search, and nEnd is the upper bound. The return value of the function is either: a) the location in arVals where a matching string was found, or b) a suitable location for inserting a string with the target value.

The recursive description of the problem is as follows (where our "test element" is the element half way between the upper and lower bound):

1. If our upper and lower bounds are at the same position (i.e., nEnd-nStart), we are done and return the upper bound (first non-recursive case).
2. If the value at our test element matches what we are looking for (i.e., nFlag==0), we are done and return the position of the test element (second non-recursive case)
3. If the test element is at the same position as the lower bound (i.e., nTest==nStart), and it is less than the value we are looking for, return the upper bound (third non-recursive case).
4. If the value at our test element is greater than the value we are looking for (i.e., nFlag>0), we return the results of a recursive function call on the range from our lower bound to the position of our test element.
5. If the value at our test element is less than the value we are looking for (i.e., nFlag<0), we return the results of a recursive function call on the range from our test element +1 to the upper bound.

Because this example is a case-insensitive search, we use stricmp() to compare the values of strings.

Test your understanding: If the third non-recursive case is true (item (3) above), what does it tell us about the relationship of the upper bound and lower bound?

3.2 Section Questions

1. Why did we set up a driver function (Fib1) in Example 15.1?

2. Can you entirely ignore performance in formulating complex recursions?

3. Why does passing an array as an argument in recursive function that is a series often help to avoid recursion inefficiencies?

4. Knowing what you know about objects and collections, what might be an OOP alternative to passing in a table in Example 15.1.

5. What are the two non-recursive cases in our insertion sort routine?

6. What was recursive in our insertion sort routine? Did it need to be?

15.3: Recursing Collections

Recursion is nearly always an alternative to iteration when sequential collections, such as **vector<>** and **list<>** collections, are involved. When recursively traversing a collection, the form of the function is frequently of the form:

- When we reach the end of the collection (in whatever direction we are going), return.
- Otherwise, do something with the current element and call the function recursively on the remainder of the collection.

Normally, iterators will be passed as arguments to such functions when STL classes are used.

15.3.1: Recursing a Vector

To illustrate recursing a vector collection in both directions, the three functions to display a simple list of integers are presented in Example 15.3.

Example 15.3: Recursive functions for traversing a vector

```
void IntVector::DisplayForward(vector<int>::iterator nPos) const
{
      if (nPos==end()) return;
      cout << *nPos;
      ++nPos;
      if (nPos!=end()) cout << "->";
      DisplayForward(nPos);
}
void IntVector::DisplayBackward(vector<int>::iterator nPos) const
{
      if (nPos==end()) --nPos;
      cout << *nPos;
      if (nPos==begin()) return;
      cout << "<-";
      DisplayBackward(--nPos);
}

void IntVector::DisplayBothWays(vector<int>::iterator nPos) const
{
      if (nPos==end()) DisplayBackward(nPos);
      else if (nPos==begin()) DisplayForward(nPos);
      else {
            vector<int>::iterator nMiddle=nPos;
            DisplayBackward(--nPos);
            cout << endl << *nMiddle << endl;
            DisplayForward(++nMiddle);
      }
}
```

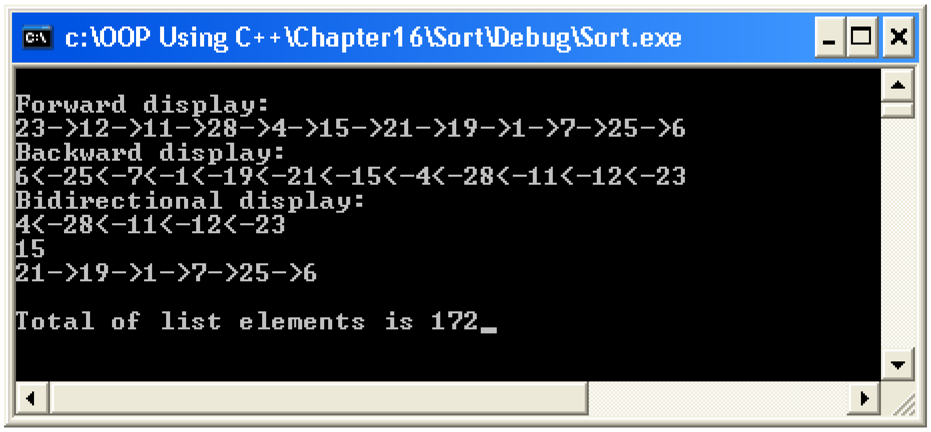
In the first two of these functions, the basic model for recursing through a collection was followed: a test for the non-recursive case, followed by an action on the current element, follows by a recursive call to the function. In the third, DisplayBothWays(), the purpose was to demonstrate that both functions work from the middle of the collection.

Examples of how the functions would be called are presented in Example 15.4.

Example 15.4: Demonstration function for IntVector class

```
void IntVectorDemo()
{
	int arUnsorted[]={23,12,11,28,4,15,21,19,1,7,25,6};
	IntVector arVals;
	int i;
	for(i=0;i<12;i++) {
		arVals.push_back(arUnsorted[i]);
	}
	cout << endl << "Forward display:" << endl;
	arVals.DisplayForward(arVals.begin());
	cout << endl << "Backward display:" << endl;
	arVals.DisplayBackward(arVals.end());
	IntVector::iterator nPos=arVals.begin();
	// Moving to the middle of the vector
	for(i=0;i<5;i++) nPos++;
	cout << endl << "Bidirectional display:" << endl;
	arVals.DisplayBothWays(nPos);
	cout << endl << endl <<
		"Total of vector elements is " << arVals.Sum(nPos);
	return;
}
```

The output resulting from running IntVectorDemo() is shown in Figure 15.4.



```
Forward display:
23->12->11->28->4->15->21->19->1->7->25->6
Backward display:
6<-25<-7<-1<-19<-21<-15<-4<-28<-11<-12<-23
Bidirectional display:
4<-28<-11<-12<-23
15
21->19->1->7->25->6

Total of list elements is 172_
```

Figure 15.4: Output from IntVectorDemo()

15.3.2: Recursing in Two Directions

A common way of setting up a recursive algorithm is to define a driver function that makes the necessary recursive calls. When dealing with linked lists and other self-referential structures, its not unusual to need such a drive function to start recursive calls going in more than one direction. Such a driver function is illustrated by the Sum () function, in Example 15.5, which sums the elements in an IntList no matter where the starting point is.

Example 15.5: Recursive functions to sum a list in two directions

```
int IntVector::Sum(vector<int>::iterator nPos) const
{
      return (nPos==begin()) ? SumForward(nPos) :
                                          SumForward(nPos)+
SumBackward(--nPos);
}
int IntVector::SumForward(vector<int>::iterator nPos) const
{
      return (nPos==end()) ? 0 : *nPos+SumForward(++nPos);
}
int IntVector::SumBackward(vector<int>::iterator nPos) const
{
      return (nPos==begin()) ? *nPos : *nPos+SumBackward(--nPos);
}
```

15.3 Section Questions

1. Why do the begin() and end() have to be handled differently from each other in recursive functions that traverse collections?
2. Is there any advantage of using recursion over iteration when traversing collections?
3. What changes would need to be made to these functions to traverse a list<int> collection, assuming that list<int> supports the same iterators?
4. Would the DisplayBothWays() function be considered an example of indirect recursion?
5. What would you need to do to change the Sum() series of functions into a Count() series of functions that counts the number of elements in a collection?

15.4: Recursion vs. Iteration

For all iterative functions, a recursive version can be constructed (and vice-versa). So how do we choose which one to write? The two most relevant concerns are clarity and performance.

15.4.1: Clarity

In today's age of high performance computers—even on the desktop—the most critical issue in choosing how to write a function is likely to be clarity. For some programming techniques, such as writing expression translators and working with tree collections, recursion is the natural way to express the programming problem. For other situations, such as our Rstrlen() function, using iterative approaches is much more natural.

15.4.2: Performance

Because of the function calls involved, a recursive solution will generally exhibit slightly lower performance than a "clean" iterative approach to the same problem. We have de-emphasized performance issues throughout this text on the assumption that: a) getting an application up and running quickly is of paramount importance—after which time performance issues can be addressed if necessary, and b) many of the techniques we are introducing will normally be implemented using standard or commercial libraries or through calls to external applications, such as databases—in which case knowing what needs to be done is what is mainly critical (and implementation details can be left to the library designers). Having said this, we should avoid incorporating functions into our code that are obviously wasteful of the processors time.

As we saw in our Fibonacci example (Section 15.2.2), it is possible for recursive functions to explode computationally. In fact, many types of problems for which we use recursion (e.g., language translation, playing games such as chess) are notorious for their computational complexity. Such complexity has proven to be a major stumbling block in the field of artificial intelligence—and there is no obvious cure. As a result, you need to be aware that "simple" recursive solutions to problems may have to be rewritten or tuned (e.g., as we demonstrated for the Fibonacci problem). In some cases, we may even have to fall back on iteration.

15.4 Section Questions

1. If performance is usually better for iterative techniques, why bother covering recursion at all?

2. If you implement a function recursively, does that mean it will have to remain recursive throughout the lifecycle of the application?

3. Is there any disadvantage to using recursive functions aside from performance?

15.5: Quicksort Walkthrough

Walkthrough available in Quicksort.avi

In Chapter 11, Section 11.5, an exercise to implement a swap-sort (a.k.a., bubble sort) was presented. Swap-sort, however, is so legendary in its inefficiency that you would be well advised never to use it in your code. Other programmers will laugh at you.

The Quicksort algorithm performs the same task—sorting an array—but with far greater efficiency. In this section, we'll walk through a highly recursive implementation of the Quicksort function.

15.5.1: Sorting and the Quicksort algorithm

It is useful to consider the properties of the swap-sort algorithm in explaining the rationale behind Quicksort. As you may recall, a typical swap-sort involves going through an array and swapping adjacent elements that are not in order. Because each pass guarantees that the largest element will reach the top of the area being sorted, we can reduce the number of elements being sorted by one every time we do another pass. As a result, our theoretical maximum number of swaps during sorting is roughly:

$$N+(N-1)+(N-2)+\ldots 2+1$$
$$\sim N^2/2$$

In other words, our sort time is going to be roughly proportional to N^2, where N is the number of elements being sorted.

Intuitively, however, we all know that it is easier to sort a lot of small piles than it is to sort a single large pile. It turns out that this observation is also justified by our formula. If we break our space into two piles, of N/2 elements each, the total sorting time will be:

$$(N/2)^2 + (N/2)^2 = N^2/2$$

In other words, it will take us half as long to sort two small piles as it will a single large pile. Furthermore, we can continue to break the piles down into smaller and smaller files. For example:

$(N/4)^2 + (N/4)^2 + (N/4)^2 + (N/4)^2 = N^2/4$
$(N/8)^2 + (N/8)^2 + (N/8)^2 + (N/8)^2 + (N/8)^2 + (N/8)^2 + (N/8)^2 + (N/8)^2 = N^2/8$
etc.

The problem with this approach is that once we've got the sorted small piles, we have to recombine them into a single large pile—which will take us some time. What would really be good would be if we could separate the higher and lower elements into separate piles before we start the sorting. Then we can combine them by stacking them on top of each other. For example, the way most of us sort a deck of cards is to separate the suits before sorting the cards within each suit.

The Quicksort algorithm, which is used to sort an array, basically takes this approach. The way it works is as follows:

- It takes a single element, that we'll call the partitioning element, which is used to divide the array into two piles.
- During each pass, it compares each element with the partitioning element. If the value is less than the partitioning element it goes in the left "pile", otherwise, it goes in the right "pile".
- It moves the partitioning element to the position in the array between the two "piles". This leaves the array in the following state:
    - Everything to the left of the portioning element is less than that element
    - Everything to the right of the partitioning element is greater than or equal to the element in value
    - The partitioning element is then guaranteed to be in its final "sorted" position.
- The left and right sides of the partitioning elements, our two "piles", are then sorted independently, using recursive calls to Quicksort.

To get an idea of the rough performance of this approach, we can make the following observations:

- During each "pass" of the array, we need to look at N elements
- The number of times we can break our array into halves without before getting down to single element element piles is our old friend Log(N).

Roughly speaking, then, our sorting performance is going to be N*Log N vs. N^2 for swap-sort. How big a difference this makes is illustrated in Table 15.1, which assumes a particular sort of 1000 elements takes 1 second for both swap-sort and Quicksort.

Number of elements	Swap-sort time	Quicksort time
1000	1 (assumed)	1 (assumed)
10000	~100 sec.	~13 sec.
100000	~3 hours	~3 min.
1000000	~11 days	~33 min

Table 15.1: Example sort time results

We will now examine an actual version of the function, used to sort an integer array.

15.5.2: The QuickSort() function

To implement the quicksort algorithm, a QuickSort function for sorting a vector<int> array of integers has been developed. There are two versions of the function:

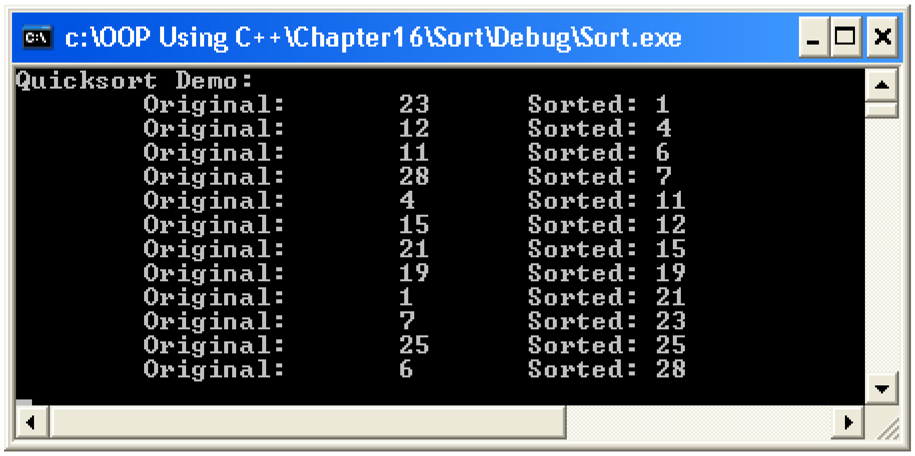
- A driver function, that just takes the vector<int> reference, then calls the recursive version.
- The recursive version, which takes three arguments: the name of the array being sorted (arVals), the starting position of the sort region (nStart) and the ending position of the sort region (nEnd).

In addition, simple swap function (taking comparable arguments) is defined for swapping element in an integer array. A demonstration function that calls QuickSort() is presented in Example 15.6.

Example 15.6: QSortDemo Function

```
void QSortDemo()
{
      int arUnsorted[]={23,12,11,28,4,15,21,19,1,7,25,6};
      vector<int> arVals;
      int i;
      for(i=0;i<12;i++) {
            arVals.push_back(arUnsorted[i]);
      }
      QuickSort(arVals);
      cout << "Quicksort Demo:" << endl;
      for(i=0;i<12;i++) {
            cout << "\tOriginal:\t" << arUnsorted[i] <<
                  "\tSorted:\t" << arVals[i] << endl;
      }
      return;
}
```

The output of calling QSortDemo is presented in Figure 15.5.



```
Quicksort Demo:
        Original:       23      Sorted: 1
        Original:       12      Sorted: 4
        Original:       11      Sorted: 6
        Original:       28      Sorted: 7
        Original:       4       Sorted: 11
        Original:       15      Sorted: 12
        Original:       21      Sorted: 15
        Original:       19      Sorted: 19
        Original:       1       Sorted: 21
        Original:       7       Sorted: 23
        Original:       25      Sorted: 25
        Original:       6       Sorted: 28
```

Figure 15.5: Output from calling QSortDemo

The functions used to implement the quicksort algorithm are presented in Example 15.7. The heart of the process is the 3-argument recursive QuickSort() function. The function begins by arbitrarily choosing a partition element in the middle of the array, at position (nStart+nEnd)/2. The function then returns in the event of the non-recursive case, where the end position is less than or equal to the start position—since an array with one element or less can't be sorted.

Example 15.7: QuickSort Functions for Sorting a vector<int>

```
void QuickSort(vector<int> &arVals)
{
      QuickSort(arVals,0,arVals.size());
}
void QuickSort(vector<int> &arVals,size_t nStart,size_t nEnd)
{
      size_t nPos,i;
      size_t nP=(nStart+nEnd)/2;
      if (nEnd-nStart<=1) return;
      Swap(arVals,nStart,nP);
      nPos=nStart;
      for(i=nStart+1;i<nEnd;i++) {
            if (arVals[i]<arVals[nStart]) Swap(arVals,++nPos,i);
      }
      Swap(arVals,nStart,nPos);
      QuickSort(arVals,nStart,nPos-1);
      QuickSort(arVals,nPos+1,nEnd);
}
void Swap(vector<int> &arVals,size_t n1,size_t n2)
{
      int nTemp=arVals[n1];
      arVals[n1]=arVals[n2];
      arVals[n2]=nTemp;
}
```

The function then enters the iterative process of dividing our array into two piles. The process takes place as follows:

- It begins by putting our partitioning element at the start of the array with a Swap() function call. What this accomplishes is to get it out of the way as we being separating our data.
- It then set nPos, which keeps track of where our partitioning element will ultimately end up, equal to nStart.
- It then loops through the remaining elements in the array, from nStart+1 to nEnd. It compares each element to the value of the portioning element (which, as you may recall, is now at arVals[nStart]). In the event the element's value is less than that of the partitioning element, it needs to be moved it into the left side "pile". This is done by:
    - Incrementing the nPos pointer, which keeps track of where our partitioning element will be
    - Swapping the element it is looking at with the element at the nPos pointer. Because nPos starts at the partitioning element, and only gets incremented when a value less than the partitioning element is encountered, this guarantees that all values in the array at position nPos or less will be less than the partitioning element—with the exception of the partitioning element itself.
- Upon completing the loop, the function swaps the partitioning element (at position nStart) with whatever value happens to be at nPos in the array—knowing that value will be less than the partitioning element as a result of the loop. At this point, we know the following:
    - All elements below nPos in the array are less than the partitioning element
    - All elements above nPos are greater than or equal to the partitioning element value.
    - The partitioning element is at position nPos, and that is the correct position for it in the final sorted array.
- The function then sorts the left portion of the array from nStart to nPos-1 by recursively calling QuickSort.
- Finally, the function sorts the right portion of the array from nPos+1 to nEnd by recursively calling QuickSort.

Upon completion of the second sort, we know that a) everything to the left of our partitioning element is sorted, c) everything to the right of it is sorted and c) the partitioning element is in the right place. It follows that we are done.

15.5.3: The TQuickSort() Template Function

In looking at the QuickSort function, you might have noticed something interesting: not a single thing in the function actually depended on the fact that we were sorting integers. In a situation like this, the opportunity to define a template function (see Chapter 13, Section 13.3) is great.

In fact, we need to define three template functions—but it doesn't really matter, since the process is almost trivial. We can see how little effort is required to implement the templates by looking at Example 15.8. All we needed to do was:

- Put template<class T> in front of each definition
- Replace vector<int> with vector<T>
- Move the functions to the .h file

To avoid confusion, the names of the three templated functions were also preceded with a T, but this wasn't necessary. Indeed, in this situation, creating a templated version of QuickSort is so superior to a single case version that we'd just throw our vector<int> version out.

Example 15.8: Template QuickSort function versions

```
template<class T>
void TQuickSort(vector<T> &arVals)
{
      TQuickSort(arVals,0,arVals.size());
}
template<class T>
void TQuickSort(vector<T> &arVals,size_t nStart,size_t nEnd)
{
      size_t nPos,i;
      size_t nP=(nStart+nEnd)/2;
      if (nEnd-nStart<=1) return;
      TSwap(arVals,nStart,nP);
      nPos=nStart;
      for(i=nStart+1;i<nEnd;i++) {
            if (arVals[i]<arVals[nStart]) TSwap(arVals,++nPos,i);
      }
      TSwap(arVals,nStart,nPos);
      TQuickSort(arVals,nStart,nPos-1);
      TQuickSort(arVals,nPos+1,nEnd);
}
template<class T>
void TSwap(vector<T> &arVals,size_t n1,size_t n2)
{
      T nTemp=arVals[n1];
      arVals[n1]=arVals[n2];
      arVals[n2]=nTemp;
}
```

What does creating a templated TQuickSort function buy us? It allows us to sort any type of vector object collection, provided:

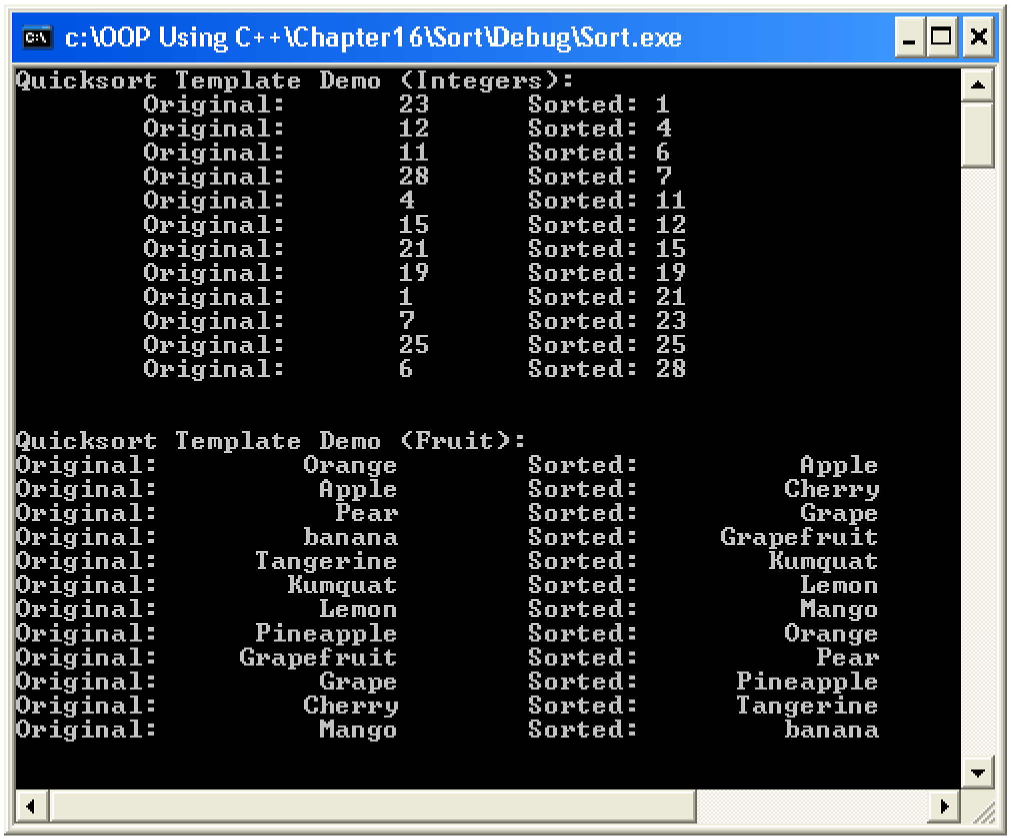
- The object has an overloaded assignment operator (needed for the TSwap() function)
- The object has an overloaded < operator, needed for TQuickSort()

The way the templated function is called is identical to our old QuickSort function, as shown in Example 15.9, where it is used for both int values and **string** values.

Example 15.9: TSortDemo() function

```
void TSortDemo()
{
	// First, we do it with integers
	int arUnsorted[]={23,12,11,28,4,15,21,19,1,7,25,6};
	vector<int> arVals;
	int i;
	for(i=0;i<12;i++) {
		arVals.push_back(arUnsorted[i]);
	}
	TQuickSort(arVals);
	cout << "Quicksort Template Demo (Integers):" << endl;
	for(i=0;i<12;i++) {
		cout << "\tOriginal:\t" << arUnsorted[i] <<
			"\tSorted:\t" << arVals[i] << endl;
	}
	char *arUFruit[12] = {"Orange", "Apple", "Pear", "banana", "Tangerine", "Kumquat",
		"Lemon", "Pineapple", "Grapefruit", "Grape", "Cherry", "Mango"};
	vector<string> arFruit;
	for(i=0;i<12;i++) {
		arFruit.push_back(arUFruit[i]);
	}
	TQuickSort(arFruit);
	cout << endl << endl << "Quicksort Template Demo (Fruit):" << endl;
	for(i=0;i<12;i++) {
		cout << "Original:";
		cout.width(15);
		cout << arUFruit[i] << "\tSorted:";
		cout.width(15);
		cout << arFruit[i] << endl;
	}
}
```

The output of running the function is presented in Figure 15.6. In looking at the output, you may notice that "banana" appears at the end of the display—instead of in between "Apple" and "Cherry". This is, of course, a result of case-sensitivity (i.e., any lower case character comes after all upper case characters). If case insensitivity is desired, an instantiated template function can be used, presented as an end-of-chapter exercise.



```
Quicksort Template Demo (Integers):
        Original:       23      Sorted: 1
        Original:       12      Sorted: 4
        Original:       11      Sorted: 6
        Original:       28      Sorted: 7
        Original:       4       Sorted: 11
        Original:       15      Sorted: 12
        Original:       21      Sorted: 15
        Original:       19      Sorted: 19
        Original:       1       Sorted: 21
        Original:       7       Sorted: 23
        Original:       25      Sorted: 25
        Original:       6       Sorted: 28


Quicksort Template Demo (Fruit):
Original:          Orange       Sorted:          Apple
Original:           Apple       Sorted:         Cherry
Original:            Pear       Sorted:          Grape
Original:          banana       Sorted:     Grapefruit
Original:       Tangerine       Sorted:        Kumquat
Original:         Kumquat       Sorted:          Lemon
Original:           Lemon       Sorted:          Mango
Original:       Pineapple       Sorted:         Orange
Original:      Grapefruit       Sorted:           Pear
Original:           Grape       Sorted:      Pineapple
Original:          Cherry       Sorted:      Tangerine
Original:           Mango       Sorted:         banana
```

Figure 15.6: TSortDemo() function output

15.5 Section Questions

1. In the quicksort algorithm, is there any need to take the partitioning element from the center of the range being sorted?

2. How would you apply quicksort to a collection of objects that you wanted to sort in many different ways (e.g., employees sorted by name, ID, salary, years at the company)?

3. If an array has just one element out of order, which will be more efficient: quicksort or swap sort?

4. Could quicksort be rewritten without any iteration?

5. How hard would it be to adapt quicksort to a list shape?

15.6: STL Sorting

Walkthrough available in STLSort.avi

Although the quicksort algorithm presented in Section 15.5 does a reasonably efficient job, when you use STL templates for your collections, or inherit from STL templates, you can often use STL functions specifically optimized for sorting and other collection management functions. In this section, we look at how to use some of these powerful tools.

15.6.1: Sorting Techniques

As part of its <algorithm> library, the STL provides a function that can be used to sort collections with the same type of NLog(N) scaling provided by quicksort. One version is provided for collections supporting random access, with a second version provided specifically for list sorting.

Random Access Sorting
To use the sort() function on a random access collection, such as a **vector<>** or **deque<>**, you'll need to have one, and possibly two libraries available, i.e.,

```
#include <algorithm>
#include <functional> // may be required
using namespace std;
```

There are two versions of the sort() function, which is a template function, available:

void sort(*iterator-begin*,*iterator-end*);
void sort(*iterator-begin*,*iterator-end,comparison-function-pointer*);

The first version takes an iterator range, specified in terms of the beginning iterator and the iterator one past the end of the included range (e.g., coll.begin() and coll.end() for some STL collection *coll*). It then sorts the collection, the elements of which must support the < operator.

For example to sort a vector of integers, we'd have code something like the following:

```
vector<int> arInt;
// some omitted initialization
sort(arInt.begin(),arInt.end());
```

The second version uses a capability that we have not used before—passing function pointers. The problem it addresses is the following: suppose the < operator available for our collection does not provide us with the order that we want? We've already seen this in the case of strings—we may want case insensitive ordering, but that is not what C++ gives us. So how do we adapt sort() to meet those needs?

There are two approaches offered in C++ to non-standard sorting. In some cases, if we simply want a reverse sort. In this case, we can call one of the pre-created STL functions (in the <functional> library header) that returns the appropriate pointer. For example:

```
    sort(arInt.begin(),arInt.end(),greater<int>());
```

This call does two things:

- It calls the greater<int>() function, which returns a function pointer to a greater than function for integers.
- The greater than function supplied tells the sort() function to swap elements if the second element is greater than the first. This has the effect of creating a reverse sort.

A second, even more flexible approach, is to write your own function. Suppose, for example we defined the following simple case-insensitive less-than function for **string** objects:

```
bool InsensitiveLT(const string &s1,const string &s2)
{
    return (stricmp(s1.c_str(),s2.c_str())<0);
}
```

We could then supply the name of the function as the third argument to sort(). By passing the name of the function, we are actually passing a pointer to the address of the function—which C++ is able to handle just like any other pointer. In fact, C++ even type-checks the function, making sure:

1. The function returns **bool**
2. The function has two arguments
3. The function's arguments match the type of data in the collection being sorted

Thus, the following code could be used:

```
    vector<string> arString;
    // some omitted initialization
    sort(arString.begin(),arString.end(),InsensitiveLT);
```

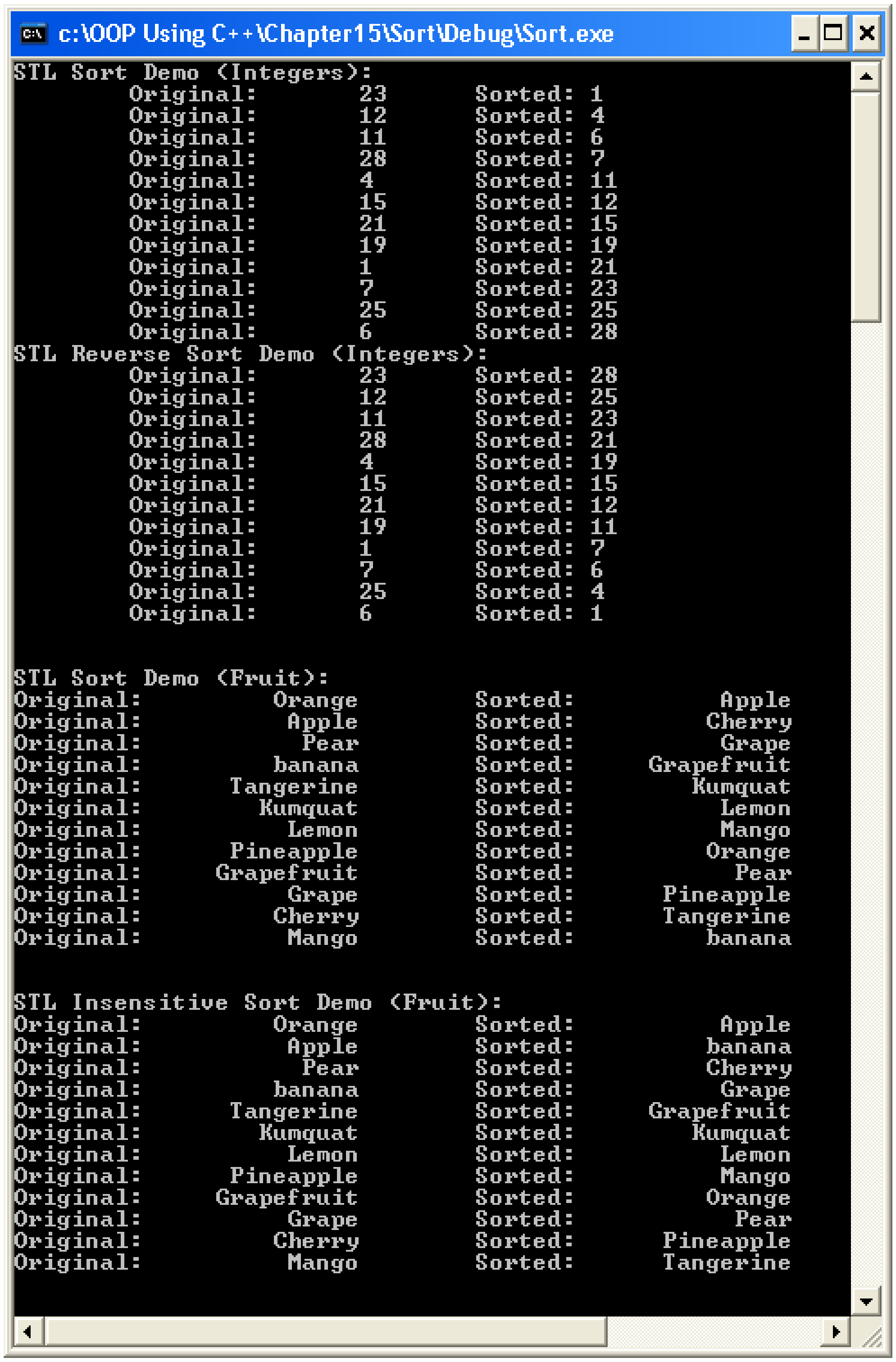
This is a very powerful capability because it allows you to customize the sort function to your objects in any way you can define. For example, you could use it to sort employees by name, by SSN, by date-of-birth—however you chose. You just define a separate function for each different sort.

Sample code demonstrating the use of the sort() function on various vector<> objects is presented in Example 15.10, with the output presented in Figure 15.7. Two additional points about sorting are also worth noting:

- Once your existing collection has been sorted, don't assume any existing iterators you've got hanging around are still valid. You'll need to call begin() again to get a valid sequence.
- The sort() function only works where random access iterators are available. If you need to sort a collection that does not support random access (most commonly, a **list<>**), you'll need to use the special member functions provided by that collection.

Example 15.10: Sample Code for Sorting vector<> Objects

```
void STLSortDemo()
{
	// First, we do it with integers
	int arUnsorted[]={23,12,11,28,4,15,21,19,1,7,25,6};
	vector<int> arVals;
	int i;
	for(i=0;i<12;i++) {
		arVals.push_back(arUnsorted[i]);
	}
	sort(arVals.begin(),arVals.end());
	cout << "STL Sort Demo (Integers):" << endl;
	for(i=0;i<12;i++) {
		cout << "\tOriginal:\t" << arUnsorted[i] <<
			"\tSorted:\t" << arVals[i] << endl;
	}
	sort(arVals.begin(),arVals.end(),greater<int>());
	cout << "STL Reverse Sort Demo (Integers):" << endl;
	for(i=0;i<12;i++) {
		cout << "\tOriginal:\t" << arUnsorted[i] <<
			"\tSorted:\t" << arVals[i] << endl;
	}
	char *arUFruit[12] = {"Orange", "Apple", "Pear", "banana",
	 "Tangerine", "Kumquat",
		"Lemon", "Pineapple", "Grapefruit", "Grape", "Cherry",
"Mango"};
	vector<string> arFruit;
	for(i=0;i<12;i++) {
		arFruit.push_back(arUFruit[i]);
	}
	sort(arFruit.begin(),arFruit.end());
	cout << endl << endl << "STL Sort Demo (Fruit):" << endl;
	for(i=0;i<12;i++) {
		cout << "Original:";
		cout.width(15);
		cout << arUFruit[i] <<  "\tSorted:";
		cout.width(15);
		cout << arFruit[i] << endl;
	}
	sort(arFruit.begin(),arFruit.end(),InsensitiveLT);
	cout << endl << endl << "STL Insensitive Sort Demo (Fruit):"
	   << endl;
	for(i=0;i<12;i++) {
		cout << "Original:";
		cout.width(15);
		cout << arUFruit[i] <<  "\tSorted:";
		cout.width(15);
		cout << arFruit[i] << endl;
	}
}
```



```
STL Sort Demo (Integers):
        Original:       23      Sorted: 1
        Original:       12      Sorted: 4
        Original:       11      Sorted: 6
        Original:       28      Sorted: 7
        Original:       4       Sorted: 11
        Original:       15      Sorted: 12
        Original:       21      Sorted: 15
        Original:       19      Sorted: 19
        Original:       1       Sorted: 21
        Original:       7       Sorted: 23
        Original:       25      Sorted: 25
        Original:       6       Sorted: 28
STL Reverse Sort Demo (Integers):
        Original:       23      Sorted: 28
        Original:       12      Sorted: 25
        Original:       11      Sorted: 23
        Original:       28      Sorted: 21
        Original:       4       Sorted: 19
        Original:       15      Sorted: 15
        Original:       21      Sorted: 12
        Original:       19      Sorted: 11
        Original:       1       Sorted: 7
        Original:       7       Sorted: 6
        Original:       25      Sorted: 4
        Original:       6       Sorted: 1

STL Sort Demo (Fruit):
Original:           Orange      Sorted:              Apple
Original:            Apple      Sorted:             Cherry
Original:             Pear      Sorted:              Grape
Original:           banana      Sorted:         Grapefruit
Original:        Tangerine      Sorted:            Kumquat
Original:          Kumquat      Sorted:              Lemon
Original:            Lemon      Sorted:              Mango
Original:        Pineapple      Sorted:             Orange
Original:       Grapefruit      Sorted:               Pear
Original:            Grape      Sorted:          Pineapple
Original:           Cherry      Sorted:          Tangerine
Original:            Mango      Sorted:             banana

STL Insensitive Sort Demo (Fruit):
Original:           Orange      Sorted:              Apple
Original:            Apple      Sorted:             banana
Original:             Pear      Sorted:             Cherry
Original:           banana      Sorted:              Grape
Original:        Tangerine      Sorted:         Grapefruit
Original:          Kumquat      Sorted:            Kumquat
Original:            Lemon      Sorted:              Lemon
Original:        Pineapple      Sorted:              Mango
Original:       Grapefruit      Sorted:             Orange
Original:            Grape      Sorted:               Pear
Original:           Cherry      Sorted:          Pineapple
Original:            Mango      Sorted:          Tangerine
```

Figure 15.7: Results of running STLSortDemo() function

List Sorting

In Chapter 16, we will introduce the **list<>** template collection. Without going into any details, however, we can deduce a lot about the collection by knowing only one thing: a **list<>** supports bi-directional iterators, but not random access iterators. Since the STL sort() function requires iterators supporting random access, it therefore cannot be used to sort a list.

For this reason, sort() member functions are commonly provided as part of collections that do not allow for random access, such as the **list<>** template. There are typically two forms of the sort() member function, namely:

> void collection<type>::sort();
> void collection<type>::sort(*comparison-function-pointer*);

Since no iterators are supplied, you cannot sort a region of a bi-directional collection the way you can with a vector. Nonetheless, you have the same ability to customize your sort, as shown in Example 15.11, which produces the same string results as shown for its **vector<>** counterpart.

Example 15.11: sort() Member of list<> Template Demonstration

```
void STLListSortDemo()
{
      int i;
      char *arUFruit[12] = {"Orange", "Apple", "Pear", "banana",
         "Tangerine", "Kumquat",
            "Lemon", "Pineapple", "Grapefruit", "Grape", "Cherry",
"Mango"};
      list<string> lstFruit;
      for(i=0;i<12;i++) {
            lstFruit.push_back(arUFruit[i]);
      }
      lstFruit.sort();
      cout << endl << endl << "STL List Sort Demo (Fruit):" << endl;
      list<string>::iterator pos=lstFruit.begin();
      for(i=0;i<12;i++) {
            cout << "Original:";
            cout.width(15);
            cout << arUFruit[i] <<  "\tSorted:";
            cout.width(15);
            cout << *pos << endl;
            pos++;
      }
      lstFruit.sort(InsensitiveLT);
      pos=lstFruit.begin();
      cout << endl << endl << "STL Insensitive Sort Demo (Fruit):"
          << endl;
      for(i=0;i<12;i++) {
            cout << "Original:";
            cout.width(15);
            cout << arUFruit[i] <<  "\tSorted:";
            cout.width(15);
            cout << *pos << endl;
            pos++;
      }
}
```

Once again, this example highlights our ability to perform actions on collections without knowing anything about their internal structure. This ability is one of the great strengths of the STL collection implementation, and justifies the time it takes to become comfortable with the iterator syntax (which can seem more than a bit peculiar as you're getting started.)

15.6.2: Other STL Algorithms

The sort() function is only one of a many (potentially) useful functions provided in the STL algorithm collection. Many of these can by customized with unary (one argument) or binary (two argument) functions, such as the above-defined InsensitiveLT() function. Some of the more potentially useful of these are summarized in Table 15.2, which is derived from the Visual Studio .Net documentation. Many additional functions, and variations are also available.

Algorithm	Description	Customizability
binary_search	Looks for an element in a collection using binary search algorithm	Comparison function (which must match sorting comparison function) may be specified
count count_if	Returns the number of elements in a range matching a value (count) or a user-specified predicate (count_if)	A single argument Boolean function must be specified for count_if
includes	Determines if one sorted collection range includes a second collection range	Comparison function may be specified, used to determine if one element is less than another
max_element min_element	Returns an interator to the maximum (minimum) element in a range	Comparison function may be specified, customizing what is meant by maximum (minimum)
merge	Takes two sorted collections (specified as iterator ranges) and merges them, placing the results in a third sorted collection (which must be sized large enough to hold the results). A separate **list<>** member version is defined.	Comparison function (which must match sorting comparison function) may be specified
partition	Takes a range and partitions it into two areas, one which meets the Boolean criteria and one which doesn't (similar to quicksort). Returns a bi-directional iterator to the partition point.	A single argument Boolean function must be specified, identifying whether a collection element meets or doesn't meet the specified criteria
sort	Takes a range a sorts it, using < criteria if no special comparison function is specified. A separate **list<>** member version is defined.	Comparison function may be specified
unique	Takes a sorted collection range and removes all duplicate elements, leaving only unique elements. A separate **list<>** member version is defined.	Comparison function (which must match sorting comparison function) may be specified

Table 15.2: Selected <Algorithm> Functions (adapted from Visual Studio .Net Documentation)

15.6 Section Questions

1. If you needed to lookup a single value in an unsorted collection, would it make sense to sort it then use binary search, to improve your lookup speed?

2. Given we have an efficient STL sort() function available to us, would it ever make sense to bother creating sorted tables, such as the KeyArray presented in Chapter 9?

3. What sorting algorithm that we have seen could be applied directly to list<> collections? Why would this not be the best way to implement list sorting?

4. Why would it make sense for the STL to have functions such as binary_search() and partition() and merge() supported within it?

5. If you were very concerned with sort efficiency, can you think of any situations where it might be beneficial to write your own sorting routines, instead of just using the STL versions?

15.7: Lab Exercise: Recursive Tokenizer

In Chapter 8, Section 8.4, we already saw how to create a powerful tokenizing class. In this lab exercise, the objective is to write a RTokenizer class that works entirely recursively. Indeed, this lab exercise cannot be judged a success if there is a single loop in your code!

15.7.1: Overview

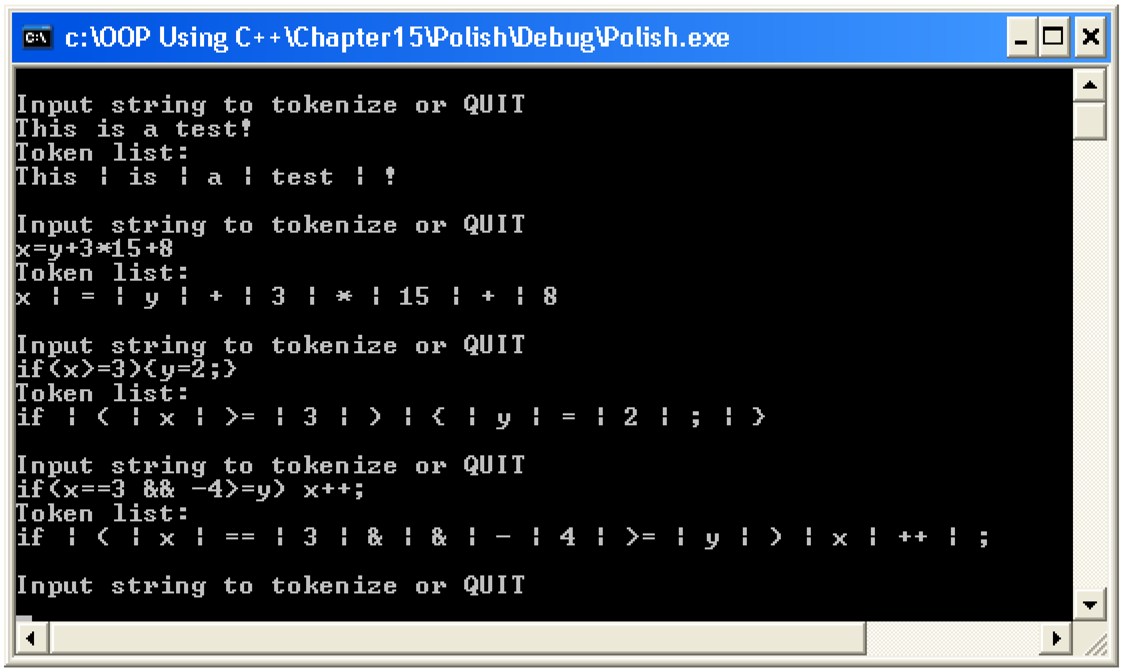
The RTokenizer class is a very simple class with only one public regular member, the Tokenize() function. A single static member function IsBreak() is also declared, returning true if its argument is a break character (defined—for our purposes—to be any character that isn't a white character, an underscore, a letter or a number).

