C++ From Scratch

Table of Contents:

C++ From Scratch

Contents

2

Getting Started

In This Chapter

- How Big Is a Small Project
- Bootstrapping Your Knowledge
- Creating the Project
- Examining the Code

- Code Spelunking
- Analyzing the Code

With a fairly simple program such as Decryptix!, my first goal is to get a version up and running--and to keep it running. After it is working, I'll add features, redesigning on-the-fly as I go.

On a large project, this can be a fatally inefficient process. As features are added, the complexity of the overall project grows, and without a good design you end up with code that is hard to maintain.

With a smaller project such as Decryptix!, however, the risk is minimal. If a new feature requires a complete redesign and rewrite of the program, no problem: It only takes a couple of days to write it in the first place.

How Big Is a Small Project?

How small must a program be to design it as you go? I'd argue that any program that takes one person more than a few weeks to program ought to be subject to a more rigorous design process. Here's why: Programs that evolve organically, rather than by design, often have to be rewritten at least once. If doing so is painful, it is worth working out the design up front. If rewriting is trivial, however, nothing was lost by diving right in.

Why Teach a Process that Is Only Good for Smaller Projects?

"Why," I hear you ask, "show an example of organic design if all large projects require formal design?" The answer is fairly straightforward: There's a lot to learn in both programming and design. This book aims to teach programming; you'll find lots of books (including a few I wrote) on object-oriented analysis and design. You can't learn everything at once.

Nothing I teach in this book is inconsistent with good design; we just won't take the time to design everything up front. I don't know about you, but I'm eager to dive into some code.

Bootstrapping Your Knowledge

In a classic C++ primer, I'd start with the structure of the program, introduce statements and expressions, add variables and constants, and then turn to classes. I'd build skill upon skill, and I'd be about 600 pages into the book before you could even begin to write your Decryptix! program.

This book is different. You're going to jump right in and wallow around awhile. Not all of it will make sense at first, and I'll gloss over lots of detail only to return to it later in the book, but the essential flow of the program can be explained pretty quickly. From time to time you'll take an "Excursion" to related--

but not strictly relevant--areas of C++.

Creating the Project

This book is designed to be of use regardless of which compiler you are using or what platform (for example, Windows or Mac) you are developing for. From time to time, however, I'll demonstrate how you can accomplish a specific task in Microsoft Visual C++ 6.0. Your compiler might be somewhat different, but the principles are the same. With the knowledge that is provided here you can easily read the documentation for your compiler and make the necessary adjustments.

I begin by creating a project. On the drive on which I installed my compiler I have created a directory called Decryptix Projects. This will house all the versions of the program I will create.

First I start Visual C++ and tell it to create a new Win32 Console Application called Decryptix, as shown in Figure 2.1.

Figure 2.1 *Microsoft Visual C++ New Project.*

If your compiler offers a *wizard* (a series of dialog boxes that helps you make these decisions), choose whatever provides you with the simplest, text-based, non-windowed, ISO-standard environment. In this case, I choose **empty application**.

After creating the project, Visual C++ drops me in the Integrated Development Environment (IDE). I choose **File**, **New**, and enter a new C++ source file named Decryptix.cpp.

In other environments, for example a text editor in UNIX, I'd just open a new file and save it as Decryptix.cpp. Often, in IDEs, saving the file with the .cpp extension signals that this is C++ source code and turns on source code indentation support (and, sometimes, color-coded text!). Source code indentation support means that when you type your source code the editor automatically indents it properly for you. Thus, if you enter

```
if ( someValue > thisValue )
```

and then press Enter, the editor automatically indents the next line. (Don't worry about what this code does, it will all be explained in time.)

NOTE: To learn what support your editor provides, please check the documentation that comes with your compiler.

Examining the Code

Now take a look at a preliminary version of Decryptix, in Listing 2.1. You can open a file in your project, save it as Decryptix.cpp, and then enter this code, exactly as shown.

TIP: I strongly advise you to enter all the source code yourself because that is the best way to learn. If you simply can't stand the thought of all that typing, however, you can retrieve this code from the CD that accompanies this book, or you can download this code--and all the code for this book--from my Web site (go to `www.libertyassociates.com` and click on Books & Resources).

This program is quite advanced, and of course you won't understand much of what you are reading. Don't be intimidated, however; this chapter and Chapter 3, "Program Flow," go over it line by line. You might find, however, that you can get a pretty good idea of what the program is doing just by reading it as prose.

Try running it and examining what it does, and then try matching the code to the output.

Listing 2.1 First Glimpse of Decryptix!

```
0:  #include <iostream>
1:
2:  int main()
3:  {
4:      std::cout << "Decryptix. Copyright 1999 Liberty ";
5:      std::cout << "Associates, Inc. Version 0.2\n " << std::endl;
6:      std::cout << "There are two ways to play Decryptix: ";
7:      std::cout << " either you can guess a pattern I create, ";
8:      std::cout << "or I can guess your pattern.\n\n";
9:
10:     std::cout << "If you are guessing, I will think of a\n ";
11:     std::cout << "pattern of letters (e.g., abcde).\n\n";
12:
13:     std::cout << "On each turn, you guess the pattern and\n";
14:     std::cout << " I will tell you how many letters you \n";
15:     std::cout << "got right, and how many of the correct\n";
16:     std::cout << " letters were in the correct position.\n\n";
```

```
17:
18:     std::cout << "The goal is to decode the puzzle as quickly\n";
19:     std::cout << "as possible. You control how many letters \n";
20:     std::cout << "can be used and how many positions\n";
21:     std::cout << " (e.g., 5 possible letters in 4 positions) \n";
22:     std::cout << "as well as whether or not the pattern might\n";
23:     std::cout << " contain duplicate letters (e.g., aabcd).\n\n";
24:
25:     std::cout << "If I'm guessing, you think of a pattern \n";
26:     std::cout << "and score each of my answers.\n\n" << std::endl;
27:
28:     const int minLetters = 2;
29:     const int maxLetters = 10;
30:     const int minPositions = 3;
31:     const int maxPositions = 10;
32:
33:     int          howManyLetters = 0, howManyPositions = 0;
34:     bool          duplicatesAllowed = false;
35:     int          round = 1;
36:
37:     std::cout << "How many letters? (";
38:     std::cout << minLetters << "-" << maxLetters << "): ";
39:     std::cin >> howManyLetters;
40:
41:     std::cout << "How many positions? (";
42:     std::cout << minPositions << "-" << maxPositions << "): ";
43:     std::cin >> howManyPositions;
44:
45:     char     choice;
46:     std::cout << "Allow duplicates (y/n)? ";
47:     std::cin >> choice;
48:
49:     return 0;
50:  }
```

Compile, link, and run this program. In Visual C++ you can do all this at once by pressing **Ctrl+F5**. Here's the output:

```
Decryptix. Copyright 1999 Liberty Associates, Inc. Version 0.2
There are two ways to play Decryptix:
 either you can guess a pattern I create,
or I can guess your pattern.
If you are guessing, I will think of a
```

```
 pattern of letters (e.g., abcde).
On each turn, you guess the pattern and
 I will tell you how many letters you
got right, and how many of the correct
 letters were in the correct position.
The goal is to decode the puzzle as quickly
as possible. You control how many letters
can be used and how many positions
 (e.g., 5 possible letters in 4 positions)
as well as whether or not the pattern might
 contain duplicate letters (e.g., aabcd).
If I'm guessing, you think of a pattern
and score each of my answers.
How many letters? (2-10):
```

Analyzing the Code

The very first line of this program (Line 0) is

```
#include <iostream>
```

The goal of this line is to add to your current file the information it needs to support *Input* and *Output streaming*: the capability to read from the keyboard (input) and write to the screen (output).

Input Stream--How data comes into your program; typically from the keyboard

Output Stream--How data leaves your program; typically to the display

---

Here's how it works: C++ now includes a group of supporting code called the standard library, which provides objects to handle input and output. `cin` is an object that handles input from the keyboard, and `cout` is an object that handles output to the screen. The details of how they work are not important at this point, but to use them you must include in your program the file iostream, which provides their definitions. The definition of an object tells the compiler what it needs to know in order for the object to be used.