The rules used by the class to break up a string into tokens are as follows:

- A token may not contain white characters, therefore any time a white character is encountered, it ends any token that is present.
- Any break character that is not part of a compound break character is a token by itself

- Certain break characters immediately adjacent to each other combine to form single tokens—examples being <=, && and +=.
- Adjacent letters and numbers combine to form a single token until ended by a white space or break character.

An example of how the class breaks up strings is presented in Figure 15.8, which was produced by typing strings into the TestToken() routine in Example 15.12.



```
Input string to tokenize or QUIT
This is a test!
Token list:
This ¦ is ¦ a ¦ test ¦ !

Input string to tokenize or QUIT
x=y+3*15+8
Token list:
x ¦ = ¦ y ¦ + ¦ 3 ¦ * ¦ 15 ¦ + ¦ 8

Input string to tokenize or QUIT
if(x>=3){y=2;}
Token list:
if ¦ ( ¦ x ¦ >= ¦ 3 ¦ ) ¦ { ¦ y ¦ = ¦ 2 ¦ ; ¦ }

Input string to tokenize or QUIT
if(x==3 && -4>=y) x++;
Token list:
if ¦ ( ¦ x ¦ == ¦ 3 ¦ & ¦ & ¦ - ¦ 4 ¦ >= ¦ y ¦ ) ¦ x ¦ ++ ¦ ;

Input string to tokenize or QUIT
```

Figure 15.8: Sample of RTokenizer class in action

Example 15.12: TokenTest() function

```
void TokenTest()
{
        RTokenizer tok;
        vector<string> arToks;
        bool bDone=false;
        while(!bDone) {
                char buf[256];
                cout << endl << "Input string to tokenize or QUIT" << endl;
                cin.getline(buf,255);
                if (stricmp("QUIT",buf)==0) break;
                tok.Tokenize(arToks,buf);
                if (arToks.size()>0) cout << "Token list:" << endl;
                else cout << "No tokens found!" << endl;
                for(int i=0;i<(int)arToks.size();i++) {
                        cout << arToks[i];
                        if (i<(int)arToks.size()-1) cout << " | ";
                        else cout << endl;
                }
        }
}
```

15.7.2: Implementation

The RTokenizer class declaration is presented in Example 15.13.

Example 15.13: RTokenizer Class Declaration

```
class RTokenizer :
      public map<string,bool>
{
public:
      RTokenizer(void);
      ~RTokenizer(void);

      void Tokenize(vector<string> &arT,const char *buf) const;
      static bool IsBreak(char cVal);
protected:
      void Tokenize(vector<string> &arT,string &current,
            const string &szBuf,int nPos) const;
};
```

The class inherits from map<string,bool>. The reason for this is as follows. We need to keep track of break characters that get grouped together, and identify them as we tokenize out string. A convenient place to hold these characters is in a lookup table. So, as part of the constructor function we initialize the table. This could be done as shown in Example 15.14. The method used to calculate nCount (which would be 10 in the example) allows us to sneak additional compound breaks (e.g., "*=" or "->") in without changing any other code.

Example 15.14: RTokenizer Constructor Function

```
RTokenizer::RTokenizer(void)
{
      char *arBreaks[] = { ">=", "<=", "==", "!="
            "&&", "||", "+=", "-=", "++", "--"
      };
      // Computes number of pairs so we don't have to...
      int nCount=sizeof(arBreaks)/sizeof(char *);
      int i;
      for(i=0;i<nCount;i++) {
            insert(pair<string,bool>(string(arBreaks[i]),true));
      }
}
```

Of the remaining functions, the 2-argument Tokenizer() function is just a driver function for the protected 4-argument version, and the IsBreak() function is self-evident from its purpose.

The 4-argument Tokenizer function, on the other hand, will take a bit more thought. The purpose of its arguments is as follows:

- *vector<string> &arT:* The array of tokens we are constructing
- *string &current:* The token we are currently building
- *const string &szBuf:* The buffer containing the string we are tokenizing. Unlike the PCalc::Tokenize() function in 15.6.2, we do not change the string entered by the user
- *int nPos:* The position in the input string that we are currently working on

The basic operation of the function should be to inspect the character at nPos and then decide whether or not to finish off the current token and add it to the array. The details are left to the reader—except that no loops are to be used.

15.7.3: Procedure

The RTokenizer() class is so small, there isn't any real way to do a lot of intermediate testing. One approach that might make sense would be to get the function working in three stages:

- Using only white space to delimit tokens, so break characters are treated like any other character
- Using spaces and single break characters to delimit tokens, ignoring the existence of combined break characters
- Making the class fully functional

15.8: Review and Questions

15.8.1: Review

A recursive function is one that calls itself, either directly or indirectly. A direct call means that the function is actually called within the body of the function itself. Indirect recursion means that some other function that the original function calls, may—ultimately—lead to the original function being called again before the other function returns.

Recursive functions are normally implemented using a stack. That means that each time an additional recursive call to a function is made, a new copy of all the functions local variables is made on the stack. That means that each function on the call stack maintains its own collection of local variable values—the only exception being static variables, for which only one copy is made.

When designing recursive functions, you tend to focus of making declarative statements about a function, rather than focusing on describing a procedure. For example, a recursive definition of factorial could be made as follows:

$$0! = 1$$
$$N! = N * (N-1)!$$

In the above definition, the second statement defines the factorial function in terms of itself, making the statement recursive. The first line, on the other hand, is non-recursive case. *Every recursive definition must have at least one non-recursive case.* Should no non-recursive case be defined, the recursion will be infinite, and will invariably lead to a stack overflow.

To design a recursive function, you typically look for two things:

- What are the non-recursive cases?
- Do the recursive cases move us in the direction of the non-recursive cases

Both of these conditions must be met in order for the recursive function to work. In addition, where a function makes more than one direct or indirect calls, you should generally try to ensure that the same function is not called with the same arguments more than once—to avoid serious inefficiencies.

Recursion is central to many sorting algorithms. The Quicksort algorithm, for example, uses iteration to partition an array into two sides, based on whether the value of all elements on each side are less than or greater than some midpoint value. It then recursively calls itself on each side.

Recursion can be used in many situations as a substitute for iteration. For example, when traversing a collection, an iterator variable can be passed into a recursive function as an argument an, in a recursive case, the function can call itself with the iterator incremented or decremented, as called for by the design of the function.

In choosing between iteration and recursion, clarity should usually be the determining factor, although recursive calls do entail more performance overhead than simple loops. Normally, recursion is considered a more advanced technique than iteration so—if the complexity of the two approaches is comparable—it makes sense to iterate (for the benefit of those who will be reading your code). On the other hand, recursive statements of a problem can sometimes be dramatically simpler than their iterative versions. In this case, it makes much more sense to use recursion. And for some problems, such as reading in and translating arbitrarily complex statements typed in by a user, recursive approaches are often the only ones that are feasible.

15.8.2: Glossary

Break Characters – Characters—such as operators in C++—that automatically separate tokens in an expression string.
Direct Recursion – A recursion where a function calls itself directly
Driver Function – A non-recursive function that is used to set up the arguments for a recursive function call
Factorial – An integer series, defined for positive numbers, where each element is the product of itself and the value of the previous element of the series
Fibonacci Series – An integer series, defined for positive numbers, where each element is the sum of the previous two elements in the series
Indirect Recursion – A recursion where a function calls another function which results in the original function being called before the other function returns
Infinite Recursion – A recursion in which a recursion doesn't terminate—either because improper non-recursive cases have been specified or recursive calls don't move the function towards the non-recursive case
Insertion Sort – A sorting technique, that may be recursive on non-recursive, that places elements in the proper position as they are added to an array.
Non-Recursive Case – A statement in a recursive problem set up that does not involve recursion (e.g., 0! = 1). At least one non-recursive case must be present in any recursive problem set up.
Quicksort – A recursive sorting algorithm that uses iteration to partition an array into two sides, based on whether the value of all elements on each side are less than or greater than some midpoint value. It then recursively calls itself on each side.
Recursion – A programming technique in which a function calls itself, either directly or indirectly

Stack Overflow – A condition, that generates an exception, that occurs when a stack exceeds its allocated space. Frequently an indication of an infinite recursion.
Tokenize – To break a string expression into individual elements, referred to as tokens.

15.8.3: Questions

15.1: Factorial class. Create a Factorial class that inherits from vector<int>. The class should override the [] operator such that whenever the user enters a coefficient that has not been calculated, it expands the series held in the vector<> before returning the value. e.g.,

```
Factorial fact;
int nV1=fact[5]; // computes 5!, placing elements in the vector<>
int nV2=fact[3]; // returns value, no computation necessary—just a lookup
int nV3=fact[7]; // computes 6! and 7! and adds them to the array
```

All computations should be done recursively.

15.2: Fibonacci class. Create a Fibonacci class that inherits from vector<int> and encapsulates the 1-call version of the Fibonacci series (Example 15.1). The class should override the [] operator such that whenever the user enters a coefficient that has not been calculated, it expands the series held in the vector<> before returning the value. (See description of Factorial class in 15.1).

15.3. Greatest Common Denominator. The GCD (also known as greatest common divisor) of two numbers is the largest number that will divide into both of them. This can be computed recursively by knowing the following facts:

- The GCD of any number, X, and 0 is X
- The GCD of X and Y is the same as the GCD of Y and X
- The GCD of X and Y is the same as the GCD of X and X % Y

Use this knowledge to write a recursive function:

```
unsigned int GCD(unsigned int X,unsigned int Y);
```

Comment: This is a rather famous recursion problem. Using the conditional operator (? :) it can be written in 1 line!

15.4: Longest repeated sequence. Write a recursive function, CountRepeats(), that takes a vector<int> argument and returns the longest number of repeated values in the sequence. For example, if our vector<int> contained:

> 1,2,2,4,6,6,6,3,2,5,7,7,7,7,7,8,3,2,2,1,1,5,5,5

the function would return 5 (reflecting the fact that there are 5 7's in a row—more than any other number). The driver function should be prototyped as follows:

```
int CountRepeats(const vector<int> &arNums);
```

(You must figure out how to prototype the recursive version) . Try to figure out a situation—any situation—where such a function might actually be useful.

15.5: Case-Insensitive Quicksort: Implement a version of Quicksort that does a case-insensitive sort of a vector<string> array.

Questions 15.6-15.10 deal with potential enhancements to the RTokenizer class

15.6: Support quoted strings. Modify your protected Tokenize() function to allow delimiters (such as " and ') to be used to hold strings together (use a string member variable, that you can set during construction, to hold the specific characters you'll support). Keeping your pure recursive approach, the idea is that once you've encountered a delimiter, you keep adding to the current token until the matching delimiter is found, or you reach the end of the input. *Hint:* If you leave the delimiter at the start of the *current* string argument while your processing, its easier for you function to tell what's going on. Strip the delimiters off before adding *current* to the array, however.

15.7: Support escape characters. Modify your protected Tokenize() function to support an escape character (\). Whenever an escape character is encountered within a delimited string, whatever character follows is added, unchanged, the the current argument. In other words, the input string (including quotes):

```
"Hello, its \"the author\" speaking"
```

will be added to the token array as a single token:

```
Hello, its "the author" speaking
```

15.8: Support escape sequences. Modify your protected Tokenize() function to support predefined character escape sequences, such as \n, \t, \r within delimited strings. Create a static function:

```
int  EscapeVal(char cVal);
```

that returns the ASCII equivalent for a specific character (e.g., EscapeVal('t') would return 9, since that's the value of \t). *Hint:* don't forget the capabilities already built into the class when implanting this.

15.9: Recursive parenthesis checker. When we get to routines like those in the general calculator (Chapter 16), nothing makes things crash faster than unbalanced parentheses, brackets or braces. Write a recursive routine, a member of the Tokenizer class, that returns false if specified elements don't balance. You should set the elements in a string of the form "(){}[]" where them first (even coefficient) of pair starts the container, and the second (odd coefficient) is the balancing element. The two overloads of the function should created, the first being the driver and the second being the (protected) recursive version:

```
bool IsBalanced(const vector<string> &arToks) const;  // Driver
bool IsBalanced(const vector<string> &arToks, int nPos,char nMatch= -1) const;
```

The arguments of the recursive version are the token array, the current token position being examined and the matching token being sought (or –1 if no container is present).

15.10: Tokenizer class modification. Using Chapter 8, end-of-chapter question 8.14 as a starting point, modify the Tokenizer class so it properly checks for matching container break characters—including nested containers of different types. You should do this by writing a member function:

```
size_t Tokenize::FindMatching(cEnd,const vector<string> &toks,

size_t nStart,size_t nEnd);
```

where cEnd is the container character you are looking for (e.g., ')', ']' or '}'), toks is the vector of tokens, nStart is the starting position (e.g., the position of the opening character plus 1) and nEnd is the position after the last allowable token position (e.g., toks.size() where no nesting has yet been encountered).

In Depth Questions:

15.11: Customized STL sort(): Write a function that orders a vector<string> object based on the length of the string elements, using the STL sort() function. Write a second version that orders a list<string>.

15.12: Quicksort reimplementation: Using the Visual Studio .Net documentation for specific information, reimplement quicksort using the *partition()* STL function.

15.13: Insertion sort reimplementation: Using the Visual Studio .Net documentation for specific information, reimplement an insertion sort algorithm using the *binary_search()* STL function.

Chapter 16

Lists and Trees

Executive Summary

Chapter 16 introduces two important collections shapes, the list and the tree. Both of these shapes have properties that distinguish them from the now-familiar array. In the case of the list, the benefits are extremely efficient insertion and deletion of elements anywhere in the collection. A tree provides the same insertion/deletion benefits, while allowing rapid access to elements by content.

The chapter begins by introducing the notion of a self-referential structure, a structure containing a pointer that points to another structure of the same type. We then demonstrate how self-referential elements can be used to create a new collection shape, the linked list. We contrast the pros and cons of linked lists with the other types of collections we have looked at, particularly the array. The STL **list<>** template class is then presented. The inherently recursive tree structure is then presented, and algorithms for creating and searching sorted binary trees are developed. More general trees, and their use as database indexes, are then described. Finally, we present a lab exercise that implements a simple doubly linked list collection and a lab that implements a general tree collection.

Learning Objectives

Upon completing this chapter, you should be able to:

- Explain what is meant by a self-referential structure
- Describe how self-referential pointers can be used to link structures together
- Explain what is meant by singly and doubly linked lists
- Iterate though the elements of a linked list
- Describe the process of insertion and removal of elements from a linked list
- Use the STL list<> template class to implement a list
- Explain what is meant by a tree
- Use recursion to create and traverse sorted binary trees
- Discuss how to remove elements from sorted binary trees
- Explain the problem of unbalanced trees and possible solutions
- Implement a general doubly linked list collection
- Implement a general tree collection

16.1: Self-Referential Structures

As the types of applications we program become more complex, we frequently find that we need to be able to represent collections of data that do not easily lend themselves to arrays and other collections we have discussed, such as hash tables. For example, a simple organizational chart, such as that presented in Figure 16.1, is straightforward enough for us understand. But how do we provide the computer with the same information that we instantly understand from looking in the diagram?

The most important tool that we have for representing complex data relationships is known as the self-referential structure. We consider the nature of such structures in this section, and then focus on an important application of such structures—the linked list—for the first half of the chapter.

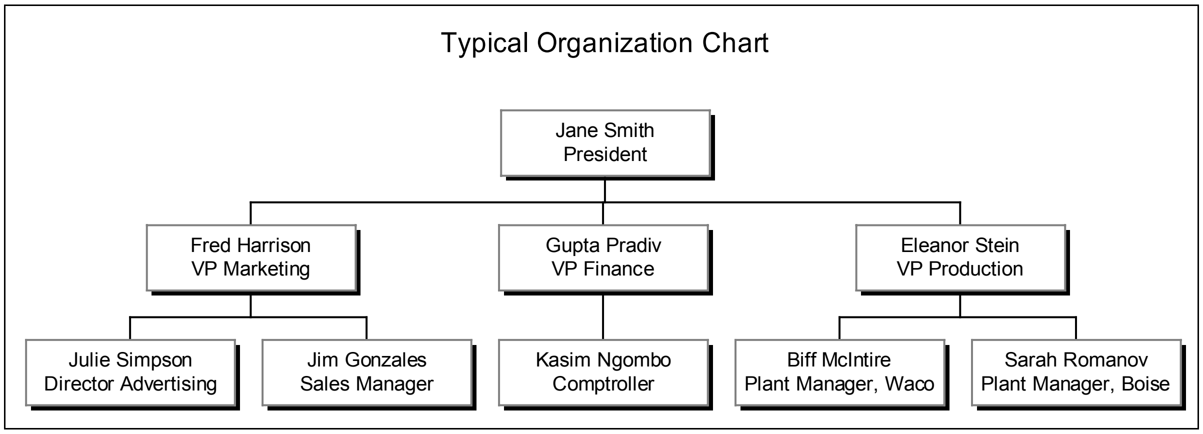

Typical Organization Chart
Jane Smith President
Fred Harrison VP Marketing
Gupta Pradiv VP Finance
Eleanor Stein VP Production
Julie Simpson Director Advertising
Jim Gonzales Sales Manager
Kasim Ngombo Comptroller
Biff McIntire Plant Manager, Waco
Sarah Romanov Plant Manager, Boise

Figure 16.1: Typical Organization Chart

16.1.1: What Are Self-Referential Structures?

What makes a structure self-referential is actually very simple. Within the structure definition, there is an included pointer to another structure of the same type. For example:

```
struct simple_self {
    int nVal;
    double dVal;
    struct simple_self *pSelf;
};
```

What makes the simple_self structure self-referential is the element pSelf, which points to another simple_self instance.

Test your understanding: Explain why self-referential structures necessarily use a pointer and why we *cannot* compose a structure within itself. (Hint: try to figure out what the sizeof operator would return if the pSelf member in our example were not a pointer, but an embedded simple_self structure).

16.1.2: Tying Objects Together with Links

Walkthrough available in OrgChart.avi

So what purpose does incorporating one or more self-referential pointers into a structure serve? The most important thing it does it to allow us to establish relationship links between our data elements. To understand how this works, consider our organization chart from Figure 16.1. Now, suppose we have the definitions in Example 16.1 available to us:

Example 16.1: Org Chart definitions

```
enum DepCode {
      dAdmin=0,
      dMarketing,
      dFinance,
      dOperations,
};
class employee
{
public:
      employee(int nId,DepCode nDept,const char *szName,const char
*szPos,
            employee *pBoss=0)
      {
            m_nId=nId;
            m_nDept=nDept;
            m_szName=szName;
            m_szPosition=szPos;
            m_pBoss=pBoss;
      }
      void AddSubordinate(employee *pSub)
             {m_arSubordinates.push_back(pSub);}
      static void DisplayDepartment(enum DepCode nVal);
      void DisplayData() const;
protected:
      unsigned int m_nId;
      enum DepCode m_nDept;
      string m_szName;
      string m_szPosition;
      employee *m_pBoss;
      vector<employee *> m_arSubordinates;
};
```

How could we use this to build a representation of our organization structure into our program? As a starting point, we can readily observe that the employee class, as defined in Example 16.1, contains two self-referential pointers. The first, m_pBoss, would—presumably—point to an individual's manager. Stated another way, it could be used to capture lines going up in Figure 16.1. Similarly, the **vector<>** of self-referential pointers, m_arSubordinates, would seem to be to be a good place to hold linkages going down.

> *Test your understanding:* What is it about a traditional organizational chart that makes pBoss and arSubordinates[] different in how they are defined.

With these insights, we can demonstrate how one could set up such relationships. Example 16.2 shows how we could initialize such structure in memory.

Example 16.2: Initializing an organization chart in memory

```
int main()
{
	employee Smith(101,dAdmin,"Smith, Jane","President");
	employee Harrison(201,dMarketing,"Harrison, Fred","VP Marketing",&Smith);
	employee Pradiv(401,dFinance,"Pradiv, Gupta","VP Finance",&Smith);
	employee Stein(301,dOperations,"Stein, Eleanor","VP Production",&Smith);
	employee Simpson(202,dMarketing,"Simpson, Julie","Director 
Advertising",&Harrison);
	employee Gonzales(203,dMarketing,"Gonzales, Jim","Sales Manager",&Harrison);
	employee Ngombo(402,dFinance,"Ngombo, Kasim","Comptroller",&Pradiv);
	employee McIntire(302,dOperations,"McIntire, Biff","Plant Manaager, Waco",&Stein);
	employee Romanov(303,dOperations,"Romanov, Sally","Plant Manager, Boise",&Stein);
	Stein.AddSubordinate(&McIntire);
	Stein.AddSubordinate(&Romanov);
	Pradiv.AddSubordinate(&Ngombo);
	Harrison.AddSubordinate(&Simpson);
	Harrison.AddSubordinate(&Gonzales);
	Smith.AddSubordinate(&Harrison);
	Smith.AddSubordinate(&Pradiv);
	Smith.AddSubordinate(&Stein);
	Harrison.DisplayData();
	cout << endl << endl;
	Smith.DisplayData();
	cout << endl << endl;
	Romanov.DisplayData();
	return 0;
}
```

The function begins by initializing a series of conveniently named objects with non-pointer information (i.e., employee ID, name, position, department code) and boss pointers (where applicable). Once the structures are present in memory, we begin to establish the subordinate relationships between them. For example, from the chart we know that Stein, in the operations department, has two people working for her: Romanov and McIntire. We create the boss-to-subordinate relationships using the AddSubordinate() member function, i.e.,:

> Stein.AddSubordinate(&McIntire);
> Stein.AddSubordinate(&Romanov);

We continue this process until all our relationships have been established. To verify that we have, indeed, captured these relationships in a meaningful way, we then call a function DisplayData()—defined in Example 16.3—to display the data for selected employees.

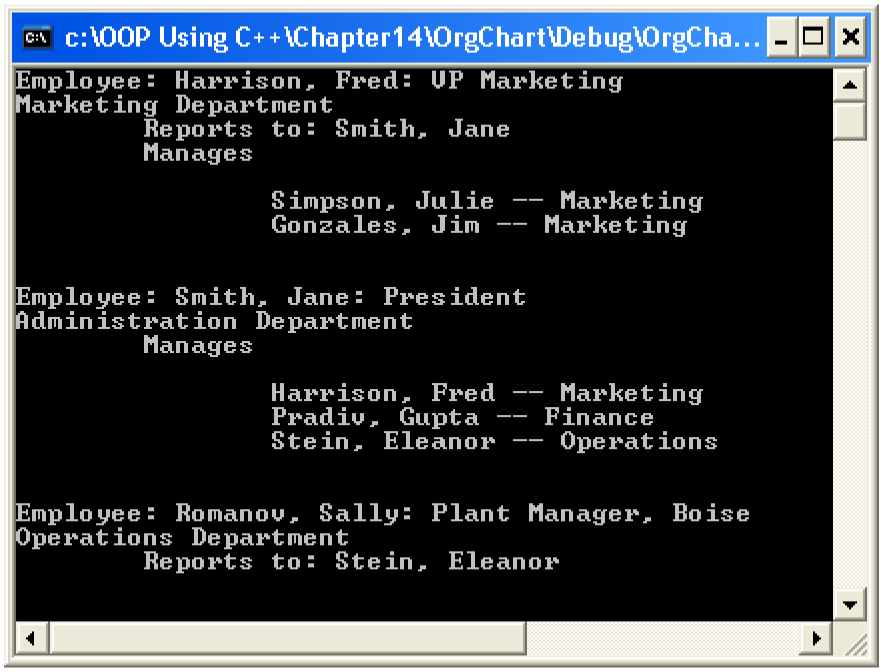
Example 16.3: DisplayData() function

```
void employee::DisplayData() const
{
	int i;
	cout << "Employee: " << m_szName << ": " << m_szPosition << endl;
	DisplayDepartment(m_nDept);
	cout << " Department" << endl;
	if (m_pBoss!=0) {
		cout << "\tReports to: " << m_pBoss->m_szName << endl;
	}
	if (m_arSubordinates.size()>0) {
		cout << "\tManages" << endl << endl ;
		for(i=0;i<(int)m_arSubordinates.size();i++) {
			cout << "\t\t" << m_arSubordinates[i]->m_szName << "
-- ";
			DisplayDepartment(m_arSubordinates[i]->m_nDept);
			cout << endl;
		}
	}
}

void employee::DisplayDepartment(enum DepCode nVal)
{
	switch(nVal)
	{
		case dAdmin:
		{
			cout << "Administration";
			break;
		}
		case dMarketing:
		{
			cout << "Marketing";
			break;
		}
		case dFinance:
		{
			cout << "Finance";
			break;
		}
		case dOperations:
		{
			cout << "Operations";
			break;
		}
		default:
		{
			cout << "Not Assigned";
			break;
		}
	}
}
```

This function works as follows:

- It first displays data for the employee passed in (including a call to a simple function DisplayDepartment() which takes a code and displays the associated department name).
- It then checks to see if the m_pBoss link is 0. If not, it uses that link to access the boss's name, which is displayed.
- Finally, it checks to see if the m_arSubordinates vector<> is empty. If not, it iterates through the subordinate array, printing the name and department of each.

The output of Example 16.3 is presented in Figure 16.2.



```
Employee: Harrison, Fred: VP Marketing
Marketing Department
        Reports to: Smith, Jane
        Manages

                Simpson, Julie -- Marketing
                Gonzales, Jim -- Marketing


Employee: Smith, Jane: President
Administration Department
        Manages

                Harrison, Fred -- Marketing
                Pradiv, Gupta -- Finance
                Stein, Eleanor -- Operations


Employee: Romanov, Sally: Plant Manager, Boise
Operations Department
        Reports to: Stein, Eleanor
```

Figure 16.2: Output of Example 16.3 (Org Chart test)

16.1 Section Questions

1. Why doesn't the compiler get confused when we specify a class name within the code that defines the class we're creating?

2. What programming elements do the boxes and lines represent in a diagram such as Figure 16.1?

3. If you have boss pointers defined matching Figure 16.1, do you really need subordinate pointers?

4. If Gonzales wanted to transfer from Marketing to Finance, would we need to delete or create any objects?

5. Could we implement a collection such as that shown in Figure 16.1 using a simple vector<>?

16.2: Linked Lists

Walkthrough available in IntList.avi

One of the most commonly used examples of self-referential structures is a linked list. Linked lists are one of the fundamental collection shapes, and are often used as an alternative for arrays and lookup tables in applications where a collection of data is accessed sequentially.

16.2.1: Singly and Doubly Linked Lists

Linked lists come in two forms, singly and doubly linked, illustrated in Figure 16.3. Each object in a singly linked list has a self-referential pointer (often given a name like pNext) that points to the next element in the list. The final element of the list—sometimes called the tail of the list—has a pointer value of NULL (or 0), signaling the end of the list. A simple structure that could be used to construct a singly linked list of float values is presented below:

```
struct float_list {
    float val;
    struct float_list *pNext;
};
```

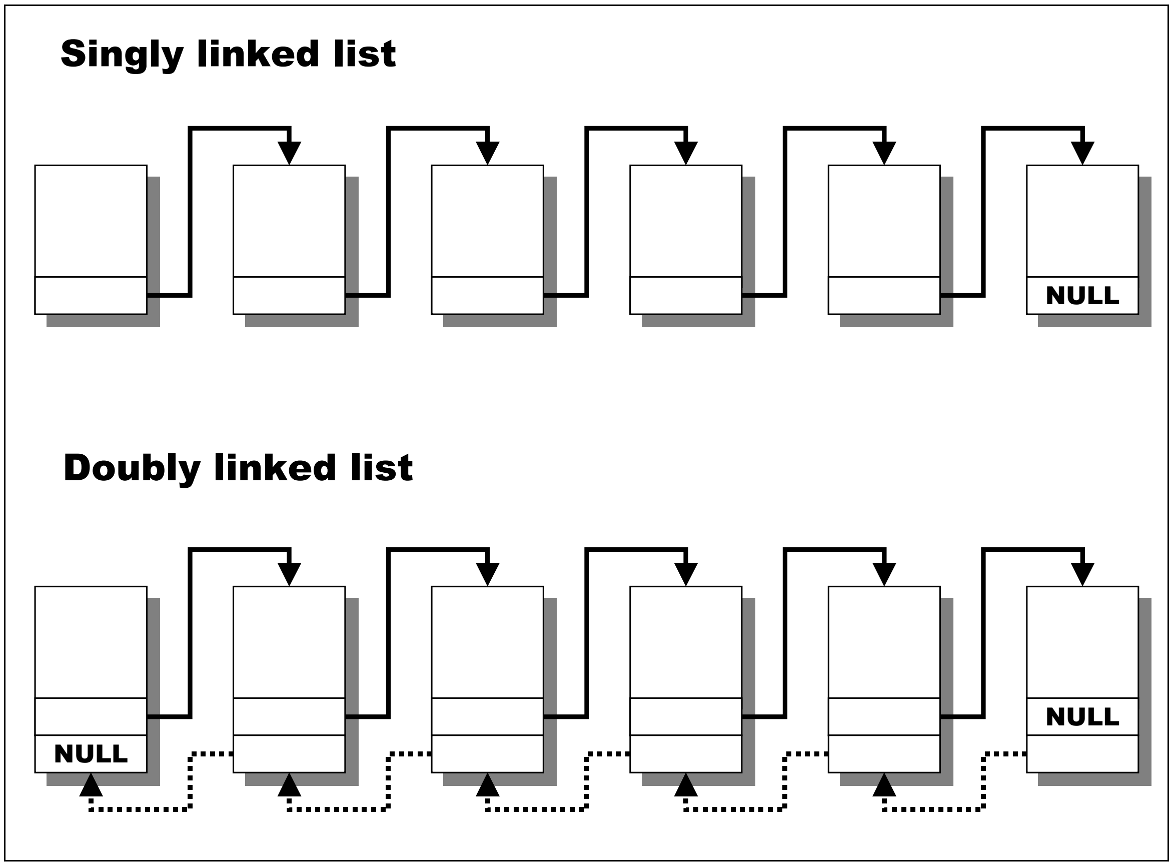


Figure 16.3: Singly and doubly linked lists

A doubly linked list uses two pointers, a pointer to the next element of the list (e.g., pNext) and a pointer to the previous element of the list (e.g., pPrev). While the resulting list is structurally similar (e.g., the difference is the dotted lines pointing backwards in Figure 16.3), the doubly linked version allows motion either towards the end (tail) of the list, or towards front—known as the head—of the list (which has a NULL value in its previous pointer). A simple class that could be used to hold a doubly linked list of integer values is presented in Example 16.4:

Example 16.4: int_list Double Linked List Class

```
class int_list {
public:
      int_list() {
            m_nVal=0;
            m_pNext=0;
            m_pPrev=0;
      }
      void Val(int nVal){m_nVal=nVal;}
      int Val() const {return m_nVal;}
      int_list *Next() const {return m_pNext;}
      int_list *Prev() const {return m_pPrev;}
      void DisplayList() const;
      void DisplayListBackwards() const;
      void DisplayListMiddle() const;
      void InsertBefore(int_list *pNew);
      void InsertAfter(int_list *pNew);
      void Remove();
protected:
      int m_nVal;
      int_list *m_pNext;
      int_list *m_pPrev;
};
```

Because doubly linked lists are an extension of singly linked lists, and are more commonly used, we will focus on them for illustrative purposes. An example of code that creates a doubly linked int_list consisting of the integers:

5   3   1   6   8

is presented in Example 16.5.

Example 16.5: Initializing Code for list

```
void ListDemo()
{
	int_list arVals[10];
	// Initializing values to position in array
	for(int i=0;i<10;i++) {
		arVals[i].Val(i);
	}
	int_list *pHead=arVals+5,*pEle,*pTail;
	// Creating list: 5 -> 3 -> 1 -> 6 -> 8
	pEle=pHead;
	pEle->InsertAfter(arVals+3);
	pEle=pEle->Next();
	pEle->InsertAfter(arVals+1);
	pEle=pEle->Next();
	pEle->InsertAfter(arVals+6);
	pEle=pEle->Next();
	pEle->InsertAfter(arVals+8);
	pTail=pEle->Next();
	pHead->DisplayList();
	cout << endl;
	pTail->DisplayListBackwards();
	cout << endl;
	pEle=arVals+6;
	pEle->DisplayList();
	cout << endl;
	pEle->DisplayListMiddle();
	cout << endl;
	// Changing list to 5 -> 3 -> 1 -> 9 -> 6 -> 8
	pEle=arVals+6;
	pEle->InsertBefore(arVals+9);
	pHead->DisplayList();
	cout << endl;
	// Changing list to 5 -> 1 -> 9 -> 6 -> 8
	pEle=arVals+3;
	pEle->Remove();
	pHead->DisplayList();
	cout << endl;
}
```

For the sake of convenience, the example begins by initializing 10 int_list objects in an array, with their assigned value being equal to their position (this allows us to work with list objects without having to give each one its own name).

Since the list we propose begins with the object whose value is 5, we put the address of that element (arVals+5, using pointer arithmetic) in a pointer pHead, that we'll use to keep track of the head of the list. We also assign that value to a temporary pointer pEle that we use as we construct the list. We then establish our two connections between the head and the next element in the target list (3) by calling the InsertAfter() function:

```
	pEle=pHead;
```

```
pEle->InsertAfter(arVals+3);
```

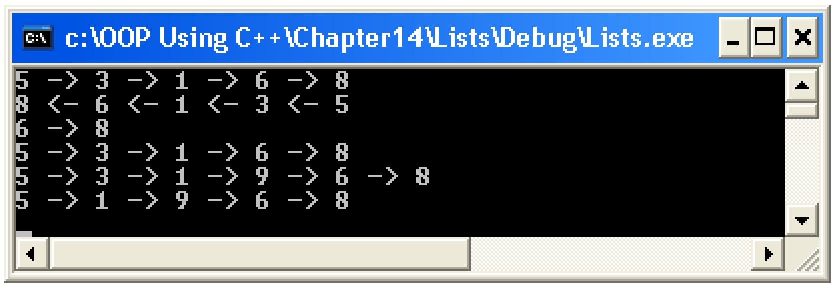
The 5 and 3 are now linked together in both directions. We then change pEle so that it points to arVals[3] with the assignment:

```
pEle=pEle->Next();
```

We can then add the next element (1) using the same procedure. Repeating the procedure until we reach the end of the list, we assign the pTail pointer to the last element (8) with the assignment:

```
pEle->InsertAfter(arVals+8);
pTail=pEle->Next();
```

The code that follows after the tail has been established demonstrates list iteration, insertion and deletion. It is presented in the next two sections. The results of running the function in 16.10 are presented in Figure 16.4.



```
5 -> 3 -> 1 -> 6 -> 8
8 <- 6 <- 1 <- 3 <- 5
6 -> 8
5 -> 3 -> 1 -> 6 -> 8
5 -> 3 -> 1 -> 9 -> 6 -> 8
5 -> 1 -> 9 -> 6 -> 8
```

Figure 16.4: List demonstration output

16.2.2: Traversing a Linked List

Traversing a linked list—iterating from element to element—is very straightforward. Three functions in our demonstration illustrate moving around the list from the head, tail and middle. These functions, DisplayList(), DisplayListBackwards() and DisplayListMiddle(), are presented in Example 16.6.

Example 16.6: List iteration examples

```
void int_list::DisplayList() const
{
	const int_list *pHead=this;
	while(pHead!=0) {
		cout << pHead->m_nVal;
		pHead=pHead->m_pNext;
		if (pHead!=0) cout << " -> ";
	}
}

void int_list::DisplayListBackwards() const
{
	const int_list *pT=this;
	while(pT!=0) {
		cout << pT->m_nVal;
		pT=pT->m_pPrev;
		if (pT!=0) cout << " <- ";
	}
}

void int_list::DisplayListMiddle() const
{
	const int_list *pT=this;
	if (pT==0) return;
	while(pT->m_pPrev!=0) pT=pT->m_pPrev;
	pT->DisplayList();
}
```

The basic mechanism for moving forwards between elements of a list is to take a temporary pointer and continue to assign it to the next element until a NULL is reached. This is illustrated by the loop in DisplayList(). Moving backwards is essentially the same process, except the m_pPrev pointer is used instead of the m_pNext pointer, as shown in DisplayListBackwards().

When given an element in the middle of a linked list, the situation is a bit more complicated. One approach, used in DisplayListMiddle(), is to iterate backwards until the head of the list is reached, e.g.:

```
while(pT->m_pPrev!=0) pT=pT->m_pPrev;
```

Once the head is known, then a function such as DisplayList() can be called without fear of missing elements.

The difficulty of moving around in a list is one of the big disadvantages of this shape of collection. Using a linked list to store data vs. an array is a bit like using a cassette instead of a CD. If you are not going to jump around a lot, the shape of the list doesn't create much of a problem. If, on the other hand, you are going to be accessing different parts of the list in random order, you'd be much better using an array (which, like a CD, you can easily jump around in) as opposed to a list, which requires you to iterate back and forth (just like rewinding and fast forwarding a cassette).

16.2.3: Inserting and Deleting List Elements

Just as a list has disadvantages over an array when it comes to moving around, it can have decided advantages if we need to keep inserting and deleting elements. Insertions and deletions within arrays typically require us to move the elements after the insertion/deletion point up or down—which can affect a lot of memory. Moreover, if our array gets too large, we may have to allocate more memory in a large block (assuming that such a large block happens available).

Linked lists are an entirely different story. Because the collection consists of small chunks tied together by pointers, you never need to worry about getting memory in blocks larger than a single list object. And insertion/deletion involves manipulating a few pointers, rather than moving around large blocks of memory.

Three functions, InsertBefore(), InsertAfter() and Remove(), used to perform insertions and deletions in our example are presented in Example 16.7.

Example 16.7: Inserting and Deleting List Elements

```
void int_list::InsertBefore(int_list *pNew)
{
      if (m_pPrev!=0) {
            m_pPrev->m_pNext=pNew;
      }
      pNew->m_pPrev=m_pPrev;
      m_pPrev=pNew;
      pNew->m_pNext=this;
}

void int_list::InsertAfter(int_list *pNew)
{
      if (m_pNext!=0) {
            m_pNext->m_pPrev=pNew;
      }
      pNew->m_pPrev=this;
      pNew->m_pNext=m_pNext;
      m_pNext=pNew;
}

void int_list::Remove()
{
      if (m_pNext!=0) m_pNext->m_pPrev=m_pPrev;
      if (m_pPrev!=0) m_pPrev->m_pNext=m_pNext;
}
```

The InsertAfter() function takes its argument, pNew, and inserts it after the object to which it is being applied. It functions as follows:

- It checks if the object is at the tail of the list (i.e., m_pNext==0). If not, it takes the element downstream of the object (i.e., m_pNext) and sets its previous pointer to the new element (i.e.,m_pNext->m_pPrev=pNew).
- It takes the m_pNext pointer (pointing upstream of our insertion point) and places the address in pNew->m_pNext, since what used to be downstream of the object is now downstream of pNew.
- It takes the m_pNext pointer puts the address of the new object (pNew) in it, since pNew will now be after the object in the list.
- It takes the pNew->m_pPrev pointer and puts the address **this** (i.e., the object) in it, completing the insertion.

The InsertBefore() function takes its argument, pNew, and inserts it in front of the object to which it is being applied. It functions as follows:

- It checks if the object is at the head of the list (i.e., m_pPrev==0). If not, it takes the element upstream of the object (i.e., m_pPrev) and sets its next pointer to the new element (i.e.,m_pPrev->m_pNext=pNew).
- It takes the m_pPrev pointer (pointing upstream of our insertion point) and places the address in pNew->m_pPrev, since what used to be upstream of the object is now upstream of pNew.
- It takes the m_pPrev pointer puts the address of the new object (pNew) in it, since pNew will now be before the object in the list.
- It takes the pNew->m_pNext pointer and puts the address **this** (i.e., the object) in it, completing the insertion.

To remove an element from a list, the pointers connecting that element into the list must be redirected to the upstream and downstream elements. This is done in Remove() as follows:

- If a downstream element is present (i.e., m_pNext!=0), the m_pPrev pointer of that element is changed to the m_pPrev pointer of the element being removed. This is written as m_pNext->m_pPrev=m_pPrev.
- If an upstream element is present (i.e., m_pPrev!=0), the m_pNext pointer of that element is changes to the m_pNext pointer of the element being removed. This is written as m_pPrev->m_pNext=m_pNext.

It is important to note that is absolutely critical that pointers be adjusted properly during list insertions and deletions. The slightest error and the program will be vectored off into the netherworld of memory—where system crashes and unhappy users are the rule, rather than the exception. For this reason, using predefined list objects and functions is generally safer than growing your own. The STL **list<>** template class is an example of such an object.

16.2 Section Questions

1. Will a linked list ever require a resizing operation the way a dynamic array can?

2. Would there be any circumstances where it would be preferable to implement a singly linked list over a doubly linked list?

3. If you were implementing a stack shape using an int_list, what members of int_list could you get rid of? Would int_list serve as the basis for an adequate stack?

4. If you were implementing a queue shape using an int_list, what members of int_list could you get rid of? Would int_list serve as the basis for an adequate queue?

5. Would the int_list class be considered a collection class?

6. Could we use binary search on a sorted linked list?

16.3: GPtrList Class

Walkthrough available in GPtrList.avi

The int_list class developed in Section 16.2 had two serious deficiencies as a list collection implementation:

1. It did not manage general collection information, such as the head pointer, tail pointer or element count.

2. It required the collection infrastructure (e.g., next and previous pointers) be embedded in with the collection objects.

In this lab walkthrough, we develop a GPtrList collection (modeled after the CPtrList class that Microsoft provides in its MFC) that remedies these two problems.

16.3.1: Overview of GPtrList Collection

The collection we'll be developing differs from the list example presented in Section 16.2 in two ways:

- The collection is separated from the data being collected—as opposed to incorporating self-referential pointers within out data structures
- The collection is implemented using two structures. A GPtrList structure is used to keep track of the head and tail of the list, along with an item count. A list_position structure is actually used to implement the list, and includes a pointer to the GPtrList object that manages the list.

The list_structure structure and typedef are presented in Example 16.8.

Example 16.8: Structures and typedefs using in the GPtrList application

```
struct list_position
{
      list_position() {
            next=0;
            prev=0;
            val=0;
      }
      list_position *next;
      list_position *prev;
      void *val;

};

typedef list_position * POSITION;
```

The GPtrList class is declared in Example 16.9. Conceptually, the data that is actually collected by the list is void pointers—meaning that we could use the list to maintain a collection of any types of objects. The flip side of this, of course, is that the programmer using the collection is responsible for keeping track of what type of data is being collected. The compiler can't help. (Obviously, we can remedy this deficiency using templates—presented as an end-of-chapter exercise).

Example 16.9: GPtrList Declaration

```
class GPtrList
{
public:
      GPtrList(){ m_nCount=0;
                              m_posHead=0;
                              m_posTail=0;}
      virtual ~GPtrList(){RemoveAll();}
      POSITION AddHead(void *p);
      POSITION AddTail(void *p);
      POSITION InsertBefore(POSITION pos,void *p);
      POSITION InsertAfter(POSITION pos,void *p);
      void RemoveAt(POSITION pos);
      void RemoveAll();
      void *GetNext(POSITION &pos) const {
            void *pVal=pos->val;
            pos=pos->next;
            return pVal;
      }
      void *GetPrev(POSITION &pos) const {
            void *pVal=pos->val;
            pos=pos->prev;
            return pVal;
      }
      void *GetAt(POSITION pos) const {return pos->val;}
      void SetAt(POSITION pos,void *p) {pos->val=p;}
      int GetCount() const {return m_nCount;}
      POSITION GetHeadPosition() const {return m_posHead;}
      POSITION GetTailPosition() const {return m_posTail;}

protected:
      POSITION m_posHead;
      POSITION m_posTail;
      int m_nCount;

};
```

Figure 16.5 diagrams the relationships between the various components of our collection for a 4-element list. The top structure represents the GPtrList structure, with 1) a pointer to the position element at the head of the list, 2) a pointer to the tail of the list, and 3) an element count (updated each time an element is added or removed). Because there are so many list_position pointers in the structures and applications, a typedef POSITION was defined to be used wherever a list_position pointer is used.

The middle structures are list_position structures, with pointers to 1) the next list element (pNext), 2) the previous list element (pPrev), 3) the GPtrList structure they are associated with (pOnwer, shown with dotted curved lines in the diagram), and 4) to the data being collected (pVal). The data being collected can be of any form, and is shown as gray amorphous blobs on the bottom row.

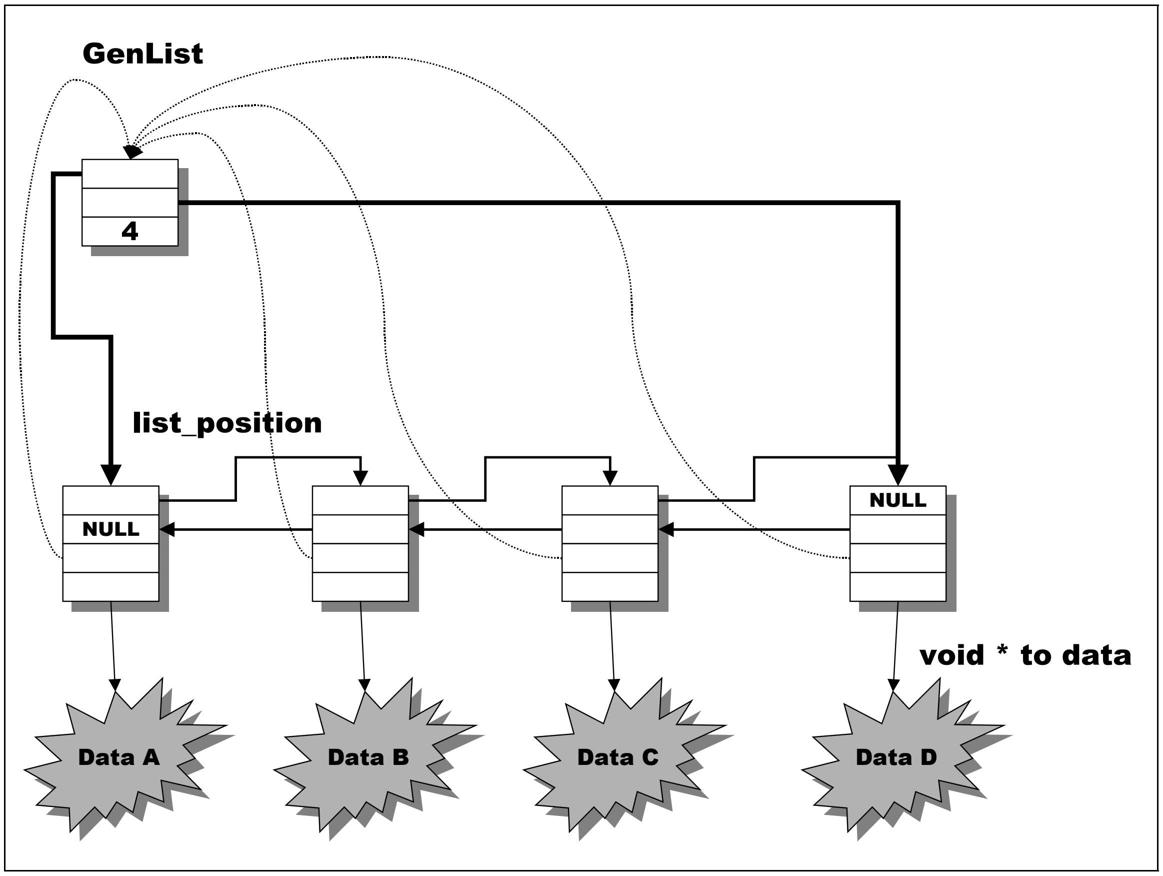


Figure 16.5: GenList collection components

16.3.2: Linked List Members

To implement the GPtrList collection, a variety of functions need to be defined. Six of these, described in Table 16.1, modify the collection. The remainder, described in Table 16.2, access data from the collection and are implemented as inline functions in Example 16.9.

GPtrList Modification Member Functions

void AddHead(void *p) Adds a new list element (with a value pointer to p) to the beginning (head) of the list, updating the m_pHead member. In the event the list is empty, the m_pHead and m_pTail members both point to the new element (since a one element list has the same element as a head and a tail). The function also increments the m_nCount member.
void AddTail(struct GenList *pList,void *p) Adds a new list element (with a value pointer to p) to the end (tail) of the list, updating the m_pTail member. In the event the list is empty, the m_pHead and m_pTail members both point to the new element (since a one element list has the same element as a head and a tail). The function also increments the m_nCount member.
void InsertBefore(POSITION pos,void *p) Adds a new list element to the list immediately before the position passed in as an argument. The function needs to create a new list_position structure and places the argument p in new object's pVal member. It must then splice the new list_position structure into the list and increment the m_nCount member. In the event pos points to the head of the list, the m_pHead member must also be updated.
void InsertAfter(POSITION pos,void *p) Adds a new list element to the list immediately after the position passed in as an argument. The function needs to create a new list_position structure and places the argument p in new object's pVal member. It must then splice the new list_position structure into the list and increment the m_nCount member. In the event pos points to the tail of the list, the m_pTail member must also be updated.
void RemoveAt(POSITION pos) Removes a list_position object from the list, splicing together the upstream and downstream position objects and freeing the memory for the position object (using delete).The m_nCount member is also decremented and—if the object being removed happens to be at the head or tail of the list—the m_pHead and/or m_pTail pointers are updated.
void RemoveAll(struct GenList *pVal) Removes all the list_position pointers from a GPtrList object, releasing their memory, and setting the associated m_pHead,m_pTail and m_nCount values to 0.

Table 16.1: Functions that modify the GPtrList collection class

GPtrList Accessor Functions

void *GetNext(POSITION &pos) const Gets the value pointer associated with pos for use a return value, *then* sets pos equal to the next position pointer in the list.
void *GetPrev(POSITION &pos) const Gets the value pointer associated with pos for use a return value, *then* sets pos equal to the previous position pointer in the list.
void *GetValue(POSITION pos) const Returns pos->m_pVal.
void SetValue(POSITION pos,void *p) Sets pos->m_pVal to p.
int GetCount(const struct GenList *pList) const Returns m_nCount.
POSITION GetHeadPosition(const struct GenList *pList) const

Returns m_pHead.
POSITION GetTailPosition(const struct GenList *pList) Returns m_pTail.

Table 16.2: Inline functions for the GPtrList collection class

An demonstration of the class usage is presented in Example 16.10.

Example 16.10: GPtrList Demonstration

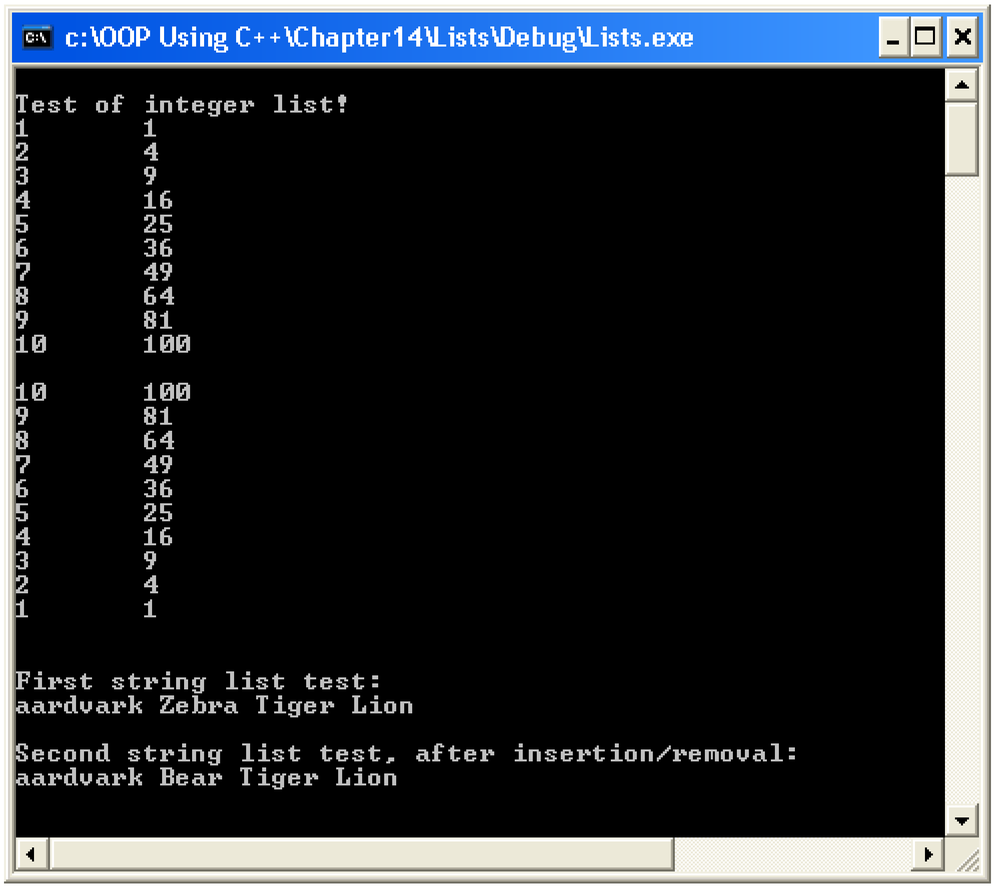
```
void GPtrListDemo()
{
	GPtrList lstSquares;
	int arSquares[10];
	// ading values to end of list
	for(unsigned int i=1;i<=10;i++) {
		arSquares[i-1]=i*i;
		lstSquares.AddTail(arSquares+(i-1));
	}
	cout << endl <<  "Test of integer list!" << endl;
	POSITION pos=lstSquares.GetHeadPosition();
	for(i=0;pos!=0;i++) {
		int *pVal=(int *)lstSquares.GetNext(pos);
		cout << i+1 << '\t' << *pVal << endl;
	}
	cout << endl;
	// Iterating backwards
	pos=lstSquares.GetTailPosition();
	for(i=lstSquares.GetCount();pos!=0;i--) {
		int *pVal=(int *)lstSquares.GetPrev(pos);
		cout << i << '\t' << *pVal << endl;
	}
	cout << endl;
	GPtrList lstText;
	// inserting values at the front of the list
	string s0("Lion"),s1("Tiger"),s2("Zebra"),s3("aardvark");
	lstText.AddHead(&s0);
	lstText.AddHead(&s1);
	// Using return value of insert
	POSITION posZebra=lstText.AddHead(&s2);
	lstText.AddHead(&s3);
	cout << endl << "First string list test:" << endl;
	pos=lstText.GetHeadPosition();
	while(pos!=0) {
		string *pVal=(string *)lstText.GetNext(pos);
		cout << *pVal << " ";
	}
	// Middle insertion
	string strBear("Bear");
	lstText.InsertAfter(posZebra,&strBear);
	lstText.RemoveAt(posZebra);
	cout << endl;
	cout << endl << "Second string list test, after 
insertion/removal:";
	cout << endl;
	pos=lstText.GetHeadPosition();
	while(pos!=0) {
		string *pVal=(string *)lstText.GetNext(pos);
		cout << *pVal << " ";
	}
	cout << endl;
	return;
}
```

In the demonstration, we first set up an array of integers (to give us something to put in the list), then illustrate forward and backward iteration through the list. Of particular interest here is the use of GetNext() and GetPrev(), an example of which is:

```
POSITION pos=lstSquares.GetHeadPosition();
for(i=0;pos!=0;i++) {
        int *pVal=(int *)lstSquares.GetNext(pos);
        cout << i+1 << '\t' << *pVal << endl;
}
```

What is critical to not about the class interface is that calling GetNext() or GetPrev() causes the value associated with the argument to be returned. The updating of the position pointer does not occur until after the value is accessed.

The function then continues with various demonstrations of list insertion and removal using string objects. The output from running the function is shown in Figure 16.6.



```
Test of integer list!
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
9       81
10      100

10      100
9       81
8       64
7       49
6       36
5       25
4       16
3       9
2       4
1       1


First string list test:
aardvark Zebra Tiger Lion

Second string list test, after insertion/removal:
aardvark Bear Tiger Lion
```

Figure 16.6: Output of GPtrListDemo() function

16.3.2: GPtrList Implementation

The GPtrList implementation is very similar to the int_list class presented in Section 16.2. What is different is the following:

1. New list_position structures need to be created whenever elements are added, and deleted when they are removed
2. Head and tail pointers need to be kept updated as the list changes
3. The count needs to be updated as the list changes

The AddHead() and AddTail() functions, presented in Example 16.11, illustrate all three of these differences. Each begins by creating a new list position structure. Splicing at the head (tail) then occurs, taking care to handle the case of an empty list, where the element becomes both head and tail. The count is then incremented at the end, with the newly created list position pointer being returned.

Example 16.11: GPtrList AddHead() and AddTail() members

```
POSITION GPtrList::AddHead(void *p)
{
      POSITION ele=new list_position;
      if (m_posHead!=0) m_posHead->prev=ele;
      else m_posTail=ele;
      ele->prev=0;
      ele->next=m_posHead;
      m_posHead=ele;
      ele->val=p;
      m_nCount++;
      return ele;
}
POSITION GPtrList::AddTail(void *p)
{
      POSITION ele=new list_position;
      if (m_posTail!=0) m_posTail->next=ele;
      else m_posHead=ele;
      ele->prev=m_posTail;
      ele->next=0;
      m_posTail=ele;
      ele->val=p;
      m_nCount++;
      return ele;
}
```

The InsertBefore() and InsertAfter() members, shown in Example 16.12, are very similar. Where they differ is that both upstream and downstream splicing needs to be accomplished. In InsertBefore(), for example, this leads to the code:

pos->prev->next=ele;

This moves to the element preceding *pos*, takes its *next* pointer and sets it to the newly created element (*ele*). So that we don't have to worry about m_pHead and m_pTail pointers, each function checks to see if we're inserting at the head (for the InsertBefore() function) or the tail (for the InsertAfter() function) and returns a call to the appropriate AddHead() or AddTail() function if we are.

Example 16.12: GPtrList InsertBefore() and InsertAfter() members

```
POSITION GPtrList::InsertBefore(POSITION pos,void *p)
{
      if (pos==0) return 0;
      if (pos->prev==0) return AddHead(p);
      POSITION ele=new list_position;
      pos->prev->next=ele;
      ele->prev=pos->prev;
      ele->next=pos;
      pos->prev=ele;
      ele->val=p;
      m_nCount++;
      return ele;
}
POSITION GPtrList::InsertAfter(POSITION pos,void *p)
{
      if (pos==0) return 0;
      if (pos->next==0) return AddTail(p);
      POSITION ele=new list_position;
      pos->next->prev=ele;
      ele->next=pos->next;
      ele->prev=pos;
      pos->next=ele;
      ele->val=p;
      m_nCount++;
      return ele;
}
```

The two list removal member functions are presented in Example 16.13. The critical special cases in the RemoveAt() member are removal of elements that happen to be at the head or the tail. Such removals not only change the splicing that must be done, they also requires us to change the class pointers to the head and/or the tail. Otherwise, the splicing required is similar to that shown in Section 16.2 for the int_list object.

Example 16.13: GPtrList RemoveAt() and RemoveAll() members

```
void GPtrList::RemoveAt(POSITION pos)
{
      if (pos==0) return;
      if (pos->prev!=0) pos->prev->next=pos->next;
      else m_posHead=pos->next;
      if (pos->next!=0) pos->next->prev=pos->prev;
      else m_posTail=pos->prev;
      m_nCount--;
      delete pos;
}
void GPtrList::RemoveAll()
{
      for(POSITION pos=GetHeadPosition();pos!=0;)
      {
            POSITION oldPos=pos;
            GetNext(pos);
            delete oldPos;
      }
      m_posHead=0;
      m_posTail=0;
      m_nCount=0;
}
```

16.3 Section Questions

1. When we iterate through a GPtrList, why do we always check the position pointer for 0?

2. Which member functions of GPtrList could have been declared static? Would there be any disadvantage to doing so?

3. Why don't we worry about tail insertions in InsertBefore() or head insertions in InsertAfter()?

4. Is there ever a case where RemoveAt() requires us to update both the head and the tail pointer?

5. Why don't we worry about splicing elements as we remove them in the RemoveAll() member function?

6. Is it important that the GPtrList destructor be virtual?

16.4: STL list<> Template Class

Walkthrough available in STLList.avi

The STL list<> template class is an example of a collection whose primary benefit is performance related in situations where lots of intermediate insertions and deletions are required—since a **list<>** can do little that a **vector<>** cannot do, and the **vector<>** offers random access as well.

16.4.1: list<> Overview

The **list<>** template provides a collection that allows iteration in either direction, but not random access. That means that the following iterator accesses are supported:

- **iter*: returns the element value referenced by the iterator
- *iter++*: moves to the next element of the list
- *iter*--: moves to the previous element of the list

The following iterator accesses, however, are ***not*** supported by the *list*<> template:

- *iter+offset*: you cannot directly access **list<>** elements without moving to them
- *iter[offset]*: random access is not permitted in a **list<>**
- **(iter+offset)*: again, random access is not permitted in a **list<>**

On the plus side, however, the **list<>** offers some features that aren't available in vector<> and other random classes:

- When an insertion or deletion is made to a **list<>**, only the iterators directly involved (or adjacent to) the modification are impacted. This is different from a **vector<>**, where all iterators (including that returned by the begin() function) can, in theory be impacted by such actions.
- Insertion and deletion performance is independent of list size. That is because, as we have seen in Section 16.2, such activities are performed by splicing elements in and out of the list, rather than by resizing an array.
- Elements from one list can be efficiently spliced into another list, without recreating all the iterators involved.

When using a **list<>** strictly for iterating through sequences, the code involved is identical to that used for a vector<>. This is illustrated in Example 16.14.

Example 16.14: Demonstration of STL list<> template

```
void STLListDemo()
{
	list<int> lstSquares;
	// adding values to end of list
	for(unsigned int i=1;i<=10;i++) {
		lstSquares.push_back(i*i);
	}
	cout << endl <<  "Test of integer list!" << endl;
	list<int>::iterator pos=lstSquares.begin();
	for(i=0;pos!=lstSquares.end();i++) {
		cout << i+1 << '\t' << *pos << endl;
		pos++;
	}
	cout << endl;
	// Iterating backwards
	pos=lstSquares.end();
	for(i=lstSquares.size();pos!=lstSquares.begin();i--) {
		pos--;
		cout << i << '\t' << *pos << endl;
	}
	cout << endl;
	list<string> lstText;
	// inserting values at the start of the list
	string s0("Lion"),s1("Tiger"),s2("Zebra"),s3("aardvark");
	lstText.push_back(s0);
	lstText.push_back(s1);
	// Using return value of insert
	list<string>::iterator posZebra=lstText.insert(lstText.end(),s2);
	lstText.push_back(s3);
	cout << endl << "First string list test:" << endl;
	list<string>::iterator spos=lstText.begin();
	for(i=0;i<lstText.size();i++) {
		cout << *spos << " ";
		spos++;
	}
	// Middle insertion
	lstText.insert(posZebra,string("Bear"));
	cout << endl;
	cout << endl << "Second string list test, after insertion:" <<
endl;
	spos=lstText.begin();
	for(i=0;i<lstText.size();i++) {
		cout << *spos << " ";
		spos++;
	}
	cout << endl;
	return;
}
```

The demonstration shows how a **list<>** is traversed using an iterator the same way a **vector<>** would be traversed for a collection of integer square values. In the list<string> demonstration, however, we see something that you would not (safely) be able to do with a vector<>. In inserting strings, we capture the return value of an insert() member in an iterator, i.e.,:

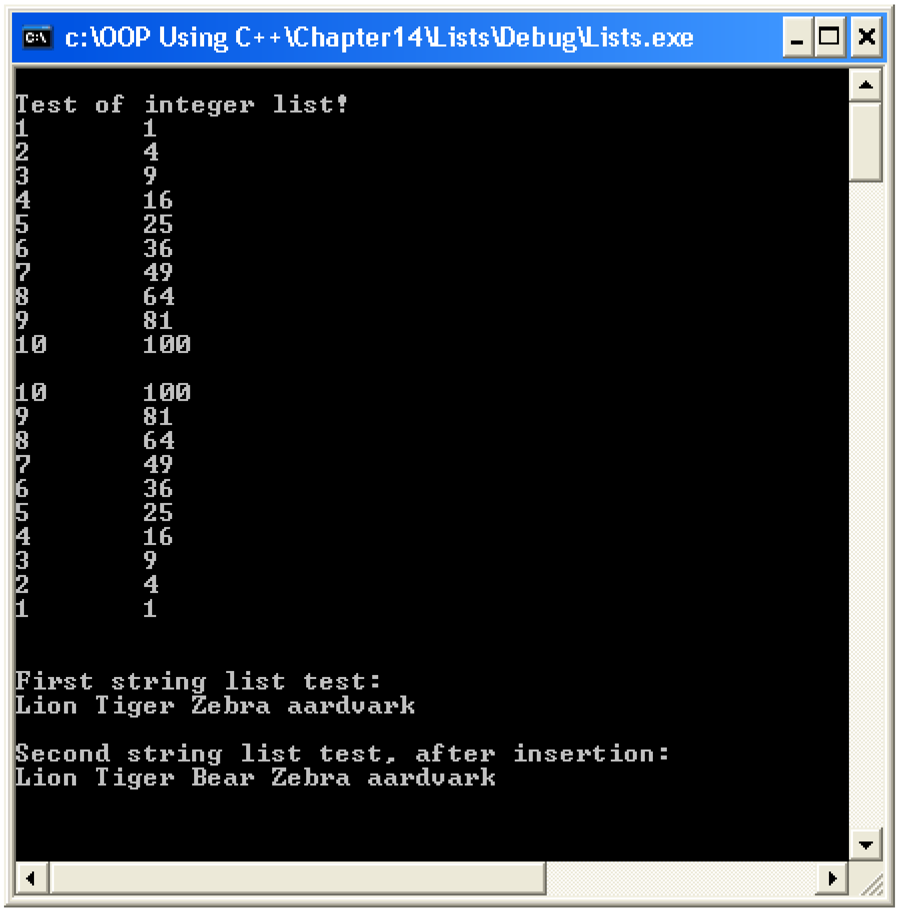
```
        list<string>::iterator posZebra=arText.insert(arText.end(),s2);
```

Then, after a subsequent insertion, we use that same iterator to make yet another insertion, i.e.,:

```
        arText.insert(posZebra,string("Bear"));
```

With a **list<>**, it is okay to use an iterator in this way because insertions and additions only change the iterator they are applied to. With a **vector<>**, you should always require your iterator (e.g., call the begin() member) after you modify the **vector<>**.

The output from running STLListDemo() is presented in Figure 16.7.



```
Test of integer list!
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
9       81
10      100

10      100
9       81
8       64
7       49
6       36
5       25
4       16
3       9
2       4
1       1


First string list test:
Lion Tiger Zebra aardvark

Second string list test, after insertion:
Lion Tiger Bear Zebra aardvark
```

Figure 16.7: STLListDemo() output

16.4.2: List Members

Commonly used **list<>** members are summarized in Table 16.3.

Operator	Arguments	Purpose
assign	3. iterator 4. iterator	Replaces the contents of the list with a collection range Arguments specify the begin and end point of assignment range from the source collection, which is not necessarily another list<>
begin	None	Returns an iterator that can be used for bi-directional access referring to the first element of the list
clear erase	None (clear) 1. iterator (erase) 2. iterator=end() (erase)	Empties the contents of a list Erase version allows an optional specified range of the list to be emptied
end	None	Returns an iterator representing the position beyond the data in the list
insert	1. iterator 2. element-data-type	Inserts an element at the specified position within the list (*Note*: other overloads also exist). Returns an iterator referring to the inserted element.
pop_back	1. element-data-type	Removes last element in the list
push_back	1. element-data-type	Adds element to the end of the list
size	None	Returns number of elements in the list
splice	1. iterator 2. list<> &arg2 3. iterator=arg2.begin() 4. iterator=arg2.end()	Splices the elements from the second list into the object at the position specified by the first argument, removing them from the second list. If the third and fourth arguments are specified, only the specified segment of the second list is spliced in.

Table 16.3: Selected list<> Member functions, compiled from Visual Studio .Net documentation

16.4 Section Questions

1. What's the principal difference between the bi-directional iterator supported by a list<> and the random access iterator supported by a vector<>?

2. If we say that a vector<> is superset of a list<> for most purposes, what would that mean?

3. Why can vector<> iterators change after insertion/deletions, while list<> iterators—excepting those directly involved in the modification—remain valid?

4. Why would a sorted vector<> tend to be much more useful than a sorted list<>?

5. What is the one significant member function shown in Table 16.1 that is supported by the list<> template class but not by the vector<>?

6. Why does it make sense to view the list<> template in comparison with the vector<> template?

16.5: Trees

One type of self-referential structure that is almost always created and traversed recursively is the tree. We have already seen an example of the tree structure in Chapter 16, the organization chart. In this section, we'll learn how to create trees and use them for purposes such as sorting.

16.5.1: What is a tree?

A tree can be viewed as a logical extension of the concept of a linked list. Because a list has only one pointer to the next element, its structure is linear. That means that tracing a list through memory is like tracing the body of a snake: no matter where it takes you, there is never any question about where to go next. The tree structure, in contrast, always has at least two pointers to other tree structures (a tree with exactly two pointers is referred to as a binary tree). For that reason, every tree element presents us with at least two branches in our path. This difference is illustrated in Figure 16.8.

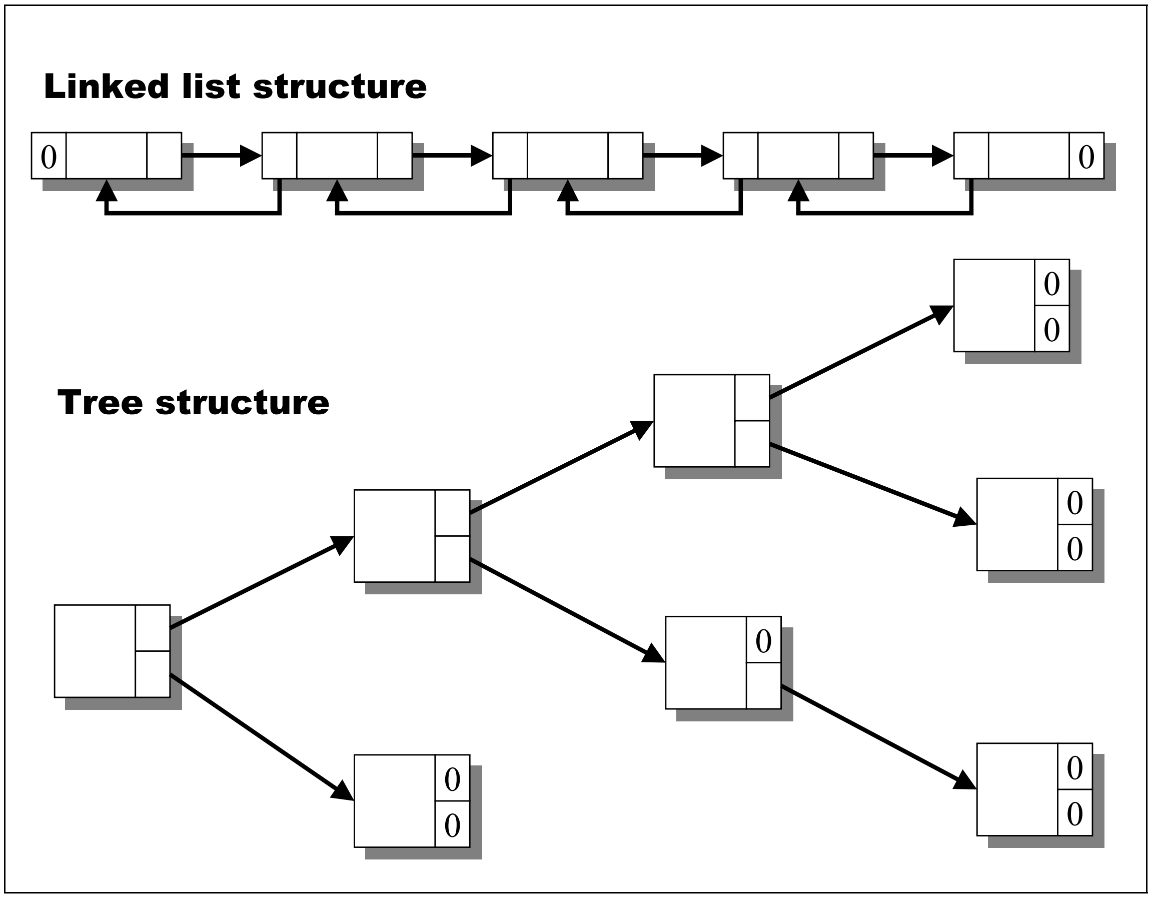


Figure 16.8: Tree vs. Linked List structures

The fact that a self-referential structure presents us with two branches does not guarantee that the structure is a tree. Two other conditions must be met:

- Each element can have one, and only one, direct parent. The direct parent of an element is defined as a tree structure that points to that element. The only element that does not have a direct parent is referred to as the root of the tree. The parents of an element consist of the chain of direct parents from that element all the way back to the root of the tree. Naturally, the root of the tree is a parent of all the other elements in the tree.
- No element can have any parent that is also a child. This restriction prevents the creation of circular trees.

Illustrations of these two conditions—illegal in trees—are presented in Figure 16.9.

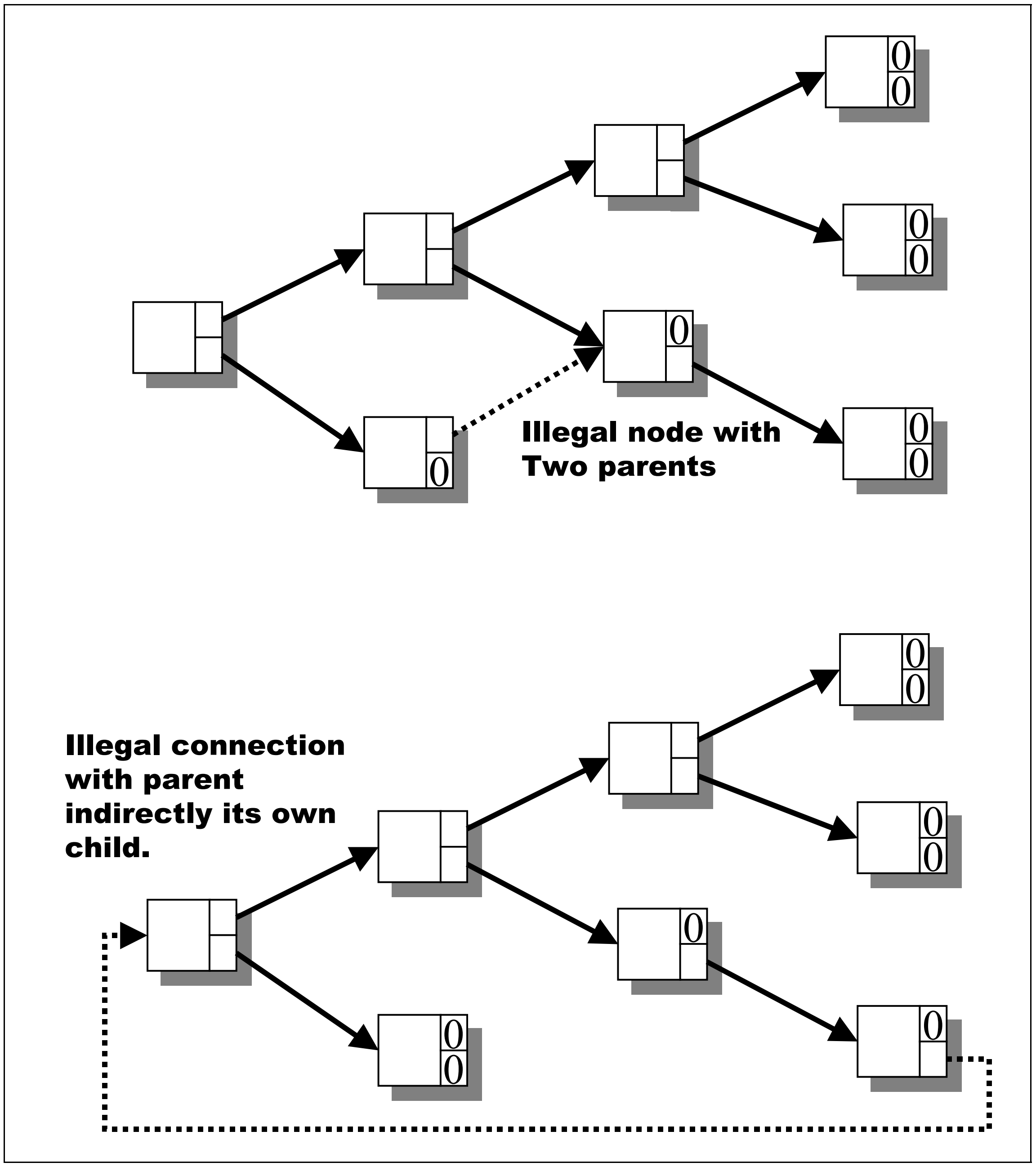


Figure 16.9: Examples of illegal tree structures

The fact that such data structures are not trees does not mean they can't exist. In fact, a more general set of self-referential structures, referred to as graphs, relax these requirements. Techniques for applying graphs, however, are beyond the scope of the current text.

Although we initially focus on binary trees, a tree can have more than two pointers. The number of pointers in a node is often referred to as the order of a tree (although for some trees, the term order sometimes refers to half the number of pointers, for reasons to be discussed later). The number of layers from the root to the bottom of the tree is generally referred to as the height or depth of a tree. Thus, in the figure showing a tree and a linked list, the order of the tree would be 2 (signifying a binary tree) and the height would be 4.

16.5.2: Binary Trees

Walkthrough available in BinaryTree.avi

A binary tree is simplest form of tree structure, which each node in the tree having pointers to two child nodes. It is also the most common form of tree structure used for trees contained entirely in memory. The simplest binary tree structure consists of an element that identifies the data associated with a node (either a pointer to the data, or an actual variable or structure containing the data itself) and two self-referential pointers that point to other tree nodes (or NULL). An example of such a tree, containing integer values, is presented in Figure 16.10.

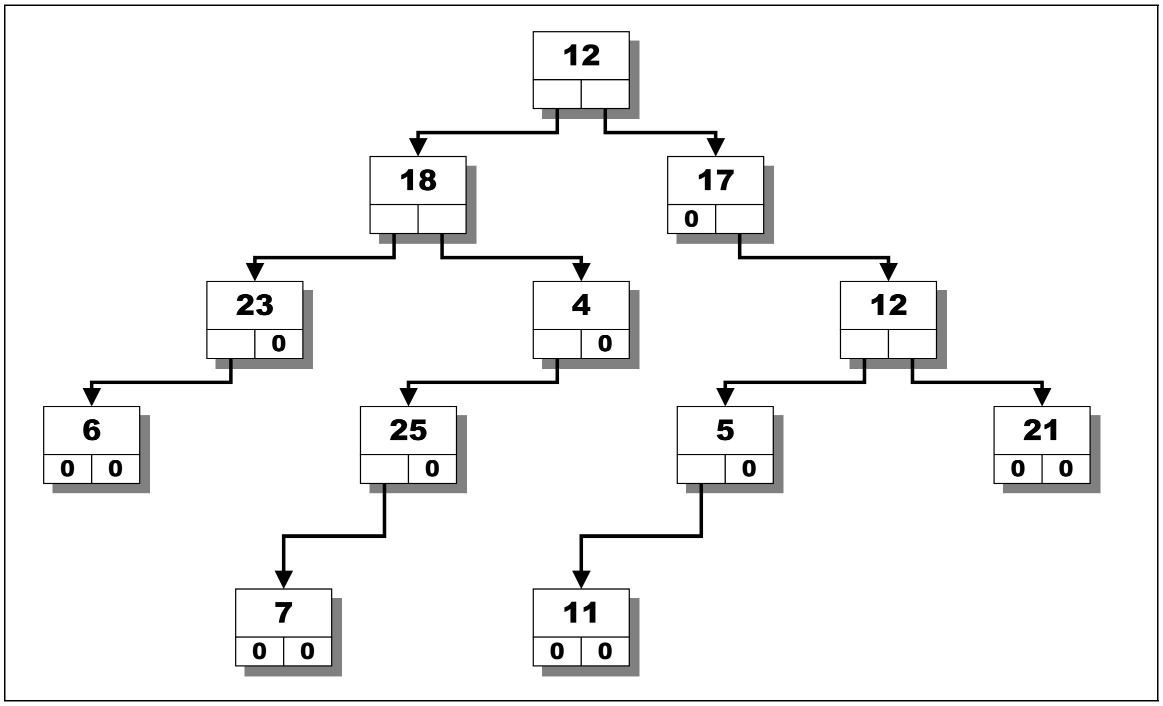


Figure 16.10: A simple binary tree containing integers.

An IntTree class that could be used to create such a tree could be constructed with the data members:

```
int nValue;
IntTree *pLeft;
IntTree *pRight;
```

Creating a simple binary tree
Example 16.15 illustrates the class declaration for an IntTree class, as well as some demonstration code that could be used to create such a tree.

Example 16.15: Creating a simple binary tree

```
class IntTree {
public:
	IntTree(int nV);
	virtual ~IntTree();
	int TotalIntTree() const;
	IntTree *FindIntTree(int nTarget) const;
	void FindEven(vector<IntTree*> &ar) const;
	int CountNodes() const;
	int nValue;
	IntTree *pLeft;
	IntTree *pRight;
};

void IntTreeDemo()
{
	int i,nTotal;
	IntTree *pRoot=new IntTree(12);
	vector<IntTree*> arNodes;
	pRoot->pLeft=new IntTree(18);
	pRoot->pLeft->pLeft=new IntTree(23);
	pRoot->pLeft->pLeft->pLeft=new IntTree(6);
	pRoot->pLeft->pRight=new IntTree(4);
	pRoot->pLeft->pRight->pLeft=new IntTree(25);
	pRoot->pLeft->pRight->pLeft->pLeft=new IntTree(7);
	pRoot->pRight=new IntTree(17);
	pRoot->pRight->pRight=new IntTree(12);
	pRoot->pRight->pRight->pLeft=new IntTree(5);
	pRoot->pRight->pRight->pLeft->pLeft=new IntTree(11);
	pRoot->pRight->pRight->pRight=new IntTree(21);
	nTotal=pRoot->TotalIntTree();
	cout << "Total of tree elements is " << nTotal << endl;
	if (pRoot->FindIntTree(25)) cout << "Found an element matching 25" << endl;
	else cout << "Didn't found an element matching 25 -- Error" << endl;
	if (pRoot->FindIntTree(24))
	cout << "Found an element matching 24-- Error" <<endl;
	else cout << "Didn't found an element matching 24" << endl;
	pRoot->FindEven(arNodes);
	cout << "Even elements: ";
	for(i=0;i<(int)arNodes.size();i++) {
		cout << arNodes[i]->nValue << " ";
	}
	cout << endl;
	delete pRoot;
}
```

There are many ways that the illustrative tree in Figure 16.10 could have been created. In the example, left branches were always added before right branches. When we get to sorted binary trees (Section 16.5.3), we'll have more to say about inserting elements into a tree.

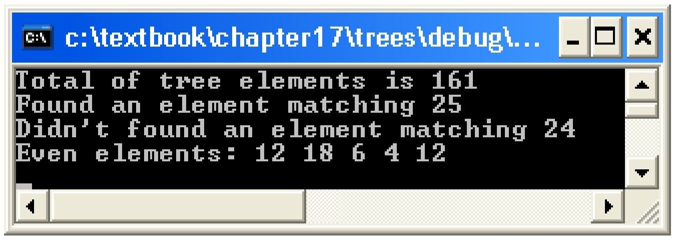
Working with binary trees
The functions that operate on binary trees nearly always work recursively. Moreover, they tend to follow a fairly consistent model:

- Perform some action (e.g., test, computation) using the value of the object being operated on.
- Call the function recursively on the left branch if that branch is not empty (sometimes the call is contingent on the results of a test).
- Call the function recursively on the right branch if that branch is not empty (sometimes call is contingent on the results of a test).

Example 16.16: Example IntTree functions

```
IntTree::~IntTree()
{
      delete pLeft;
      delete pRight;
}
int IntTree::TotalIntTree() const
{
      int nVal=nValue;
      if (pLeft!=0) nVal+=pLeft->TotalIntTree();
      if (pRight!=0) nVal+=pRight->TotalIntTree();
      return nVal;
}
IntTree *IntTree::FindIntTree(int nTarget) const
{
      IntTree *pFound=0;
      if (nTarget==nValue) pFound=(IntTree *)this;
      else if (pLeft!=0) pFound=pLeft->FindIntTree(nTarget);
      if (pFound==0 && pRight!=0) pFound=pRight->FindIntTree(nTarget);
      return pFound;
}

void IntTree::FindEven(vector<IntTree*> &ar) const
{
      if (nValue%2==0) ar.push_back((IntTree *)this);
      if (pLeft!=0) pLeft->FindEven(ar);
      if (pRight!=0) pRight->FindEven(ar);
}
```

A number of functions illustrating this general approach are called in the IntTreeDemo() function (Example 16.15). The function definitions are presented in Example 16.16, the output of which are presented in Figure 16.11.



```
Total of tree elements is 161
Found an element matching 25
Didn't found an element matching 24
Even elements: 12 18 6 4 12
```

Figure 16.11: Results of calling IntTreeDemo()

One of the most amazing things about tree functions is that they tend to be very small, yet very powerful. The destructor function provides a good example of this, calling the destructors on the left and right branches (effectively implementing a recursive destruction of the entire tree).

Tree functions also tend to look a lot alike, as is illustrated by the three remaining functions. The first of these, the TotalIntTree() function sums the integer values of every element in the tree. Declaratively, it views the sum as the total of three components:

- The value of the root node
- The sum of the elements in the tree whose root is pointed to by the left pointer, if it is not empty
- The sum of the elements in the tree whose root is pointed to by the right pointer, if it is not empty.

The FindIntTree() function is used to find the first node matching a certain value, returning its address. It works as follows:

- If the root node matches the target value, its address is returned.
- It searches the left branch, if it is not empty, returning the pointer if a value is found
- If the previous searches were unsuccessful, and the right branch is not empty, it returns the result of searching the right branch.

The FindEven() function is intended fill an array of node pointers with every node in the tree that holds an even argument. Its argument is a vector<IntTree*> object to which even nodes are added as they are encountered.

The function works as follows:

- If the current element value is even, it is added to the vector<> argument using the push_back() member.
- A recursive call to the tree on the left branch is then made, if it is not empty, adding all even elements on that branch
- A recursive call to the tree on the right branch is then made, if it is not empty, adding all empty elements on that branch.

The function then returns.

16.5.3: Sorted Binary Trees

Trees are generally used for two of purposes. The first is as a means of representing complex data structures, such as organization charts, grammars and inheritance hierarchies. The second—and more common—purpose is as an alternative collection shape to the array and map for storing sorted values. As we have already noted, the linear shape of a list makes it a bad choice for any collection that is not processed sequentially, but offers rapid insertion and deletion. A sorted tree has the same ability to facilitate insertion and deletion, but also allows relatively efficient search.

Example sorted tree
A typical sorted binary tree of integers is presented in Figure 16.12. While it looks structurally similar to the unsorted tree in Figure 16.3, it has two important properties that the earlier tree does not have. Specifically, for every root:

- Any node coming off the root's left branch has a value less than the root
- Any node coming of the root's right branch has a value greater than the root

This is true for the main root, and for the roots of all subtrees. (In the event we had duplicate values in the tree, one of the two branches would also be assigned to hold equal values to the root, e.g., a pLessThan and a pGreaterThanOrEqual pointer).