`cin` is pronounced *see-in*, and `cout` is pronounced *see-out*.

You include this file in your program with the `#include` statement. When your compiler is invoked, the precompiler runs, reading through your program and looking for lines that begin with the # symbol. When it sees `#include`, it knows it must read in a file. The angle brackets (`<` and `>`) say "look in the usual place." When you installed your compiler, it should have set up "the usual place" to look for these files.

Some folks pronounce # as *hash*, others as *cross-hash*. I call it `pound`, so I pronounce this line of code *pound include eye-oh-stream* .

The net effect is that the file iostream is read into your program at this point, which is just what you want. You can now use the `cout` object, as you'll see in a few moments.

NOTE: Using the angle brackets, `<iostream>` indicates that the precompiler is to "look in the usual place." An alternative is to use double quote marks--for instance `"myfile.h"`--which say "look in the current project directory and, failing that, look in the usual place."

Namespaces

Unlike the code in Chapter 1, "Introduction," this version uses the new ANSI/ISO standard library header file <iostream> rather than <iostream**.h**> (note that the new header doesn't use .h).

These headers support the new namespace protocols, which enable you to avoid conflicts in the names of objects and methods when working with code you buy from other vendors. For example, there might be two objects named `cout`. We solve this by "qualifying" the name with `std::`, as shown on lines 4-26. This qualification with `std::` indicates to the compiler that it is to use the `cout` object that is defined in the standard (`std`) library, which comes with your compiler.

Unfortunately, this makes the code look much more complicated and difficult to read.

using namespace std

To simplify this code and to make it easier for us to focus on the issues we care about, I'll rewrite the preceding example by adding the keywords

```
using namespace std;
```

This signals to the compiler that the code I'm writing is within the `std` (standard) namespace. In effect, it tells the compiler that when it sees `cout` it is to treat it like `std::cout`.

NOTE: All the rest of the code in the book uses this trick, which makes the code much easier to read and follow, at the cost of undermining the protection that namespaces afford.

When you write your commercial applications you might want to eschew the `using namespace` idiom because you might want to ensure namespace protection.

Listing 2.1a is an exact replica of Listing 2.1, except that it takes advantage of the `using namespace` idiom.

```
0:  #include <iostream>
1: using namespace std;
2:  int main()
3:  {
4:       cout << "Decryptix. Copyright 1999 Liberty ";
5:       cout << "Associates, Inc. Version 0.2\n " << endl;
6:
7:       cout << "There are two ways to play Decryptix: ";
8:       cout << " either you can guess a pattern I create, ";
9:       cout << "or I can guess your pattern.\n\n";
10:
11:       cout << "If you are guessing, I will think of a\n ";
12:       cout << "pattern of letters (e.g., abcde).\n\n";
13:
14:       cout << "On each turn, you guess the pattern and\n";
15:       cout << " I will tell you how many letters you \n";
16:       cout << "got right, and how many of the correct\n";
17:       cout << " letters were in the correct position.\n\n";
18:
```

```
19:       cout << "The goal is to decode the puzzle as quickly\n";
20:       cout << "as possible. You control how many letters \n";
21:       cout << "can be used and how many positions\n";
22:       cout << " (e.g., 5 possible letters in 4 positions) \n";
23:       cout << "as well as whether or not the pattern might\n";
24:       cout << " contain duplicate letters (e.g., aabcd).\n\n";
25:
26:       cout << "If I'm guessing, you think of a pattern \n";
27:       cout << "and score each of my answers.\n\n" << endl;
28:
29:       const int minLetters = 2;
30:       const int maxLetters = 10;
31:       const int minPositions = 3;
32:       const int maxPositions = 10;
33:
34:       int          howManyLetters = 0, howManyPositions = 0;
35:       bool          duplicatesAllowed = false;
36:       int          round = 1;
37:
38:       cout << "How many letters? (";
39:       cout << minLetters << "-" << maxLetters << "): ";
40:       cin >> howManyLetters;
41:
42:       cout << "How many positions? (";
43:       cout << minPositions << "-" << maxPositions << "): ";
44:       cin >> howManyPositions;
45:
46:       char     choice;
47:       cout << "Allow duplicates (y/n)? ";
48:       cin >> choice;
49:
50:       return 0;
51:  }
```

Code Spelunking

One of the most powerful ways to learn C++ is to use your debugger. I highly recommend that immediately after entering this code into your project (or downloading it from my site), you compile, link, and run it. You'll need to check your documentation for how to do this, but most modern IDEs offer a menu choice to "Build the entire project."

If you are using Visual C++, you can simply point your cursor at the buttons on the toolbar until you find the ones that compile and link or that build the entire project.

After it is working, set this book aside and pick up the documentation for your debugger, which you'll find with the documentation for your compiler. Set a break point on the first line of code in `main()` (see line 5 in Listing 2.1). In Visual C++ you just put your cursor on that line and press **F9**, or press the break point toolbar button. Once the break point is set, run to the break point (in Visual C++, press **F5**). Step over each line of code and try to guess what is going on. Again, you'll need to check your documentation for how to step over each line of code (in Visual C++ it is **F10**).

The debugger is one of the last things most primers introduce; I feel that it needs to be one of the very *first* things you learn. If you get stuck, see the exploration of debugging at the end of this chapter.

Every C++ program has a `main()` function (Listing 2.1, line 2). The general purpose of a function is to run a little code and then return to whomever called you.

All functions begin and end with parentheses, as you can see on lines 3 and 51. A *function* consists of a series of statements, which are all the lines that are shown between the parentheses.

This is the essence of a structured program. Program flow continues in the order in which the code appears in the file until a function is called. The flow then branches off to the function and follows line by line until another function is called or until the function returns (see Figure 2.2).

In a sense, a function is a subprogram. In some languages, it is called a *subroutine* or a *procedure*. The job of a function is to accomplish some work and then return control to whatever invoked the function.

Figure 2.2 *When a program calls a fuction, execution switches to the function and then resumes at the line after the function call.*

When `main()` executes, we execute `Statement1`. We then branch to line 1 of `Func1()`. `Func1`'s three lines execute, and then processing returns to `main()`, where we execute `Statement2`. `Func2` is then called, which in turn calls `Func3()`. When `Func3` completes it returns to `Func2()`, which continues to run until its own `return` statement, at which time we return to `main()` and execute `Statement3`. We then call `Func4()`, which executes its own code and then returns to `main()`, where we execute `Statement4`.

Returning a Value

When a function returns to whoever called it, it can return a value. You'll see later what the calling

function can do with that value.

Every function must declare what kind of value it returns: For example, does it return an integer or a character? If a function does not return a value, it declares itself to return `void`, which means that it returns nothing.

main() Is More Equal than Others

`main()` is a special function in C++. All C++ programs begin with `main()`; when `main` ends, the program ends. In a sense, the operating system (Windows, DOS, and so on) calls `main()`.

`main()` always returns an `int` (integer). I'll discuss the various types of values later in the book; for now it is sufficient to know that you must always declare `main` to return an integer.

NOTE: On some older compilers, you can have `main()` return `void`, but that is not legal under the new ISO standard. It is a good idea to get into the habit of having `main()` return an `int` every time.

You'll notice that `main()` does return an integer (in this case, `0`) on line 50. When programs are run from batch files or scripts, you can examine these values. For the programs in this book (and probably for most of the programs you will write), this value is discarded. By convention, you'll return `0` to indicate that the program ran without incident.

Using cout to Print to the Screen

Most of the statements in this very first program are designed to write to the screen. Use the standard output object `cout`. You send a string of characters to `cout` by enclosing them in quotation marks and by using the output redirection operator (`<<`), which you create by holding the Shift key and pressing the comma key twice.

This actually takes advantage of a very advanced feature in C++ called *operator overloading*, which is discussed in detail in Chapter 6, "Using Linked Lists." Fortunately, for now you can use this feature without fully understanding it. The net effect is that the words

```
Decryptix. Copyright 1999 Liberty
```

are sent to the screen.

Operator Overloading--The capability of user-created types to use the operators that built-in types use, such as +, =, and ==. I explain how to do this in Chapter 6.

Special Printing Characters

Line 5 prints the words

```
Associates, Inc. Version 0.2
```

to the screen. Notice that before the closing quotes, line 5 includes `\n`. These are two special marks within quoted strings. The slash is called an *escape character*, and when it is found in a quoted string it means "what follows is a special instruction to the compiler." The letter *n*, when it follows the escape character, stands for "new line." Thus, the effect is to print, to the output, a new line.

Escape character--A character that serves as a signal to the compiler or precompiler that the letter that follows requires special treatment. For example, the precompiler usually treats the character *n* as a letter, but when it is preceded by the escape character (`\n`), it indicates a new line.

Notice also that this line ends with

```
<< endl;
```

`cout` can receive more than just strings. In this case, the redirection operator is being used to send `endl`.

NOTE: `endl` is pronounced *end-ell* and stands for "end line."

This sends another new line to the output and flushes out the buffers. Buffers will be explained later, when I talk about streams, but the net effect ensures that all the text is written to the screen immediately.

Line 7 begins to print another line, which is continued on line 8 and completed on line 9.

Together, these lines print the following output:

```
Decryptix. Copyright 1999 Liberty Associates, Inc. Version 0.2
There are two ways to play Decryptix: either you can guess a pattern
I create,
or I can guess your pattern.
```

Note first that there is no new line after Liberty and before Associates. There was no instruction to `cout` to print a new line, so none was printed. Two new lines appear after `0.2`. The first, created by the `\n` character, ends the line; the second, created by `endl`, skips a line.

You can achieve the effect of skipping a line by putting in two `\n` characters, as shown on line 9.

Table 2.1 illustrates the other special printing characters.

Table 2.1Special Printing Characters

Character	What it means
`\n`	new line
`\t`	tab
`\b`	rings the bell
`\"`	prints a double quote
`\'`	prints a single quote
`\?`	prints a question mark

\\	prints a backslash

Variables

A *variable* is a place to store a value during the progress of your program.

Variable--A place to store a value

In this case, at line 36, you want to keep track of what round of play you are up to. Store this information in a variable named `round`:

```
int round = 1;
```

One way to think of your computer's memory is as a series of cubbyholes. Each cubbyhole is one byte, and every byte is numbered sequentially: The number is the address of that memory. Each variable reserves one or more bytes in which you can store a value.

Your variable's name (`round`) is a label on one of these cubbyholes, which enables you to find it easily without knowing its actual memory address.

Think of it like this: When you jump in a cab in Washington, D.C., you can ask for 1600 Pennsylvania Avenue, or you can ask for The White House. The identifier "the White House" is the name of that address.

Figure 2.3 is a schematic representation of this idea. As you can see from the figure, `round` starts at memory address 103. Depending on the size of `round`, it can take up one or more memory addresses.

Figure 2.3 *A schematic representation of memory.*

RAM is *random access memory*. When you run your program, it is loaded into RAM from the disk file. All variables are also created in RAM. When programmers talk about memory, they are usually referring to RAM.

Setting Aside Memory

When you define a variable in C++, you must tell the compiler what kind of variable you are declaring: an `int`, `char`, and so forth. The type tells the compiler the size of the variable. For example, a `char` is 1 byte, and on modern computers an `int` is 4 bytes; thus, the variable `round` consumes four bytes (cubbyholes) of memory.

Defining a Variable

You define a variable by stating its type, followed by one or more spaces, the variable name, and a semicolon:

```
int round;
```

The variable name can be virtually any combination of letters, but it cannot contain spaces. Legal variable names include `x`, `J23qrsnf`, and `myAge`. It is good programming practice to use variable names that tell you what the variables are for. This makes them easier to understand, which makes it easier for you to maintain your program.

Case Sensitivity

C++ is case sensitive; therefore, a variable named `round` is different from `Round`, which is different from `ROUND`. Avoid using multiple variables whose names differ only by capitalization--it can be terribly confusing.

NOTE: Some compilers enable you to turn case sensitivity off. Don't be tempted to do this. Your programs won't work with other compilers, and other C++ programmers will be very confused by your code.

Keywords

C++ reserves some words, and you cannot use them as variable names. These are keywords that are used by the compiler to control your program. Keywords include `if`, `while`, `for`, and `main`. Your compiler manual probably provides a complete list, but generally, any reasonable name for a variable is almost certainly not a keyword.

Creating More Than One Variable at a Time

You can create more than one variable of the same type in one statement by writing the type and then the variable names, separated by commas. For example

```
int howManyLetters, howManyPositions;
bool valid, duplicatesAllowed;
```

Assigning Values to Your Variables

Back in listing 2.1, at line 36, a local variable is defined by stating the type (`int`) and the variable name (`round`).

This actually allocates memory for the variable. Because an `int` is four bytes, this allocates four bytes of memory. When the compiler allocates memory, it reserves the memory for the use of your variable and assigns the name that you provide (in this case, `round`).

Scope

Scope refers to the region of a program in which an identifier--something that is named, such as an object, variable, function, or constant--is valid. When I say a variable has *local scope*, I mean that it is valid within a particular function.

Scope--The region of a program in which an identifier (that is, the name of something) is valid.

Local scope--When an identifier has local scope, it is valid within a particular function.

There are other levels of scope (global, static member, and so on) that I will discuss as I progress through the program.

The Value of Variables

Local variables, such as `round`, have a value when they are created regardless of whether you initialize them. If you don't initialize them (as shown here), whatever happened to already be in the bit of memory is assigned to them--that is, a random *garbage* value.

It is good programming practice to *initialize* your variables. When you initialize a variable, you create it and give it a specific value, all in one step:

```
int round = 1;
```

This creates the variable `round` and initializes it with the value `1`.

Just as you can define more than one variable at a time, you can initialize more than one variable. For example,

```
int howManyLetters = 0, howManyPositions = 0;
```

initializes the two variables `howManyLetters`, each to the value `0`. You can even mix definitions and initializations:

```
int howManyLetters = 0, round, howManyPositions = 2;
```

This example defines three variables of type `int`, and it initializes the first and third.

Characters

On line 46 of Listing 2.1, you created a character variable (type `char`) named `choice`. On most computers, character variables are 1 byte, enough to hold 256 values. A `char` can be interpreted as a small number (0-255) or as a member of the ASCII set. *ASCII* stands for the *American Standard Code for Information Interchange*. The ASCII character set and its *ISO* (*International Standards Organization*) equivalent are a way to encode all the letters, numerals, and punctuation marks.

ASCII--The American Standard Code for Information Interchange

ISO--The International Standards Organization

You create a character by placing the letter in single quotes. Therefore, `'a'` creates the character *a*.

In the ASCII code, the lowercase letter *a* is assigned the value `97`. All the lower- and uppercase letters, all the numerals, and all the punctuation marks are assigned values between `1` and `128`. Another 128 marks and symbols are reserved for use by the computer maker.

Characters and Numbers

When you insert a character--`'a'`, for example--into a `char` variable, what is really there is just a number between 0 and 255. The compiler knows, however, how to translate back and forth between characters and one of the ASCII values.

The value/letter relationship is arbitrary; there is no particular reason that the lowercase *a* is assigned the value `97`. As long as everyone (your keyboard, compiler, and screen) agrees, there is no problem. It is important to realize, however, that there is a big difference between the value `5` and the character `'5'`. The latter is actually valued at `53`, much as the letter `'a'` is valued at `97`.

Listing 2.2 is a simple program that prints the character values for the integers 32-127. Pay no attention to the details of this program--we will walk through how this works later in the book.

Listing 2.2 Printing out the Characters

```
   #include <iostream >
using namespace std;
int main()
   {
   for (int i = 32; i<128; i++)
     cout << (char) i;
   return 0;
   }
 !"#$%G'()*+,./0123456789:;<>?@ABCDEFGHIJKLMNOP
_QRSTUVWXYZ[\]^'abcdefghijklmnopqrstuvwxyz<|>~s
```

NOTE: Your computer might print a slightly different list.