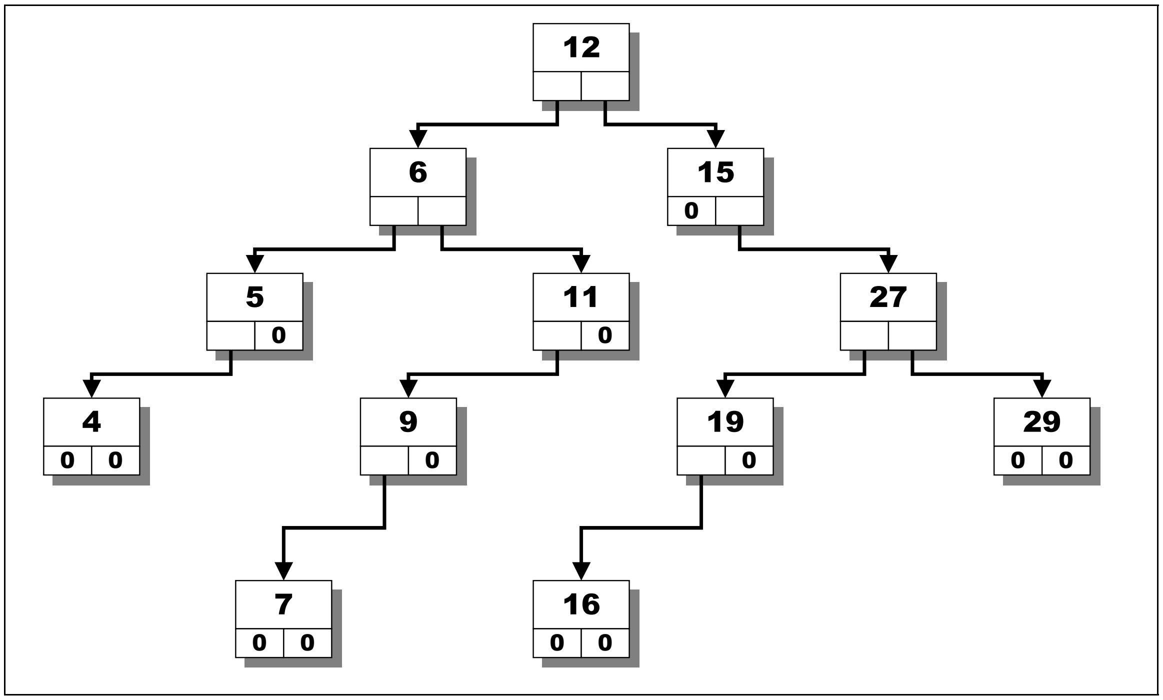


Figure 16.12: A sorted binary tree

Working with sorted binary trees
The manner in which the sorted tree is organized has a major impact on performance—particularly for activities involving search. To understand this, it is useful to compare two search functions—one which takes advantage of the sorted nature of the tree (the SearchNode() function) an one that does not (the SearchNodeDumb() function, essentially the same as our FindIntTree() function in Example 16.16). These are presented in Example 16.17, along with the definition of a SortedIntTree class that inherits from our original IntTree class.

Example 16.17: Two recursive search functions for a binary tree

```
class SortedIntTree : public IntTree
{
public:
	SortedIntTree(int nValue) : IntTree(nValue) {}
	virtual ~SortedIntTree() {}
	SortedIntTree *Left() const {return (SortedIntTree *)pLeft;}
	SortedIntTree *Right() const {return (SortedIntTree *)pRight;}
	void AddNode(int nValue);
	SortedIntTree *FindNode(int nTarget) const;
	SortedIntTree *FindNodeDumb(int nTarget) const;
	void SortNodes(vector<IntTree*> &ar) const;
	SortedIntTree *RebalanceRoot();
	SortedIntTree *Rebalance(const vector<IntTree*> &ar,int nStart,int nEnd);
	SortedIntTree *RemoveNode();
};

SortedIntTree *SortedIntTree::FindNode(int nTarget) const
{
	SortedIntTree *pFound=0;
	nFindCount++;
	if (nTarget==nValue) return (SortedIntTree *)this;
	if (nTarget<nValue && Left()!=0) pFound=Left()->FindNode(nTarget);
	if (nTarget>nValue && Right()!=0) pFound=Right()->FindNode(nTarget);
	return pFound;
}
SortedIntTree *SortedIntTree::FindNodeDumb(int nTarget) const
{
	SortedIntTree *pFound=0;
	nFindCount++;
	if (nTarget==nValue) return (SortedIntTree *)this;
	if (Left()!=0) pFound=Left()->FindNodeDumb(nTarget);
	if (pFound==0 && pRight!=0) pFound=Right()->FindNodeDumb(nTarget);
	return pFound;
}
```

Essentially, the only difference between the two functions is that FindNode() will only search one side (pLeft or pRight) if the node doesn't match, while FindNodeDumb() will search both sides—if it doesn’t find the value on the left branch. So the question becomes: how important is the difference?

As it turns out, for large search spaces, the difference is huge. Because the FindNode() function cuts the search space in half each time it drops down a level, it can exhibit exactly the same type of performance we saw in our binary search algorithm for an array. In other words, search time can be proportional to the logarithm of the number of elements being searched. In FindNodeDumb(), on the other hand, we'll end up—on average—searching half the search space if the element is there, and all the search space if it is not.

In our example tree, the number of nodes searched by the two functions is presented in Table 16.1. For larger search spaces, the differences would grow increasing huge (e.g., for a million elements, we'd be looking at ~20 for the FindNode() function vs. ~1,000,000 for the FindNodeDumb() function).

Value being searched for	SearchNode() count	SearchNodeDumb() count
7	5	7
29	4	12
28	4	12

Table 16.1: Example search results

Once a sorted tree is established, it is also very easy to turn that tree into a sorted array. An example of a function that accomplishes this is the SortNodes() function presented in Example 16.18.

Example 16.18: Generating a sorted array from a binary tree

```
void SortedIntTree::SortNodes(vector<IntTree*> &ar) const
{
	if (Left()!=0) Left()->SortNodes(ar);
	ar.push_back((IntTree *)this);
	if (Right()!=0) Right()->SortNodes(ar);
}
```

The SortNodes() function has the same pattern of arguments as the FindEven() function (Example 16.16) and works as follows:

- A recursive call is made on the left branch if it is not empty. This places lower-valued elements in the array prior to the current element.
- The current node is added to the array with a push_back() call
- A recursive call is made on the right branch if it is not empty. This places the higher value elements on the array after the current element.

The function then returns.

Test your understanding: Why must we make the recursive call on the left branch *before* we add the **this** pointer to the array in the SortNodes() function?

Creating a sorted tree

Having seen the benefits of a sorted tree for searching and sorting, the final question that remains is how to create such a tree. Not surprisingly, the answer is another simple recursive function. This function, AddNode() is presented in Example 16.19, along with the function CreateWithAdds() used to generate the Figure 16.12 tree.

Example 16.19: AddNode() function

```
void SortedIntTree::AddNode(int nVal)
{
      if (nVal<nValue) {
            if (pLeft==0) pLeft=new SortedIntTree(nVal);
            else Left()->AddNode(nVal);
      }
      else  {
            if (pRight==0) pRight=new SortedIntTree(nVal);
            else Right()->AddNode(nVal);
      }
}

SortedIntTree *CreateWithAdds()
{
      SortedIntTree *pRoot=new SortedIntTree(12);
      pRoot->AddNode(6);
      pRoot->AddNode(5);
      pRoot->AddNode(4);
      pRoot->AddNode(11);
      pRoot->AddNode(9);
      pRoot->AddNode(7);
      pRoot->AddNode(15);
      pRoot->AddNode(27);
      pRoot->AddNode(19);
      pRoot->AddNode(16);
      pRoot->AddNode(29);
      return pRoot;
}
```

The AddNode() function is quite simple in operation. It takes an integer to be added and compares it to the nValue integer.

- If the added integer is less, it looks at the pLeft pointer. If it's empty, it creates the node and places it there. If it is not empty, a recursive call is made to add the value to the left subtree.
- Otherwise, it looks at the pRight pointer. If it's empty, it creates the node and places it there. If it is not empty, a recursive call is made to add the value to the right subtree.

By looking at the sequence of calls in CreateWithAdds(), you can verify that the Figure 16.12 tree was created.

Unfortunately, our simple binary tree is not perfect. In fact, its performance depends on the fact that the elements we are adding are roughly random in order. If they are not, our AddNode() algorithm can create some serious problems. Suppose, for example, we created a new 6-element tree with the calls:

```
struct IntTree *pRoot=CreateNode(5);
AddNode(pRoot,10);
AddNode(pRoot,15);
AddNode(pRoot,20);
AddNode(pRoot,25);
AddNode(pRoot,30);
```

The resulting tree would look like the illustration in Figure 16.13. Such trees are called "unbalanced" and offer search performance properties roughly comparable to those of a linked list—that is to say, appalling.

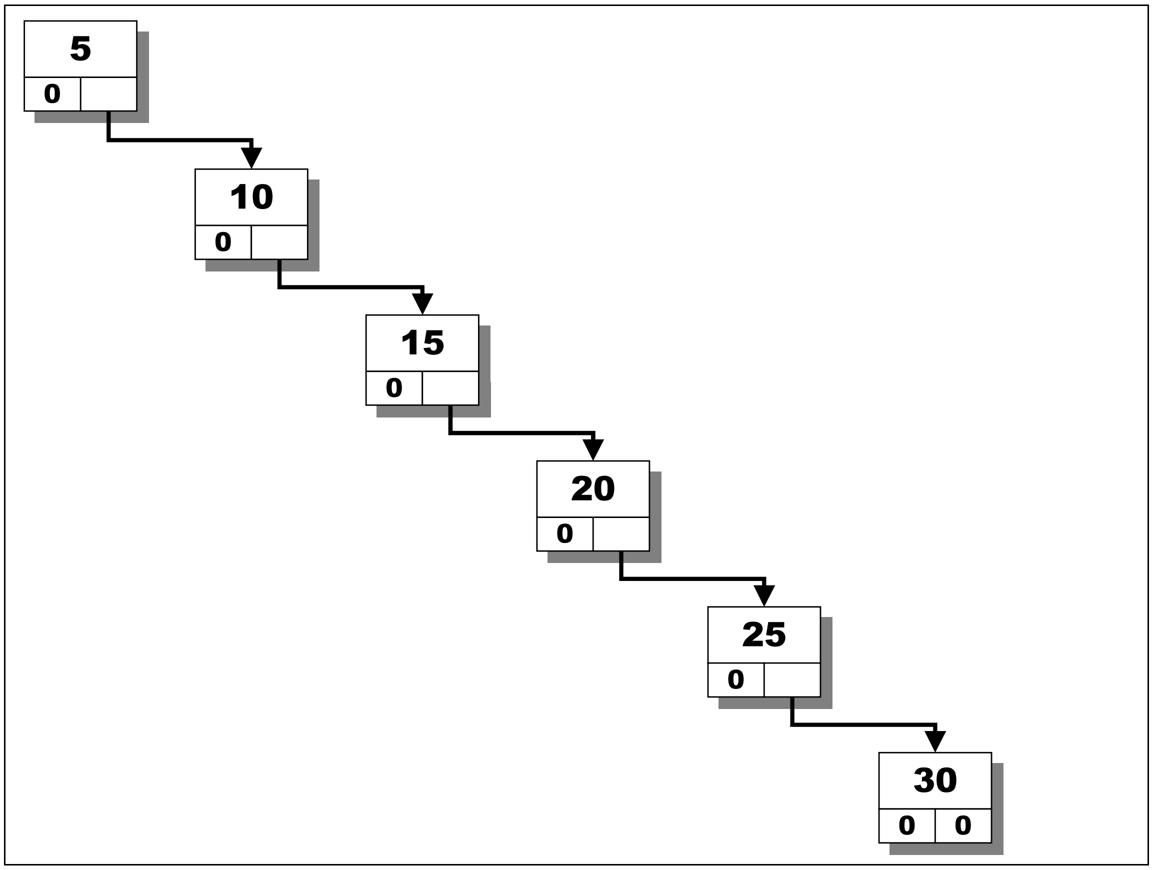


Figure 16.13: Unbalanced binary tree

So how do we avoid unbalanced trees? There are basically two ways:

1. Make sure collections are randomized with respect to their sorting key before placing them in a tree. We could use a shuffling algorithm to accomplish this, before adding elements.

2. Use an algorithm that guarantees a balanced tree is created or, once a tree has been create, rebalance it. We take this approach in Example 16.20.

So, how do we rebalance a tree? The process is actually very simple—once you start thinking recursively. Suppose you have all the elements from a tree in a sorted array. Which would you want for your new root? The obvious answer—if you want a balanced tree—is the element in the middle. Then recursion comes in. Your new problem is to choose the roots of the left and right branch trees from the arrays on the left-hand side of the root and the right hand side of the root (remembering that our array is already sorted). But, effectively, this is the same problem we just solved. So, we call our function recursively on the left side to get the left branch node, and on the right side to get the right branch node—with the non-recursive case being where the start and end of the array being placed on the tree are the same. (The reader may note the similarity of this approach to that found in Quicksort, Chapter 16, Section 16.4).

Example 16.20: Tree balancing functions

```
SortedIntTree *SortedIntTree::RebalanceRoot()
{
      vector<IntTree*> ar;
      SortNodes(ar);
      SortedIntTree *pRoot=Rebalance(ar,0,ar.size());
      return pRoot;
}

SortedIntTree *SortedIntTree::Rebalance(const vector<IntTree*> &ar,
               int nStart,int nEnd)
{
      int nMid=(nStart+nEnd)/2;
      SortedIntTree *pRoot;
      if (nStart>=nEnd) return 0;
      pRoot=dynamic_cast<SortedIntTree*>(ar[nMid]);
      if (pRoot==0) return 0;
      pRoot->pLeft=Rebalance(ar,nStart,nMid);
      pRoot->pRight=Rebalance(ar,nMid+1,nEnd);
      return pRoot;
}
```

The functions work together as follows:

- *SortedIntTree *RebalanceRoot()*: A driver function to rebalance an entire tree. It first calls SortNodes() to create the sorted array of nodes. Having done this, it can make a call to the Rebanace() function, which recursively creates the balanced tree.
- *SortedIntTree *Rebalance(vector<IntTree *> &arNodes,int nStart,int nEnd)*: Takes a sorted vector<> array (arNodes) and creates a tree using the elements from nStart to nEnd-1. It is recursive, and returns 0 when nStart>=nEnd, the non-recursive case. Otherwise, it selects the node at middle position (nStart+nEnd)/2

as the root of the new tree, then makes recursive calls on the left and right side of the array.

Removing Nodes from a Tree
Unfortunately, adding nodes to a binary tree is a lot simpler than removing them. There is a "simple way" to handle the problem, but it is not entirely satisfactory. Once again, the problem of keeping the tree as balanced as possible comes into play.

The "simple" approach to deleting a node is to perform one of two possible operations:

- Take the former right branch of the deleted node and add it to the rightmost path associated with the left node. The former left node can then be attached to the parent of the deleted node.
- Take the former left branch and add it to the leftmost path associated with the former right node. The former right node can then be attached to the parent of the deleted node.

These two approaches, using our sample tree (Figure 16.12) are illustrated in Figure 16.14. While the first approach seems to work okay, in this example, the second clearly makes the tree unbalanced. Indeed, the problem with this approach—in general—is that it tends to produce increasingly unbalanced trees. Since, most of the time, insertions will be much more common than deletions in memory-based trees, this may not be a problem. But, there are other ways to approach the problem.

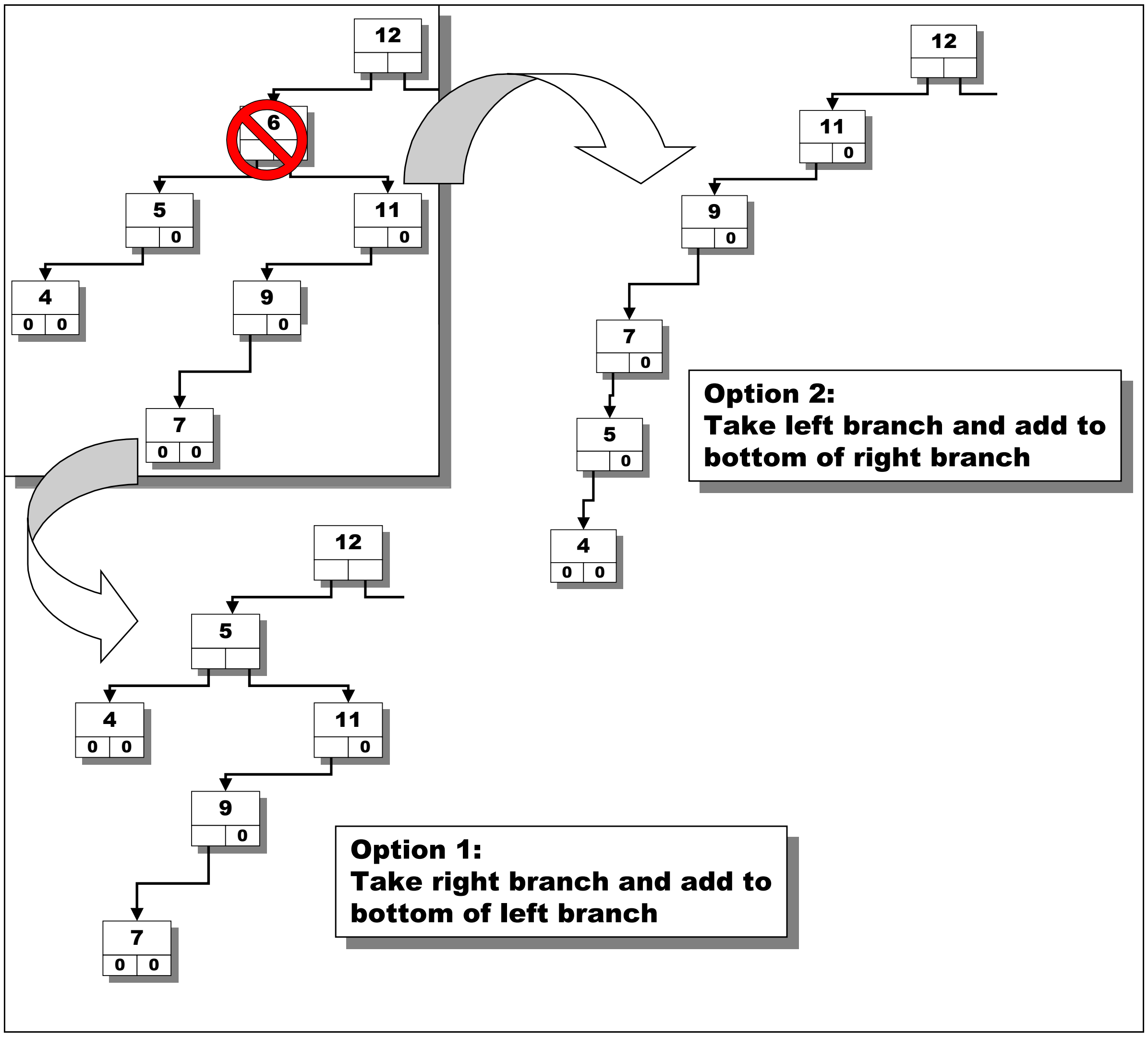

12
6
5
0
11
0
4
0 0
9
0
7
0 0
12
11
0
9
0
7
0
5
0
4
0 0
Option 2:
Take left branch and add to bottom of right branch
12
5
4
0 0
11
0
9
0
7
0 0
Option 1:
Take right branch and add to bottom of left branch

Figure 16.14: Reassembling nodes after removal

An alternative approach—and the one that we will illustrate—is to create a rebalanced tree using the nodes below the deleted node. This is accomplished in the function RemoveNode(), presented in Example 16.21. The function takes the node we are removing as its argument, and returns a new balanced tree containing all the nodes *except* the removed node. As a consequence, we'd tend to call it in a way that reassigns a branch, for example:

```
pTree->pLeft=RemoveNode(pTree->pLeft);
```

It would also be possible to write a version of the function that also takes the root of the tree as an argument, along with the node to be removed. That function is presented as an end-of-chapter exercise.

Example 16.21: RemoveNode() function

```
SortedIntTree *SortedIntTree::RemoveNode()
{
	SortedIntTree *pRoot=0;
	vector<IntTree*> ar;
	// Use malloc() in C
	if (Left()!=0) Left()->SortNodes(ar);
	if (Right()!=0) Right()->SortNodes(ar);
	pRoot=Rebalance(ar,0,ar.size());
	pLeft=0;
	pRight=0;
	return pRoot;
}
```

The function, which is very similar to the RebalanceRoot() function in Example 16.20, works as follows:

- We call SortNodes() on the left and right branches of the node we are removing, but we do not add the node itself (since we'll be deleting it). This gives us a sorted array of all the nodes beneath the one we are deleting.
- We rebalance our sorted array with a call to Rebalance() (Example 16.20). The return value of that function will become the new root of the tree that used to belong to the deleted node
- Finally, we reset the left and right pointers of the current node to 0, so when we delete the node we are removing we do not recursively delete nodes that still exist in the tree.

In the event our count of nodes under the deleted node is 0, we will be returning 0 (the value returned by Rebalance(). To actually accomplish the node remove we need code such as the following:

```
// Deleting left-branch node of tree
SortedIntTree *pOld=pTree1->Left();
pTree1->pLeft=pTree1->Left()->RemoveNode();
delete pOld;
// Replacing the root of the tree
pOld=pTree1;
pTree1=pTree1->RemoveNode();
delete pOld;
```

In the first removal, we were eliminating the node on the left branch of the tree, (e.g., the 6 branch in Figure 16.12). We first get a pointer to that node so we can delete it once it has been removed. We then call remove node, replacing the tree's left pointer with the new balanced tree. We then delete the node that was removed.

The same procedure is used in deleting the root node of the tree. Again, we get our pointer to the old root, then call RemoveNode() on the root—placing the return value in our pTree1 pointer (which is needed to keep track of the root of the tree). We can then delete the old root node.

16.5.4: General Trees

Trees are often used to implement large indexes, such as might be used in database applications. In such cases, the trees are normally not binary trees but, instead, can have arbitrarily large numbers of children.

A particularly common form of higher order tree is referred to as a balanced tree. Balanced trees, or b-trees as they are more commonly called (with variations like the b+-tree), are tree-based data structures which remedy a number of the deficiencies in binary trees described earlier in the chapter. In particular, b-tree can be designed to:

- Remain balanced no matter what the order of input data.
- Operate reasonably efficiently from disk, instead of having to reside entirely in memory.

As a consequence, these trees are commonly encountered in on-disk indexing or sorting applications.

The ability to operate efficiently from disk is mainly a property of the ability to adjust the size of tree nodes to conform to hardware capabilities. Unlike binary trees, which have a fixed number of pointers (2), btree nodes can be designed with any number of pointers—so that the size of a node can be made to precisely match the minimum amount of data read from disk in a single access, referred to as the "block size". Conceptually, such nodes are contrasted with binary nodes in Figure 16.15.

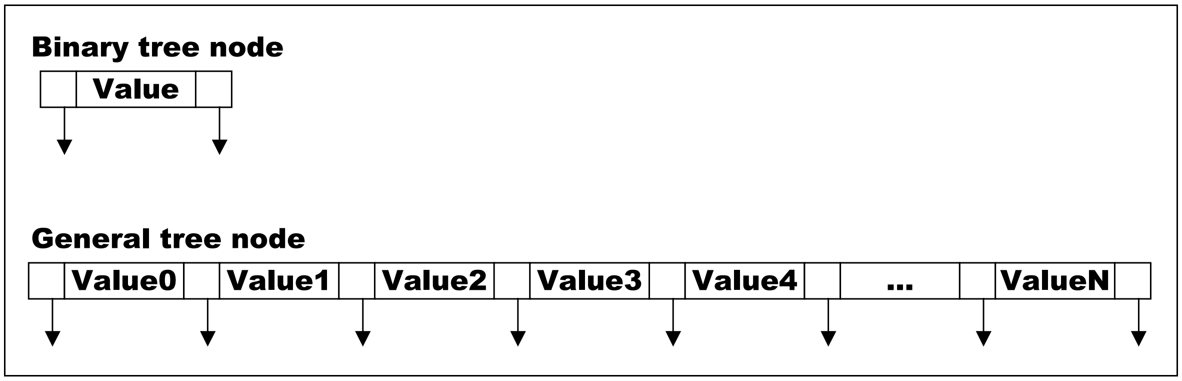


Figure 16.15: Binary tree nodes vs. General tree nodes

By convention, for a b-tree of order N, there are 2N possible data elements and 2N+1 node pointers.

The other way in which b-trees are different from our binary tree algorithm relates to how they grow. In our algorithm, new nodes always were added to the bottom of the tree, with no attention being paid to how deep the particular path was relative to the size of the tree. In b-tree algorithms, on the other hand, data is always located on the same level—the leaf level—of the tree. As new data is added, eventually the nodes at higher levels begin to fill up. When this happens, they split apart and—eventually—a new node may be created at the top of the tree. The practical result of this algorithm is that the tree remains in balance, regardless of the order in which elements are added.

16.5 Section Questions

1. Explain why a tree shape allows you to reach a node faster than a linked list—if you know where the node is located in the tree.
2. Explain why recursion is so much more commonly used in tree than with other collection shapes.
3. Explain why traversing a sorted binary tree is a lot like binary search.
4. If you had an array that was sorted and wanted to add its elements to a binary tree, whate are two approaches you might take to avoid an unbalanced tree?
5. Why is an unbalanced tree like a linked list?
6. Why is "block size" important in general tree design?
7. Why would the size of a collection's key impact the structure of a general tree?

16.6: Lab Exercise: A Generic Tree Collection

Walkthrough available in GenTree.avi

Just as it is possible to write a general-purpose list collection, it is possible to write a reasonably generic sorted tree collection. The one difference between the tree and the list, however, is that our generic tree elements need to keep a copy of their sorting key, as well as a pointer to the associated data object. Thus, the collection can be viewed as "general purpose" for objects having the same key type (e.g., int, char *, double). It could, for example, be viewed as an alternative to a hash_map<> or map<> collection where many insertions and deletions were anticipated.

16.2.1: Overview of StringTree Collection

The StringTree collection is sufficiently similar to the IntTree structure discussed in this chapter that its functionality needs little discussion. The class declaration is presented in Example 16.22.

Example 16.22: The StringTree class declaration

```
class StringTree {
public:
      StringTree(const char *szK,void *pV);
      virtual ~StringTree();
      void AddNode(const char *szK,void *pV);
      StringTree *FindNode(const char *szK) const;
      void SortNodes(vector<StringTree*> &arNodes) const;
      StringTree *RebalanceRoot();
      StringTree *Rebalance(const vector<StringTree*> &arNodes,
                            int nStart,int nEnd);
      StringTree *RemoveNode();
      string szKey;
      void *pValue;
      StringTree *pLeft;
      StringTree *pRight;
};
```

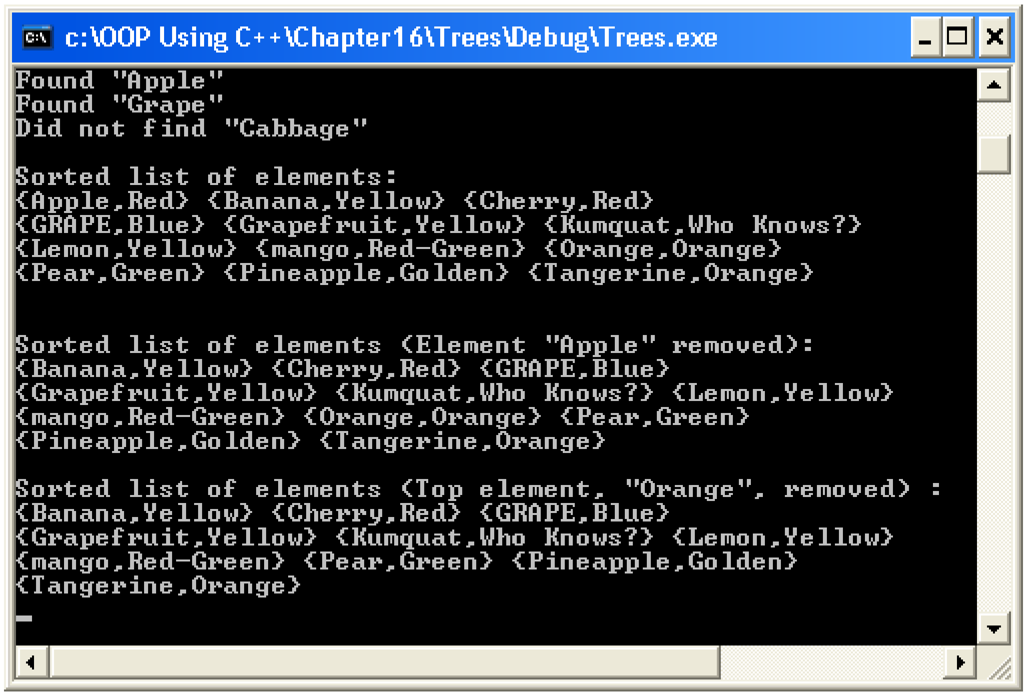
Its key difference from the IntTree collection is that it has a **string** key—which we'll assume is case *insensitive*—and a void pointer to an associated data object. In this way, it resembles our generic list collection, presented in Section 16.3.

16.6.2: Overview of StringTree Functions

Because the StringTree member functions are identical in purpose and general design to the SortedIntTree functions presented in Section 16.1.3, their purpose is not described here. The use of the StringTree functions is illustrated in Example 16.23. The resulting output is presented in Figure 16.16.

Example 16.23: StringTree demonstration function

```
void SortedStringTreeDemo()
{
        int i;
        string arFruit[12] = {"Orange", "Apple", "Pear", "Banana", "Tangerine", "Kumquat",
                "Lemon", "Pineapple", "Grapefruit", "GRAPE", "Cherry", "mango"};
        const char *arColor[12] = {"Orange", "Red", "Green", "Yellow", "Orange", "Who
Knows?",
                "Yellow", "Golden", "Yellow", "Blue", "Red", "Red-Green"};
        vector<StringTree*> arNodes;
        StringTree *pTree=new StringTree(arFruit[0].c_str(),(void *)arColor[0]);
        for(i=1;i<12;i++) pTree->AddNode(arFruit[i].c_str(),(void *)arColor[i]);
        cout << endl;
        if (pTree->FindNode("Apple")) cout << "Found \"Apple\"";
        else cout << "Error: Did not find \"Apple\"";
        cout << endl;
        if (pTree->FindNode("Grape")) cout << "Found \"Grape\"";
        else cout << "Error: Did not find \"Grape\"";
        cout << endl;
        if (pTree->FindNode("Cabbage")) cout << "Error: Found \"Cabbage\"";
        else cout << "Did not find \"Cabbage\"";
        cout << endl << endl;
        pTree->SortNodes(arNodes);
        cout << "Sorted list of elements: ";
        cout << endl;
        for(i=0;i<(int)arNodes.size();i++) {
                cout << "{" << arNodes[i]->szKey << "," << (char *)(arNodes[i]->pValue) <<
"} ";
                if (i%3==2) cout << endl;
        }
        cout << endl << endl;
        StringTree *pOld=pTree->pLeft;
        pTree->pLeft=pTree->pLeft->RemoveNode();
        delete pOld;
        arNodes.clear();
        pTree->SortNodes(arNodes);
        cout << "Sorted list of elements (Element \"Apple\" removed): ";
        cout << endl;
        for(i=0;i<(int)arNodes.size();i++) {
                cout << "{" << arNodes[i]->szKey << "," << (char *)(arNodes[i]->pValue) <<
"} ";
                if (i%3==2) cout << endl;
        }
        cout << endl << endl;
        pOld=pTree;
        pTree=pTree->RemoveNode();
        delete pOld;
        arNodes.clear();
        pTree->SortNodes(arNodes);
        cout << "Sorted list of elements (Top element, \"Orange\", removed) : ";
        cout << endl;
        for(i=0;i<(int)arNodes.size();i++) {
                cout << "{" << arNodes[i]->szKey << "," << (char *)(arNodes[i]->pValue) <<
"} ";
                if (i%3==2) cout << endl;
        }
        cout << endl;
        delete pTree;
}
```



```
Found "Apple"
Found "Grape"
Did not find "Cabbage"

Sorted list of elements:
{Apple,Red} {Banana,Yellow} {Cherry,Red}
{GRAPE,Blue} {Grapefruit,Yellow} {Kumquat,Who Knows?}
{Lemon,Yellow} {mango,Red-Green} {Orange,Orange}
{Pear,Green} {Pineapple,Golden} {Tangerine,Orange}


Sorted list of elements (Element "Apple" removed):
{Banana,Yellow} {Cherry,Red} {GRAPE,Blue}
{Grapefruit,Yellow} {Kumquat,Who Knows?} {Lemon,Yellow}
{mango,Red-Green} {Orange,Orange} {Pear,Green}
{Pineapple,Golden} {Tangerine,Orange}

Sorted list of elements (Top element, "Orange", removed) :
{Banana,Yellow} {Cherry,Red} {GRAPE,Blue}
{Grapefruit,Yellow} {Kumquat,Who Knows?} {Lemon,Yellow}
{mango,Red-Green} {Pear,Green} {Pineapple,Golden}
{Tangerine,Orange}
_
```

Figure 16.16: Output from StringTree demonstration function

16.6.3: Procedure

The procedure for the Generic Tree lab is straightforward:

- Use the SortedIntTree (Section 16.1.3) as a starting point for creating your class
- Add the appropriate data members and modify the function arguments
- Implement the case-insensitivity in your insertion and search functions

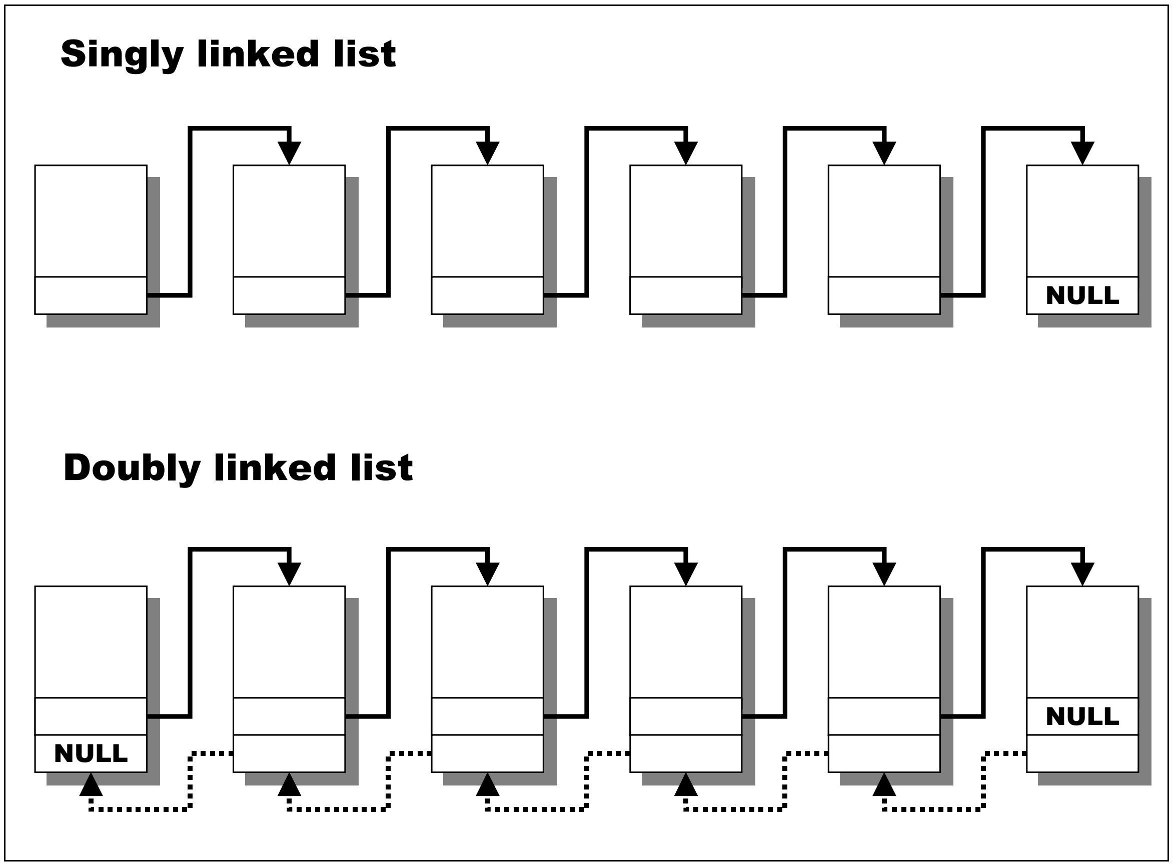
Write code to test the class, such as that in Example 16.23.

16.7: Review and Questions

16.7.1: Review

The chapter begins by introducing self-referential structures—structures or class objects that contain embedded pointers to objects of the same type. Using these objects, it is possible to represent rich conceptual relationships between objects, such as an organization chart, as opposed to mere collections of objects.

One of the most important collections utilizing self-referential structures is the linked list. The embedded object pointers in such lists allow chains of objects to be maintained, as illustrated below. Where pointers link objects in only one direction, the list is known as a singly linked list. Where pointers link objects in both directions, the collection is known as a doubly linked list.



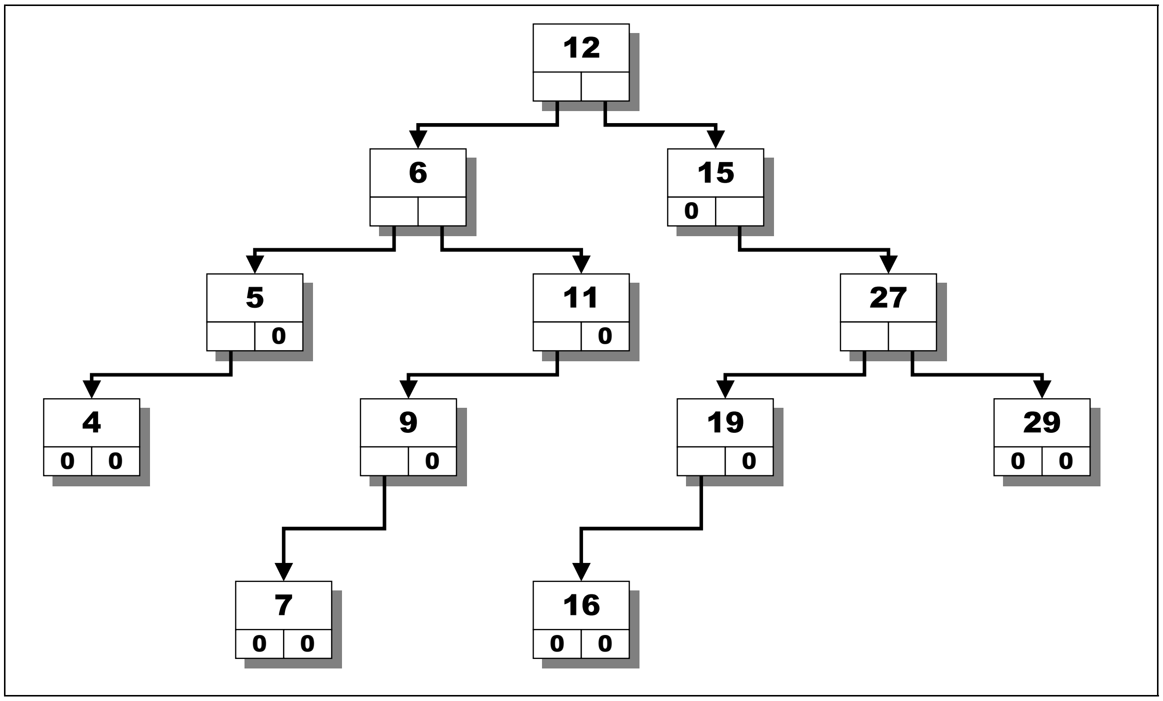
The key benefit of the linked list collection shape is the ability to insert or remove objects anywhere in the collection very quickly. In this regard, the linked list is far superior to the array shape. On the other hand, linked lists do not support random access, making them more like a tape than a CD.

The STL provides a **list<>** template class that implements a linked list. The class takes a single parameter, the type of object being collected. It supports bi-directional iterators, consistent with the doubly linked list shape.

The tree shape is also a self-referential structure consisting of elements—normally called nodes—that are linked together. The linkages in a tree are often called branches. If node A has a branch that points to node B, node A is referred to as the parent node of B. Node B, on the other hand, is referred to as child node of A. What distinguishes a tree from a list and other collections (e.g., a graph), is that the linkages are in the following pattern:

- Each node has one—and only one—parent. The only exception element is the node at the top of the tree, the root, which has no parent.
- Each tree node can point to two or more child nodes. Any number of these branches may be empty.

A tree with only two branches is known as a binary tree. Binary trees are the most common form of trees used when trees are constructed in memory. A particularly important type of binary tree is the sorted binary tree, which has the properties that the node values of all elements on the left (or less-than) branch of a node are less the node's value, whereas all values on the right (or greater-than) branch are greater than the node's value (one of the two branches may also hold equal values). A depiction of such a tree is shown in the figure below.



Functions that operator on sorted binary trees are nearly always recursive, tend to be small, and generally follow a pattern along the following lines:

- Perform some action based on the contents of the current node
- Perform a test, then—depending upon the results—perform some action on the left branch (most commonly a recursive function call).
- Perform a test, then—depending upon the results—perform some action on the right branch (most commonly a recursive function call).

The order of these actions depends upon the nature of the task to be performed. For example, an ascending sort routine for a binary tree would:

- if left branch is not empty, sort left branch
- add node element
- if right branch is not empty, sort right branch

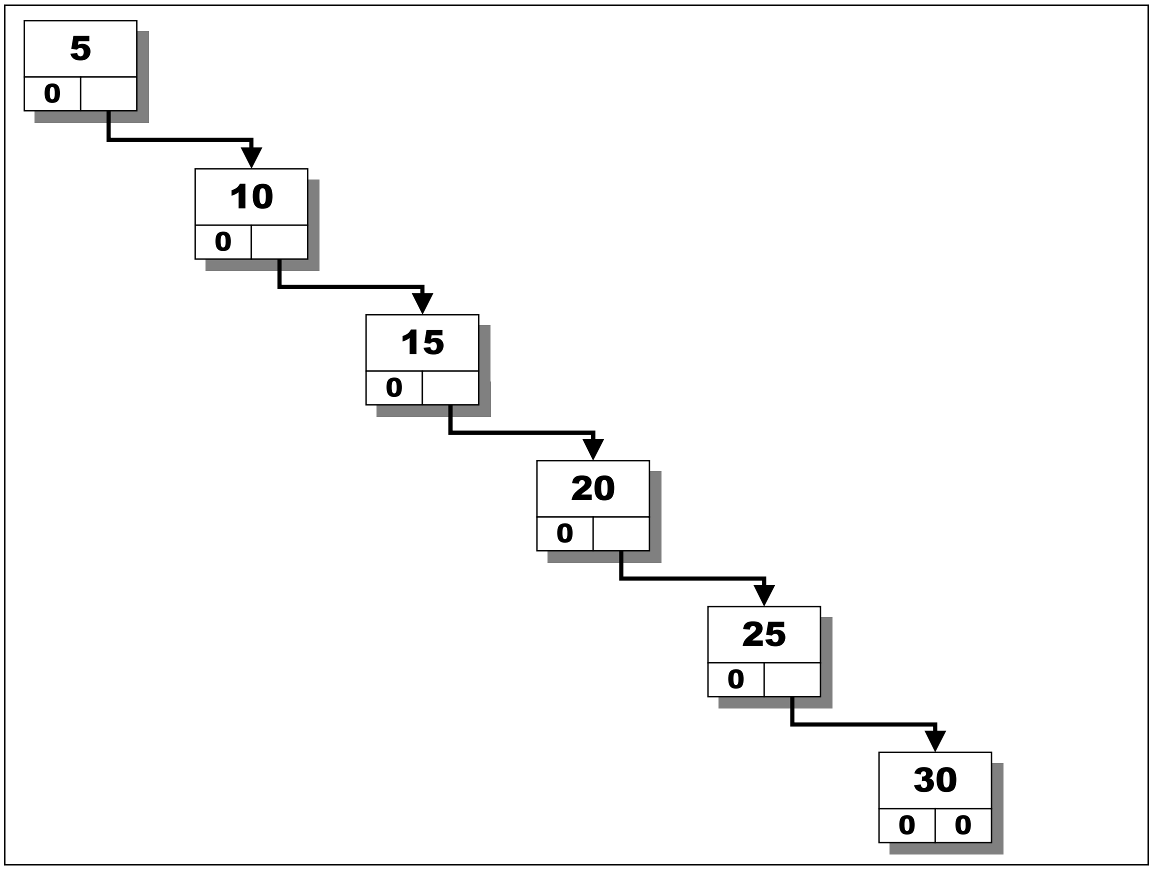
A descending sort, on the other hand would:

- if right branch is not empty, sort right branch
- add node element
- if left branch is not empty, sort left branch

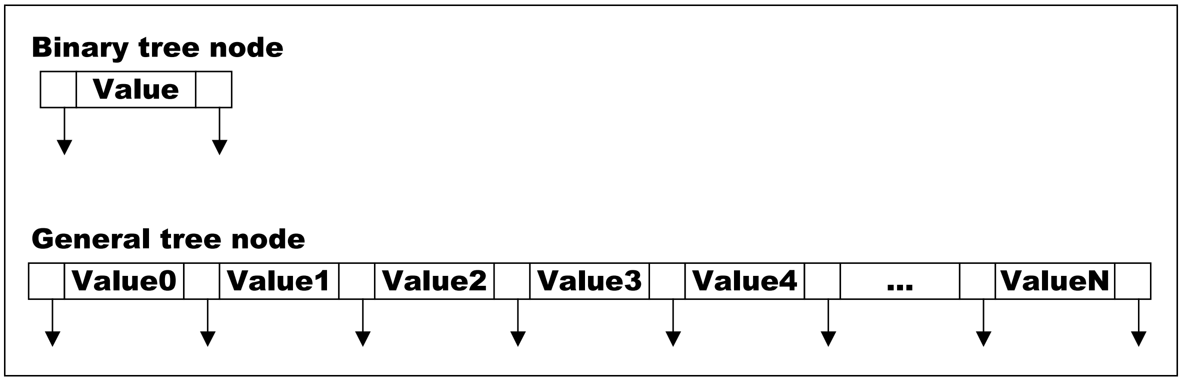
A node addition routine, in contrast, would:

- compare the current node to the value being added
- if added value is less than current node, check left branch. If it is empty, add the node directly, otherwise make a recursive call.
- otherwise, check right branch. If it is empty, add the node directly, otherwise make a recursive call.