Built-In Types

C++ comes right out of the box with knowledge of a number of primitive built-in types. The type of a variable or object defines its size, its attributes, and its capabilities.

For example, an `int` is, on modern compilers, 4 bytes in size. It holds a value from `-2,147,483,648` to `2,147,483,647`. For more on bytes and why `2,147,483,648` is a round number, see Appendix A, "Binary and Hexadecimal."

You might think that an integer is an integer, but it isn't quite. The keyword `integer` refers to a four-byte value, but only if you are using a modern compiler on a modern 32-bit computer. If your software or computer is 16-bit, however, an integer might be only two bytes. The keyword `short` usually refers to a two-byte integer, and the keyword `long` most often refers to a four-byte integer, but neither of these is certain. The language requires only that a `short` is shorter than or equal to an integer, and an integer is shorter than or equal to a `long`. On my computer, a `short` is 2 bytes and an integer is 4, as is a `long`.

ISO C++ provides the types that are listed in Table 2.2.

Table 2.2 Variable Types

Type	Size	Values
unsigned short `int`	2 bytes	`0` to `65,535`
short `int`	2 bytes	`-32,768` to `32,767`
unsigned long `int`	4 bytes	`0` to `4,294,967,295`
long `int`	4 bytes	`-2,147,483,648` to `2,147,483,647`
`int` (16-bit)	2 bytes	`-32,768` to `32,767`
`int` (32-bit)	4 bytes	`-2,147,483,648` to `2,147,483,647`
unsigned `int` (16-bit)	2 bytes	`0` to `65,535`
unsigned `int` (32-bit)	4 bytes	`0` to `4,294,967,295`

`char`	1 byte	256 character values
`float`	4 bytes	`1.2e-38` to `3.4e38`
`double`	8 bytes	`2.2e-308` to `1.8e308`
`bool`	1 byte	`true` or `false`

NOTE: ISO C++ recently added a new type, `bool`, which is a `true` or `false` value. `bool` is named after the British mathematician George Bool (1815-1864), who invented Boolean algebra, a system of symbolic logic.

Size of Integers

This book assumes that you are using a 32-bit computer (for example, a Pentium) and that you are programming with a 32-bit compiler. With that development environment, an integer is always 4 bytes. Listing 2.3 can help you determine the size of the built-in types on your computer, using your compiler.

Listing 2.3 Finding the Size of Built-In Types

```
1:   #include <iostream>
2:   using namespace std;
3:   int main()
4:   {
5:     cout << "The size of an int is:\t\t";
5a:    cout << sizeof(int)    << " bytes.\n";
6:     cout << "The size of a short int is:\t";
6a:    cout << sizeof(short)  << " bytes.\n";
7:     cout << "The size of a long int is:\t";
7a:    cout << sizeof(long)   << " bytes.\n";
8:     cout << "The size of a char is:\t\t";
8a:    cout << sizeof(char)   << " bytes.\n";
9:     cout << "The size of a float is:\t\t";
```

```
9a:    cout << sizeof(float)  << " bytes.\n";
10:    cout << "The size of a double is:\t";
10a:   cout << sizeof(double) << " bytes.\n";
11:    cout << "The size of a bool is:\t";
11a:   cout << sizeof(bool) << " bytes.\n";
12:        return 0;
13:  }
The size of an int is:          4 bytes.
The size of a short int is:     2 bytes.
The size of a long int is:      4 bytes.
The size of a char is:          1 bytes.
The size of a float is:         4 bytes.
The size of a double is:        8 bytes.
The size of a bool is:          1 bytes.
```

NOTE: On your computer, the number of bytes presented might be different. If the number of bytes reported for an `int` (the first line of output) is 2 bytes, you are using an older (and probably obsolete) 16-bit compiler.

TIP: If you are using Visual C++, you can run your program with **Ctrl+F5**. Your ouput displays, followed by

```
Press any key to continue
```

This gives you time to look at the output; then, when you press a key, the window closes.

If your compiler does not provide this service, you might find that the text scrolls by very quickly, and then you are returned to your IDE.

In this case, add the following line to the top of the file:

```
#include <conio.h>
```

Add the following line just before `return 0;`:

```
_getch();
```

This causes your program to pause after all the output is completed; it waits for you to press the spacebar or other character key. Add these lines to every sample program.

Most of Listing 2.3 is probably pretty familiar to you. The one new feature is the use of the `sizeof()` operator, which is provided by your compiler and tells you the size of the object you pass as a parameter. For example, on line 5, the keyword `int` is passed into `sizeof()`. Using `sizeof()`, I determined that on my computer an `int` is equal to a long `int`, which is 4 bytes.

Using an Integer Variable

In Decryptix!, we want to keep track of how many different letters the code can contain. Because this is a number (between `1` and `26`), you can keep track of this value with an `int`. In fact, because this number is very small, it can be a short `int`, and because it must be a positive number, in can be an unsigned short `int`.

Using a short `int` might save me two bytes. There was a time when such a savings was significant; today, however, I rarely bother. Short `int`s can be written just as short, so it is the height of profligacy to waste bytes because I'm too lazy to write short rather than `int`.

When I was a boy, bytes were worth something, and I watched every one. Today, for most applications, bytes are cheap. They are the pennies of programming, and these days most programmers just keep a small cup of bytes out on the counter and let customers take them when they need a few, occasionally tossing extra bytes into the cup for the next person to use.

NOTE: In the 1960s, many programmers worked in primitive languages, which represented dates as characters. Each number in the date consumed one byte (so 1999 consumed 4 bytes). Obsessive concern about saving a byte here and a byte there led many programmers to shorten dates from four digits (1999) to two (99), thus saving two bytes and creating the Y2K problem.

For the vast majority of programs, the only built-in types to be concerned with are `int`, `char`, and `bool`. Now and again you'll use unsigned `int`s, `float`s, and `double`s. Of course, most of the time you'll use your own programmer-created types.

Signed and Unsigned

After you determine the size of an integer, you're not quite done. An integer (and a short and a long) can be *signed* or *unsigned*. If it is signed, it can store negative and positive numbers. If it is unsigned, it can store only positive numbers.

Because signed numbers can store negative as well as positive numbers, the absolute value they can store is only half as large.

Wrapping Around an Unsigned Integer

The fact that unsigned long integers have a limit to the values they can hold is only rarely a problem--but what happens if you *do* run out of room?

When an unsigned integer reaches its maximum value, it wraps around and starts over, much like a car odometer. Listing 2.4 shows what happens if you try to put too large a value into a short integer.

Listing 2.4 Wrapping Around an Unsigned Integer

```
1: #include <iostream>
2: using namespace std;
3:  int main()
4:  {
5:     unsigned short int smallNumber;
6:     smallNumber = 65535;
7:     cout << "small number:" << smallNumber << endl;
8:     smallNumber++;
9:     cout << "small number:" << smallNumber << endl;
10:    smallNumber++;
11:     cout << "small number:" << smallNumber << endl;
12:     return 0;
13:  }
small number:65535
small number:0
small number:1
```

On line 4, `smallNumber` is declared to be an unsigned short `int`, which on my computer is a two-byte variable that can hold a value between `0` and `65,535`. On line 5, the maximum value is assigned to `smallNumber`, and it is printed on line 6 using the standard output library function. Note that because we did not add the line `using namespace std;` we must explicitly identify `cout`. The keyword endl is also part of the standard library and must be explicitly identified.

On line 7, `smallNumber` is *incremented*; that is, one is added to it. The symbol for incrementing is `++`

(as in the name *C++*, an incremental increase from C). Thus, the value in `smallNumber` is `65,536`. However, unsigned short integers can't hold a number larger than `65,535`, so the value is wrapped around to `0`, which is printed on line 8.

Incremented--When a value is incremented, it is increased by one.

On line 9, `smallNumber` is incremented again, and then its new value, `1`, is printed.

Wrapping Around a Signed Integer

A signed integer is different from an unsigned integer in that half of the values you can represent are negative. Instead of picturing a traditional car odometer, you might picture one that rotates up for positive numbers and down for negative numbers. One mile from `0` is either `1` or `-1`. When you run out of positive numbers, you run right into the largest negative numbers and then count back down to `0`. Listing 2.5 shows what happens when you add `1` to the maximum positive number in short integer.

Listing 2.5 Wrapping Around a Signed Integer

```
1:  #include <iostream>
2: using namespace std;
3:  int main()
4:  {
5:      short int smallNumber;
6:      smallNumber = 32767;
7:      cout << "small number:" << smallNumber << endl;
8:      smallNumber++;
9:      cout << "small number:" << smallNumber << endl;
10:     smallNumber++;
11:     cout << "small number:" << smallNumber << endl;
12:     return 0;
13:  }
small number:32767
small number:-32768
small number:-32767
```

On line 4, `smallNumber` is declared to be a signed short integer. (If you don't explicitly say that it is unsigned, it is assumed that it is signed.) The program proceeds much as the preceding program does, but the output is quite different.

The bottom line is that just like an unsigned integer, the signed integer wraps around from its highest positive value to its highest negative value.

Constants

The point of a variable is to store a value. I call it a variable because it might vary: That is, the value might change over the course of the program. `howManyLetters` starts out as `0`, but is assigned a new value based on the user's input.

At times, however, you need a fixed value--one that won't change over the course of the program. The minimum number of letters a user is allowed to choose is an example of a fixed value; the programmer determines it long before the program runs.

A fixed value is called a *constant*. There are two flavors of constants: literal and symbolic.

Literal Constants

A *literal constant* is a value that is typed directly into your program wherever it is needed. For example

```
int howManyLetters = 7;
```

`howManyLetters` is a variable of type `int`; `7` is a literal constant. You can't assign a value to `7`, and its value can't be changed.

Symbolic Constants

Like variables, *symbolic constants* are storage locations, but their contents never change during the course of your program. When you declare a constant, what you are really doing is saying to the compiler, "Treat this like a variable, but if I ever change the value stored here, let me know." The compiler then tells you with a compiler error.

You'll define `minLetters` to be a symbolic constant--specifically, a constant integer whose value is `2`. Note that the constant `minLetters` is used on a number of different lines of code. If you decide later to change the value to `3`, you need to change it only in one place--the change affects many lines of code. This helps you avoid bugs in your code. Changes are localized, so there is little chance of one line assuming that the minimum number of letters is two, whereas another line assumes that it is three.

There are actually two ways to declare a symbolic constant in C++. The old, traditional, and now obsolete way is with a preprocessor directive: `#define`.

Defining Constants with `#define`

To define a constant in the traditional way, enter the following code:

```
#define minLetters 2
```

Note that when it is declared this way, `minLetters` is of no particular type (`int`, `char`, and so on). `#define` does a simple text substitution. Every time the preprocessor sees the word `minLetters`, it puts in the text `2`.

Because the preprocessor runs before the compiler, your compiler never sees your constant; it sees the number `2`. Later, when I discuss debugging, you'll find that `#define`d constants do not appear as symbolic constants in the debugger; you see only the literal value (`2`).

Defining Constants with `const`

Although `#define` works, there is a newer, much better way to define constants in C++:

```
const int minLetters = 2;
```

This creates the symbolic constant `minLetters` but ensures that it has a particular type--`integer`. If you try to write

```
const int minLetters = 3.2;
```

the compiler complains that you have declared it to be an `int` but that you are trying to initialize it with a `float`.

The purpose of a debugger is to enable you to peek inside the machine and watch your variables change as the program runs. You can accomplish a lot with a debugger, and aside from your text editor, it is your most important tool.

Unfortunately, most novices don't become comfortable with their debugger until very late in their experience of C++. This is a shame because the debugger is a terrific learning tool.

Although every debugger is different and you definitely want to consult your documentation, this excursion illustrates how you might debug Listing 2.1 using the Microsoft Visual C++ 6.0 Enterprise Edition debugger. Your exact experience might vary, but the principles are the same.

Follow these steps:

1. Create a project called Decryptix.

2. Open a new file called decryptix.cpp.

3. Enter the program as it is written or download it from my Web site.

4. Place the cursor on the first line after the opening brace and press **F9**. You see a red dot in the margin next to that line, indicating a break point (see Figure 2.4).

Figure 2.4 *A break point showing in Visual C++.*

5. Press **Go** (**F5**). The debugger starts, and your code runs.

6. Choose **View/Debug Windows** and make sure that the Watch and Variables windows are open (see Figure 2.5).

Figure 2.5 *Checking that the Watch and Variables windows are open.*

7. Press **Step Over** (**F10**) to walk through the code line by line.

8. Scroll down to the line on which `duplicatesAllowed` is defined, and place a break point there. Press **Go** (**F5**) to run until this second break point.

9. Note in the variables window that `howManyLetters` and `howManyPositions` are zero, but that `duplicatesAllowed` has a random value (see Figure 2.6).

Figure 2.6 *Looking at values in the debugger.*

10. Press **Step Over** (**F10**) to step over this one line of code. This causes `duplicatesAllowed` to be created and initialized. Note that the value that is shown in the variables window is now correct (`false` is indicated as `0`, and `true` is indicated as `1`). Note also that `round` has a random value. Press **F10** again; `round` is initialized to `1`.

11. Explore the debugger and read through the documentation and help files. The more time you spend in the debugger, the more you will come to appreciate its tremendous value, both for finding bugs and for helping you understand how programs work.

Contents

C++ From Scratch

Online Copyright

C++ From Scratch

Contents

3

Program Flow

In this Chapter

- Building Robustness
- What Are You Trying to Accomplish?
- Solving the Problem with Loops
- The `if` Statement

This chapter takes a look at how programs progress, how loops are created, and how programs branch based on user input and other conditions.

Building Robustness

Listing 2.1 in Chapter 2, "Getting Started," is vulnerable to incorrect user input. For example, what happens if the user asks for 35 letters or says, "Use five letters in six positions without duplicates." This, of course, is impossible: You can't put five letters into six positions without duplicating at least one letter. How do you prevent the user from giving you bogus data?

A *robust program* is one that can handle any user input without crashing. Writing highly robust code is difficult and complex. Therefore, we won't set a goal of handling absolutely any bogus data, but we will attempt to deal with a few highly predictable errors.

Robust--A program is robust when it can handle incorrect user input or other unanticipated events without crashing.

Listing 3.1 illustrates a more complex version of the code you just considered. Before it is discussed, take a look at it and see if you can make some guesses about what it is doing.