When creating sorted binary trees using the above adding routine, it is possible for the tree to become unbalanced—meaning a preponderance of nodes are hanging off the branch on one side (as shown in the figure below). Such a tree, for example, could be created the values being added (5,10,15,20,25,30) were already in ascending order. Various algorithms are available for rebalancing trees. There are also algorithms that can be used to keep trees in balance as nodes are added.



Tree collections are commonly used to store information, such as file indices, on disk. The nodes for such trees are typically designed to accommodate the disk performance characteristics, such as block size. For this reason, such trees are typically not binary trees, but have a more general node structure, as shown below. Such trees normally employ self-balancing algorithms for additions and deletions, since their entire purpose is to enhance search performance.



16.7.2: Glossary

Balanced Tree – A sorted tree where the number of right children and left children of each node tend to be roughly equal (true for all branches, for general trees).
Balancing Algorithm – An algorithm for rebalancing a tree, or for keeping a tree balanced as nodes are added or removed.
Bi-directional Iterator – An iterator that can be used front to back or back to front but cannot be used for random access
Binary Tree – A tree where each node has two branches to child nodes.
Block Size – The minimum amount of data processed in disk read or write operation. Tends to be operating system specific.
Branch – A term used to describe a self-referential link from one tree node to another.
Child Node – If node A has a branch that points to node B, node B is referred to as child node of A.
Doubly Linked List – A list collection shape supporting bi-directional iteration.
General Tree – A tree with whose maximum number of child branches is not limited to 2. Nodes for such trees are often designed to accommodate performance characteristics of a system, such as block size.
Index File – A file (or part of a larger database file) that contains a keyed index of pointers designed to enhance database performance, normally stored using a general tree collection.
Linked List (or List) – A collection shape that implements bi-directional iteration and rapid element insertion and deletion, but does not support random access by position or content.
list<> – The STL class template that implements a doubly linked list.
Node – A term used to describe an element in a tree collection.
Parent Node – If node A has a branch that points to node B, node A is referred to as the parent node of B.
Reverse Iterator – An iterator that can be used to traverse a collection from back to front. By definition, a bi-directional iterator implies the existence of a reverse iterator.
Self-balancing Tree – A tree where node addition and deletion algorithms incorporate provisions the keep a tree continually balanced.
Self-referential Structure – A structure or class that contain embedded pointers to objects of the same type.
Singly Linked List – A list shape allowing iteration in only one direction (normally forward).
Sorted Binary Tree – A sorted binary tree present where all child nodes on the left (or less-than) branch are less a node's value, while all child node values on the right (or greater-than) branch are greater than the node's value.
Tree – A collection of node objects where each node has, at most, a single parent node and can point to at least two child nodes.
Unbalanced Tree – A sorted tree where the number of right children and left children of each node tend to be substantially different.

16.7.3: Questions

16.1: Promotion function. Add a member function Promote() to the employee class in Example 16.1 that moves an employee up a level in the organizational hierarchy. You may assume that a promoted employee is: a) moved to a level equivalent to that of his/her former boss, b) shares the same boss with his/her former boss, and c) keeps the same subordinates. You should also prevent promotion of the CEO and any promotion that would lead to two CEOs.

16.2: Replace employee function. Add a member function Replace() to the employee class in Example 16.1 that removes an employee (the object being operated on) from the organizational hierarchy and adds a replacement employee (specified as an reference argument). You may assume that the new employee will: a) have the same boss as the replaced employee, b) takes on any subordinates of the replaced employee, and c) keeps any existing subordinates subordinates. The replaced employee, on the other hand, should have his/her boss and any subordinates removed.

16.3: string List. Implement a StringList class, modeled after the int_list (Section 16.2.1) that maintains a list of STL string objects.

16.4: Resetting a list. Add a new member function to the int_list class (Section 16.2.1) called ResetList(). This function, returning void and taking no arguments, should set all pointers in the list (i.e., links) to NULL, no matter what list object it is applied to.

16.5: Splicing in a list before a position. Add a new overload to the InsertBefore() member of the GPtrList collection (Section 16.3) that takes a list as an argument and splices its objects into the list object the function is applied to, effectively transferring the argument list into the list object. The function should be prototyped as follows:

POSITION InsertBefore(POSITION pos,GPtrList &list);

After performing the splice (which should use the same position objects used in the argument list), the function should remove all elements from the argument list. It should return the first POSITION pointer in the splice.

16.6: Splicing in a list after a position. Add a new overload to the InsertAfter() member of the GPtrList collection (Section 16.3) that takes a list as an argument and splices its objects into the list object the function is applied to, effectively transferring the argument list into the list object. The function should be prototyped as follows:

POSITION InsertAfter(POSITION pos,GPtrList &list);

After performing the splice (which should use the same position objects used in the argument list), the function should remove all elements from the argument list. It should return the first POSITION pointer in the splice.

16.7: Maximum value in a list: Write a recursive function to return the maximum value in an int_list that starts from anywhere in the list. (Hint: once you figure out the non-recursive case—which is at, but not past, the end of the list—the function is straightforward).

16.8: Find all matching elements: Suppose we wanted to modify the FindIntTree() function in Example 16.16 so that it found all the matching elements in the tree (e.g., the tree in Figure 16.14 has two nodes with a value of 12), as opposed to just finding the first. How would we need to modify the function arguments and the function itself? (Hint: the FindEven() function in Example 16.16 is a good model).

16.9: Modified node removal function. Write a function, with the prototype:

```
struct IntTree *RemoveNodeTree(struct IntTree *pRoot,struct IntTree *pRemove);
```

that removes a node (pRemove) from a tree (pRoot). The function should return the root of the tree (which will be the same as pRoot unless pRemove==pRoot). You may, if you wish, call RemoveNode() within your function. (Hint: you'll probably want to write a function:

```
struct IntTree *FindParent(struct IntTree *pRoot,struct IntTree *pNode);
```

that returns the parent of its argument (pNode) in the tree (pTree) help you.)

16.10: Simple node removal function: Write a function that performs node removal either Option 1 or Option 2 node removal (as described in Figure 16.16). Use the same prototype as the RemoveNodeTree() function above.

16.11. Tracing though a tree. Identify the path through the nodes that resulted in the numbers in Table 16.1 for each function.

16.12: Recursive tree traversal function. A simple loop that will find the rightmost branch of any node (pNode) in a tree such as the int_tree is as follows:

```
while(pNode->pRight!=0) pNode=pNode->pRight;
```

Rewrite it as a recursive function.

Chapter 17

Introduction to Grammars

Executive Summary

Chapter 17 introduces the concept of a grammar, which is a set of rules used to determine if an expression is legal or illegal. Grammars are used extensively both in everyday life (e.g., language) and in programming (e.g., programming language grammars, expression grammars). Although the study of grammars is highly involved, even a quick introduction can lead to a significant enhancement in an individual's programming skills and range.

The chapter begins by explaining the function of a grammar. A simplified version of the Backus Naur/Normal form (BNF) notation for representing a grammar is then presented. Variations on this representation are also presented. Exercises in applying grammars to expressions are then explained, and a sample grammar describing arithmetic and logical expressions is used as an illustration. Two approaches for translating Roman Numerals to integers are then contrasted: one code-based and one grammar-based. The superiority of the grammar-based approach should be readily evident to the reader. Finally, a lab exercise involving the development of a symbolic calculator—using the previously developed sample grammar—is specified.

Learning Objectives

Upon completing this chapter, you should be able to:

- Explain the role of a grammar
- Write simple grammars in BNF notation
- Determine whether expressions are valid or invalid based on a BNF grammar
- Create and interpret a tree-based diagram that relates an expression to a grammar
- Identify some issues that contribute to the complexity of a grammar
- Describe the key role played by recursion in most grammars
- Derive your own grammars for simple expressions
- Identify situations where processing of user input can benefit from the use of a grammar
- Write functions that determining if a particular expression is legal according to a simple grammar
- Write functions to evaluate expressions, using a grammar to organize the program structure

17.1: Grammars

In this section, we introduce the concept of a grammar—a useful tool for defining I/O data that is often represented in tree form. We begin by looking at grammars in an abstract sense, then introduce a simplified notation (based on Backus-Naur form, or BNF) for specifying a grammar.

17.1.1: What is a Grammar?

A grammar is a set of rules for determining if an expression is legal or not. For example, languages have their own grammars for determining if a sentence is valid. We learn these rules through listening to others speak and, more formally, as part of our education. Indeed, the importance of grammars is underscored by the fact that the school years in which grammar was taught were traditionally referred to as "grammar school".

Consider how we are taught grammars in English. We might begin with one rule that states that an example of a legal sentence structure is a *subject* followed by a *predicate*. We might then learn that an example of a legal *subject* is an *article* (e.g., the or a) followed by a *noun phrase*. We might then be told that a noun phrase is either a *noun* or an *adjective* followed by another *noun phrase*. And so forth... To determine if a sentence is legal, we need to demonstrate that is constructed according to the rules of the grammar.

Just as in programming, recursion in a grammar occurs when an object defined by the grammar references itself, either directly or indirectly. The above definition of *noun phrase*, for example, is recursive because it states that a noun phrase can consist of an *adjective* followed by another *noun phrase*. By allowing for such recursion in our definitions, we can compactly specify the legality of infinitely many possible combinations. Continuing with our example, suppose speedy, red and shiny are all adjectives, while bicycle is a noun. Then our sentence grammar would assert all of the following to be noun phrases:

- bicycle
- red bicycle
- shiny speedy red bicycle

On the other hand, neither "red", "bicycle red" nor "shiny speedy bicycle red" would be legal noun phrases according to the rules we specified. Thus, unless other rules (not specified here in our simple grammar) make them legal, they would not be part of a legal sentence.

> *Test your understanding:* In the above grammar, a subject must always end with a noun. Explain why.

The correspondence between grammars and trees is illustrated in Figure 17.1, which created simple sentence diagram for the phrase "The bright red bicycle travels swiftly".

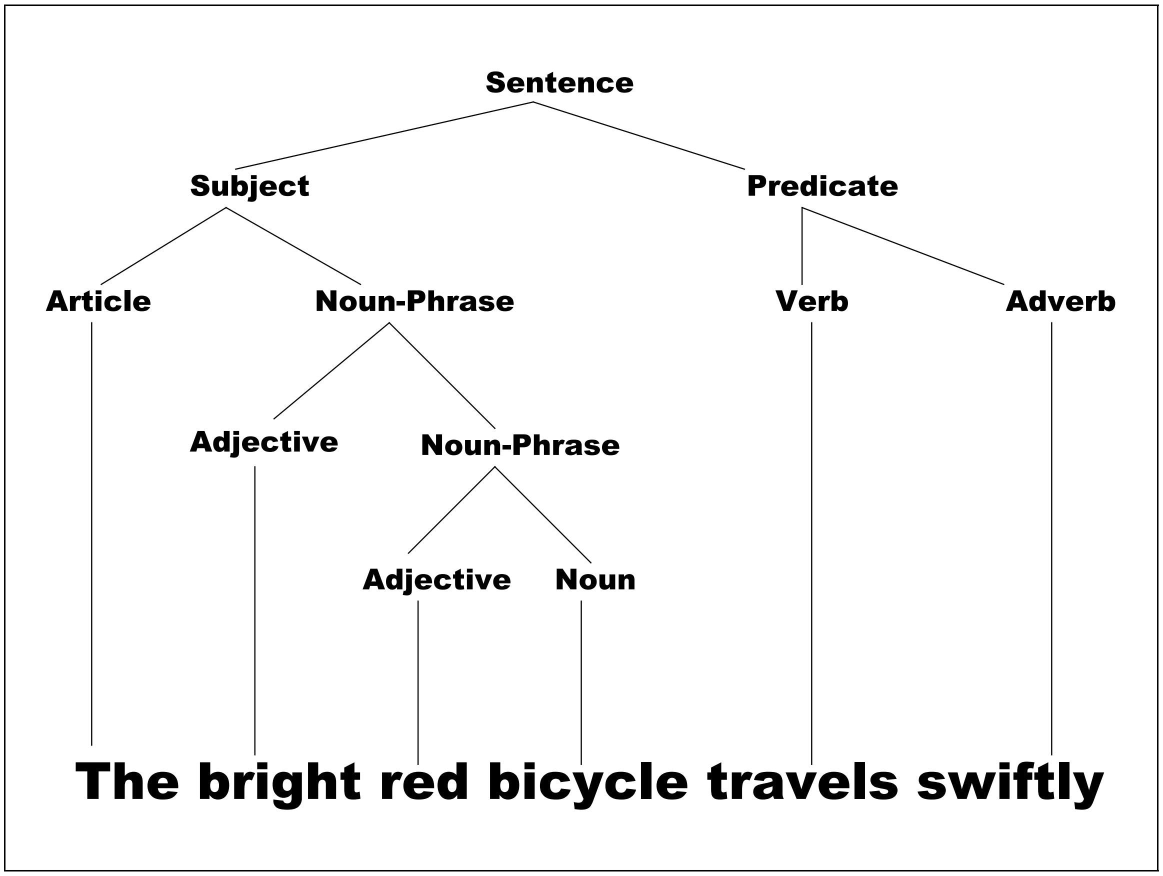


Figure 17.1: A sentence diagram showing the tree structure of a grammar

17.1.2: A Notation for Specifying a Grammar

A common way to describe simple grammars uses a notation called Backus Naur/Normal form (BNF). In its simplest form, it has the following structure:

- names of the elements of our grammar are placed between < and >
- the symbol ::= is used to indicate "is replaced by"
- the vertical bar (|) is used to signify or, and iidentifies alternative ways of replacing expressions.
- Literal constants, or tokens, are as written.

Example Grammar
The easiest way to understand the notation is with an example. Consider the following hypothetical grammar:

(1) <valid-expression> ::= <a-expression> | <b-expression> | <c-expression>
(2) <a-expression> ::= A <b-expression> | A
(3) <b-expression> ::= B <a-expression> | B <b-expression>
(4) <c-expression> ::= C <b-expression> | C

According to these four rules, written in BNF notation, we can determine if a sequence of letters represents a valid expression. Consider, for example, the following expressions:

i. A
ii. AB
iii. ABA
iv. ADAAAAB
v. CBBBABBA
vi. ABBBCBA

Using the grammar we just specified, we can assess whether or not each expression is legal or not. The results are discussed in Table 17.1.

Expression	Legal?	Explanation
i) A	Legal	It is an <a-expression> (Rule 2) which is a <valid-expression>, according to Rule (1)
ii) AB	Not Legal	Because it begins with an A, it would be valid only if we could show B was a valid <b-expression> (Rule 2). However, a <b-expression> cannot terminate with a B (Rule 3), so B is not a valid B expression.
iii) ABA	Legal	Again, we look for an A expression followed by a B expression (Rule 2). BA is a valid B expression (Rule 3) because it consists of a B followed by an <a-expression> (since A, by itself, is a valid <a-expression>, according to Rule 2).
iv) ADAAAAB	Not Legal	There is no symbol D in the grammar.
v) CBBBABBA	Legal	Since it begins with C, we begin by applying Rule 4. We then apply Rule 3 on BBBABBA. Since we can always place a B in front of a B expression, the problem becomes is ABBA a valid <a-expression>? Applying Rule 2, the question becomes is BBA a valid B expression? Back to Rule 3, we strip off two Bs and are left with the question is A a valid A expression. It is, based on Rule 2.
vi) ABBBCBA	Not Legal	The only place a <c-expression> is found is in Rule 1 and Rule 4. Based on this, the only place a C can appear is at the head of the expression

Table 17.1: Examples of Legal and Illegal Expressions

The use of | operators in BNF can be a bit cumbersome as grammars grow large. Thus, a slightly modified notation of BNF is often used to describe the syntax of computer languages, such as C/C++, and expressions. Rather than using the | and ::=, the notation places possible values for each expression on separate lines. For example, the hypothetical grammar specified in could be rewritten as:

```
valid-expression :
        a-expression
        b-expression
        c-expression

a-expression :
        A b-expression
        A

b-expression :
        B a-expression
        B b-expression

c-expression :
        C b-expression
        C
```

Which notation is used to describe a grammar is largely a matter of convenience.

Grammar Complexity
In the field of computer science, a great deal has been written about the complexity associated with different grammars. Analogous to collection shapes, the complexity properties of a grammar address issues such as:

- *Is the grammar ambiguous?* Some grammars, such as those for natural languages, allow multiple interpretations of the same expression. A well-worn phrase illustrating this is "Time flies like an arrow", which can be interpreted in 4 ways: as a philosophical statement about the passage of time, as a statement about the eating preferences of a particular species of fly (think "fruit flies like a banana"), and as two sets of moderately incomprehensible instructions to an individual with a stopwatch (e.g., "time hurdlers like [i.e., the way you would time] a sprinter" or "time hurdlers like [i.e, the way a] a referee [would]").
- *Combinatorial properties:* What minimum rate of increase in processing is required as the length of expressions being evaluated increases (e.g., exponential, N-squared, etc.)? Processing requirements as the size of the grammar grows (e.g., number of expressions) are also considered.
- *Completeness:* Can the grammar categorize all legal and illegal expressions?

Even relatively simple grammars can have very different properties. Our hypothetical letter grammar, for example, was simplified by the fact that each of our expressions (excluding the top level expression) began with a token. It would be possible, however, to construct a grammar where this was not the case, e.g.:

(1) <valid-expression> ::= <a-expression> | <b-expression> | <c-expression>
(2) <a-expression> ::= A <b-expression> | A
(3) <b-expression> ::= B <a-expression> | <c-expression> B
(4) <c-expression> ::= C <b-expression> | <a-expression> C

In rules (3) and (4) of this revised grammar, literals appear after the sub-expression (e.g., <c-expression> B and <a-expression> C). These changes, you would discover, would make the process of determining if a string of As, Bs and Cs is legal much more difficult for non-trivial strings. It is even possible for a grammar to embed expressions within other expressions or tokens, e.g. rules (2) and (3) in the following hypothetical grammar:

(1) <valid-expression> ::= <a-expression> | <b-expression> | <c-expression>
(2) <a-expression> ::= A <b-expression> | <a-expression> A <c-expression>
(3) <b-expression> ::= B <a-expression> | B <b-expression> B
(4) <c-expression> ::= C <b-expression> | <a-expression> C

Complex grammars, such as those used in natural language, are important domains in fields such as linguistics and artificial intelligence. Some understanding of grammar concepts is beneficial to any programmer, however. Any time a program takes input from a user that is more complex than a menu item number, it is advisable to determine a grammar for legal and illegal inputs.

17.1.3: A Simple Grammar for Arithmetic Expressions

A common use of a grammar is to evaluate a numeric expression typed in by a user. A good example of this is what MS-Excel must do when a user types a formula into a spreadsheet. It is also the core of the calculator walkthrough and lab exercise presented in Section 17.5.

The arithmetic expression grammar is presented in Example 17.1. In addition to the grammar itself, we also have specifically certain primitives:

- VARIABLE: a collection of characters, beginning with a letter, that does not contain any white characters (i.e., characters less than or equal to space, which is ASCII 32, or greater than ~, which is ASCII 127) or operators embedded in it, and is not a number.
- NUMBER: a string of decimal digits, preceded by and ending with an operator or white character, which may or may not contain a single period within it.
- STRING: a collection of characters, which may include spaces, within " or ' delimiters.
- DATE: A delimited string of three integers separated by / or – characters.

We could have, of course, included these definitions within our grammar. Doing so precisely, however, would have added considerably to the grammar's length and would have made it more—rather than less confusing.

Example 17.1: Full Calculator Grammar

```
    expression :
                    assign-expression
                    test-expression
                    arith-expression
      simple-token
    assign-expression :
                    VARIABLE = arith-expression
                    VARIABLE = bool-expression
          VARIABLE = non-number-token
  test-expression :
                    ? bool-expression
  bool-expression :
                    arith-expression comparision-operator  arith-expression
                    string-expression comparison-operator string-expression
          date-expression comparison-operator date-expression
                    bool-expression equality-operator  bool-expression
          ! bool-expression
                    bool-expression logical-operator bool-expression
      ( bool-expression )
      TRUE
      FALSE
    arith-expression :
                    item
                    item arith-operator arith-expression
    item :
                    value
                    + value
                    - value
    value :
                    VARIABLE
                    NUMBER
                    ( arith-expression )
    string-expression :
                    VARIABLE
                    STRING
    date-expression :
                    VARIABLE
                    DATE
    simple-token :
                    VARIABLE
                    TRUE | FALSE
                    DATE
                    STRING
                    NUMBER
    arith-operator : one of
                     +  -  *  /
    comparison-operator : one of
                    > <  ==  >=  <=  !=
    equality-operator : one of
                    ==  !=
  logical-operator  : one of
                    &&  ||
```

Our grammar can be interpreted as follows:

- **Expression**: Our grammar is designed to accommodate four types of expressions:
  - Assignments, i.e., X=3, Y=X*2 or Affirmative=TRUE
  - Tests, i.e., ? X<7*Y/8 or ? X<5 || Y>6/2
  - Arithmetic expressions, i.e., Y*4-3/(X+1)
  - Simple expressions (i.e., variables or the 4 literals—numbers, strings, dates and Boolean values)
- **Assignment:** An assignment must always have a variable followed by an equal sign followed by an arithmetic expression, a Boolean expression or a simple token (variable or literal).
- **Test Expression:** A bool-expression preceded by a ?, which signifies the result will be a true/false value, rather than a number.
- **Boolean Expression:** Can be variety of tests, including:
  - a comparison of arithmetic expressions (e.g., X>Y), string expressions, date expressions, or other Boolean expressions (only == and != can be used to compare two Boolean expressions),
  - a negation of a Boolean expression (e.g., ! (X>Y)),
  - a logical combination of Boolean expressions
  - a parenthesized Boolean expression. This use of parentheses in the grammar is important because it allows us to specify order of operation (e.g., (X>3 || Y<4) && Z !=3)
  - a variable that has been assigned a Boolean value
  - TRUE or FALSE (case insensitive)
- **Arithmetic Expression:** Any expression that evaluates to a number. The *item* definition is used to allow richer expressions than pure VARIABLE and NUMBER primitives would allow.
- **Item:** A definition specifically intended to handle leading + and – signs.
- **Value:** A definition that handles numbers, variables and parenthesized expressions.
- **String Expression:** A quoted string or a variable that has been assigned a string value.
- **Date Expression:** A quoted date or a variable that has been assigned a date value.
- **Simple Token:** A variable or a literal number, date, string or Boolean value.

An example of how the expression (X+7)*-3 would be interpreted is presented in Figure 17.2.

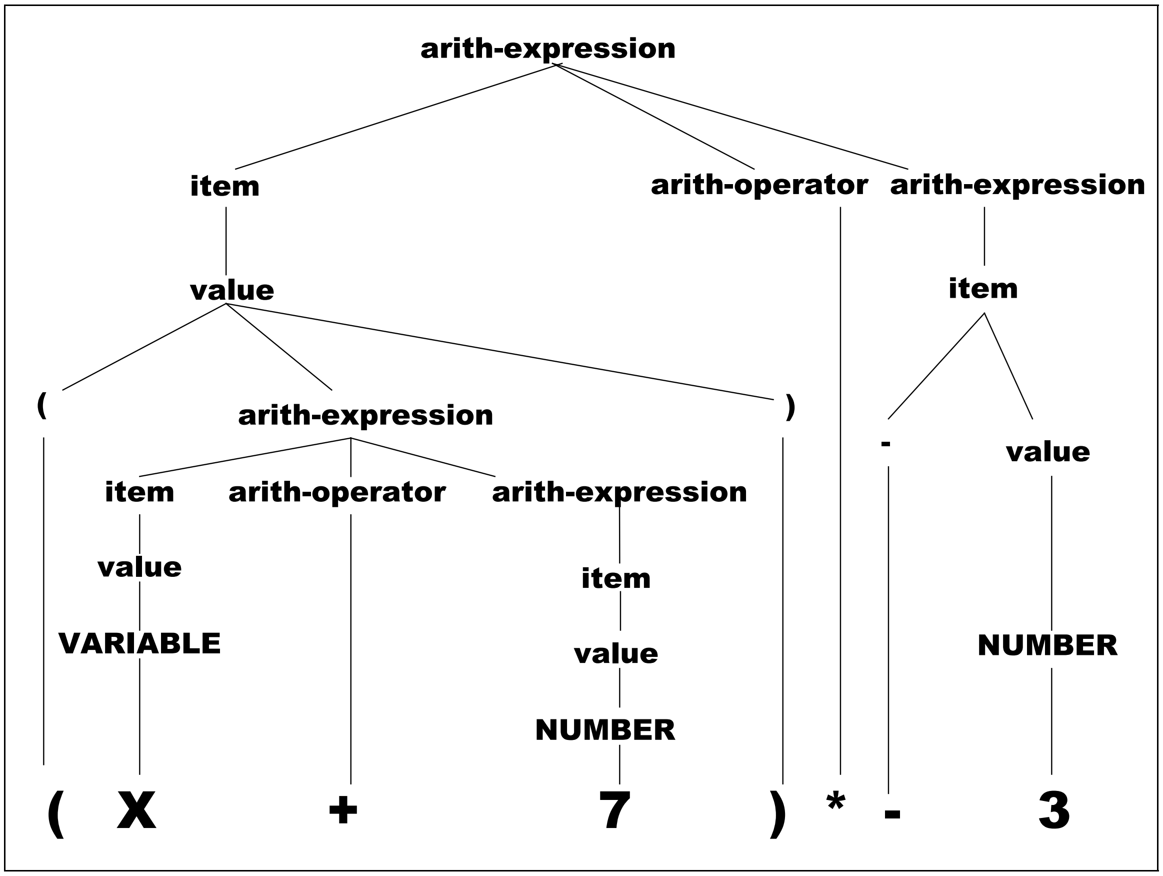


Figure 17.2: Tree illustration of expression (X+7)*-3 using expression grammar

Some examples of expression created according to the grammar are presented in Table 17.3.

Expression	Legal?	Explanation
X = 3	Legal	A legal assignment, beginning with variable = and ending with an expression.
? X + 3 = Y*4	Not Legal	The problem here is that X+3=Y*4 is not a valid bool-expression because = is not a value test. Has it been a == (double equal), it would have been fine.
(4*Y + 7)/-(3+X)	Legal	A valid arithmetic expression. The top level breaks into item -> (4*Y+7), arith-operator -> / and arith-expression -> - (3+X).The 1st and 3rd items further decompose into legal expressions, and so forth.
(3+4 < Y)	Not Legal	Tests must be preceded by a ? symbol, since a plain Boolean expression is not a legal expression.
X=(Y>X)	Not Legal	There is no provision in the grammar for assigning a boolean value to a variable.
? !(X+3)>Y	Legal	The (X+3)>Y is a legal Boolean expression, so placing a ! in front of it is okay. The whole Boolean expression is preceded by a ?

Table 17.2: Examples of Legal and Illegal Expressions

17.1 Section Questions

1. Does a grammar tell you what an expression means?
2. What is the purpose of BNF?
3. Why would grammars where all expressions begin with primitive tokens tend to be faster to translate than more complex grammars?
4. Would the expression x++Y be legal in the calculator grammar?
5. Would the expression x+++Y be legal in the calculator grammar?
6. What impact would it have, it any, if item were defined as:
   ```
   item :
   value
   + item
           item
   ```
7. Is "? Y<2 && !!!! X>3" a legal expression in the calculator grammar?
8. Why isn't operator precedence specified as part of our calculator grammar?

17.2: Computing Roman Numerals

Walkthrough available in Roman2Int.avi

Translating Roman numerals to integers provides an interesting task for demonstrating how problem representation—in this case, the use of a grammar—can dramatically reduce the complexity of a program.

17.2.1: Roman Numerals 101

The basic Roman numbering system constructs a number using a set of symbols, each of which has a value. In our function, the symbols we will support are:

I – 1
V – 5
X – 10
L – 50
C – 100
D – 500
M – 1000

Higher symbol values were achieved by overlining to multiply by a factor of 1000, e.g.,

$$\overline{\text{V}} = 5000$$
$$\overline{\text{X}} = 10000$$

For simplicity's, sake we'll content ourselves with handling numbers of 4999 or less.

The basic framework for creating a number is that symbols are written in descending order, and their values are added. For example, LXXVIII would be 73 (50+10+10+5+1+1+1). There are, however, some complex observations that we can make about any numeral that is legal:

1. No half-way symbol (e.g., V, L, D) a) may appear more than once in a numeral, nor b) can it appear in front of a larger symbol.
2. If a single symbol—representing an even power of 10 (i.e., I, X, C) is placed to the left of a symbol of greater value, the lesser symbol's value is subtracted from the greater symbol's value. This, 9 is normally written IX (10-1) instead of VIIII (5+1+1+1+1), and 4 can be written IV instead of IIII, although both forms are legal.
3. Any symbol to the left of another symbol can be no less than 1/10th of the value of any symbol to its right. Thus, IX, IV, XC, XL are okay, but IC, IM, IXC and XM are not.

4. Once a symbol has been placed to the left of a greater symbol, a) it cannot be repeated in the number (e.g., XIXI is not legal) except when it follows a lesser symbol (e.g., repeating the C in MCMXCVII is permissible because the repeated C comes after an X), nor can b) the symbol 5 times its value be used (e.g., XIXV is not legal).
5. If a lesser symbol is placed in a series of higher-value symbols, it must be in front of the right-most of the higher value symbols (e.g., CCXC is legal, CXCC is not).
6. No single symbol may appear more than four times in a numeral

These exceptions provide some interesting sequencing issues, as they mean an X in our number could either add 10 or –10 to the total, depending upon what comes next.

> *Test your understanding:* for each of the following Roman numerals, identify: a) if its legal, and b) its value (if it is legal):
> - XLVIII
> - MIM
> - CLXIXX
> - MCMXCIX
> - CVLII
> - LIIII

The translation of a Roman numeral is an example of a general class of translation problems. In tackling the problem of translating a sequence, it is usually a good idea to break it down into two separate problems:

1. Is the input legal?
2. What is the value of the input?

Doing so has two benefits. First, it simplifies the function for evaluating the value—since that function can assume its input is legal without doing any testing. Second, it forces us to address the problem of what to do about illegal input. By organizing our code in this way—a good practice whenever user-input is involved, we can avoid problems like the nasty assertion failure in the debugging exercise (Chapter 9, end of Section 9.4.2) that resulted from an illegal digit being supplied by the user.

17.2.2: Finding Numeral Value

In order to solve this problem, we're going to need a series of functions. The first, obviously needed, is a function that returns the value of any symbol. We can prototype that function as follows:

```
unsigned int NumeralValue(char cNum);
```

The argument *cNum* contains a symbol—'I', 'V', 'X', 'L', 'C', 'D', 'M'—which can be either uppercase or lower case. It returns its integer value—1, 5, 10, 50, 100, 500, 1000—or 0, if an illegal numeral (e.g., 'G') is supplied. If there was ever a natural fit with a case statement, it is this function, which is provided as Example 17.2.

Example 17.2: NumeralValue()

```
// NumeralValue(char cVal) returns the integer equivalent of the character cVal
unsigned int NumeralValue(char cNum)
{
	unsigned int nVal=0;
	switch(cNum)
	{
		case 'I':
		case 'i':
		{
			nVal=1;
			break;
		}
		case 'V':
		case 'v':
		{
			nVal=5;
			break;
		}
		case 'X':
		case 'x':
		{
			nVal=10;
			break;
		}
		case 'L':
		case 'l':
		{
			nVal=50;
			break;
		}
		case 'C':
		case 'c':
		{
			nVal=100;
			break;
		}
		case 'D':
		case 'd':
		{
			nVal=500;
			break;
		}
		case 'M':
		case 'm':
		{
			nVal=1000;
			break;
		}
	}
	return nVal;
}
```

17.2.3: Testing Legal Numeral

Our next function, IsLegalRoman(), is intended to test if a Roman numeral is legal. It is prototyped as follows:

bool IsLegalRoman(const char szNum[]);

The argument szNum[] contains an string representing a Roman numeral (e.g., "MMIII"). It returns a Boolean value of 1 (true) or 0 (false).

We will implement two versions of the same function:

- a "brute-force" method, that attempts to employ our rules for Roman numerals exactly as they were written, and
- an "elegant" method, in which we revisit how Roman numerals are presented and come up with a method of testing that takes half as much code(and is ten times more likely to be bug-free).

IsLegalRoman1(): Brute Force Method
In doing a straightforward application of our rules for Roman numerals, it is useful to have a function that tells us if a numeral is a half-way symbol (e.g., V, L, D). Thus, we define a short test function:

bool IsHalfWaySymbol(char cSym)

This function takes a character argument and returns **true** if it is 'V', 'v', 'L', 'l', 'D' or 'd', **false** otherwise.

IsHalfwaySymbol() is presented in Example 17.3.

Example 17.3: IsHalfwaySymbol()

```
bool IsHalfWaySymbol(char cSym)
{
	char cUp=(char)toupper(cSym);
	return (cUp=='V' || cUp=='L' || cUp=='D');
}
```

Our brute-force approach to determining if a numeral is valid, called IsLegalRoman1(), is presented in Example 17.4.

Example 17.4: IsLegalRoman1()—brute force approach

```
bool IsLegalRoman1(const char szNum[])
{
        int i;
        bool bRet=true;
        for(i=0;bRet && szNum[i]!=0;i++) {
                unsigned int nVal=NumeralValue(szNum[i]);
                unsigned int nNext=NumeralValue(szNum[i+1]);
                // Illegal digit
                if (nVal==0) bRet=false;
                else if (IsHalfWaySymbol(szNum[i]))
                {
                        // Halfway symbols can never precede larger symbols
                        if (nVal<nNext) bRet=false;
                        // Check for repeated halfway symbol
                        else {
                                int j;
                                for(j=i+1;bRet && szNum[j]!=0;j++) {
                                        if (nVal==NumeralValue(szNum[j])) bRet=false;
                                }
                        }
                }
                // Symbol prior to higher symbol
                else if (nVal<nNext)
                {
                        // Test for no less than 1/10th rule
                        if (nNext/nVal>10) bRet=false;
                        else {
                                // Need to catch problems like CXCC instead of CCXC
                                // and CXCL, which can't really be interpreted
                                int j;
                                for(j=i+2;szNum[j]!=0;j++) {
                                        unsigned int nDownStream=NumeralValue(szNum[j]);
                                        if (nNext<=nDownStream || nVal*5==nDownStream)
bRet=false;
                                        // Check repeated digit exception
                                        // e.g., MCMXCVII repeated C is okay
                                        if (nVal==nDownStream && NumeralValue(szNum[j-
1])>=nVal) bRet=false;
                                }
                        }
                }
                // Check for more than 4 non-halfway symbols
                else {
                        int j,nCount=1;
                        for(j=i+1;szNum[j]!=0;j++) {
                                if (nVal==NumeralValue(szNum[j])) nCount++;
                        }
                        if (nCount>4) bRet=false;
                }
        }
        if (i==0) return false; // Romans have no 0, so empty string is not allowed!
        return bRet;
}
```

The basic logic of the function is as follows:

- We have a local variable, bRet, that becomes **false** as soon as there is evidence that our numeral is not valid.
- We go through our sequence of digits until we reach the NUL terminator or until bRet becomes **false**—which occurs when any of our tests for a legal numeral fails.
- We test first for an illegal numeral digit (e.g., 'K'), which can be determined by the fact that NumeralValue() returns 0.
- If the digit we're looking at is a half-way symbol, we make sure:
    - that it does not precede a larger symbol (Rule 1b), and
    - that it is the only digit of that type present in the numeral (Rule 1a).
- If the digit is not a halfway symbol, we look ahead at the sequence and ask: is the digit that comes after it (nNext) a higher value than the current symbol (nVal). If so:
    - We check to make sure it is no less than 1/10th the value of the higher symbol by computing nNext and nVal (Rule 3)
    - We loop through the remaining symbols to be sure:
        - nVal is not present, except where it is preceded by a lesser digit, such as the C in 1997—MCMXCVII (Rule 4a)
        - nVal*5 is not present (Rule 4b)
        - nNext is not present (Rule 5)
- If our digit is not out of numeric order (relative to the digit to the right), we check the remaining digits to ensure the same symbol does not appear more than 4 times (Rule 6).

If our original string was empty or bRet got set to **false** in the loop, the function returns **false**, otherwise it returns **true**.

As you look over the code for the brute force version of IsLegalRoman1(), an uncomfortable feeling should start creeping up the back of your neck. Even though the function is small (without comments, the function is less than 40 lines), it is packed full of characteristics that might provoke anxiety:

- *Serious inefficiencies:* The code has no less than three loops that look ahead through the digits in the numeral looking for violations. While we are not particularly concerned with software performance in this text, such inefficiency is suggestive of sloppy thought processes.
- *Numerous in-code comments:* Throughout this book, the philosophy has been to minimize in-code comments except in situations where the code is not easily understood. The in-code comments in this particular function were critical, however, since none of the code was easily understood (even to its author!)
- *Deep nesting:* Some of the code in the routine was nested in branches and loops five levels deep (in fact, the FlowC chart took up 6 pages!). At such depths, the potential for logic errors to creep in is very high.

Having said all this, there are certainly worse ways to implement Roman numeral translation. Probably, with extensive testing, we could develop a high confidence level that the routine works. But imagine an application consisting of 1000 functions written like this—still a "small" application, by most standards. Yikes! You'd have to clone yourself for ten successive generations to test it enough to get the whole thing running.

The bottom line is this: whenever you write a function like this, you should immediately start thinking to yourself, "How can I make it simpler?". You can think of your entire software project as having a budget limit for "nasty" functions like this one—go over a certain number—say 5 or 10—and you've got yourself a testing nightmare. Don't squander your precious “nasty function” budget on functions that can be cleaned up!

IsLegalRoman2(): The Elegant Approach
In the vast majority of situations, simplifications to a problem come about as a result of changing the way we look at the problem. In our brute force approach, we applied the rules we were given pretty much as stated. Are there other ways to look at the problem? In particular, can we find the "holy grail" of simplification: *taking a single complex problem and then breaking it up into a collection of simpler problems that can be solved independently?*

As it turns out, there's a nice simplification for Roman numerals that immediately suggests itself with a slight change in the way we write the numerals. Take, for example, the year 1997. Normally, it is written:

MCMXCVII

Suppose, however, we were to break it up into decimal groups. In this case, it becomes:

M-CM-XC-VII (i.e., 1-9-9-7)

This suggests a grammar along the lines of that in Example 17.5.

Example 17.5: Example Roman Numeral Grammar

```
numeral-expression :
        thousand-expr
        hundred-expr
        ten-expr
        unit-expr

thousand-expr :
        thousand-group
        thousand-group hundred-expr

hundred-expr :
        hundred-group
        hundred-group ten-expr


ten-expr :
        ten-group
        ten-group unit-expr

unit-expr :
        unit-group
```

Furthermore, we can say something about each of the decimal group expressions:

- Each group can contain only 3 possible symbols—we'll call them U (unit size), F (five size) and T (ten times the unit size).
- Legal combinations of the 3 symbols are as follows:
  - UT (e.g., IX, XC)
  - UF (e.g., IV, XL)
  - {Unit-cluster}—up to four repetitions of the unit (e.g., III, XX)
  - F (e.g., V, L)
  - F{Unit-cluster} (e.g., VII, LXXX)

Now, the next thing we can do is write two functions—the purpose of which is to tell us where a group ends:

```
int UnitCluster(const char szN[],int nStart,char cUnit);
int DecimalGroup(const char szN[],int nStart,char cUnit,char cFive,char cTen);
```

The arguments for the two functions are as follows:

- szN: a string containing the Roman numeral
- nStart: our current position in the string (0 is the first character)

- cUnit: the unit symbol (e.g., I,X,C,M)
- cFive: the five symbol for the current position (e.g., V,L,D)
- cTen: the ten symbol for the current position (e.g., X,C,M)

The return values for the UnitCluster() and DecimalGroup() functions are the character positions immediately after the current group. If there is no match for the type of group each function is looking for, the functions just return nStart. These two functions are presented in Example 17.6.

Example 17.6: UnitCluster() and DecimalGroup() functions

```
/* UnitCluster finds the end of a group of Roman numeral symbols matching cUnit
(in szN[], starting at nStart)returning the position of the end. If more than 4
are present, it returns nStart+4 (since a legal numeral can have no more than
4 of a given digit in a row). It there is no match, it just returns nStart. */
int UnitCluster(const char szN[],int nStart,char cUnit)
{
	int i;
	cUnit=(char)toupper(cUnit);
	for(i=0;i<4 && toupper(szN[nStart+i])==cUnit;i++){}
	return nStart+i;
}

/* DecimalGroup finds the end of a group of Roman numeral symbols representing
a single decimal digit. cUnit is the unit symbol for that digit (e.g., 'I'),
cFive is the five symbol for that digit (e.g., V) and cTen is the ten symbol
(e.g., X). It returns the starting position in the string of the next decimal
place, or nStart if there is no match. */
int DecimalGroup(const char szN[],int nStart,char cUnit,char cFive,char cTen)
{
	int nPos=nStart;
	cUnit=(char)toupper(cUnit);
	cFive=(char)toupper(cFive);
	cTen=(char)toupper(cTen);
	// cFive and cTen may be '\0' for thousands place
	if (cFive !=0 && toupper(szN[nStart])==cFive)
		nPos=UnitCluster(szN,nStart+1,cUnit);
	else if (toupper(szN[nStart])==cUnit) {
		if (cTen!=0 && toupper(szN[nStart+1])==cTen) nPos=nStart+2;
		else if (cFive!=0 && toupper(szN[nStart+1])==cFive) nPos=nStart+2;
		else nPos=UnitCluster(szN,nStart,cUnit);
	}
	return nPos;
}
```

We immediately notice a few things about these functions:

- They are much smaller than IsLegalRoman1()
- Virtually all the comments relate to the arguments, as opposed to how the code works. This is because once you understand what the functions are supposed to do, their actual implementation is relatively trivial.

Naturally, it is not really fair to compare the size of the two functions to IsLegalRoman1(), because we haven't included IsLegalRoman2(). That function, however, turns out to be pretty trivial, as shown in Example 17.7.

Example 17.7: IsLegalRoman2()—the "elegant" approach

```
// Tells if a Roman numeral, in szNum[], is legal
bool IsLegalRoman2(const char szNum[])
{
	unsigned int nStart=0;
	nStart=DecimalGroup(szNum,nStart,'M',0,0);
	nStart=DecimalGroup(szNum,nStart,'C','D','M');
	nStart=DecimalGroup(szNum,nStart,'X','L','C');
	nStart=DecimalGroup(szNum,nStart,'I','V','X');
	if (nStart==0 || nStart!=strlen(szNum)) return false;
	return true;
}
```

The IsLegalRoman2() function is very simple in its operation. Since we know that Roman numerals have to be written higher to lower, left to right, we:

- Process each decimal group, in descending order.
    - If a group isn't present, the DecimalGroup() function returns the original position, and we go to the next group.
    - If a group is present, nStart gets moved to the start of the next group
- Once we have processed all the decimal places, we test to see if we're at the end of the string containing the numeral—by comparing nStart to strlen(szNum).
    - If we are, it was a valid string—since all the groups were processed successfully, and in order
    - If we're not at the end of the string, we were stopped by an invalid character somewhere—that means we'll be returning **false**.