Listing 3.1[em]A More Complex Version of Decryptix!

```
0:  #include <iostream>
1:  using namespace std;
2:  int main()
3:  {
4:    cout << "Decryptix. Copyright 1999 Liberty";
```

```
5:    cout << "Associates, Inc. Version 0.2\n\n" << endl;
6:    cout << "There are two ways to play Decryptix: either";
7:    cout << "you can guess a pattern I create,\n";
8:    cout << "or I can guess your pattern.\n\n";
9:    cout << "If you are guessing, I will think of a pattern\n";
10:   cout << "of letters (e.g., abcde).\n\n";
11:   cout << "On each turn, you guess the pattern and I will\n";
12:   cout << "tell you how many letters you got right, and how many
\n";
13:   cout << "of the correct letters were in the correct position.\n
\n";
14:   cout << "The goal is to decode the puzzle as quickly as\n";
15:   cout << "possible. You control how many letters can be\n";
16:   cout << "used and how many positions (e.g., 5 possible \n";
17:   cout << "letters in 4 positions) as well as whether or not\n";
18:   cout << "the pattern might contain duplicate \n";
19:   cout << "letters (e.g., aabcd).\n\n";
20:   cout << "If I'm guessing, you think of a pattern and score \n";
21:   cout << "each of my answers.\n\n" << endl;
22:
23:
24:    int round = 1;
25:    int howManyLetters = 0, howManyPositions = 0;
26:    bool duplicatesAllowed = false;
27:    bool valid = false;
28:
29:    const int minLetters = 2;
30:    const int maxLetters = 10;
31:    const int minPositions = 3;
32:    const int maxPositions = 10;
33:
34:
35:    while ( ! valid )
36:    {
37:       while ( howManyLetters < minLetters
38:          || howManyLetters > maxLetters )
39:       {
40:          cout << "How many letters? (";
41:          cout << minLetters << "-" << maxLetters << "): ";
42:          cin >> howManyLetters;
43:          if ( howManyLetters < minLetters
44:             || howManyLetters > maxLetters )
45:          {
```

```
46:             cout << "please enter a number between ";
47:             cout << minLetters << " and " << maxLetters << endl;
48:          }
49:       }
50:
51:       while ( howManyPositions < minPositions
52:          || howManyPositions > maxPositions )
53:       {
54:          cout << "How many positions? (";
55:          cout << minPositions << "-" << maxPositions << "): ";
56:          cin >> howManyPositions;
57:          if ( howManyPositions < minPositions
58:             || howManyPositions > maxPositions )
59:          {
60:             cout << "please enter a number between ";
61:             cout << minPositions <<" and " << maxPositions <<
endl;
62:          }
63:       }
64:
65:       char choice = ' ';
66:       while ( choice != 'y' && choice != 'n' )
67:       {
68:          cout << "Allow duplicates (y/n)? ";
69:          cin >> choice;
70:       }
71:
72:       duplicatesAllowed = choice == 'y' ? true : false;
73:
74:       if ( ! duplicatesAllowed
75:          && howManyPositions > howManyLetters )
76:       {
77:         cout << "I can't put " << howManyLetters;
78:         cout << " letters in " << howManyPositions;
79:         cout << " positions without duplicates! Please try again.
\n";
80:          howManyLetters = 0;
81:          howManyPositions = 0;
82:       }
83:       else
84:       valid = true;
85:    }
86:
```

```
87:    return 0;
88:  }
```

What Are You Trying to Accomplish?

Line 35 brings us to the first line of code after the initialization of the local variables and constants.

The goal with this piece of code is to prompt the user for a series of pieces of information. Specifically, you want to know how many letters he or she will use (for example, five letters means *a*, *b*, *c*, *d*, and *e*), how many positions (for example, three positions means there are three letters that are actually used in the code), and whether you'll allow duplicates (can one letter repeat?)

The problem is that the user might not give you valid information. For example, the user might tell you to use four letters in five positions with no duplicates. This, unfortunately, is not physically possible. You want to make sure that you have reasonable choices before moving forward with the program.

NOTE: Let me pause and point out that in a real commercial program, it is not unusual for literally dozens--or even hundreds--of lines of code to be devoted to catching and responding appropriately to bogus user input. You will not endeavor to be quite that robust here, but you do want to trap the obvious mistakes and ask the user to try again.

Solving the Problem with Loops

The essential approach to solving this problem is to do some work (ask the user for input), test a condition (determine whether the data makes sense), and, if the condition fails, start over.

This is called a *loop*; C++ supports a number of different looping mechanisms.

---

loop--A section of code that repeats.

Remember that you've created two constant integers for the values you need: `minLetters` and `maxLetters`. Because you initialized `howManyLetters` to zero, when you start out, howManyLetters is of course less than `minLetters`, assuming that `minLetters` is greater than zero.

You want to continue to prompt and then reprompt the user while howManyLetters is either less than the minimum or more than the maximum. To do this, you'll create a `while` loop.

The syntax for the `while` statement is as follows:

```
while ( condition )
statement;
```

`condition` is any C++ expression, and *`statement`* is any valid C++ statement or block of statements. When *`condition`* evaluates to `true`, *`statement`* executes, and then *`condition`* is tested again. This continues until *`condition`* tests false, at which time the `while` loop terminates and execution continues on the first line following *`statement`*.

Your `while` statement might be

```
while ( howManyLetters < minLetters )
{
     //...
}
```

NOTE: The symbol

```
//...
```

indicates that I've left out code that you're not considering at the moment.

Relational Operators

Relational operators determine whether two numbers are equal, or whether one is greater or less than the other. Every relational statement evaluates to either `true` or `false`.

Relational Operator--A symbol (for example, > or <) that is used to determine the relative size of two objects. Relational operators evaluate to `true` or `false`.

If the integer variable `howManyLetters` has the value `1` and the constant `minLetters` has the value `2`, the expression

```
howManyLetters < minLetters
```

returns `true`.

A `while` loop continues while the expression is true, so if `howManyLetters` is `1` and `minLetters` is `2`, the expression is true and the `while` loop will in fact execute.

Note that I talk about a single statement executing. It is also possible to execute a *block* (that is, a group) of statements.

Blocks and Compound Statements

A statement can be a single line or it can be a block of code that is surrounded by braces, which is treated as a single statement. Although every statement in the block must end with a semicolon, the block itself does not end with a semicolon.

The block of code itself can consist of any number of statements, but it is treated as a single statement.

This enables you to run several statements as the execution of a single `while` loop.

Not only can you test whether one variable is less than another, you can test whether one is larger, or even whether they are the same.

There are six relational operators listed in Table 3.1, which also shows each operator's use and some sample code.

Table 3.1 Relational Operators

Name	Operator	Sample	Evaluates
Equals	==	`100 == 50;`	`false`
		`50 == 50;`	`true`

Not Equals	`!=`	`100 != 50;`	`true`
		`50 != 50;`	`false`
Greater Than	`>`	`100 > 50;`	`true`
		`50 > 50;`	`false`
Greater Than	`>=`	`100 >= 50;`	`true`
or Equals		`50 >= 50;`	`true`
Less Than	`<`	`100 < 50;`	`false`
		`50 < 50;`	`false`
Less Than	`<=`	`100 <= 50;`	`false`
or Equals		`50 <= 50;`	`true`

WARNING: Many novice C++ programmers confuse the assignment operator (=) with the equals operator (==). This can create a nasty bug in your program.

Logical Operators

The problem with this `while` loop is that it tests only whether `howManyLetters` is less than the constant `minLetters`; you also need to test to find out whether `howManyLetters` is greater than `maxLetters`.

You *can* test them separately:

```
while ( howManyLetters < minLetters )
{
     //...
}
while ( howManyLetters > maxLetters )
{
     //...
}
```

This will work, but the code within both `while` loops will be identical. In fact, what you are really trying to say is that you want to repeat this work while `howManyLetters` is less than `minLetters` or while `howManyLetters` is greater than `maxLetters`. C++ enables you to make exactly that test using the *logical OR operator* (`||`). You create the logical OR operator by pressing **Shift+** twice.

The Logical OR Operator

In this case, you are asking for the `while` loop to continue as long as either condition is true, so you use logical OR:

```
while ( howManyLetters < minLetters || howManyLetters > maxLetters )
{
     //...
}
```

This code says that the statement (between the braces) is executed if it is true that `howManyLetters` is less than `minLetters` or if it is true that `howManyLetters` is greater than `maxLetters` (or if both conditions are true).

The Logical AND Operator

At other times, you might want to continue only if both conditions are true, in which case you want to use `while (`*`condition 1`* `and` *`condition 2`*`)`. The *logical AND operator* (&&) handles this condition. A logical AND statement evaluates two expressions, and if both expressions are true, the logical AND statement is true as well.

For example, you can test the following:

```
while ( howManyLetters > minLetters && howManyLetters < maxLetters )
{
```

```
    //...
}
```

this statement executes only if it is true that `howManyLetters` is greater than `minLetters` and if it is also true that `howManyLetters` is less than `maxLetters`.

Logical operator OR--**| |** created by two vertical lines, by pressing **Shift+backslash** (\) twice.

Logical operator AND--&& created by two ampersands, by pressing **Shift+7** twice.

The `if` StatementAn `if` statement allows you to take action only if a condition is true (and to skip the action or do something else if the condition is false). You use `if` statements every day:

```
If it is raining, I'll take my umbrella.

If I have time, I'll walk the dog.
```

If I don't walk the dog, I'll be sorry.

The simplest form of an `if` statement is this:

```
if (expression)
    statement;
```

The *`expression`* in the parentheses can be any expression at all, but it usually contains one of the relational expressions. If the expression has the value `false`, the statement is skipped. If it evaluates to `true`, the statement executes. Once again, the statement can certainly be a compound statement between braces, as you see here.

The Logical NOT Operator

A *logical NOT statement* (`!`) evaluates `true` if the expression that is being tested is false. This is confusing at first, but an example will help. I might start by saying, "If it is raining, I'll bring my umbrella":

```
if ( raining )
```

```
  BringUmbrella();
```

How do I express that I'll only go for a walk if it is *not* raining?

```
if ( ! raining )
   GoForWalk();
```

I can also reverse these:

```
if ( ! raining )
  LeaveUmbrella;
if ( raining )
  GoForWalk;
```

You get the idea.

Thus

```
if ( ! valid )
```

is true only if `valid` is false.

Logical NOT--Evalutes `true` when something is not true, and `false` when it *is* true.

You can use this nifty construct to turn your logical OR statement into a logical AND statement without changing its meaning. For example

```
while ( howManyLetters < minLetters || howManyLetters > maxLetters )
```

is exactly the same thing as

```
while ( (! (howManyLetters > minLetters) ) &&
(! (howManyLetters > maxLetters ) _) )
```

The logic of this is easy to understand if you use values. Assume that `minLetters` is `2`, `maxLetters` is `10`, and `howManyLetters` is `0`.

In that case, the `while` loop executes because the left part of the statement is true (0 is less than 2). In an OR statement, only one side must be true for the entire statement to return `true`.

Thus,

```
while ( howManyLetters < minLetters || howManyLetters > maxLetters )
```

becomes

```
while ( 0 < 2 || 0 > 10 ) // substitute the values
```

becomes

```
while ( true || false ) // evaluate the truth of each side
```

becomes

```
while ( true ) // if either is true, the statement is true
```

The second statement,

```
while ( (! (howManyLetters > minLetters) ) &&
(! (howManyLetters > maxLetters ) ) )
```

becomes

```
while ( (! (0 > 2) ) && (! (0 > 10 ) _) )
```

Now each side must be evaluated. The NOT symbol reverses the truth of what follows. It is as if this said, "While it is *not* true that zero is greater than 2 *and* it is *not* true that zero is greater than 10."

Thus you get

```
while ( (! (false) ) && (! (false ) _) )
```

When you apply NOT to `false`, you get `true`:

```
while ( (true) ) && (true) )
```

With an AND statement, both sides must be true; in this case they are, so the statement will execute.

Short Circuit Evaluation

When the compiler is evaluating an AND statement such as

```
while ( (x == 5) && (y == 5) )
```

the compiler evaluates the truth of the first statement (`x==5`); if this fails (that is, if x is not equal to five), the compiler does not go on to evaluate the truth or falsity of the second statement (`y == 5`) because AND requires that both be true.

Similarly, if the compiler is evaluating an OR statement such as

```
while ( (x == 5) || (y == 5) )
```

if the first statement is `true` (`x == 5`), the compiler never evaluates the second statement (`y == 5`) because the truth of either is sufficient in an OR statement.

Relational Precedence

Relational operators and logical operators, because they are C++ expressions, each return a value of `true` or `false`. Like all expressions, they have a precedence order (see Appendix B, "Operator Precedence") that determines which relations are evaluated first. This fact is important when determining the value of the statement

```
if ( x > 5 &&  y > 5  || z > 5)
```

It might be that the programmer wanted this expression to evaluate `true` if both x and y are greater than `5` or if z is greater than `5`. On the other hand, the programmer might have wanted this expression to evaluate `true` only if x is greater than `5`, and if it is also true that either y is greater than `5` or z is greater than `5`.

If x is `3` and y and z are both `10`, the first interpretation is true (z is greater than `5`, so ignore x and y), but the second is false (it isn't true that x is greater than `5`, and it therefore doesn't matter what is on the right side of the && symbol because both sides must be true.)

Although precedence determines which relation is evaluated first, parentheses can both change the order and make the statement clearer:

```
if (  (x > 5)  && (y > 5 ||  z > 5) )
```

Using the values that were mentioned earlier, this statement is false. Because it is not true that x is

greater than `5`, the left side of the AND statement fails, so the entire statement is false. Remember that an AND statement requires that both sides be true: Something isn't both "good tasting" AND "good for you" if it isn't good tasting.

NOTE: It is often a good idea to use extra parentheses to clarify what you want to group. Remember, the goal is to write programs that work and that are easy to read and understand. It is easier to understand

```
(8 * 5) + 3
```

than

```
8 * 5 + 3
```

even though the result is the same.

Putting It All Together

Following is the `while` statement you'll use to see whether you have a reasonable number of letters:

```
while ( howManyLetters < minLetters || howManyLetters > maxLetters )
{
     //...
}
```

This reads "As long as the condition is true, do the work between the braces." The condition that is tested is that either `howManyLetters` is less than `minLetters` OR `howManyLetters` is greater than `maxLetters`.

Thus, if the user enters `0` or `1`, `howManyLetters` is less than `minLetters`, the condition is true, and the body of the `while` loop executes.

do while

Because `howManyLetters` is initialized to zero, you know that this `while` loop will run at least once. If you do not want to rely on the initial value of `howManyLetters` but you want to ensure that the loop runs at least once in any case, you can use a slight variant on the `while` loop--the `do while`

loop:

```
do statement
    while ( condition )
```

This says that you will do the body of the loop while the condition is true. The loop must run at least once because the condition is not tested until after the statement executes the first time. So you can rewrite your loop as follows:

```
do
{
     //...
} while ( howManyLetters < minLetters || howManyLetters > maxLetters )
```

You know you need a `do while` loop when you are staring at a `while` loop and find your self saying, "Dang, I want this to run at least once!"

`do while`--A `while` loop that executes at least once and continues to exit while the condition that is tested is true.

Enumerated Constants

When I have constants that belong together, I can create *enumerated* constants. An enumerated constant is not quite a *type*; it is more of a collection of related constants.

The syntax for enumerated constants is to write the keyword `enum`, followed by the enumeration name, an open brace, each of the legal values (separated by commas), and a closing brace and a semicolon. Here's an example:

```
enum COLOR { RED, BLUE, GREEN, WHITE, BLACK };
```

This statement performs two tasks:

1. It makes `COLOR` the name of an enumeration.

2. It makes `RED` a symbolic constant with the value `0`, `BLUE` a symbolic constant with the value `1`, `GREEN` a symbolic constant with the value `2`, and so on.

Every enumerated constant has an integer value. If you don't specify otherwise, the first constant has the value `0` and the rest count up from there. Any one of the constants can be initialized with a particular value, however, and those that are not initialized count upward from the ones before them. Thus, if you write

```
enum Color { RED=100, BLUE, GREEN=500, WHITE, BLACK=700 };
```

`RED` has the value `100`; `BLUE`, the value `101`; `GREEN`, the value `500`; `WHITE`, the value `501`; and `BLACK`, the value `700`.

In this case you'll create an `enum` called `BoundedValues` and establish the values you need:

```
enum BoundedValues
{
minPos = 2,
maxPos = 10,
minLetters = 2,
maxLetters = 26
};
```

This replaces the four constant integers described previously. Frankly, there often is little advantage to enumerated constants, except that they keep these values together in one place. If, on the other hand, you are creating a number of constants and you don't particularly care what their value is so long as they all have unique values, enumerated constants can be quite useful.

Enumerated constants are most often used for comparison or testing, which is how you use them here. You'll test whether `minLetters` is greater or less than these enumerated values.

Returning to the Code

Let's look at the code beginning on line 37 and ending on line 49:

```
37:        while ( howManyLetters < minLetters
38:           || howManyLetters > maxLetters )
39:        {
40:           cout << "How many letters? (";
41:           cout << minLetters << "-" << maxLetters << "): ";
42:           cin >> howManyLetters;
43:           if ( howManyLetters < minLetters
44:              || howManyLetters > maxLetters )
45:           {
```

```
46:              cout << "please enter a number between ";
47:              cout << minLetters << " and " << maxLetters << endl;
48:           }
49:        }
```

The goal of this statement is to continue to prompt the user for an entry that is greater than `minLetters` and smaller than `maxLetters`.