The procedure is illustrated for the integer MCMVII (1907) in Figure 17.3.

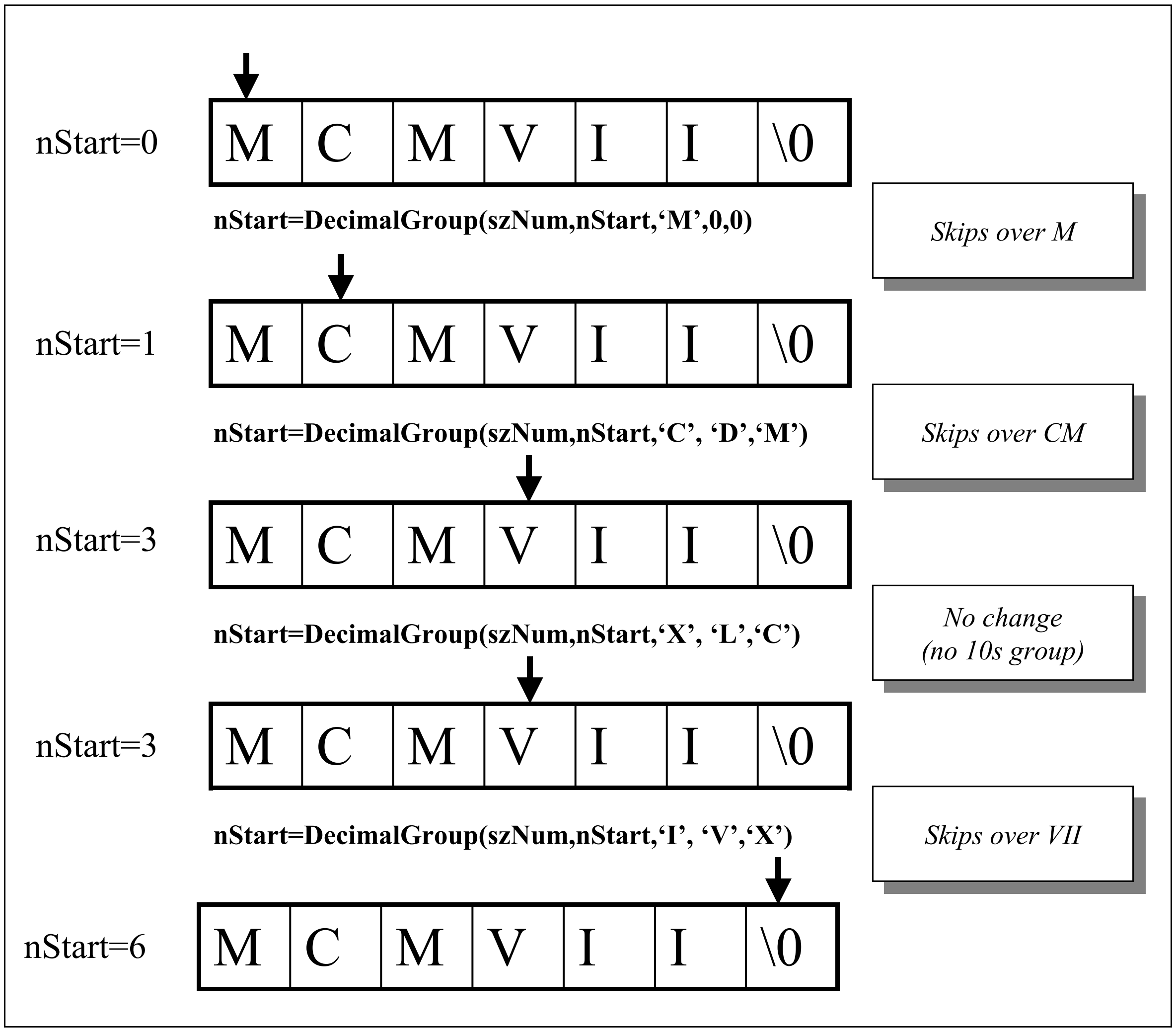

nStart=0
M C M V I I \0
nStart=DecimalGroup(szNum,nStart,'M',0,0)
Skips over M
nStart=1
M C M V I I \0
nStart=DecimalGroup(szNum,nStart,'C', 'D','M')
Skips over CM
nStart=3
M C M V I I \0
nStart=DecimalGroup(szNum,nStart,'X', 'L','C')
No change
(no 10s group)
nStart=3
M C M V I I \0
nStart=DecimalGroup(szNum,nStart,'I', 'V','X')
Skips over VII
nStart=6
M C M V I I \0

Figure 17.3: IsLegalRoman("MCMVII") illustration

17.2.4: Roman2Int Function

Once we know a string containing a Roman numeral is legal, actually finding the value is practically trivial. Our basic algorithm is as follows:

- Read in a numeral digit
- If its value is >= to the value of the digit that follows, add the digit to the total
- If its value is < than the digit that follows, subtract the value

The function can be prototyped as follows:

```
unsigned int Roman2Int(const char szNum[]);
```

The return value is the value of the Roman numeral in szNum, or 0 if the numeral is illegal. For example:

- Roman2Int("MCMLXXXIII") would return 1983
- Roman2Int("MIM") would return 0, since it's an illegal number form.

The function is presented in Example 17.18.

Example 17.8: Roman2Int() function

```
// Returns the integer value of a Roman numeral in szNum,
// or 0 if the numeral is illegal
unsigned int Roman2Int(const char szNum[])
{
      unsigned int nTotal=0;
      int i;
      if (!IsLegalRoman2(szNum)) return 0;
      for(i=0;szNum[i]!=0;i++) {
            unsigned int nVal=NumeralValue(szNum[i]);
            unsigned int nNext=NumeralValue(szNum[i+1]);
            if (nVal<nNext) nTotal=nTotal-nVal;
            else nTotal=nTotal+nVal;
      }
      return nTotal;
}
```

17.3: Lab Exercise: General Calculator

Walkthrough available in Calculator.avi

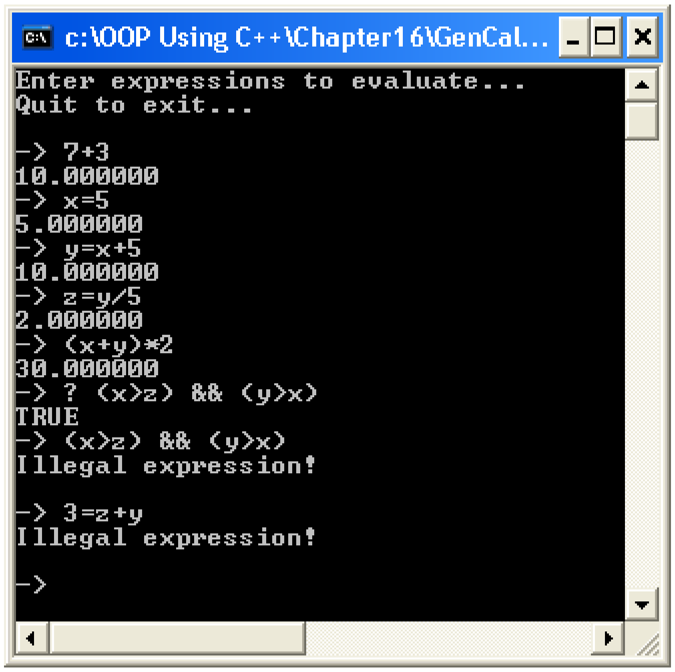
In this section, we create a general calculator that is capable of evaluating arithmetic expressions, logical expressions and implements symbolic variables. It is based upon the calculator grammar, Example 17.1 discussed in Section 17.1.3, and builds upon the Variables class, introduced in Chapter 14, Section 14.4.

17.3.1: General Calculator Design

The general calculator that we are constructing allows us to perform three distinct tasks:

- Evaluate arithmetic expressions containing numbers, variables and operators
- Evaluate logical expressions (preceded by a ?), determining if their value is TRUE or FALSE
- Assign numeric values to variables, whose values can subsequently be used in later expressions.

An example of the calculator in operation is presented in Figure 17.4.



```
c:\OOP Using C++\Chapter16\GenCal...
Enter expressions to evaluate...
Quit to exit...

-> 7+3
10.000000
-> x=5
5.000000
-> y=x+5
10.000000
-> z=y/5
2.000000
-> (x+y)*2
30.000000
-> ? (x>z) && (y>x)
TRUE
-> (x>z) && (y>x)
Illegal expression!

-> 3=z+y
Illegal expression!

->
```

Figure 17.4: General calculator in operation

The lab exercise involves creating the calculator in a series of three stages, the first being a walkthrough of the evaluation of arithmetic expressions.

17.3.2: Stage 1: Evaluating Arithmetic Expressions

The first stage of the calculator project is to create a class that can evaluate arithmetic expressions of arbitrary complexity. These expressions can contain numbers, operators and variables, all of which are in a sequence that conforms to our grammar. To implement this version, we use a subset of the complete grammar, shown in Example 17.9.

Example 17.9: Arithmetic Calculator Grammar Subset

```
expression :
                arith-expression
assign-expression :
                VARIABLE = arith-expression
arith-expression :
                item
                item arith-operator arith-expression
item :
                value
                + value
                - value
value :
                VARIABLE
                NUMBER
                ( arith-expression )
arith-operator : one of
                 + - * /
```

Class Overview
As a starting point, we can make use of the Variables (and Token) classes that were developed in Chapter 14, Section 14.4. As you may recall, the Variables class was designed with two purposes in mind:

- to tokenize an expression using Tokens which identify the token type (e.g., number, variable, break) of each token.
- to allow variable names to become associated with variable values—a critical capability in the calculator we are designing.

To take advantage of the functionality in the Variables class, we inherit the new class, BasicCalc, from Variables, as shown in Example 17.10.

Example 17.10: BasicCalc declaration

```
class BasicCalc : public Variables
{
public:
      BasicCalc(void);
      virtual ~BasicCalc(void);
      void MainLoop();
      bool IsArithExpression(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const;
      bool IsItem(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const;
      bool IsValue(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const;
      bool IsArithOperator(const Token &tok) const;
      size_t NextOperator(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const;
      size_t MatchParenthesis(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const;
      size_t MidExpression(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const;
      virtual int Precedence(const char *szOp) const;
      // Evaluation functions
      double EvaluateArithExpression(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const;
      double EvaluateItem(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const;
      double EvaluateValue(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const;
      double EvaluateOperator(const char *szOp,double d1,double d2)
const;
protected:
      string m_szArithOperators;
};
```

The BasicCalcDemo() function and BasicCalc::MainLoop() member function, presented in Example 17.11, illustrate the basic use of the BasicCalc class, which can only be used to evaluate arithmetic expressions (not logical expressions or assignments).

Example 17.11: Functions illustrating the BasicCalc interface

```
void BasicCalcDemo()
{
	BasicCalc calc;
	calc.SetValue("X","24");
	calc.SetValue("Y","25");
	calc.SetValue("Z","26");
	calc.MainLoop();
}

void BasicCalc::MainLoop()
{
	cout << "Enter arithmetic expressions to evaluate..." << endl;
	cout << "Quit to exit..." << endl << endl;
	bool bEnd=false;
	while (!bEnd) {
		char buf[256]={0};
		cout << "-> ";
		cin.getline(buf,255);
		vector<Token> ar;
		TokenizeString(buf,ar);
		if (ar.size()>0 && ar[0].Name()=="Quit") bEnd=true;
		else {
			if (ar.size()==0) continue;
			else if (IsArithExpression(ar,0,ar.size())) {
				double
dVal=EvaluateArithExpression(ar,0,ar.size());
				cout << dVal;
			}
			else cout << "Illegal expression!" << endl;
		}
		cout << endl;
	}
}
```

The BasicCalcDemo() function is used to set values for X,Y and Z (using the Variables interface), so we have variable values to work with. The MainLoop() function implement's the BasicCalc user interface, which:

- Prompts the user for an expression with a -> ouput
- Reads the user expression, tokenizes it, the validates it with a call to IsArithExpression()
  - If the expression is legal, it evaluates it using EvaluateArithExpression(), then displays the return value.
  - If the expression is not legal, it outputs "Illegal Expression!"
- The loop continues until the user types "Quit"

The design of the class is entirely driven by the Example 17.9 grammar. For each expression in the grammar, a pair of functions is defined:

- *ArithExpression:* IsArithExpression() and EvaluateArithExpression()
- *Item:* IsItem() and EvaluateItem()
- *Value:* IsValue() and EvaluateValue()

In addition, the Variables class provides us with a number of additional validating functions, such as IsVariable(), IsNumber(), IsOperator() and IsLegalName().

The paired functions for each expression type accomplish two complementary purposes: determining if an expression is legal and evaluating the expression. An advantage of constructing our class this way (as opposed to simply trying to evaluate expressions) it that it allows us to assume (within our evaluation function) that the expression is legal. This has two benefits: 1) it makes the evaluation function shorted, and 2) it allows us to use assert() statements within evaluation functions to assist in debugging.

Another thing the paired functions have in common is their arguments. In each case, the same three arguments are passed:

- The array of tokens
- The starting position for the expression
- The position after the end of the expression

Because our grammar definition is in inherently recursive, this allows us to specify smaller and smaller segments of the token array in both validity checking and evaluation. This technique is similar to that used in Quicksort (Chapter 15, Section 15.5).

Validity Testing Functions
Both the validity checking (Is… functions) and evaluation (Evaluate…) functions correspond closely to the grammar. The main validity testing functions are presented in Example 17.11.

Example 17.11: Top level validation functions

```
bool BasicCalc::IsArithExpression(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const
{
      size_t nOp=NextOperator(ar,nStart,nEnd);
      if (nOp==nEnd) return IsItem(ar,nStart,nEnd);
      else if (nOp>nEnd || !IsArithOperator(ar[nOp])) return false;
      else return (IsItem(ar,nStart,nOp) &&
            IsArithExpression(ar,nOp+1,nEnd));
}
bool BasicCalc::IsItem(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const
{
      if (ar[nStart]=="+" || ar[nStart]=="-")
                  return IsValue(ar,nStart+1,nEnd);
      else return IsValue(ar,nStart,nEnd);

}
bool BasicCalc::IsValue(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const
{
      size_t nCount=nEnd-nStart;
      if (nCount==1) return (ar[nStart].Type()==tNumber);
      else if (nCount<3) return false;
      else if (ar[nStart]=="(" && ar[nEnd-1]==")") {
            return IsArithExpression(ar,nStart+1,nEnd-1);
      }
      else return false;
}
```

The top-level validation functions work as follows. IsArithExpression() needs to verify that our expression (the entire token array or some subsection of it) is of one of two forms:

- *item*
- *item arith-operator arith-expression*

To do this, we try to skip to the operator in the second form by calling NextItem() (which is discussed shortly). If there is no operator, the skipping function returns nEnd and we validate the expression by calling IsItem() on the entire range (i.e., addressing the first form of arith-expression). If we find an operator, on the other hand, we verify that:

- The token at position nOp is a valid arithmetic operator (a string of legal arithmetic operators, the member m_szArithOperators, is set to "+-*/" in the class constructor)
- The subarray of tokens prior to the operator is a legal item (by calling IsItem() on the range from nStart to nOp)
- The subarray of tokens following the operator is a legal arith-expression (by calling IsArithExpression() on the range from nOp+1 to nEnd).

Unless all three tests return true, the IsArithExpression() returns false.

IsItem() is even simpler. An *item* can be one of three things:

- *value*
- + *value*
- - *value*

As a result, the function checks for a leading + or -, incrementing nStart it if it is present, then calls IsValue() on the range from nStart to nEnd.

IsValue() is also simple. A value can be either a variable, a number or an expression within parentheses. If nEnd-nStart is 1, it must be either a variable or a number, so we call the two test functions (both from the Variables class). Otherwise we verify: 1) that that the tokens at nStart and nEnd-1 are "(" and ")" respectively, and 2) that the range from nStart+1 to nEnd-1 (i.e., the range inside the parentheses) contains a valid expression (by calling the IsArithExpression() function on that range).

Example 17.12: Helper functions for validation and evaluation

```
size_t BasicCalc::NextOperator(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const
{
      if (ar[nStart]=="+" || ar[nStart]=="-") nStart++;
      if (ar[nStart]=="(") nStart=MatchParenthesis(ar,nStart,nEnd);
      nStart++;
      return nStart;
}
size_t BasicCalc::MatchParenthesis(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const
{
      assert(ar[nStart]=='(');
      size_t nPos=nStart;
      int nBalance=1;
      while(nPos<nEnd && nBalance>0) {
            nPos++;
            if (ar[nPos]=="(") nBalance++;
            else if (ar[nPos]==")") nBalance--;
      }
      return nPos;
}
bool BasicCalc::IsArithOperator(const Token &tok) const
{
      return (strchr(m_szArithOperators.c_str(),*(tok.Name()))!=0);
}
```

As we already noted, the top-level functions in Example 17.11 rely on certain helper functions, presented in Example 17.12. The NextOperator() function, for example, finds the next operator in an expression. As we already noted, an *arith-expression* can be:

- *item*
- *item arith-operator arith-expression*

That means the NextOperator() function has to skip over an item. An item, as already noted, can be:

- *value*
- + *value*
- - *value*

We therefore need to skip over the first token if it is a + or – sign, then skip over the value. A value, in turn, is either a variable, number or expression in parentheses. So, we increment our position by 1 unless the nStart token is a left parenthesis—in which case we skip to the balancing right parenthesis (calling the MatchParenthesis() function) then add 1 to our position.

Since parentheses can be nested, the MatchParenthesis() function cannot simply look for a ")" token after verifying that nStart is "(". Instead, it uses a balancing counter that is incremented each time a left parenthesis is encountered, and decremented each time a right parenthesis is encountered. The loop ends when the balancing counter is 0, or when the end is reached (in which case nEnd is returned).

Evaluation Functions
Because our grammar only tells us if an expression is legal—and not what to do with it—we need to provide some operators-specific knowledge to the class in order to perform evaluations. Three functions serve this purpose:

- *Precedence()*: which returns an operator's precedence
- *MidExpression()*, which uses operator precedence to determine where to break up an expression for evaluation
- *EvaluateOperator()*: Which applies an operator to two arithmetic (double) arguments.

These functions are presented in Example 17.13.

Example 17.13: Operator Evaluation Functions

```
int BasicCalc::Precedence(const char *szOp) const
{
	IString sOp=szOp;
	if (sOp=="+" || sOp=="-") return 50;
	else if (sOp=="*" || sOp=="/") return 60;
	else return 0;
}
size_t BasicCalc::MidExpression(const vector<Token> &ar,
                                 size_t nStart,size_t nEnd) const
{
	int nLowest=-1;
	size_t nPos;
	do
	{
		nStart=NextOperator(ar,nStart,nEnd);
		if (nLowest<0) nPos=nStart;
		if (nStart<nEnd) {
			int nNew=Precedence(ar[nStart].c_str());
			if (nNew<nLowest || nLowest<0) {
				nLowest=nNew;
				nPos=nStart;
			}
			nStart++;
		}
	}
	while (nStart<nEnd);
	return nPos;
}
double BasicCalc::EvaluateOperator(const char *szOp,
                                    double dV1,double dV2) const
{
	double dVal=0.0;
	switch(*szOp)
	{
		case '+': {
			dVal=dV1+dV2;
			break;
		}
		case '-': {
			dVal=dV1-dV2;
			break;
		}
		case '*': {
			dVal=dV1*dV2;
			break;
		}
		case '/': {
			if (dV2==0.0) cerr << "Divide by zero error!" << endl;
			else dVal=dV1/dV2;
			break;
		}
		default: {
	    assert(false);
			cerr << "Unknown arithmetic operator!" << endl;
			break;
		}
	}
	return dVal;
}
```

The Precedence() function, returns arbitrary precedence values of 50 for + and -, and 60 for * and /. This will ultimately ensure that * and ? are performed prior to + and – (unless parentheses are present).

The MidExpression() makes use of the precedence information in determining where to break up an expression for evaluation purposes. Though a series of calls to NextOperator(), it checks the precedence at each position, keeping track of the precedence and position of the lowest precedence operator encountered (in *nLowest* and *nPos*). After iterating through all operators, it returns the position held in nPos. Thus, if the string:

X * 3 + Y *2 - Z

is encountered, it would return the position of the + sign (since + and – have the same precedence, the leftmost operator is returned).

The EvaluateOperator() function is also relatively trivial. It uses a case statement to determine what operation to apply to its two arguments, then returns the result. To avoid an exception, it returns 0.00 (and an error message) when a divide by zero is encountered. It also returns 0.00 and generates an assertion failure if an illegal operator reaches the function.

> *Test your understanding:* Why does it makes sense to handle divide-by-zero and illegal operators (assertion failure) differently?

As was the case for validation, three functions drive the validation process. These are presented in Example 17.14 and—once again—are tructured around the grammar.

Example 17.14: Functions driving arithmetic evaluation

```
double BasicCalc::EvaluateArithExpression(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const
{
      size_t nPos=MidExpression(ar,nStart,nEnd);
      if (nPos<nEnd) {
            double t1=EvaluateArithExpression(ar,nStart,nPos);
            double t2=EvaluateArithExpression(ar,nPos+1,nEnd);
            return EvaluateOperator(ar[nPos],t1,t2);
      }
      else return EvaluateItem(ar,nStart,nEnd);
}
double BasicCalc::EvaluateItem(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const
{
      double dSign=1.00;
      if (ar[nStart]=="-") {
            dSign=-1.00;
            nStart++;
      }
      else if (ar[nStart]=="+") nStart++;
      return dSign*EvaluateValue(ar,nStart,nEnd);

}
double BasicCalc::EvaluateValue(const vector<Token> &ar,
            size_t nStart,size_t nEnd) const
{
      if (nEnd-nStart==1) {
            string szVal;
            GetValue(ar[nStart].c_str(),szVal);
            return atof(szVal.c_str());
      }
      assert((nEnd-nStart>=3) && (ar[nStart]=="(") && (ar[nEnd-
1]==")"));
      return EvaluateArithExpression(ar,nStart+1,nEnd-1);
}
```

The EvaluateArithExpression() function breaks up the expression according to operator precedence. If no operator is found in the expression, it calls EvaluateItem(). If an operator is found, however, it calls EvaluateArithExpression() on both tides of the operator, placing the values in *dVal1* and *dVal2*. It then calls EvaluateOperator() to appl;y the operator to the two values. Even though our grammar specifies an *arith-expression* to be:

item arith-operator arith-expression

we can still apply EvalArithExpression() to the left hand side because an *item* is also al legal *arith-expression*.

The EvaluateItem() function skips over any +/- sign, then calls EvaluateValue(). If a – sign was present, the return of EvaluateValue() is multiplied by –1.

If a single element is passed into the EvaluateValue() function, the GetValue() member of the Variables class is used to retrieve the text value associated with the variable or number token, which is then converted to double using atof() library function. If, however, a parenthesized expression is passed in to the function, a call to EvaluateArithExpression() is made on the sub array of tokens inside the parentheses.

17.3.3: Step Two: Logical Calculator

Once the arithmetic calculator is constructed, we can add support for logical and comparison operators. The class should be called LogicCalc and should inherit from BasicCalc(). Its MainLoop() function should be able to evaluate both Boolean expressions (preceded by a ?) and arithmetic expressions.

Class Declaration
The declaration of the LogicCalc class is presented in Example 17.15.

Example 17.15: LogicCalc Class Declaration

```
class LogicCalc : public BasicCalc
{
public:
      LogicCalc(void);
      virtual ~LogicCalc(void);
      void MainLoop();

      bool IsBoolExpression(const vector<Token> &ar,
                              size_t nStart,size_t nEnd) const;
      bool EvaluateBoolExpression(const vector<Token> &ar,
                              size_t nStart,size_t nEnd) const;
      bool EvaluateCompareOp(const char *szOp,
                              const Token &t1,const Token &t2) const;
      bool EvaluateBoolOp(const char *szOp,bool b1,bool b2) const;
      bool IsComparisonOp(const Token &tok) const;
      bool IsLogicalOp(const Token &tok) const;
      virtual int Precedence(const char *szOp) const;
};
```

Function Descriptions
The core functions for the LogicCalc class are described in Table 17.3.

Key LogicCalc Member Functions

bool IsBoolExpression(const vector<Token> &ar,size_t nStart,size_t nEnd) const
Returns true if the expression within the range is either a valid logical expression (i.e., Boolean expressions joined by && or || operators) or a valid comparison. According to the grammar, comparisons may consist of:

- Arithmetic expressions separated by comparison operators.
- Boolean expressions separated by comparison operators Only == and != can be used)
- Date tokens (variables or literal dates in the form 'mm/dd/yyyy') separated by comparison operators.
- String tokens (variables or literal quoted strings) separated by comparison operators.
- ! followed by a Boolean expression
- A Boolean expression within parenthesis
- TRUE or FALSE literal values.

(Hint: start with the arithmetic cases, get them working, the do the other cases—which tend to be simpler).

bool EvaluateBoolExpression(const vector<Token> &ar,size_t nStart,size_t nEnd) const
Returns true if the specified range evaluates to a true expression false otherwise.

bool EvaluateCompareOp(const char *szOp,const Token &t1,const Token &t2) const
Evaluates operations of the form *value* **op** *value*, where **op** is >,>=,<,<=,== or !=. To do this, it needs to determine the token type (number, date, string or Boolean) and perform the appropriate comparison of values. The Number(), Date(), and Bool() functions in the Variables class can be quite helpful in this regard. [Note: only == and != are evaluated if Boolean tokens are based in].

bool EvaluateBoolOp(const char *szOp,bool b1,bool b2) const
Evaluates Boolean expressions joined by && or ||.

bool IsComparisonOp(const Token &tok) const
Returns **true** if the token passed as an argument is >,>=,<,<=,== or !=. Otherwise, it returns **false.**

bool IsLogicalOp(const Token &tok) const
Returns **true** if the token passed as an argument is&& or ||. Otherwise, it returns **false.**

virtual int Precedence(const char *szOp) const
Returns precedence of the operator passed in as an argument (0 if operator is not recognized). Should return values for arithmetic, comparison and logical operators. Reasonable values would be:

- * and /: 60
- + and -: 50
- >, >=, <, <=, == and !=: 40
- &&: 30
- ||: 20

(Hint: you can call the base class version, then check for other operators).

Table 17.3: LogicCalc member descriptions

17.3.3: Step Three: General Calculator

Once the LogicCalc class is constructed, we can add support for assignments, at which point the calculator is complete. The new class, ExprCalc should inherit from LogicCalc, and should support all expressions in the grammar.

Class Declaration
The declaration of the ExprCalc class is presented in Example 17.16.

Example 17.16: ExprCalc Class Declaration

```
class ExprCalc :  public LogicCalc
{
public:
      ExprCalc(void);
      virtual ~ExprCalc(void);
      bool IsLegalExpression(const vector<Token> &ar) const;
      bool EvaluateExpression(const char *buf,string &out);
      bool EvaluateExpression(const vector<Token> &ar,string &out);
      void MainLoop();
};
```

Function Descriptions
The core functions for the ExprCalc class are described in Table 17.4.

Key LogicCalc Member Functions

bool IsLegalExpression(const vector<Token> &ar) const Determines if a string of tokens is legal according to the grammar.
bool EvaluateExpression(const char *buf,string &out) The driver for the validation and evaluation process. The function should tokenize the expression in buf, call EvaluateExpression() to assure it is legal and—if it is—call the second version of EvaluateExpression to determine the result (which will be placed in the out string argument). The function return **true** if buf contains a legal expression, **false** otherwise. The function is not const because assignments can change the values of variables in the class.
bool EvaluateExpression(const vector<Token> &ar,string &out) Evaluates the expression passed in, placing the result (as a string) in out. For assignments and arith-expressions, out will be a text representation of a number (e.g., use sprintf). For bool-epressions, out will contain either TRUE or FALSE. For date expressions, it can contain the date integer representation (e.g., 20030906)

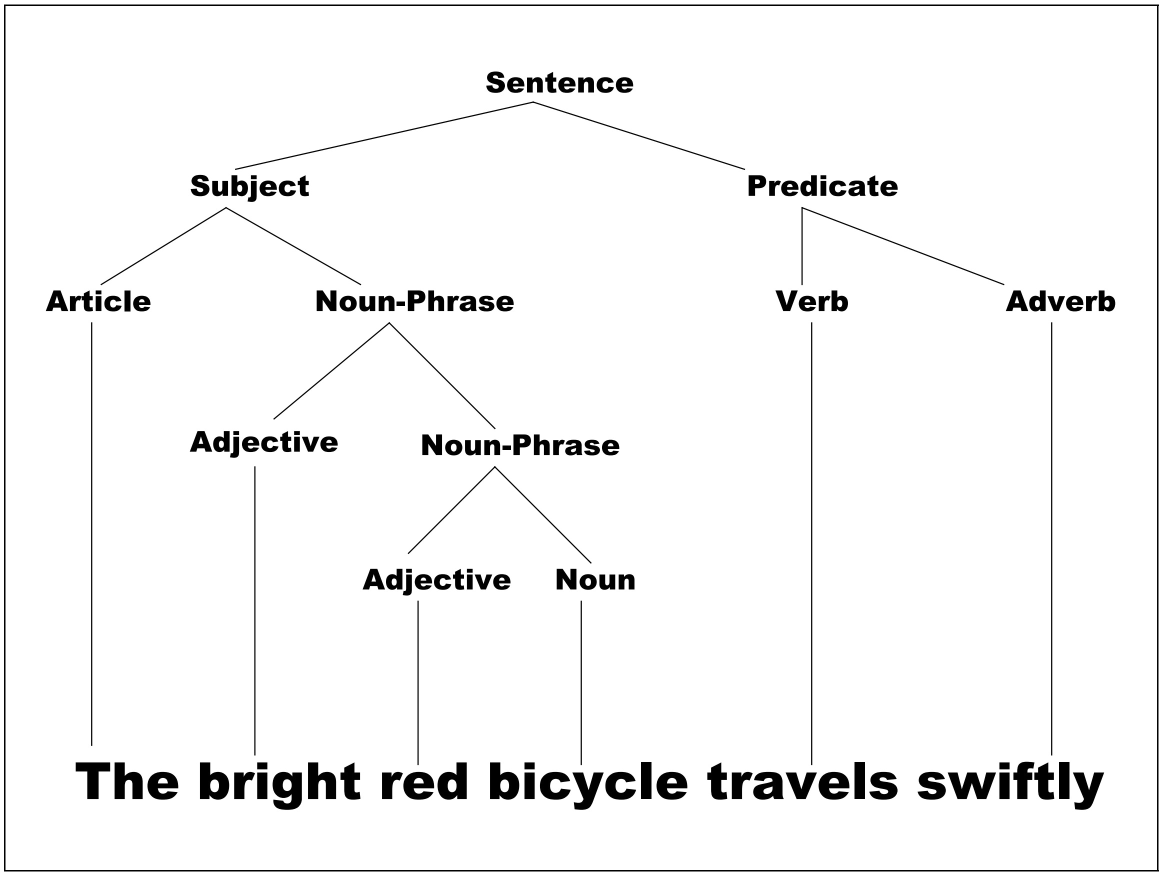
Table 17.4: ExprCalc member descriptions

17.4: Review and Questions

17.4.1: Review

A grammar provides a set of rules for determining if a collection of elements or tokens is a legal expression. The expression can technical (e.g., an Excel formula) or non-technical (e.g., a sentence). Grammars *do not* determine if the expression "makes sense" or how it is evaluated. Thus, a sentence can be grammatically correct without making any sense and determining that an arithmetic expression is valid does not necessarily provide a road map for how it should be evaluated.

Grammars tend to be recursive in their definition and are usually defined in terms of primitives (e.g., building blocks that cannot be further decomposed, such as tokens) and expressions that are defined in terms of sequences of other expressions and primitives. An expression is valid when it can be shown that all primitives match its pattern, and all sub-expressions are valid. This is sometimes depicted in tree form, as shown below.



A common notation for simple grammars is Backus Naur/Normal form (BNF). In this notation, all expressions are placed with {} and primitives are written "as is". A BNF grammar is written as a series of statements that specify what each expression can be replaced by (using the =:: symbol), with alternative replacements being separated by bar (|) symbols. e.g.:

```
<valid-expression> ::= <a-expression> | <b-expression> | <c-expression>
<a-expression> ::= A <b-expression> | A
<b-expression> ::= B <a-expression> | B <b-expression>
<c-expression> ::= C <b-expression> | C
```

Another common form places each possible replacement on a separate line, e.g.:

```
valid-expression :
        a-expression
        b-expression
        c-expression

a-expression :
        A b-expression
        A

b-expression :
        B a-expression
        B b-expression

c-expression :
        C b-expression
        C
```

Grammars can be simple or highly complex. Sources of complexity include ambiguity (more than one way of arriving at a "legal expression" may exist), combinatorial properties (how processing time changes as the size of the expression or grammar grows) and indeterminacy (it may not be possible to tell if an expression is legal).

Formally defining a grammar can lead to dramatic improvements in coding speed, size and quality. It is often possible to use a grammar as the basis for application design—even to the point of naming functions after the expressions in the grammar. Doing so provides a number of benefits:

- The grammar itself provides a powerful documentation tool
- The grammar itself can be studied for design defects, instead of waiting for code to be written before debugging begins
- The grammar helps in the construction of highly recursive programs, which tend to be much smaller than iterative versions

Examples of these benefits were found in several chapter examples.

17.4.2: Glossary

Backus Naur/Normal form (BNF) – A common notation for specifying grammars
Evaluation – determining the value or meaning of an expression—normally not a property of a grammar
Expressions – Elements of a grammar defined in terms of sequences of other expressions and primitives. Alternative paths for determining an expression is legal are often provided.
Grammar – a set of rules for determining if a collection of elements or tokens is a legal expression.
Legal Expression - An expression that can be decomposed into a series of primitives and legal sub expressions that match the pattern specified by the grammar.
Primitives – Building blocks used to construct a grammar that cannot be further decomposed, such as tokens or atoms.

17.4.3: Questions

17.1: Grammar Translation. Assume the following grammar:

```
<full-expression> := <apple-expression> | <pear-expression> | <kiwi-expression>
<apple-expression> := A | A <kiwi-expression>
<pear-expression> := P <apple-expression> P
<kiwi-expression> := K | K <full-expression>
```

Identify which of the following are legal full expressions:

1) A K P
2) P A K P
3) K K P A P
4) A A K P A P
5) K K K A K A K P A K P A P P

Draw a tree for each legal expression. Identify any characteristics of illegal expression that force them to be illegal.

17.2: Grammar Translation. Suppose I had the following grammar:

```
{legal-expression} ::= }x-expression} | {y-expression} | {z-expression}
{x-expression} ::= X | Z {y-expression}
{y-expression}::= Y | Z {x-expression}
```

{z-expression} ::= Z | Z {z-expression}

Identify which of the following are legal-expressions:

- XY
- ZYZZY
- ZZZZZZ
- XZYZZZ
- ZZZZZZZZX

Draw a tree for each legal expression. Identify any characteristics of illegal expression that force them to be illegal.

17.3. Correspondence between grammar and code. In Example 17.6, identify how each line of code in UnitCluster() and NumeralGroup() corresponds to our rules for decimal places (using U, F and T)

17.4. Extended Roman Numerals: Assume that an archeologist just unearthed a scroll, penned by RESI (Roman Empire Standards Institute) that instituted the following new symbols:

F – 5000
T – 10000
B – 50000
H – 100000

Roman2Int application to accommodate these symbols.

17.5: Period symbol. Explain the types of problems that could arise if the period (.) symbol were specified to be a break character in the calculator. Are there any advantages to doing so?

17.6: Default variable values. How would we need to modify the calculator so that a whenever a variable name appears, it is automatically initialized to zero?

17.7: External data sources. Suppose we a function, prototyped as follows:

```
string GetExternal(const char *szName);
bool SetExternal(const char *szName,const char *szValue);
```

that could be called to get and set the value of some variable (szName) from some external source (e.g., the Internet, a database, spawning another program, whatever). What types of modifications would be required so that our calculator program could use such a function seamlessly.

Questions 17.11-17.15 deal with extensions to the calculator application

17.11: Keyword support. Modify the general calculator (by inheritance, most likely) to support keywords (such as AND and OR) that can be used in place of operators.

Part Four:

Projects

Chapter 22

dBase Interpreter Project

Executive Summary

Chapter 22 provides detailed instructions for the design and implementation of a simple single-table database interpreter. It is the first of three major project chapters whose purpose is to provide the reader with the opportunity to take a design specification—along with some useful hints—and transform it into an operational application that is an order of magnitude more complex than any end-of-chapter lab exercise. As is the case with all three projects provide in this text, it is organized so that earlier project stages can be completed well before the reader has completed the book. Each section provides guidance regarding how far the reader needs to be. For this project, the ANSI C++ chapters (Chapter 6 through Chapter 17) are sufficient to complete the entire project.

The chapter begins by introducing the dBase table, a mainstay of programming in the mid-1980s and still widely used (through the MS FoxPro application) through the late 1990s. The project then proceeds through a series of stages, wherein the reader learns to 1) extract the structure of a table from a .dbf file header, 2) read and write data in the dBase record format, 3) implement memory-based sorting and searching algorithms, and 4) interact with the user using an interpreter-based interface.

Learning Objectives

Upon completing this project, you should be able to:

- Comfortably extract and write data within a binary file of known format using the STL file system
- Describe some of the issues involved when working with data of different types
- Understand how a C++ program typically interacts with external data sources
- Explain a number of issues that are routinely addressed in database design (e.g., deletion vs. erasing data, implementation of indices, views vs. tables)
- Use various STL classes to support memory-based sorting and filtering tables
- Use a grammar to implement testing required for searching
- Implement a simple grammar-based interpreter
- Use inheritance effectively in the creation of multi-stage projects

22.1: Introduction to dBase

In this section we provide some background on the dBase application, overview the entire project, then discuss—in detail—the dBase III table file format.

22.1.1: dBase Background

In the early days of personal computing, the dBase application, developed by Ashton-Tate, was the premier tool for implementing databases. Long before SQL-based PC applications (e.g., MS Access) became available, dBase and dBase-inspired applications (such as FoxPlus and FoxPro) were used to implement local databases at many companies. Indeed, well into the late 1990s, dBase-inspired tools (most notably FoxPro, which had become MS Visual FoxPro by that time) were still commonly used in small and mid-sized companies because of their outstanding performance.

By the time the third version of dBase has been released, the tool had many advanced capabilities that included the ability to:

- Create tables with up to 128 fields (i.e., columns) containing text, numeric, data and logical data. In dBase, each table was stored in a separate file with the .dbf extension.
- Assign indexes to tables, stored in separate files with the .ndx extension, used to order a table.
- Set filters that limited visible records to a subset of a table, creating a type of query.
- Iterate though tables to process their contents.
- Establish relationships between indexed tables, implementing a form of natural join.
- Write complex programs using the dBase programming language.

By today's standards, many of the capabilities of dBase III seem quite primitive. Even the entry level tools of today, such as MS Access, provide support for advanced capabilities—such as a GUI interface, relational integrity enforcement and SQL-based queries—that far exceed those provided by dBase. Nonetheless, the dBase family of applications set the standard for PC databases for more than a decade.

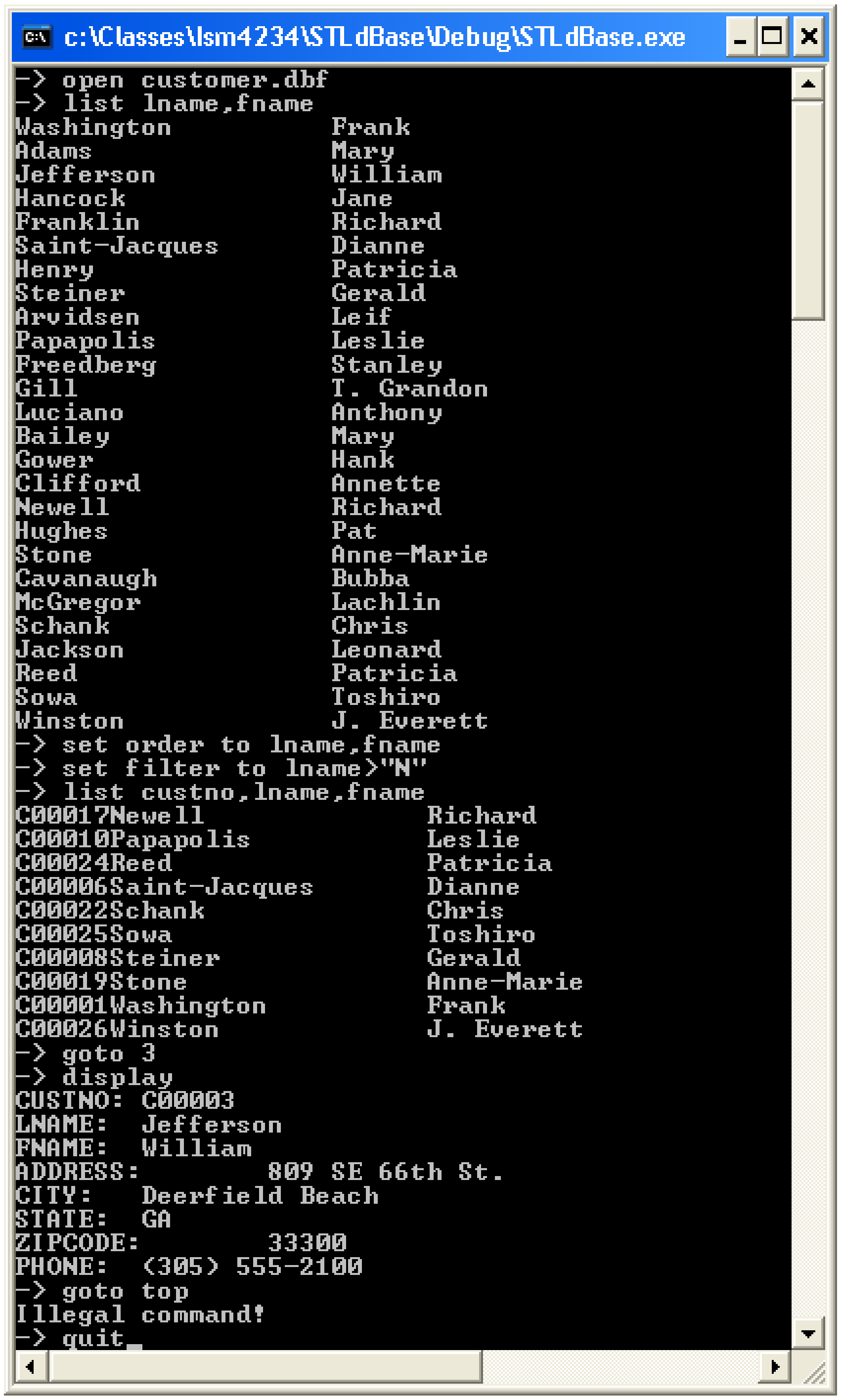
22.1.2: Project Overview

Walkthrough available in dBase.avi

In this project, we’ll be creating a “mini” dBase, capable of working with a single .dbf formatted file.

Project Interface

The interface of the project will, as might be expected, change as you proceed through development. By the final stage of the project, the application will provide a command interpreter as a user-interface, meaning that the user will type in commands and the application will respond to them. A example of the interpreter in operation is presented in Figure 22.1.



```
-> open customer.dbf
-> list lname,fname
Washington          Frank
Adams               Mary
Jefferson           William
Hancock             Jane
Franklin            Richard
Saint-Jacques       Dianne
Henry               Patricia
Steiner             Gerald
Arvidsen            Leif
Papapolis           Leslie
Freedberg           Stanley
Gill                T. Grandon
Luciano             Anthony
Bailey              Mary
Gower               Hank
Clifford            Annette
Newell              Richard
Hughes              Pat
Stone               Anne-Marie
Cavanaugh           Bubba
McGregor            Lachlin
Schank              Chris
Jackson             Leonard
Reed                Patricia
Sowa                Toshiro
Winston             J. Everett
-> set order to lname,fname
-> set filter to lname>"N"
-> list custno,lname,fname
C00017Newell              Richard
C00010Papapolis           Leslie
C00024Reed                Patricia
C00006Saint-Jacques       Dianne
C00022Schank              Chris
C00025Sowa                Toshiro
C00008Steiner             Gerald
C00019Stone               Anne-Marie
C00001Washington          Frank
C00026Winston             J. Everett
-> goto 3
-> display
CUSTNO: C00003
LNAME:  Jefferson
FNAME:  William
ADDRESS:        809 SE 66th St.
CITY:   Deerfield Beach
STATE:  GA
ZIPCODE:        33300
PHONE:  (305) 555-2100
-> goto top
Illegal command!
-> quit_
```

Figure 22.1: Example of Completed dBase Project Interface

Project Organization
The core of the project is the creation a CDB3File class, which will inherit from the STL *fstream* class. This class will allow you to load, change, and display dBase III formatted files. Using this as a base class, you will then create a CDB3Sorted class that sorts and

filters data from a dBase III+ formatted file. Finally, you will create a CDB3Command class, which will allow you to send commands to your file using a command interpreter.

The project will be constructed in a series of stages:

1. *Encapsulating the .dbf table structure.* The first stage of the project involves creating classes that read and write table structure information. It involves the creation of three classes:
   - *Header class:* You will create a CHeader class that reads and writes to the first 32 bytes of a dBase file (known as the header). This header includes information such as the type of file, the number of records, and the date it was last modified.
   - *Field class:* You will create a CField class that captures the definition information for each field in the table. This information includes field name, field type, field length and precision (for numeric fields).
   - *Field collection class*: You will create a *CFieldSet* class, which consists of a collection of *CField* objects composed with a single *CHeader* object. This class, effectively, will contain all the definition information associated with a .dbf file.
2. *Encapsulating record reading and writing.* You will create a *CDB3File* class (that inherits from *fstream* and is composed with a *CFieldSet* object) that allows you to manipulate a single table in a .dbf format. In a series of steps you will be implementing display, editing, adding, and deleting.
3. *Sorted database class*: You will create a *CDB3Sorted* class, inheriting from *CDB3File*, that allows you to manipulate a single table in a .dbf format. In a series of steps you will implement sorting and selecting records from a data file. As part of implementing searching, you will incorporate elements of the Calculator lab exercise presented in Chapter 17, Section 17.3.
4. *Command interpreter:* As the final part of the project, you will be creating a command interpreter class, *CDB3Command*, that inherits from *CDB3Sorted* and takes commands (as text strings) from the user and then executes appropriate member functions on the database object. This class will involve a further extension of the Calculator lab exercise.

Command Set

The commands to be supported by the final stage of the project are listed in Table 22.1.

Command	Description
QUIT	Exits the interpreter
OPEN file-name	Opens a new .dbf file, closing the existing file if one is already open.
GOTO location	Moves to a specific record in the table. Location may be either TOP, BOTTOM or a record number.
LIST [fieldlist]	Lists all the records in the table, one per row. If a fieldlist is provided (a list of field names, separated by commas), it only displays those fields.
DISPLAY [ALL] [fieldlist]	Displays records and field names, one row per field. If ALL is present, all records are displayed, otherwise only the active record is displayed. If a fieldlist is provided (a list of field names, separated by commas), it only displays those fields.
SET field TO value	Sets the value of the specified field to the specified value (which must be the same type as the field) for the active record.
SET ORDER TO fieldlist	Changes the order of display and list commands to the order specified by the field list.
SET FILTER TO expression	Changes the DISPLAY ALL and LIST commands so that only those meeting the specified expression (which can be tested using the CFilterExpression class provided to you) are visible.
DELETE [recordnumber]	Deletes the active record (i.e., marks it as deleted), without actually removing it from the file.
PACK	Removes all deleted records from the file.
APPEND [COPY]	Adds a record to the end of the file. If COPY is provided, the record added is a copy. Otherwise, a record consisting entirely of spaces is added.
RECALL [recordnumber]	Removes the deleted mark from the specified record (or the active record, if none is specified).
SKIP	Moves to the next active record, using the index if available
BACK	Moves to the previous active record, using the index if available
RUN cmd-file [out-file]	Runs a series of commands in a batch file (cmd-file). If the out-file argument is present, it sends the output to a file Otherwise, output is sent to the console.

Table 22.1: dBase Commands to be Implemented

22.1.3: DBF File Structure

In many ways, actual data is stored in a .DBF file in a manner similar to that found in the fixed length text format in that all record data is stored as text, and all records are the same length (implying a constant length for each field). There is, however, a key difference between DBF and text files: DBF files contain information about how the file is laid out in a section of the file known as the header section. Specifically, the header contains information such as:

- The date when the table was last modified
- The number of records (rows) in the table
- The length of each record
- The size and type of each data element

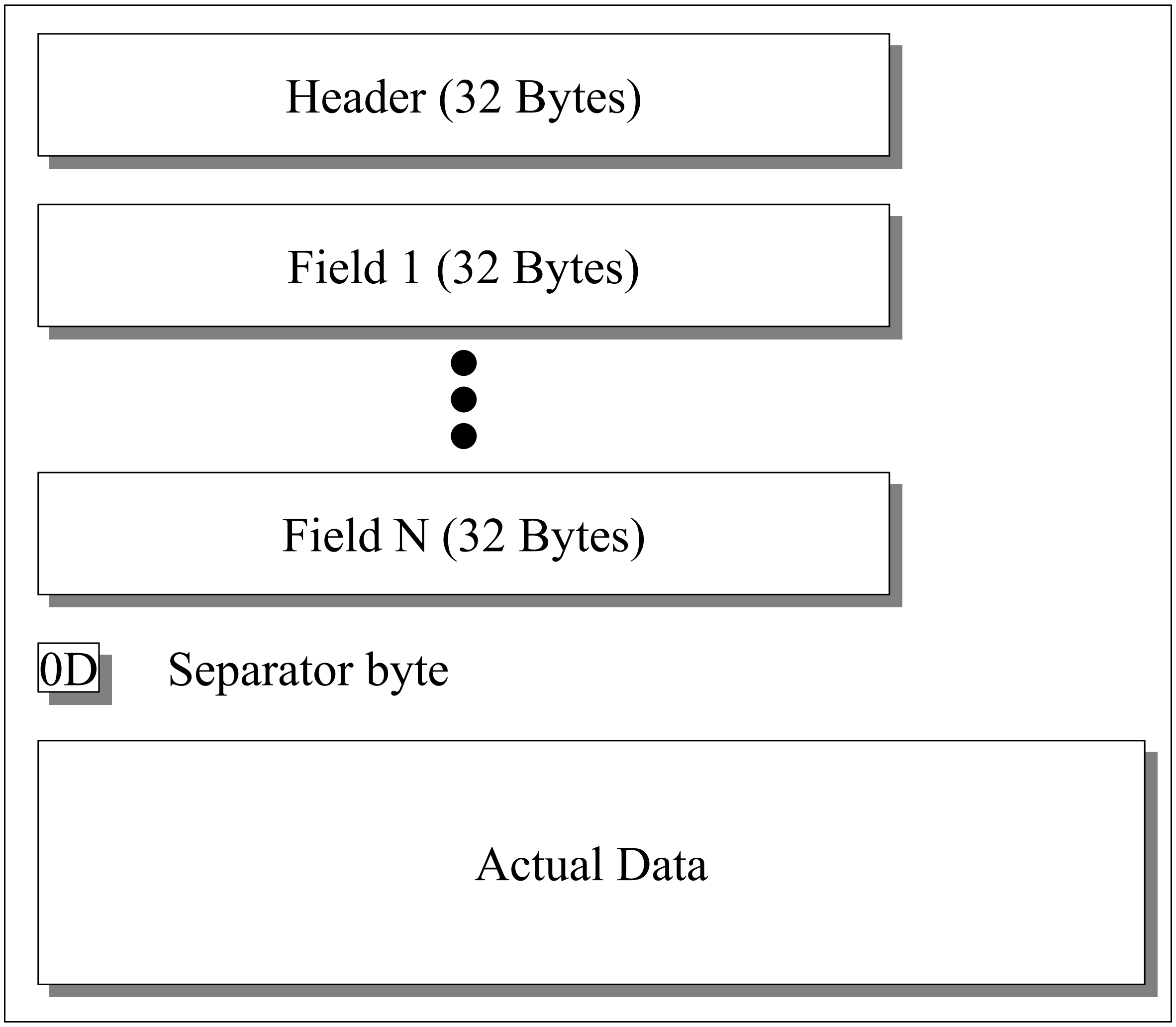
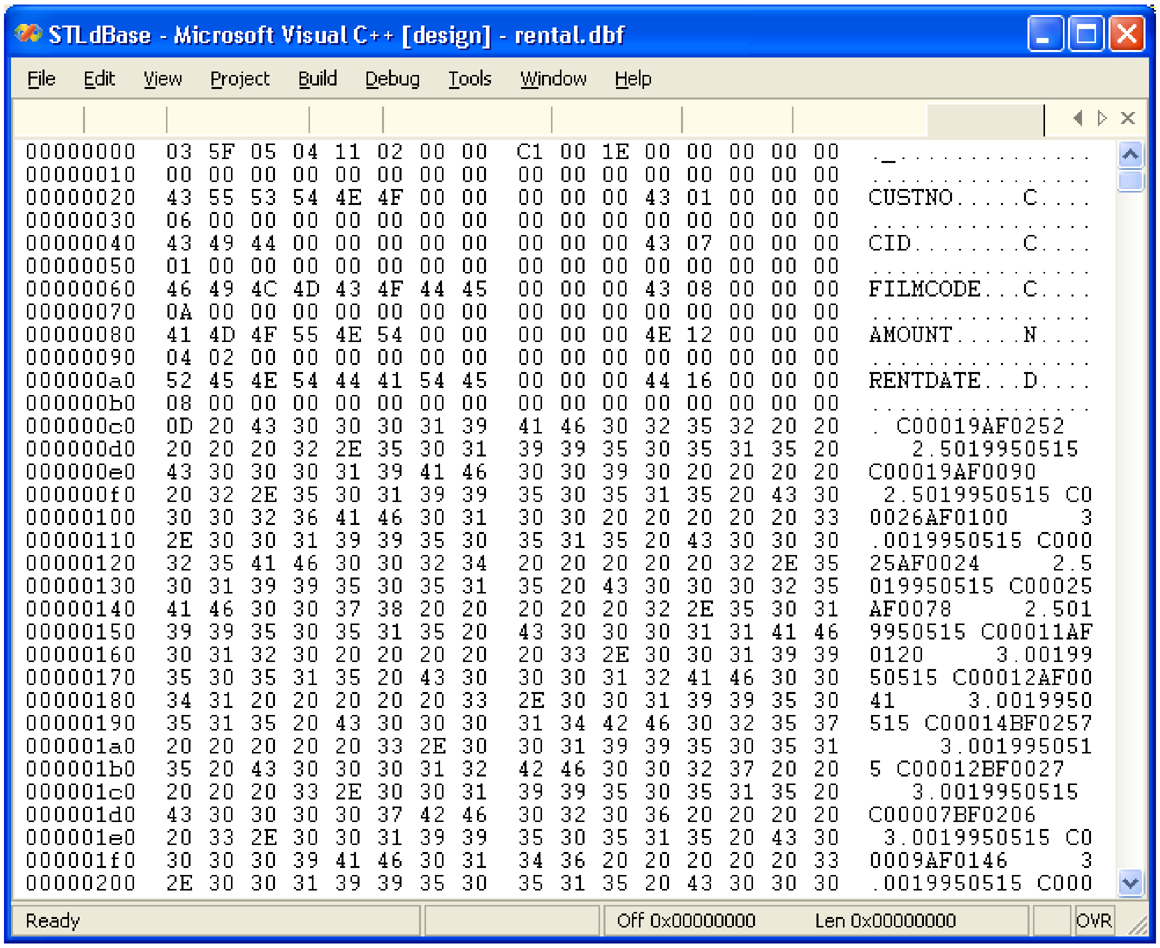


Figure 22.2: DBF file organization

The overall organization of a .dbf file, illustrated in Figure 22.2, is as follows:

- The file begins with 32 bytes of information related to the organization of the entire file, including date modified, record length and number of records.
- Definitions of individual fields (up to 128) then follow. Each definition is 32 bytes and contains information about the field name, data type and size.
- A single carriage return, '\n' follows the field definitions.
- The actual data for the table then begins.

A binary display of the contents of an actual DBF file (Rental.DBF) is presented in Figure 22.3.


STLdBase - Microsoft Visual C++ [design] - rental.dbf
File Edit View Project Build Debug Tools Window Help
Ready
Off 0x00000000
Len 0x00000000
OVR

```
00000000  03 5F 05 04 11 02 00 00  C1 00 1E 00 00 00 00 00  ._..............
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000020  43 55 53 54 4E 4F 00 00  00 00 00 43 01 00 00 00  CUSTNO.....C....
00000030  06 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000040  43 49 44 00 00 00 00 00  00 00 00 43 07 00 00 00  CID........C....
00000050  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000060  46 49 4C 4D 43 4F 44 45  00 00 00 43 08 00 00 00  FILMCODE...C....
00000070  0A 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000080  41 4D 4F 55 4E 54 00 00  00 00 00 4E 12 00 00 00  AMOUNT.....N....
00000090  04 02 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000a0  52 45 4E 54 44 41 54 45  00 00 00 44 16 00 00 00  RENTDATE...D....
000000b0  08 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000c0  0D 20 43 30 30 30 31 39  41 46 30 32 35 32 20 20  . C00019AF0252
000000d0  20 20 20 32 2E 35 30 31  39 39 35 30 35 31 35 20     2.5019950515
000000e0  43 30 30 30 31 39 41 46  30 30 39 30 20 20 20 20  C00019AF0090
000000f0  20 32 2E 35 30 31 39 39  35 30 35 31 35 20 43 30   2.5019950515 C0
00000100  30 30 32 36 41 46 30 31  30 30 20 20 20 20 20 33  0026AF0100     3
00000110  2E 30 30 31 39 39 35 30  35 31 35 20 43 30 30 30  .0019950515 C000
00000120  32 35 41 46 30 30 32 34  20 20 20 20 20 32 2E 35  25AF0024     2.5
00000130  30 31 39 39 35 30 35 31  35 20 43 30 30 30 32 35  019950515 C00025
00000140  41 46 30 30 37 38 20 20  20 20 20 32 2E 35 30 31  AF0078     2.501
00000150  39 39 35 30 35 31 35 20  43 30 30 30 31 31 41 46  9950515 C00011AF
00000160  30 31 32 30 20 20 20 20  20 33 2E 30 30 31 39 39  0120     3.00199
00000170  35 30 35 31 35 20 43 30  30 30 31 32 41 46 30 30  50515 C00012AF00
00000180  34 31 20 20 20 20 20 33  2E 30 30 31 39 39 35 30  41     3.0019950
00000190  35 31 35 20 43 30 30 30  31 34 42 46 30 32 35 37  515 C00014BF0257
000001a0  20 20 20 20 20 33 2E 30  30 31 39 39 35 30 35 31       3.001995051
000001b0  35 20 43 30 30 30 31 32  42 46 30 30 32 37 20 20  5 C00012BF0027
000001c0  20 20 20 33 2E 30 30 31  39 39 35 30 35 31 35 20     3.0019950515
000001d0  43 30 30 30 30 37 42 46  30 32 30 36 20 20 20 20  C00007BF0206
000001e0  20 33 2E 30 30 31 39 39  35 30 35 31 35 20 43 30   3.0019950515 C0
000001f0  30 30 30 39 41 46 30 31  34 36 20 20 20 20 20 33  0009AF0146     3
00000200  2E 30 30 31 39 39 35 30  35 31 35 20 43 30 30 30  .0019950515 C000
```

Figure 22.3: Binary Display of Rental.DBF file

Because the MS Visual Studio binary viewer displays 16 bytes per row, it is relatively easy to locate data in the file. The first two rows correspond to the header block of information. The key elements of the header are as follows:

- **Byte 0:** Should be 0x03 for dBase III files. (0x83 is also legal, but it implies a separate memo file, which will not be relevant here).

- **Byte 1:** Year the file was last modified (1900 is 0). In Figure 22.3, the value 0x5F corresponds to 95 decimal, or 1900+95=1995.
- **Byte 2:** Month the file was last modified. In Figure 22.3, the value 0x05 signifies May.
- **Byte 3:** Day the file was last modified. In Figure 22.3, the value 0x04 signifies 4. Thus, the file was last modified on 5/4/1994.
- **Bytes 4-7:** A 4-byte integer indicating the number of records in the file. Reversing the bytes, this becomes 0x00000211, which is 1041.
- **Bytes 8-9:** A 2-byte integer specifying where the actual data in the file begins. In this case (reversing the bytes), the position is 0x00C1. Looking at the addresses on the left, you can verify that this is—in fact—the actual location where the data appears to be located (immediately after the 0x0D separator byte shown in Figure 22.3).
- **Bytes 10-11:** A 2-byte integer specifying the record length. In dBase III, the maximum record length was 4000—so this number should always be less than or equal to that value. In Figure 22.3, the value 0x001E specifies that each record in 30 bytes long.

The remaining bytes in the 32-byte header are ignored. Following the header block comes the field definitions. The number of fields defined for the table can be computed from the information in the header block. Specifically, you take the location where the data begins (0xC1) and subtract the bytes used for the header (0x20 bytes) and the separator byte (0x01 byte). This leaves 0xA0 bytes for all the field definitions (160 bytes, in decimal). We then divide this by the size of each field definition (0x20 bytes, or 32) and the result is 5 fields (160/32=5). This is appears consistent with the general pattern on the right hand side (text display) of the viewer.

Each 32-byte field definition is laid out as follows:

- **Bytes 0-10:** A NUL terminated string containing the field name. In dBase III, field names were limited to 10 characters (with the additional byte allocated for the NUL terminator). For example, the field beginning at 0x00000020 is named "CUSTNO", the field beginning at 0x00000060 is named"FILMCODE", etc.
- **Byte 11:** A single character indicating the field type. Allowable types included 'N' (numeric), 'C' (character, or text), 'D' (date) and 'L' (logical, or TRUE/FALSE). For example, the CUSTNO field has a value of 0x43 (ASCII for 'C'), meaning its text. The AMOUNT field has a value of 0x4E (ASCII for 'N'), meaning it's a numeric field. The RENTDATE field has a value of 0x44 (ASCII for 'D'), meaning it is a date field
- **Byte 16:** A single character specifying the field length (individual dBase III fields were limited to 255 bytes). For example, the CUSTNO field is 0x06 (6) characters long. The CID field is 0x01 (1) character long, and so forth.
- **Byte 17:** Relevant only for numeric fields, it's the number of places to the right of the decimal place—roughly equivalent to the precision of a numeric field. For example, the AMOUNT field has a precision of 0x02 (2), meaning it is written

will 2 decimal places, while the length of the field is 4, meaning the maximum amount we can handle is 9.99 (since the decimal is included in the length).

The remaining bytes in each field definition can be ignored for our purposes. A list of field types, and their interpretation, is presented in Table 22.2.

Code	Type	Storage
C	Text	Stored as ASCII characters, based on the field length, with no NUL terminators
N	Numeric	Stored as ASCII characters. The field length includes the decimal point character.
D	Date	Stored as an 8 character integer in the form YYYYMMDD (e.g., 20000826 for 8/26/2000)
L	Boolean	Single byte, stored as 'T' (true) or 'F' (false).
M	Memo	Not supported by our interpreter. Refers to a field with no length limitations in a separate file.

Table 22.3: Allowable dBase 3 Field Types

22.2: Working with DBF Table Structure

In this section, we develop a series of classes that allow use to read and change the header section of the dbf file. In this project, we will not implement table structure modifications (which are substantially more complex that data structure modifications). The classes are as follows:

- CHeader: works with the 32-byte block at the beginning of the dbf file.
- CField: Works with individual 32-byte field definitions
- CFieldSet: Manages the collection of CField object and the CHeader object.

22.2.1: CHeader class

The CHeader class needs to be able to read and write header information. Its declaration is presented in Example 22.1.

Example 22.1: CHeader class declaration

```
class CHeader
{
public:
	CHeader();
	virtual ~CHeader();
	long Date() const;
	void Date(long nDate);
	long Records() const;
	void Records(long nRecords);
	short int HeaderLength() const;
	void HeaderLength(short int nHdr);
	short int RecordLength() const;
	void RecordLength(short int nRecLen);
	bool Load(fstream &strm);
	bool Save(fstream &strm) const;
	short int FieldCount() const;
	friend ostream &operator<<(ostream &out,const CHeader &hdr);
protected:
	long m_nDate;
	long m_nRecords;
	short int m_nHeaderLength;
	short int m_nRecordLength;
};
```

A summary of the member functions, and general implementation instructions, are presented in Table 22.4. Approximate length guidelines are also provided, which provide a rough indicator of function complexity.

CHeader Function Descriptions
Defaults for CHeader(),~CHeader() are sufficient.
Date(), Records(), HeaderLength(), RecordLength() are all 1 line accessor functions.
FieldCount()[1 line] Computes the number of fields using m_nHeaderLength. (See what the header length value consists of to determine the calculation.)
bool Load(fstream &strm) [10 lines]: • Returns false if NULL (0) file pointer is passed in. • Moves to the beginning of the file then reads 32 bytes into a character buffer. • Checks that byte 0 is a legal value (see header specs) • Takes the three date bytes and transforms them into a long integer in the form YYYYMMDD • Using memcpy(), copies the remaining values into the appropriate data members. • returns true
bool Save(fstream &strm) const [13 lines] • Returns false if stream is not open • Creates 32 byte buffer and initializes it to all 0x00's using memset() • Places legal value in byte 0 • Converts long date member to year, month and day bytes • Using memcpy(), copies remaining members into appropriate positions in buffer • Moves to start of file and writes 32 byte header • returns true
friend ostream &operator<<(ostream &out,const CHeader &hdr) [5 lines] • Sends member values to screen, so they can be examined for test purposes • Returns out

Table 22.4: CHeader Function Descriptions

22.2.2: CField class

The CField class holds information on an individual field structure. Its declaration is presented in Example 22.2. Of particular note is that the class uses the IString class—implemented for case-insensitive lookup in Chapter 14, Section 14.4—as a return type for the Name() function. This class will be used throughout the application, because dBase was generally case insensitive.

Example 22.2: CField class

```
class CField
{
public:
      CField();
      virtual ~CField();
      const CField &operator=(const CField &oth);
      CField(const CField &oth);

      IString Name() const;
      void Name(const char * szName);
      char Type() const;
      void Type(char nType);
      unsigned char Length() const;
      void Length(unsigned char nLen);
      unsigned char Precision() const;
      void Precision(unsigned char nPrec);
      short int StartPosition() const;
      void StartPosition(short int nPos);
      short int NextPosition() const;

      bool Load(fstream &strm);
      bool Save(fstream &strm) const;

      friend ostream &operator<<(ostream &out,const CField &hdr);

protected:
      void Copy(const CField &oth);

      string m_szName;
      char m_nType;
      unsigned char m_nLength;
      unsigned char m_nPrecision;
      short int m_nStartPosition;

};
```

A summary of the CField member functions, and general implementation instructions, are presented in Table 22.5.

CFieldFunction Descriptions
Defaults for CField(),~CField() are sufficient.
Name(), Type(), Length(), Precision(), StartPosition() are all 1 line accessor functions
NextPosition() returns the start position plus the field length [1 line]
CField(const CField &oth) [1 line] • Copy constructor needs to be defined because field structures will be used in templates that make copies and projects • Calls the protected member Copy()
const CField &operator=(const CField &oth) [3 lines] • Checks to make sure that the left and right side of the project aren't the same • Calls the protected member Copy() • Returns oth
void Copy(const CField &oth) [5 lines] • Used in both project operator and copy constructor • Assigns each member the value of the associated member of oth
bool Load(fstream &strm) [9 lines] • Returns false if file pointer is 0 • Reads from the current position of the file into a 32 character buffer (Note: this function assumes the file pointer is positioned properly before it is called) • Pulls values for various data members from the buffer (see specs). (Hint: since the field name is NUL terminated, the value of m_szName can be established using an assignment) • Returns true
bool Save(fstream &strm) const [10 lines] • Returns false if file pointer is 0 • Initializes a 32 byte buffer to 0x00's using memset() • Puts values from various data members into the buffer (see specs). • Writes to the current position of the file from the 32 character buffer (Note: this function assumes the file pointer is positioned properly before it is called) • Returns true
friend ostream &operator<<(ostream &out,const CField &hdr) [6 lines] • Writes the values of the various member variables to out. • Returns out • Used for testing purposes only

Table 22.5: CField Function Descriptions

22.2.3: CFieldSet class

Now that we have defined individual header and field objects, we're going to compose them into a single object capable of reading and writing the complete definition sections of .dbf files. Since every table is a collection of fields, we choose to inherit the class from the STL vector<> template. This way, we can use all the vector<> accessors to get at individual fields. The CFieldSet class declaration is presented in Example 22.3.

Example 22.3: CFieldSet Declaration

```
class CFieldSet : protected vector<CField>
{
// First stage
public:
	CFieldSet();
	virtual ~CFieldSet();
	long RecordCount() const;
	void RecordCount(long nCount);
	long Date() const;
	void Date(long nDate);
	short int RecordLength() const;
	short int HeaderLength() const;
	bool Load(fstream &strm);
	bool Save(fstream &strm,bool bHeaderOnly=false) const;
	friend ostream &operator<<(ostream &out,const CFieldSet &flds);
protected:
	CHeader m_hdr;
// Second stage
public:
	void RemoveAll();
	int Add(CField &fld);
	void Copy(const CFieldSet &ar);
	void InsertAt(int nPos,CField &fld);
	void RemoveAt(int nPos);
	int Field(const char * szName) const;
	bool Field(const char * szName,CField &fld) const;
	// Useful if protected inheritance is used
	const CField &operator[](size_t i) const;
	int FieldCount() const;
protected:
	hash_map<IString,int> m_mapFields;
	bool Remap();
};
```

Since creating the CFieldSet is substantially more involved than the previous CHeader and CField classes, we describe the functions in two stages. You can implement the first group, test it, then implement the second group.

The first group of functions performs the basic loading and saving of CFieldSet objects that sit at the beginning of a dbf file. These functions are described in Table 22.6.

CFieldSet Function Descriptions: Group 1
CFieldSet(),~CFieldSet(): Default versions are sufficient
RecordCount(), Date(), RecordLength(), HeaderLength() are all 1 line accessor functions that, in turn, call the equivalent m_hdr accessor functions. You often define such functions, for convenience, when you compose one class within another.
bool Load(fstream &strm) [8 lines] • Calls the load function on the m_hdr object to load the header. Returns false if this fails • Calls RemoveAll() member (uses vector<CField>::clear(), for the time being) to get rid on any existing definitions. • Loops through the fields (you can get a count from the header), reading each field into a CField object then calling push_back() (inherited) to add it to the array. • Returns true
bool Save(fstream &strm,bool bHeaderOnly=false) const [6 lines] • Calls the Save() member on m_hdr to save the header. Returns false if this fails. • If bHeaderOnly argument is true, it returns true. This argument will be useful later in the project because when working with databases, we need to update the header frequently (e.g., when records are added, or the date changes), but rarely need to change the field definitions for an existing .dbf file. • Loops through its elements and calls the Save() member on each (because we inherit from a template which is based on a CField array, it knows to use the CField save). • Writes a single 0x0D byte to the file after the last field (the delimiter byte). • Returns true
friend ostream &operator<<(ostream &out,const CFieldSet &flds) [4 lines] • Outputs header using << operator • Iterates through the fields, outputting each using the << operator • Returns out

Table 22.6: CFieldSet Function Descriptions (Group 1)

Although it is convenient to use a vector<> array to hold the field definitions, in a working program such an array would not be best collection shape. Specifically, a program using a database is likely to be continually accessing fields by name (rather than by their position in the file definition). To improve performance, we can therefore add an STL **hash_map<>** lookup table to the CFieldSet class, that associates the name of each field with its position in the array. Once again, IString objects as used as key, implementing appropriate case-insensitivity.

When we compose the lookup table with the CFieldSet class, however, we should realize that a lot of the default members of vector<> are no longer satisfactory. The reason is that every time we either add or remove an element from the array, we need to modify the lookup table. To avoid problems, we implement our own interface member functions, such as Add(), InsertAt(), RemoveAt() and RemoveAll(). Furthermore, it is probably a good idea to change to protected inheritance, preventing a class user from using the native vector<> members, such as push_back(). Indeed, a reasonable case could also be made that the class should really inherit from CHeader, and compose the vector<Field> collection as a member. More than one reasonable design path is quite common in object-oriented design. The second group of CFieldSet function definitions are presented in Table 22.7.

CFieldSet Function Descriptions: Group 2

void RemoveAll() [2 lines]
- Calls the base class clear()
- Calls clear() on the m_mapFields member.

int Add(CField &fld) [3 lines]
- Calls the base class push_back()
- Calls the protected Remap() member, to regenerate the lookup table
- Returns the position of the add (the size of the vector<> minus 1).

void Copy(const CFieldSet &ar) [3 lines]
- Calls the base class assign() member, which takes two iterators as arguments reflecting the start and end of the range being copied (i.e., ar.begin() and ar.end()).
- Copies the header from the argument (using an project)
- Calls the protected Remap() member, to regenerate the lookup table

void InsertAt(int nPos,CField &fld) [2 lines]
- Calls base class member of insert (you'll need to create an iterator)
- Calls protected Remap() member, to regenerate the lookup table

void RemoveAt(int nPos) [2 lines]
- Calls base class member of erase() at the specified position (you'll need to create an iterator)
- Calls protected Remap() member, to regenerate the lookup table

int Field(const char * szName) const [5 lines]
- Places szName in a temporary string, then makes the string upper case (so field names are case insensitive)
- Performs a lookup operation on the m_mapFields member. If found, it returns the associated array position. If not, it returns -1.

bool Field(const char * szName,CField &fld) const [4 lines]
- Calls the Field(const char *) member to determine position. If less than 0, returns false
- Assigns the associated CField object to the fld referenced argument
- Returns true

bool Remap() [12 lines]
- Calls the clear() member on the m_mapFields lookup table to remove the old data
- Iterates through the array (using size() and [] operator) to get each field.
- Places the name of each field in a temporary string, which is then made uppercase
- Uses Lookup() member to find out if the field is already present. If so, the function returns false since duplicate fields are not permitted
- Uses the hash_map [] overload to associate each upper case name with its position in the array
- Returns true

const CField &operator[](size_t i) const [1 line]
Overloads the [] operator (only necessary if protected inheritance is used).

int FieldCount() const [1 line]
Substitutes for the size() member (only necessary if protected inheritance is used).

Table 22.7: CFieldSet Function Descriptions (Group 2)

22.3: CDB3File class

In this section, you will create a CDB3File class, which encapsulates the various functions needed to work with dBase formatted files. We have chosen to inherit the CDB3File class (with inheriting from CFieldSet also being a perfectly reasonable alternative). The complete class declaration is presented in Example 22.4.

Because the class is substantially more complex than the previous classes, design will be staged. You should test each group of functions before going on to the next group.

Example 22.4: CDB3File Class Declaration

```
class CDB3File : public fstream
{
        // Group 1: Basic Elements
public:
        CDB3File();
        CDB3File(const char * szName);
        virtual ~CDB3File();

        bool Open(const char * szName);
        virtual void Close();

        long Seek(long lOff);
        virtual unsigned long length() const;
        virtual unsigned long GetPosition( ) const;
protected:
        long ComputeFilePosition(long nRecNo) const;
        virtual bool Read(string &szBuf, long nRecPos);
        virtual void Write(const char * lpBuf, long nRecPos);
        CFieldSet m_setFields;
        long m_nRecordPosition;
        string m_szRecordBuffer;
        bool m_bOpen;
        string m_szFileName;
        // Group 2: Getting Field Data
public:
        string GetField(const char * szName) const;
        string GetField(const CField *pFld) const;
        char FieldType(const char * szName) const;
        bool GetField(const char * szName,string &szValue) const;
        bool GetField(const char * szName,bool &bValue) const;
        bool GetField(const char * szName,double &dValue) const;
        bool GetField(const char * szName,long &lValue) const;
        bool Display(ostream &out) const;
        bool Display(ostream &out,const vector<string> &arFields) const;
        friend ostream &operator<<(ostream &out,const CDB3File &rec);
protected:
        bool Field(const char * szName,CField &fld) const;
        // Group 3: Setting Field Data
public:
        bool SetField(const char * szName,const char * szValue);
        bool SetField(const char * szName,bool bValue);
        bool SetField(const char * szName,double dValue);
        bool SetField(const char * szName,long lValue);
protected:
        virtual bool ReplaceField(const CField *fld,const char * szVal);
        void Update();
        // Group 4: Table and record operations
public:
        bool AppendBlank();
        virtual bool AppendCopy(const char * szBuf=0);
        bool IsDeleted();
        virtual bool Delete(long nOff=-1);
        virtual bool Recall(long nOff=-1);
        virtual bool Pack();
        virtual void DisplayAll(ostream &out,const vector<string> &arExpr);
        virtual void DisplayAll(ostream &out);
        virtual void List(ostream &out,const vector<string> &arExpr,char cSep=0);
        virtual void List(ostream &out,char cSep=0);
        string GetFields(long nPos,const vector<string> &arList,char cSep=0);
protected:
        bool Create(const char * szName,const CFieldSet &set);
};
```

22.3.1: Establishing Basic Record I/O

The first group of functions, presented in Table 22.8, implements basic file operations and block reading/writing. Individual field information is not required for these activities because our interpreter always reads and writes data as an entire record.

CDB3File Function Descriptions: Group 1
Data members: • m_setFields: will be used to work with the header and field definitions • m_nRecordPostion: at any given time, the data from a single record will be loaded into m_szRecordBuffer. m_nRecordPosition keeps track of what record is loaded. Note: according to dBase convention, the first record is record 1 (not record 0, as it would typically be in C++) • m_szRecordBuffer: a string object that holds the data for the current record. Because dBase files hold their data using ASCII text, a string is a perfectly reasonable way of storing the entire record. • m_bOpen: Keeps track of whether the file is open or not. Sometimes referred to as the open flag.
CDB3File() [2 line] • Sets the open flag to false • Sets m_nRecordPosition to –1 (signifying no record is loaded) • Towards the end of the project, additional initialization operations will need to be added
CDB3File(const char * szName) [1 line] • Calls Open() member function. • Assigns the return value of Open() to m_bOpen.
Default version of ~CDB3File() will initially serve, since the CDB3File class doesn't work with any pointers and the fstream object automatically closes the file when it is destroyed. Towards the end of the project, cleanup operations will need to be added.
bool Open(const char * szName) [9 lines] • Checks if a table is already open and, if so, closes it • Calls fstream::open() member, passing it the name of the file and appropriate flags (e.g., for binary, read-write opening, not truncated). • Loads the header into m_setFields, returning false if it fails. If successful, sets the m_szFileName member to its argument. • If the file is empty (no records), sets the position member to 0, empties the record buffer and sets the open flag to false. • If records are present, sets the open flag to the return value of a seek operation on the first record. • Returns the open flag.

void Close() [4 lines*]
- Empties the field set with a call to its RemoveAll() member
- Empties the record buffer
- Sets the open flag to false
- Calls the base class close() to close the file
- Note: Some additional cleanup will need to be added later, when sorting and filters are present

long ComputeFilePosition(long nRecNo) const [2 lines]
- Purpose: Computes the actual position in the file where a record begins
- Checks to make sure the record number exists, returning –1 if not.
- Returns the file position, computed with a formula based on the header length, the field length and the record number

virtual bool Read(string &szBuf, long nRecPos) [8 lines]
- Purpose: Read the contents of a single record (at position nRecPos) a single record into a string (szBuf).
- Checks to make sure nRecPos is a legal record, returning false otherwise.
- Determines the file position where the record begins (using ComputeFilePosition()) and calls the base class seekp() & seekg() to move there.
- Creates a temporary character buffer, size MAXREC+1 (you should probably define MAXREC as 4000 in your .h file), and initializes it to 0x00's using memset().
- Using the base class Read() member, reads the bytes (hint: number of bytes is equal to the record length), returning false if the read fails to get the required number of bytes.
- Assigns the temporary record buffer to the szBuf string.
- Returns true

virtual void Write(const char * lpBuf, long nRecPos) [4 lines]
- Purpose: Writes the contents of a single record (contained in the buffer lpBuf) to the appropriate position in the file.
- Checks to make sure nRecPos is a legal record, returning otherwise.
- Determines the file position where the record begins (using ComputeFilePosition()) and calls the seekg() and seekp() members to move there.
- Writes the bytes to the file (using the base class write() member)

long Seek(long lOff) [7 lines]
- Purpose: To go to a particular record in the file (lOff) and load the data into the m_szRecordBuffer member. Returns the record number if successful, 0 otherwise.
- If lOff is the same as the current record number, returns lOff (since the record is already loaded)
- Verifies that lOff is a legal record
- Calls the protected Read() member (using m_szRecordBuffer as the string argument)
- If the Read fails, sets m_nRecordPosition to –1, empties the record buffer, and returns 0.
- Sets the current record number to lOff and returns its value.

virtual long GetLength() const [1 line]
- Returns the number of records in the file (accessible through the field set member)

virtual long GetPosition( ) const [1 line]
- Simple accessor function returning the current record number in the file.

Table 22.8: CDB3File Function Descriptions (Group 1)

22.3.2: Reading and Displaying Field Values

Once a record has been loaded into the buffer string (m_szRecordBuffer), pulling out the actual field values as strings is relatively straightforward . (Remember: the CField objects tell us where a particular field is located in the string). The only real problem is caused by the fact that we have four distinct field types (C, N, L and D). Character fields are no problem, but numeric fields need to be converted to doubles, logical fields to Boolean values (true or false) and date fields to long integers (e.g., 20000816 for 8/16/2000). As a result, we need to define a series of members to extract the data in its appropriate format.

The two versions of the Display functions also warrant some comment. Both are designed to display the data from the active record to an output stream in the form:

FieldName [tab] Value [\n].

For example:

```
CUSTNO     C10009
LNAME      Gill
FNAME      Grandon
etc…
```

The first version (with one argument) lists all the fields. The second version, however, uses its second argument (a vector<string> containing field names) to determine what fields appear in the display. (Note: if you look at how the DISPLAY command will be used in the command interpreter, you can see why both versions are useful).

The member functions to be implemented for field display are presented in Table 22.9. You should implement and test these functions prior to moving on to the next group.

CDB3File Function Descriptions: Group 2
bool Field(const char * szName,CField &fld) const [1 line] • Looks up a CField structure by name from the field set member, assigning it to the reference argument if found • Returns true if successful, false otherwise..

string GetField(const CField *pFld) const [5 lines]

- Purpose: extracts the text value of a particular field (based on the CField structure it is passed). The function is convenient because in many other functions, we will need both the CField structure (e.g., to determine what type of field it is) and its test representation. It returns an empty string if the operation fails.
- Creates a temporary string.
- If the field pointer is 0, the record position is not valid or the record buffer is empty, returns the empty temporary string.
- Extracts the text field from the record buffer (Hint: the string substr() member is very useful here) and assigns it to the temporary string.
- Returns the temporary string.

string GetField(const char * szName) const [2 lines]

- Purpose: Provides an alternative to the previous function for accessing the text value of a field.
- Calls the Field() member to get a pointer to the CField structure named szName, then returns the value of the GetField(const CField*) version of the function.

char FieldType(const char * szName) const [3 lines]

- Purpose: Provides convenient way of getting the field type using the field name as an argument.
- Calls the Field() member to get a pointer to the CField structure named szName
- If the pointer is 0, it returns 0. Otherwise, it returns the Type() member of the CField object returned.

bool GetField(const char * szName,string &szValue) const [3 lines]

- Purpose: Extracts a character field, named szName, as a string, checking to make sure that the field type is valid (i.e., ‘C’). The value of the field is placed in szName.
- If the field type is not ‘C’, it returns false.
- It calls the GetField(const char *) member and assigns the value to szValue.
- If the string returned is empty, the function returns false. Otherwise, it returns true.

bool GetField(const char * szName,bool &bValue) const [5 lines]

- Purpose: Extracts a logical field, named szName, as a □irfiel value, checking to make sure that the field type is valid (i.e., ‘L’). The value of the field is placed in bValue.
- If the field type is not ‘L’, it returns false.
- It calls the GetField(const char *) member and assigns the value to a temporary string.
- If the string returned is empty, the function returns false.
- If the first character of the temporary string is ‘T’, bValue is set to true. Otherwise, bValue is set to false.
- Returns true.

bool GetField(const char * szName,double &dValue) const [5 lines]

- Purpose: Extracts a numeric field, named szName, as a double precision real number, checking to make sure that the field type is valid (i.e., ‘N’). The value of the field is converted to double, and placed in dValue.
- If the field type is not ‘N’, it returns false.
- It calls the GetField(const char *) member and assigns the value to a temporary string.
- If the string returned is empty, the function returns false.
- The string is converted to float (Hint: using atof() is easiest) and assigned to dValue.
- Returns true.

bool GetField(const char * szName,long &lValue) const [5 lines] • Purpose: Extracts a date field, named szName, as a long integer value (e.g., 20000816), checking to make sure that the field type is valid (i.e., 'D'). The value of the field is converted to a long integer, and placed in lValue. • If the field type is not 'D', it returns false. • It calls the GetField(const char *) member and assigns the value to a temporary string. • If the string returned is empty, the function returns false. • The string is converted to long and assigned to lValue. • Returns true.
bool Display(ostream &out) const [10 lines] • Purpose: Loops through the fields of the active record and displays their names and values. Returns false if no record is active, true otherwise. • If the record buffer is empty, sends "{Empty}" to the output stream and returns false. • Loops through the fields in the field set and sends their name and string value to the output stream (one line per field). • Returns true.
bool Display(ostream &out,const vector<string> &arFields) const [8 lines] • Purpose: To display a specified set of fields, whose names are contained in the argument arFields. If the arFields array is empty, it displays all fields. Returns false if any values in the arFields array are not valid field names or if the record buffer is empty, true otherwise. • If the record buffer is empty or the arFields array is empty (i.e., size is 0), returns the single argument version of the Display() function. • Sets a temporary return flag to true • Loops through the fields in the arFields array and sends their name and string value to the output stream (one line per field). If any values are empty (signifying an invalid field name), sets the return flag to false. • Returns the value of the return flag.
friend ostream &operator<<(ostream &out,const CDB3File &rec) [2 lines] • Purpose: alternative to calling the Display() member. • Calls Display() on the rec argument. • Returns the output stream argument

Table 22.9: CDB3File Function Descriptions (Group 2)

22.3.3: Setting Field Values and Modifying the Database File

In this step, we need to add members that actually change the values of fields and update the records in the database. There are a number of considerations that need to be addressed here.

- Any time the database is updated, there is a good chance that we will need to write to the header (e.g., to update the date, change the number of records, etc.)

For this reason, it makes sense to encapsulate the updating of a single records in a protected member function, Update(), that will be called any time a change is made.

- Since we will need to update the date in the file header, we need to be able to access the system date. The C standard library <time.h> is useful in this regard. For example, to get today's date as a long integer, the following code can be used:

```
#include <time.h>
// etc…
time_t theTime=time(NULL);
struct tm *pTm, tmD;
pTm=localtime(&theTime);
// Copying data from static structure
tmD=(*pTm);
long nDate=
(tmD.tm_year+1900)*10000+(tmD.tm_mon+1)*100+tmD.tm_mday;
```

- One particularly tricky aspect of moving data into dBase files is formatting the strings correctly. dBase stores numbers as text, with a specified length and precision. The question is, how to get our numbers formatted according to a field specification. One technique is to use two sprintf formatting commands. For example, suppose our value is in the double *dVal*, our field length is in the integer variable *nLen*, and our precision is in the integer variable *nPrec*. Consider the following sample code:

```
        char szFmt[80],szValue[80];
        sprintf(szFmt,"%%%d.%dlf",nLen,nPrec);
        // e.g., if nLen were 10, nPrec were 2, this would create the string
"%10.2lf"
        // (the effect of %% is to output a single '%')
        sprintf(szValue,szFmt,dVal);
        // Same as writing sprintf(szValue,"%10.2lf",dVal), in above example
```

The field update member function descriptions are presented in Table 22.10.

CDB3File Function Descriptions: Group 3

void Update() [6 lines]

- Purpose: To take the current record buffer and write it to the file, updating the file header so it reflects the current date.
- If the record position is valid, calls the Write() member to write the buffer to the file.
- Gets the date, using the <time.h> functions (see example above), and updates the field set to reflect the current date.
- Saves the header (field set Save() member), using the header-only option.

bool ReplaceField(const CField *fld,const char * szVal) [15 lines*]

- Purpose: Takes the string szVal and places it in the appropriate position in the record buffer (as determined by pFld), truncating or padding with blanks (as necessary) to ensure it fits exactly. It then updates the file. The function returns false if the record position is invalid or if the pFld passed is 0, true otherwise.
- Checks for valid record position and field pointer, returning false if an error is detected.
- Takes szVal and places it in a temporary string.
- If the length of szVal is greater than the field length, it truncates the temporary string to the proper length (Hint: use the Left() member)
- If the length of szVal is less than the field length, it adds trailing blanks. (Hint: This can be done by creating a temporary character buffer, filling it with the required number of blanks, then concatenating it to the temporary string).
- Replaces the old field value with the temporary string. (Hint: This can easily be done by using the Delete() and Insert() string members)
- Calls Update() member to write the record to disk.
- Returns true.
- Note: This function will need to be modified when sorting and filters are added

Table 22.10: CDB3File Function Definitions (Group 3)

22.3.4: Table and Record Operations

The final group of functions related to the table as a whole or a particular record. In implementing these functions, it is particularly important to note the following distinction:

- The Delete() function marks a record for deletion, but does not actually remove the record.
- The Pack() function involves rebuilding the database after removing all deleted records. As is often the case for high risk activities (i.e., activities that could lead to file corruption in the event the disk failed in the middle), it makes sense to build a temporary file containing the records we want. Then, upon completion of the temporary file, we can delete the original file and rename the temporary file. Two <direct.h> functions that are useful in this regard are:

remove(const char * szName)//
deletes a file (which cannot be open), and
rename(const char * szFrom,const char * szTo)
// changes the name of szFrom to szTo.

The table and record member function descriptions are presented in Table 22.11.

CDB3File Function Descriptions: Group 4
bool Create(const char * szName,const CFieldSet &set) [5 lines] • Purpose: Creates a file named szName, consisting of a copy of database header contained in the set argument, with no records. Returns true if successful. This member will be useful in creating a temporary file during packing operations. • Calls the base class Open member to open the szName file (binary,write or create). Returns false if the operation fails. • Copies the set argument into its m_setFields member. (Hint: this can easily be done with the field set Copy() member). • Saves the field set member to the open file. • Returns true
bool SetField(const char * szName,const char * szValue) [3 lines] • Purpose: Set the value of a character field, returning true if successful, false otherwise. • Checks for appropriate field type, returning false if not. • Gets a pointer to the associated field structure, then returns the value of a call to ReplaceField(), passing the field pointer and szValue.
bool SetField(const char * szName,bool bValue) [4 lines] • Purpose: Set the value of a logical field, returning true if successful, false otherwise. • Checks for appropriate field type, returning false if not. • Creates a temporary string, holding the value "T" or "F" (as determined by bValue) (Hint: you can do this in one line using the ternary operator). • Gets a pointer to the associated field structure, then returns the value of a call to ReplaceField(), passing the field pointer and the temporary string.
bool SetField(const char * szName,double dValue) [7 lines] • Purpose: Set the value of a numeric field, returning true if successful, false otherwise. • Checks for appropriate field type, returning false if not. • Creates a temporary string, holding the properly formatted value of the dValue number (Hint: you can do this with a series of Format() calls, as discussed earlier). • Gets a pointer to the associated field structure, then returns the value of a call to ReplaceField(), passing the field pointer and the temporary string.
bool SetField(const char * szName,long lValue) [5 lines] • Purpose: Set the value of a date field, returning true if successful, false otherwise. • Checks for appropriate field type, returning false if not. • Creates a temporary string, holding the properly formatted value of the lValue number (Hint: you can do this with a Format() call). • Gets a pointer to the associated field structure, then returns the value of a call to ReplaceField(), passing the field pointer and the temporary string.

bool AppendCopy(const char * szBuf=0) [5 lines*]

- Purpose: Appends a copy of the current record (if szBuf==0) or a specified buffer (if szBuf!=0) to the end of the file. Returns true if successful, false otherwise.
- Sets the length of the database to the current length plus 1.
- If a valid pointer has been passed in, it assigns the current record buffer to that pointer.
- Sets the position to the last record (i.e., the one just added).
- Calls Update() to write the buffer to the file and update the header.
- Returns true.
- Note: This function will need to be modified when sorting and filters are added

bool AppendBlank() [4 lines]

- Purpose: Adds a blank record (filled with spaces) record to the end of the file. Returns true if successful.
- Creates a temporary character buffer filled with a number of spaces equal to the field length.
- Returns the value of a call to AppendCopy(), passed the buffer as an argument.

bool IsDeleted() [2 lines]

- Purpose: Returns true if the current record is marked for deletion, false otherwise.
- If the record buffer is empty, or has a deleted byte containing '*', returns true. Otherwise, returns false.

bool Delete(long nOffset=-1) [5 lines*]

- Purpose: Marks the specified record (lOff>0) or the current record (lOff<1) for deletion, returning true if successful, false otherwise.
- If a valid (>0) offset is supplied, calls Seek() to load the appropriate record. Returns false if the seek fails.
- If the record buffer is empty, or the deleted byte is already set, returns false.
- Sets the deleted byte (character 0) of the record buffer to '*', then calls Update() to save it.
- Returns true.
- Note: This function will need to be modified when sorting and filters are added

bool Recall(long nOffset=-1) [5 lines*]

- Purpose: Removes the deleted byte from the specified record (lOff>0) or the current record (lOff<1), returning true if successful, false otherwise.
- If a valid (>0) offset is supplied, calls Seek() to load the appropriate record. Returns false if the seek fails.
- If the record buffer is empty, or the deleted byte is already off, returns false.
- Sets the deleted byte (character 0) of the record buffer to a space (ASCII 32), then calls Update() to save it.
- Returns true.
- Note: This function will need to be modified when sorting and filters are added

bool Pack() [12 lines*]

- Purpose: Regenerates the file, removing all deleted records. Returns true if successful, false otherwise.
- Declares a temporary CDB3File object and calls its Create() member function, using a name such as “temp.dbf”, and passing it a copy of the current field set. This creates an empty file with the same structure as the original on disk.
- Iterates through all the records in the current file, loading them and using AppendCopy() to copy all non-deleted records to the temporary file.
- Saves the current file name in a temporary string, with a call to the GetFilePath()
- Closes the current file
- Closes the temporary file
- Deletes the current file (using the name saved in the temporary string). Use the <stdio.h> function remove() to accomplish this.
- Renames the temporary file (.e.g, temp.dbf) to the original file name. Use the <stdio.h> function rename() to accomplish this.
- Opens the original file name, returning the value returned by Open
- Note: This function will need to be modified when sorting and filters are added

virtual void DisplayAll(ostream &out,const vector<string> &arExpr) [6 lines]

- Purpose: Displays all non-deleted records in the file. If the arExpr argument is empty, it displays all fields–otherwise it displays only the fields in that array.
- Starts at the beginning of the file and does a Seek() on each record. If the record is not deleted, it calls Display() on that record.

virtual void DisplayAll(ostream &out) [2 lines]

- Purpose: Displays all non-deleted records in the file. It displays all fields for each record.
- Creates a temporary string array (with no elements), then call the two argument version of DisplayAll(), passing it the temporary array as its second argument.

string GetFields(long nPos,const vector<string> &arList,char cSep=0) [23 lines]

- Purpose: Used to create a string consisting of the values of specified fields (arList) or all fields (if arList is empty) for a record in a given file position (nPos). In the event that nPos or arList contains illegal values, it returns an empty string. The cSep argument allows a character (e.g., a comma, bar or whatever) to be used to separate the fields in the resultant string. e.g., if arList is a three element array containing "ZIPCODE", "LNAME" and "FNAME", it might produce:

  "33486Gill   Grandon   " with no separator, or

  "33486|Gill |Grandon   " if '|' was specified as the separator
- Sets a temporary boolean value (e.g., bAll) to true if arList has no elements, false otherwise.
- Creates a temporary string to hold the expression
- Does a Seek() to the specified position (nPos), returning the empty temporary string if the seek fails or the record is marked as deleted.
- Creates a temporary string to hold the separator (cSep), if any is supplied (i.e., if cSep does not equal 0).
- If bAll is false, does loop through all the elements of arList, getting the value for each field and appending it (plus the separator, if required) to the temporary string. If any non-field is present in arList, it empties the temporary string and returns it.
- If bAll is true, iterates through all the fields in the definition, appending the value of each one (plus the separator, if specified) to the temporary string.
- Returns the value of the temporary string.

virtual void List(ostream &out,const vector<string> &arExpr,char cSep=0) [6 lines]

- Purpose: Lists all records (one line per record) in the database. The values displayed for each record is controlled by the field names in arExpr (or all fields, if arExpr is empty). The cSep argument determines what, if any, separator appears between fields in the listing. (See GetFields() for explanation of the separator).
- It starts at the beginning of the file and does a Seek() on each record. If the record is not deleted, it calls GetFields() on that record, sending the resultant string to out.
- Hint: This function is nearly identical to DisplayAll(), except it calls GetFields() instead of Display().

virtual void List(ostream &out,char cSep=0) [2 lines]

- Purpose: Lists all records (one line per record) in the database. All field values are displayed for each record. The cSep argument determines what, if any, separator appears between fields in the listing. (See GetFields() for explanation of the separator.
- Creates an empty temporary string array, then calls the two argument version of List(), supplying the array as the second argument.

Table 22.11: CDB3File Function Descriptions (Group 4)

22.4: CDB3Sorted Class

In this section, we implement sorting and selection. We do this by adding an m_index member that contains all active records (i.e., records that meet our section criteria and have not been deleted) sorted according to a key that we have specified.

Rather than modifying the CDB3File class, it makes sense to do this by creating a new class, CDB3Sorted. There are a number of justifications for this approach:

- It would allow subsequent users to decide whether or not they want to use the indexing functionality. If not, they create a CDB3File object, if so, a CDB3Sorted object.
- Nearly all the members that are overridden call the base class members as part of their implementation (e.g., within CDB3Sorted::DisplayAll() there is a call to CDB3File::DisplayAll() function). This makes the code for both the base and sorted members simpler. (In fact, when I originally modified the base class, the many of the functions were nearly twice as long).
- It provides a good example of the type of situation where you would use inheritance to modify an existing class.

The class is implemented in three stages:

1. We implement the index is implemented purely for sorting
2. We take an existing class, the LogicCalc class (developed in Chapter 17, Section 17.3 lab exercise), and inherit from it to create the DBCalc class which supports validation and evaluation of selection criteria (a.k.a. filter expressions)
3. We compose the filter expression (m_filter) and the evaluation class (m_parser) into the CDB3Sorted class and modify the members (many of which were already modified in the first task) to support filtering.

Although the filtering capability could have added to a new class inheriting from CDB3Sorted (e.g., CDB3SortedAndFiltered), there did not seem to be much practical benefit of doing so. Since we use the index to control both ordering and selection activities, the resulting functions are more compact if both activities are performed together.

22.4.1: Sorting Records

In this section we will be focusing on implementing sorting. The CDB3Sorted class, however, will ultimately implement both sorting and filters (i.e., searching for a collection of records). We will therefore continue our modification of the class in Section 22.4.3.

In order to make much use of a database, we need to be able to change the order in which each table is presented, and to select specific sets of records. In the "real world", we'd probably do this by creating some tree-oriented index that could be stored on disk. Rather than go to that extreme, however, we will content ourselves with creating a **multimap<>** (which is based on a tree shape) that relates IString keys to long record numbers, i.e.:

```
multimap<IString,long> m_index;
```

This is not a serious a concession to reality as it might first seem. Since we'll always access the index using encapsulated member functions, were we to decide to change to a disk-based index, the basic structure of our class would remain nearly identical. We'd just have to change the implementation of the functions that actually access the index.

We will also need an array to identify the field or fields determining the sort order. This can be done with a string array:

```
vector<string> m_arKey;
```

which will contain the field names. For example, if m_arKey contained two elements, with m_arKey[0]==”ZIPCODE” and m_arKey[1]==”LNAME”, the m_index collection would list elements sorted by zip code, and by last name (within each zip code).

The multimap<> template you will be using is very similar to the map<> template that was introduced in Chapter 14. The major difference is that multiple values with the same key can be held in the collection. This is critical, since many possible sorts (e.g., sorting by STATE) will lead to indexes with keys that are the same across many records.

The CDB3Sorted class declaration is presented in Example 22.5.

Example 22.5: CDB3Sorted Class Declaration

```
class CDB3Sorted : public CDB3File
{
public:
      CDB3Sorted();
      virtual ~CDB3Sorted();

      virtual void Close();
      virtual void DisplayAll(ostream &out,const vector<string>
&arExpr);
      virtual void DisplayAll(ostream &out);
      virtual void List(ostream &out,const vector<string> &arExpr,
                              char cSep=0);
      virtual void List(ostream &out,char cSep=0);
      void SetKey(const vector<string> &arKey);
      string GetKey(long nPos);
      void InsertKey(long nPos);
      void RemoveKey(long nPos);
      long Find(const char * szKey);
      virtual bool Next();
      virtual bool Previous();
   void Sort();
      virtual bool AppendCopy(const char * szBuf=0);
      virtual bool Delete(long nOff=-1);
      virtual bool Recall(long nOff=-1);
      virtual bool Pack();

protected:
      virtual bool ReplaceField(const CField *fld,const char * szVal);
      multimap<IString,long> m_index;
      vector<string> m_arKey;
// Filter-related
      bool IsFiltered() const;
      void RemoveFilter();
      bool ReplaceFilter(const char *szFilter);
      bool ReplaceFilter(const vector<Token> &szInput,size_t nStart);
      vector<Token> m_filter;
      DBCalc m_parser;
};
```

The sorting-related functions associated with the CDB3Sorted class are described in Table 22.12.

CDB3Sorted Function Descriptions: Sorting-Related

CDB3Sorted(), ~CDB3Sorted() default constructors and destructors are adequate

virtual void Close() [4 lines]

- Purpose: To close the file and remove all data structures that could interfere with opening a new file.
- Call the base class version of close
- Clears the filter (discussed in Section 22.5), the contents of the key array (m_arKey) and the index (m_index).

void DisplayAll(ostream &out,const vector<string> &arExpr) [10 lines]

- Purpose: Display records, ordered according to m_index, in a manner equivalent to the prevously defined Display() functions. If the arExpr argument is empty, it displays all fields–otherwise it displays only the fields in that array. If m_index is empty, it displays non-deleted records in the order in which the appear in the file.
- If the sort array contains elements, it iterates through the sort array, doing a seek on each element and displaying the loaded record with a call to Display().
- If the sort array is empty, calls the base class version of DisplayAll().

void DisplayAll(ostream &out) [2 lines]

- Purpose: Display records, ordered according to m_index, in a manner equivalent to the previously defined Display() functions. It displays all fields for each record.
- Creates a temporary string array (with no elements), then calls the two argument version of DisplayAll(), passing it the temporary array as its second argument.

void List(ostream &out,const vector<string> &arExpr,char cSep=0) [10 lines]

- Purpose: Lists each record (one line per record) in the database, using the m_index map to determine the order in which they appear. The values displayed for each record is controlled by the field names in arExpr (or all fields, if arExpr is empty). The cSep argument determines what, if any, separator appears between fields in the listing. (See GetFields() for explanation of the separator.
- If the sort array contains elements, it iterates through the index map, doing a seek on each element and displaying the loaded record with a call to GetFields(), which is then sent to out.
- If the sort array is empty, it starts at the beginning of the file and does a Seek() on each record. If the record is not deleted, it calls GetFields() on that record, sending the resultant string to out.

(*Hint*: This function is nearly identical to DisplayAll(), except it calls GetFields() instead of the Display() function.)

void List(ostream &out,char cSep=0) [2 lines]

- Purpose: Lists each record (one line per record) in the database, using the m_index map to determine the order in which they appear. All field values are displayed for each record. The cSep argument determines what, if any, separator appears between fields in the listing. (See GetFields() for explanation of the separator.
- Creates an empty temporary string array, then calls the two argument version of List(), supplying the array as the second argument.

void SetKey(const vector<string> &arKey) [1 line]

- Purpose: Sets the value of the key definition array for the file.
- Sets the value of m_arKey to its argument, using the vector<string>::Copy() member.

string GetKey(long nPos) [1 line]

- Purpose: Returns the value of the sort key for a specified record (nPos).
- Calls GetFields(), passing it the sort key (m_arKey) as its second argument.

long Find(const char * szKey) [5 lines]

- Purpose: Identifies a location in the file (by record number) where a particular key occurs. If no record with that key is present, returns -1.
- Calls FindPosition() to search m_index for a particular key, placing the return value in a temporary integer.
- Does a Seek() on the m_index position identified by the return value. If the seek is successful, and the key of the record matches the szKey argument, the location of the record is returned.
- If the temporary return value integer is not within m_index, or if the corresponding record does not match the key, it returns -1.

bool Skip() [10 lines]

- Purpose: Moves to the next record in a table. If an index is present, it moves to the next record in the index. Returns false if it is at the end of the table or index.
- For non-indexed tables, calls Seek(m_nRecordPosition+1) if not at the end of the table.
- For indexed table:
    - Finds the active record in the index
    - Uses the iterator to move to the next index element
    - If not at the end of the index, calls Seek() using the value in the iterator (->second member), otherwise, returns false.

bool Previous() [8 lines]

- Purpose: Moves to the previous record in a table. If an index is present, it moves to the previous record in the index. Returns false if it is at the beginning of the table or index.
- For non-indexed tables, calls Seek(m_nRecordPosition-1) if not at the beginning of the file.
- For indexed table:
    - Finds the active record in the index
    - If the current record is not the first element in the index, uses the iterator to move to the previous index element. Otherwise, returns false.
    - Calls Seek() using the value in the iterator (->second member).

void InsertKey(long nPos) [7 lines]

- Purpose: Adds a reference to the record at position nPos to the m_index map.
- Performs a Seek() of position nPos to load the key. If the seek fails, or if the record is deleted, or if the record does not meet the active filter, the function returns.
- Adds the key-position association to the m_index map.

void RemoveKey(long nPos) [11 lines]

- Purpose: Removes a reference to a particular record (nPos) from the m_index.array.
- If the key array is empty or a Seek() at nPos fails, returns.
- Places the key at nPos into a temporary string.
- Iterates though m_index, looking for a record at nPos. When nPos is found, it removes the reference from m_index (using the erase() member), then returns.
- If the referenced record is not found (i.e., the key changes in the index prior to a matching nPos being found), it returns having done nothing.

void Sort() [8 lines]

- Purpose: Generates the m_index array for a given key array (m_arKey).
- If the key array is empty, returns.
- Empties the m_index array.
- Iterates through the file, adding all non-deleted records that meet the active filter to m_index.
- Calls QuickSort() on the start and end of m_index to sort it.

virtual bool ReplaceField(const CField *fld,const char * szVal) [5 lines]

- Purpose: Modifies the base class ReplaceField() so that any time a field value is changed, the index is updated.
- If an illegal field is passed in, or there is no active record, it returns false.
- RemoveKey() is called on the current record number to remove the active record from the index (if it is present).
- The base class version of ReplaceField() is called, saving the return value in a temporary variable.
- InsertKey() is called on the current record number, to replace the record in the index.
- The temporary variable is returned.
- Design Note: In our CDB3File design, we ensured that all modifications to field values were ultimately accomplished in ReplaceField(). As a result, we do not need to modify any of the SetField() functions. This is why it is good to design members such that key actions only take place within a single member.

virtual bool AppendCopy(const char * szBuf=0) [3 lines]

- Purpose: Modifies the base class AppendCopy() to ensure index array is updated.
- Calls base class AppendCopy(), holding value in a temporary variable.
- If temporary variable is true, calls InsertKey() to update index
- Returns value of temporary variable
- Design Note: AppendBlank() does not need to be overridden because it performs its action by calling AppendCopy(). [See design comment for ReplaceField()]

virtual bool Delete(long nOff=_1) [3 lines]

- Purpose: Modifies the base class Delete() to ensure index array is updated.
- Calls base class Delete(), holding value in a temporary variable.
- If temporary variable is true, calls RemoveKey() to update index
- Returns value of temporary variable

virtual bool Recall(long nOff=_1) [3 lines]

- Purpose: Modifies the base class Recall() to ensure index array is updated.
- Calls base class Recall(), holding value in a temporary variable.
- If temporary variable is true, calls InsertKey() to update index
- Returns value of temporary variable

virtual bool Pack() [14 lines]

- Purpose: Modifies the base class Pack() to ensure index array is regenerated after deleted records have been removed from the file.
- Makes a copy of the key array in a temporary string array.
- Puts the address of the active filter in a temporary pointer, then sets the m_pFilter variable to 0 (Note: this is critical because otherwise the filter will be deleted when the file closed during the packing process).
- Removes all the elements of the index array (m_index).
- Calls the base class version of Pack(). If successful, sets the filter and the index array, then calls Sort() to regenerate the index array.
- Returns the value previously returned by the base class version of Pack().

Table 22.12: CDB3Sorted Function Descriptions (Sorting-related)

After you have debugged and tested your sorting routines, you can turn to the more complex problem of selecting a set of records.

22.4.2: The DBCalc Class

Because we are limiting our ordering expression to vector<string> collections of field names, they are relatively easy to verify. A filter expression, which is a logical selection criteria applied to each record, such as:

```
LNAME>"M" && AMOUNT<6.00
```

is much harder to deal with. Fortunately, we have already done most of the legwork (in Chapter 17, Section 17.3) when we implemented the calculator. (Note: if you have not implemented the first part of that lab exercise, the LogicCalc class, you need to so prior to proceeding).

The LogicCalc class that we created was custom-tailored for filter purposes, since it supports the following capabilities:

- Evaluation of Boolean expressions
- Support for 4 data types: numeric, string, date and logical
- Even the data type codes, tString=='C', tDate=='D', tNumber='N' and tBoolean='L' happen to match the codes used by dBase for its field types.

It is also reasonable to suppose that these capabilities are not entirely coincidental. Unfortunately, the LogicCalc has one significant drawback: it uses variables names established by assignment, and not field values. As a result, we need to create a new class, the DBCalc class, which allows us to acquire our values from a table. The class declaration for DBCalc is presented in Example 22.6.

Example 22.6: DBCalc Class Declaration

```
class DBCalc :    public LogicCalc
{
public:
      DBCalc(void);
      virtual ~DBCalc(void);
      bool InitValue(Token &tok,const CDB3File &db) const;
      bool InitType(Token &tok,const CDB3File &db) const;
      bool TokenizeString(const char *buf,vector<Token> &arDest,
            const CDB3File &db,bool bTypeOnly=false) const;
      void UpdateValues(vector<Token> &arDest,const CDB3File &db)
const;
};
```

In the declaration, we have overridden two functions—InitValue() and TokenizeString()—which happen to be the two functions where variable values are established (way back in the Variables class, presented in Chapter 14, Section 14.4). We have added an additional argument to each of these functions—a CDB3File object reference. Using that reference, we can gain access to field names and field values that we can use to establish the variable values and types we'll need to evaluate our filter expressions.

In addition, we have two more functions designed to enhance efficiency. InitType() is just like InitValues() except it doesn't worry about setting token values, only token types. This is useful because it allows us to validate a filter expression even if no data is present (since LogicCalc::IsBoolExpression() only needs type information, not actual values). Similarly, UpdateValues() takes a filter expression and acquires the field values for the current record, without validating it. After all, once we know an expression is valid (syntax-wise) for one record in a table, we know it is going to be valid for all records, since field types don't change from record to record.

DBCalc function descriptions are presented Table 22.13.

DBCalc Function Descriptions
DBCalc(void), virtual ~DBCalc(void) Default constructor and destructor are adequate.
bool InitValue(Token &tok,const CDB3File &db) const [10 lines] • Purpose: Establishes a token's value, returning false if the value cannot be established. • Checks to see if the token is a field name in the db table. If so, it establishes the value and type from the field type and field value, with its source set to tVariable (you could alternatively add a tField source to the Token enumeration, if you were so inclined) • If not a field name, uses InitFromData() to establish type and value from literal tokens. • Otherwise, it returns false.
bool InitType(Token &tok,const CDB3File &db) const [7 lines] • Purpose: Establishes a token's type (without bothering to establish its value), returning false if the value cannot be established. • Checks to see if the token is a field name in the db table. If so, it establishes the type from the field type, with its source set to tVariable • If not a field name, uses InitFromData() to establish type (and value) from literal tokens. • Otherwise, it returns false.
bool TokenizeString(const char *buf,vector<Token> &arDest,const CDB3File &db, bool bTypeOnly=false) const [10 lines] • Purpose: To tokenize a buffer and establish token types and, if required, values • Nearly identical to Variables::ToeknizeString() except that it calls the DBCalc::InitValue() function (if bTypeOnly is false) • If bTypeOnly is true, calls the DBCalc::InitType() function instead of InitValue() • Returns true unless tokenization fails.
void UpdateValues(vector<Token> &arDest,const CDB3File &db) const [5 lines] • Purpose: to refresh the token values in arDest with updated field values from the table. • Iterates through arDest, acquiring the field value (from db) for any token of type tVariable

Table 22.13: DBCalc Function Descriptions

After testing your DBCalc class, you can integrate it into the CDB3Sorted class, which is described next.

22.4.3: Filtering Records

A filter is an expression used by database applications—such as dBase, MS FoxPro and MS Access—to hide records not meeting specified criteria. Once a filter is in place, activities like display/listing records and iterating through a table (e.g., using the Next() and Previous() functions) should act as if the records aren't there. Other actions, such as jumping to a record position, should not be affected by a filter.

In a certain sense, our m_index map, which controls sort order, already acts like a filter. What we therefore need to do is use our filter expression to remove records not meeting the filter. Doing so allows us to use the same record navigation functions for tables that are filtered, sorted or both.

The CDB3Sorted member functions that must be specifically implemented for filtering are listed in Table 22.14.

CDB3Sorted Functions Descriptions : Filter Functions
bool IsFiltered() const [2 lines] • Purpose: To determine if the active record passes the filter criteria. Returns true if it does, false otherwise. • If the filter is not set (m_pFilter==0), returns true. • Returns the result of the CFilterExpression::Evaluate() member otherwise.
void RemoveFilter() [2 lines] • Purpose: Removes the active filter. • Clears the m_filter member. • Calls Sort() to regenerate the index.
void ReplaceFilter(const char *pFilter) [3 lines] • Purpose: Replaces the active filter with another filter (passed in as a NUL-terminated string argument) • Uses m_parser to tokenize the argument, then calls the second version of ReplaceFilter() using that array as an argument. • Returns true unless the tokenizing fails or call to ReplaceFilter() fails.
bool ReplaceFilter(const vector<Token> &szInput,size_t nStart) [5 lines] • Purpose: Replaces the active filter with another filter (passed in as a vector<Token> argument identifying the start of the filter expression as nStart) • Calls m_parser.IsBoolExpression() to verify the filter is valid • Assigns the szInput vector to the m_filter member. • Calls Sort() to regenerate the index map (m_index).

Table 22.14: CDB3Sorted Function Descriptions (Filter Functions)

22.5: CDB3Command class

In this final section, you will create CDB3Command class, providing an interpreter that allows you to invoke the key functions provided by the CDB3Sorted and CDB3File classes from the command line. We choose to create this class using inheritance from CDB3Sorted. Unlike our earlier example, however, this class doesn't require us to override nearly any base class members. The reason: it adds capability (i.e., the ability to interpret command lines) but doesn't really change any of the behaviors of the base class it inherits from.

The CDB3Command class declaration is presented in Example 22.7.

Example 22.7: CDB3Command Class Declaration

```
class CDB3Command : public CDB3Sorted
{
public:
      CDB3Command();
      virtual ~CDB3Command();

      bool Execute(const char * szInputLine,IString &szCommand,
                  ostream &out);
      void CommandLoop(istream &in,ostream &out);
      bool ExecuteGoto(const char * szName);
      bool ExecuteRun(const vector<Token> &arToks,ostream &out);
      bool ExecuteDisplay(const vector<Token> &arToks,ostream &out);
      bool ExecuteList(const vector<Token> &arToks,ostream &out);
      bool SetOrder(const vector<Token> &arToks);
      bool SetValue(const vector<Token> &arToks);
      bool SetFilter(const vector<Token> &arToks);
      bool ValidateFieldList(vector<string> &arFields,
                                    const vector<Token> &arToks,size_t
nStart=0);

};
```

Member function descriptions for the class are presented in Table 22.15.

CDB3Command Function Descriptions
Default versions of CDB3Command() and ~CDB3Command() suffice. This is not surprising as the class has no data members.
bool Execute(const char * szInputLine,IString &szCommand,ostream &out) [31 lines] • Purpose: Takes an input string (szInputLine), that may come from the user or a file, breaks it into tokens, then executes it by calling the appropriate function depending upon what command was given. The command (e.g., OPEN, SET, LOAD, SKIP, etc.) is placed in szCommand, and the function returns true or false depending upon whether or not it was successful. • Creates a temporary string array to hold the tokens, then tokenizes the input string (using CParser::Tokenize()), returning false if the string cannot be tokenized or is empty. • Places the first token (which holds the command) in szCommand. • In a large if statement, it compares the command string to the various legal commands (see table at the beginning of the project), performing the action directly if it is simple (e.g., if the command is “QUIT”, it returns true; if the command is “OPEN”, it calls the Open() function passing it the second element of the string array–position 1–as its argument) or by calling a function. In each case, the value returned is an indication of the success or failure of the function.

void CommandLoop(istream &in,ostream &out) [7 lines]

- Purpose: Keeps calling Execute() on each line typed in by the user (or from a file, depending on how the value of istream is established), returning when the user types a line beginning with “QUIT”.
- Sets up a temporary string to hold each command.
- Enters an infinite loop that: a) prints a prompt to out (e.g., ->), b) reads a line from in, c) calls Execute(), passing it the input string, the temporary string used to hold commands, and the output stream (Note: if Execute() returns false, it sends an “Illegal line” message to out), d) Checks if the command string set by Execute() is “QUIT”, if so, it breaks the loop.

bool ExecuteGoto(const char * szName) [5 lines]

- Purpose: Issues a Seek() command to move to the position in the file specified by szName, which may be “TOP”, “BOTTOM” or a record number.
- Checks the value of szName for TOP, BOTTOM or a record number.
- If a record number is passed, it returns the result of the appropriate Seek() function.
- If TOP or BOTTOM is passed, location depends on whther or not an index is in place.
  - o If no index is present, goes to record 1 or to last record.
  - o If an index is present, uses an iterator to move to the beginning or end of the index

bool ExecuteDisplay(const vector<string> &arToks,ostream &out) [19 lines]

- Purpose: The function is passed a tokenized command line, that should have “DISPLAY” as its initial element. It verifies that the syntax is legal (see table at the start of the project), then calls the appropriate Display() or DisplayAll() function.

bool ExecuteList(const vector<string> &arToks,ostream &out) [12 lines]

- Purpose: The function is passed a tokenized command line, that should have “LIST” as its initial element. It verifies that the syntax is legal (see table at the start of the project), then calls the appropriate List() function.

bool ExecuteRun(const vector<string> &arToks,ostream &out) [20 lines]

- Purpose: The function is passed a tokenized command line, that should have “RUN” as its initial element and a command file (a text file containing commands) as its next element. It verifies that the syntax is legal (see table at the start of the project), then enters a loop that reads each line from the command file and executes it.
- Verifies that the initial element is “RUN”, then opens the next element as a ifstream object. If the open fails, it returns false.
- If a third token is present, opens that file using a temporary output file stream object. Otherwise, it assigns the value of out to the same temporary output file stream object that was passed into the CommandLoop() function (Hint: you could pass in cout, if you are just testing it by itself).
- Enters a command loop (similar to that in CommandLoop()) that reads each line from the command file, echoes it to the output stream with a prompt (e.g., *->), and executes it, ending when there are no more lines or a “QUIT” command is encountered.

bool SetOrder(const vector<string> &arToks) [17 lines]

- Purpose: The function is passed a tokenized command line, that should have “SET” and “ORDER” as its initial two elements. It verifies that the syntax is legal (see table at the start of the project), then issues the appropriate command to remove the index (if “OFF” is the third element) or set the index (if “TO” is the third element, followed by a comma-separated field list). It returns true if successful, false otherwise.
- Verifies first three elements of arToks are “SET”, “ORDER” and “TO” or “OFF”.
- If the third element is “OFF”, removes the key array and calls Sort().
- If the third is “TO”, creates a field list using the ValidateFieldList() function starting with position 3 to ensure the remaining elements constitute a valid field list, sets the key array and sorts the index.
- Returns true unless the syntax is invalid or the index cannot be set.

bool SetFilter(const vector<string> &arToks) [10 lines]

- Purpose: The function is passed a tokenized command line, that should have “SET” and “FILTER” as its initial two elements. It verifies that the syntax is legal (see table at the start of the project), then issues the appropriate command to remove the index (if “OFF” is the third element) or set the index (if “TO” is the third element, followed by a legal filter expression). It returns true if successful, false otherwise.
- Verifies first three elements of arToks are “SET”, “FILTER” and “TO” or “OFF”.
- If the third element is “OFF”, removes the active filter then calls Sort().
- If the third is “TO”, calls ReplaceFilter() to establish the new filter.
- Returns true unless the syntax is invalid or the filter cannot be set.

bool SetValue(const vector<string> &arToks) [39 lines]

- Purpose: The function is passed a tokenized command line, that should have “SET” as its initial argument, a valid field name as its next argument, “TO” as its third argument, and a field name or literal as its final argument. It verifies that the syntax is legal (see table at the start of the project), then issues the appropriate command to set the value of the field. It returns true if successful, false otherwise.
- Verifies the array is the proper size (4 elements), and that elements 0 and 2 are “SET” and “TO” respectively. Returns false if the syntax is not valid.
- Verifies that element 1 is a field, and that element 3 is a token of the same type. Returns false if the field type and token types don’t match.
- Using a case statement, calls the appropriate SetValue() member, depending on the field type. The data functions CVariables::Date(),CVariables::Number() and CVariables::Bool() can be useful in getting the data from the element 3 token.

bool ValidateFieldList(vector<string> &arFields,const vector<Token> arToks,
size_t nStart=0) const [10 lines]

- Purpose: takes a token vector and verifies that it is a legal field list (valid field names separated by comma characters) returning true if it is, false otherwise. Useful in setting sort order and in display/list operations.
- Copies the field's names (only) into the string vector first argument.

Table 22.15: CDB3Command Function Descriptions

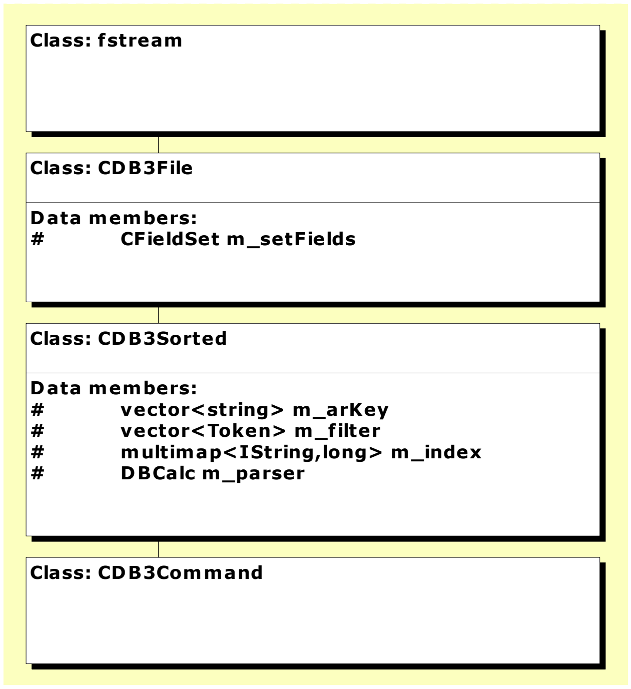
22.6: Review and Questions

22.6.1: Review

The core of the dBase interpreter project is a series of inherited classes, each of which add functionality to the project. These classes are:

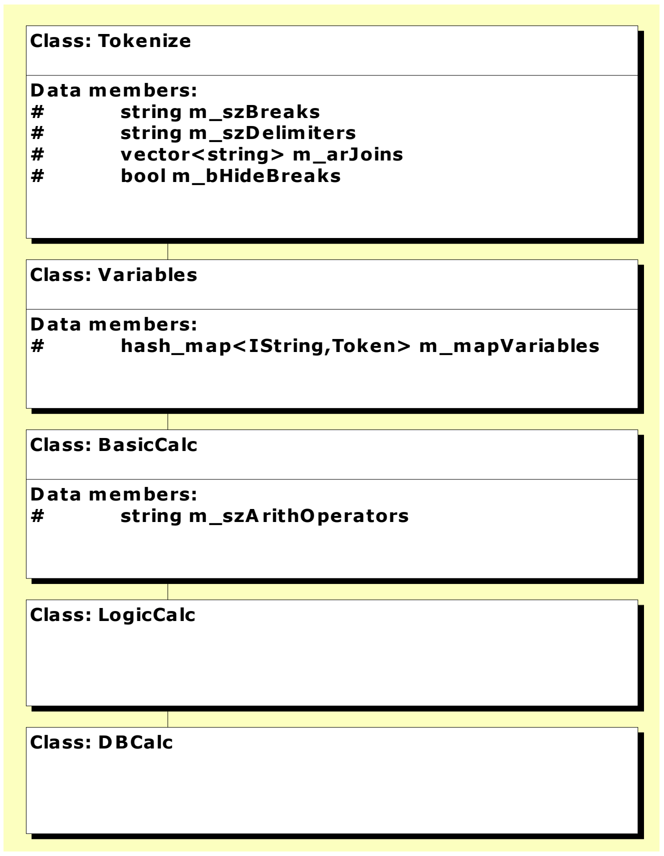
- **CDB3File:** Inheriting from the STL fstream class, this class encapsulates the basic operations for viewing and editing data in a dBase III formatted table file. Within it is composed a CFieldSet object, encapsulating the header and field definitions for the file. It is developed in Section 22.3.
- **CDB3Sorted:** Inheriting from CDB3File, the class implements sorting and filtering of records in the table file. The m_arKey vector<string> data member holds a list of field names that determines the ordering in the m_index multimap object (relating key values to record position). The vector<Token> m_filter member holds an expression that uses field names and evaluates to **true** or **false** for each field name. Only records for which the filter is **true** appear when all records are displayed or listed. It is developed in Section 22.4.
- **CDB3Command:** Inheriting from CDB3Sorted, the class implements an interpreter that provides command line access to the member functions of the CDB3Sorted and CDB3File classes. It is developed in Section 22.5.

The main class inheritance network is illustrated below.



In addition to the main classes, a particularly important set of classes inherit from walkthroughs and labs provided in previous chapters. Specifically:

- **Tokenize**: The Tokenize class, originally developed in Chapter 8, Section 8.4, breaks input strings into token strings and can be customized with respect to break characters, string delimiters and joined breaks (e.g., <= or &&).
- **Variables**: Inheriting from Tokenize, the class was originally developed in Chapter 14, Section 14.4, and provides an associative map between variables and values, using an STL hash_map<> template class. The class also provides token typing based on the appearance of literal tokens, such as "2.34" (number), '9/6/03' (date) and "Hello World" (string).
- **BasicCalc**: Inheriting from Variables, the class was originally developed in Chapter 17, Section 17.3.1, where it was used to evaluate arithmetic expressions in using a basic grammar.
- **LogicCalc**: Inheriting from BasicCalc, the class was presented to readers as a lab exercise in Chapter 17, Section 17.3.2. The calculator extended the basic calculator by providing the ability to evaluate Boolean expression using both logical and comparison operators. In comparison operations, 4 distinct data types (number, text, date and logic) were supported.
- **DBCalc**: Inheriting from LogicCalc, this user-constructed class is developed in Section 22.4.2 for the purposes of evaluating filter expressions. It extends the LogicCalc class by allowing variable values to be established from field values.



Other classes that are relevant to the project include:

- **IString:** A class inheriting from the STL string class that implements case-insensitive comparison and supports hashing. Originally developed in Chapter 14, Section 14.4.
- **Token:** Inheriting from IString, provides a means for associating names, values, data types and data sources into a single object.
- **CHeader:** A class developed in Section 22.2.1 that encapsulates and interprets the header information contained in the first 32 bytes of a dbf file.
- **CField:** A class developed in Section 22.2.2 that encapsulates a single 32 byte field definition contained in a block after the header in a dbf file.
- **CFieldSet:** A class, developed in Section 22.2.3, that inherits from vector<Field> to hold the complete set of field definitions from a dbf file. In addition, it is composed with a single CHeader object and a hash_map<IString,int> collection that maps field names to their position in the CFieldSet vector<> in order to achieve rapid lookup.

The interpreter grammar is summarized in Example 22.8.

Example 22.8: dBase Interpreter Grammar

```
<command-expression> ::= <single-command> | <multi-command>
<single-command>     ::= QUIT | PACK | SKIP | BACK
<multi-command>      ::= GOTO plus-integer | GOTO TOP | GOTO bottom
                     ::= OPEN text-string
                     ::= LIST | LIST <field-list>
                     ::= DELETE | DELETE plus-integer
                     ::= RECALL | RECALL plus-integer
                     ::= APPEND | APPEND COPY
                     ::= RUN text-string | RUN text-string text-string
                     ::= <display-command>
                     ::= <set-command>
<display-command>    ::= <single-display> | <multi-display>
<single-display>     ::= DISPLAY | DISPLAY <field-list>
<multi-display>      ::= DISPLAY ALL | DISPLAY ALL <field-list>
<set-command>        ::= <order-set> | <filter-set> | <field-set>
<order-set>          ::= SET ORDER TO <field-list> | SET ORDER OFF
<filter-set>         ::= SET FILTER TO <filter-expression> | SET FILTER OFF
<field-set>          ::= SET numeric-field TO number
                     ::= SET text-field TO string
                     ::= SET logic-field TO TRUE | SET logic-field TO FALSE
                     ::= SET date-field TO date
<field-list>         ::= <field> | <field> , <field-list>
<field>              ::= numeric-field | text-field | logic-field | date-field
<filter-expression>  ::= <bool-expression>
<bool-expression>    ::= ! <bool-expression>
                     ::= ( <bool-expression> )
                     ::=<bool-expression> LOperator  <bool-Expression>
                     ::=<bool-expression> EqOperator  <bool-Expression>
                     ::= <arith-expression> ROperator <arith-expression>
                     ::= <DAtom> ROperator <DAtom>
                     ::= <TAtom> ROperator  <TAtom>
                     ::= <LAtom>
<arith-expression>   ::= <item> | <item> <arith-operator> <arith-expression>
<item>               ::= <value> | + <value> | - <value>
<value>              ::= <NAtom> | ( <arith-expression> )
<NAtom>              ::= numeric-field | number
<DAtom>              ::= date-field | <delim> integer / integer/ integer <delim>
<TAtom>              ::= text-field | <delim> text-string <delim>
<LAtom>              ::= logic-field | TRUE | FALSE
 <LOperator>         ::= ||  | &&
<ROperator>          ::= <EqOperator> | > | < | >= | <=
<EqOperator>         ::= != | ==
<arith-operation>    ::= + | - | * | /
<delim>              ::= " | '
```

22.6.2: Glossary

dBase – A PC database family of products originally developed by Ashton Tate.
dbf – The file extension used for dBaseIII table files.
FoxPro – A PC database family of dBase-compatible products that originated with Foxbase, evolved into FoxPlus and eventually was incorporated into Visual Studio.
Field – A column in a database table
Filter – An expression that determines visible records in a database
Fixed Record Format – A database format whereby all rows are the same size, padded with blanks if so required.
Header – The section of a dBase file containing overall descriptions of the table (e.g., date modified, record length, record count) and field definitions
Index – An expression used to order in which records are presented in a database
Key – A term use for an expression used to identify or order records in a table
Logical Delete – Marking a record in a database as deleted, without actually removing
multimap<K,T> - An STL template collection object allowing multiple elements to be added with the same key expression (K).
Packing – The process of physically removing logically deleted records from a table
Recall – Restoring a record that has been logically deleted
Record – A row in a database table

22.6.3: Questions

22.1: Modify Structure. Explain some of the challenges that would result for implementing table structure modification within the interpreter.

22.2: Reimplemented Filter and Sort. Instead of implementing a combined filter and sort index, filtering and sorting could be implemented independently (e.g., using an STL set<> to implement the filter). What are the pros and cons of implementing the two independently?