The purpose of the `if` statement is to issue a reminder message if the number that is entered is out of bounds. Here's how the code reads in words:

```
37:        while ( howManyLetters < minLetters
38:           || howManyLetters > maxLetters )
```

While it is either true that the value `howManyLetters` is smaller than `minLetters` or it is true that `howManyLetters` is greater than `maxLetters`,

```
    cout << "How many letters? (";
    cout << minLetters << "-" << maxLetters << "): ";
    cin >> howManyLetters;
```

prompts the user and captures the user's response in the variable `howManyLetters`:

```
    if ( howManyLetters < minLetters
    || howManyLetters > maxLetters )
```

Test the response; if it is either smaller than `minLetters` or greater than `maxLetters`,

```
{
     cout << "please enter a number between ";
     cout << minLetters << " and " << maxLetters << endl;
}
```

prints out the reminder message.

The logic of this next `while` loop, shown on line 51, is identical to the preceding one.

Getting a Boolean Answer from the User

It is now time to ask the user whether he or she wants to allow duplicates (on line 72). You have a problem, however. The local variable `duplicatesAllowed` is of type `bool`, which, you'll

remember, is a type that evaluates either to `true` or `false`.

You cannot capture a Boolean value from the user. The user can enter a number (using `cin` to save it in an `int` variable) or a character (using `cin` to save it in a character variable). There are some other choices as well, but Boolean is not one of them.

Here's how you'll do it: You'll prompt the user to enter a letter, *y* or *n*, and you'll then set the Boolean value based on what is entered.

The first task is to capture the response, and here you need something very much like the `while` logic that was shown previously for the letters. That is, you create a variable, as shown on line 65, initialize it to an invalid answer (in this case, space), and then continue to prompt until the user gives you an acceptable answer (`'y'` or `'n'`):

```
char choice = ' ';
while ( choice != 'y' && choice != 'n' )
{
    cout << "Allow duplicates (y/n)? ";
    cin >> choice;
}
```

Begin by defining and initializing a character variable, `choice`. You can initialize it to a space by enclosing a space in single quotes, as described in Chapter 2.

Once again, you use a `while` loop to test whether you have valid data. This time, you will test to see whether `choice` is not equal to `'y'` or `'n'`.

If it is true that `choice` is not equal to (`!=`) `'y'`, and it is also true that `choice` is not equal to `'n'`, the expression returns `true` and the `while` statement executes.

Your next task is to test the value in `choice` (which must now be `'y'` or `'n'`) and set `duplicatesAllowed` accordingly. You can certainly use an `if` statement:

```
if ( choice == 'y')
    duplicatesAllowed = true;
else
    duplicatesAllowed = false;
```

Equality Operator ==

The equality operator (==) tests whether two objects are the same. With integers, two variables are equal

if they have the same value (for example, if x is assigned the value `4` and y is assigned the value `2*2`, they are equal). With character variables, they are equal if they have the same character value. No surprises here. We test for equality on line 72 to see if `choice` is equal to the letter `'y'`.

Equality operator (==)--Determines whether two objects have the same value. Be careful with this; you need two equal signs. A single equal sign (=) indicates *assignment* in C++. Thus, if you write

a = b

in C++ you assign the value currently in b to the variable a. If you want to test if they are equal, you must write

a == b

---

else

Often your program wants to take one branch if your condition is true, another if it is false. The keyword `else` indicates what the compiler is to execute if the tested expression evaluates `false`:

```
if (expression)
    statement;
else
    statement;
```

`else`--An `else` statement is executed only when an `if` statement evaluates to `false`.

Thus, the code shown says, "If `choice` is equal to `y`, set `duplicatesAllowed` to `true`; otherwise (`else`), set it to `false`."

The Conditional (or ternary) Operator

You're trying to assign the value `duplicatesAllowed` depending on the value of `choice`. In

English you might want to say, "Is `choice` equal to `y`? If so, set `duplicatesAllowed` equal to `true`; otherwise, set it to `false`."

C++ has an operator that does exactly what you want.

The conditional operator (`?:`) is C++'s only ternary operator: It is the only operator to take three terms.

NOTE: The *arity* of an operator describes how many terms are used. For example, a *binary* operator, such as the addition operator (`+`), uses two terms: `a+b`. In this case, `a` and `b` are the two terms.

C++ has a few *unary* operators, but you've not seen them yet. The conditional operator is C ++'s only ternary operator, and thus the terms *conditional* operator and *ternary* operator are often used interchangeably.

---

arity--How many terms an operator uses

unary--An operator that uses only one term

binary--An operator that uses two terms

ternary--An operator that uses three terms

The conditional operator takes three terms and returns a value. In fact, all three terms are expressions; that is, they can be statements that return a value:

```
(expression1) ? (expression2) : (expression3)
```

This line is read as follows: "If `expression1` is true, return the value of `expression2`; otherwise, return the value of `expression3`." Typically, this value is assigned to a variable.

Thus, line 72 shows

```
duplicatesAllowed = (choice == y) ? true : false;
```

Figure 3.1 illustrates each of the operators and terms.

Figure 3.1 *Dissecting a statement.*

This line is read as follows: "Is it true that `choice` equals the character `'y'`? If so, assign `true` to `duplicatesAllowed`; otherwise, assign `false`."

After you are comfortable with the conditional operator, it is clean, quick, and easy to use.

Putting It All Together

You are now ready to analyze the `while` loop, beginning at line 35. You start at line 27 by establishing `valid` as a Boolean operator that is initialized to `false`.

```
while ( ! valid )
```

The `while` loop executes while `valid` is false. Because `valid` was initialized to `false`, the `while` loop will certainly execute the first time through. This `while` loop begins with the opening brace on line 36 and ends at the closing brace on line 86.

This entire loop continues to execute until and unless `valid` is set to `true`.

Within this `while` loop are a series of interior `while` loops that solicit and test the values for `howManyLetters`, `howManyPositions`, and, ultimately (if indirectly), `duplicatesAllowed`:

```
 ( ! duplicatesAllowed && howManyPositions > howManyLetters )
```

Finally, after the values are gathered, on line 74 you test the logic of the choices. If it is true that duplicates are not allowed, and it is also true that `howManyPositions` (provided by the user) is greater than the number the user chose for `howManyLetters`, you have a problem: You need to put five letters in six positions without duplicating any letters--it can't be done.

In this case, you execute the `if` statement and write to the screen, "I can't put five letters in six positions without duplicates! Please try again." You then reinitialize `howManyLetters` and `howManyPositions` to zero. Make sure that you understand why. Hint: check the `while` loops in which these variables are assigned the user's choice.

If, on the other hand, the `if` statement fails (if duplicates are allowed or if `howManyPositions` is not greater than `howManyLetters`), the `else` statement executes, `valid` is set to `true`, and the

`while` loop terminates.

Contents

C++ From Scratch

Contents

4

Creating Classes

In Chapter 3, "Program Flow," you began to put logic into `main()` to gather the user's preference. In this chapter you'll look at creating classes to do this work.

Why Classes?

Although it is possible--and perhaps tempting--to just flesh out `main()` with the additional functionality you want to add to this program, it is a very bad idea.

The point of object-oriented programming is to create objects and assign them responsibility for specific aspects of the game. This fosters encapsulation, and with it maintainability and extensibility.

Maintainability means that the programs can be maintained at less expense. *Extensibility* means that you can add features without breaking the existing code.

As we design and implement classes, I'll discuss *design heuristics*: guidelines for designing excellent software.

---

design heuristics--Guidelines for quality in design

The very first--and perhaps most important--object-oriented design heuristic is that each class needs to have a single area of responsibility, and each object needs to collaborate with other objects to accomplish more complicated tasks.

As a rule, C++ programmers tend to keep `main()` very simple. Its job is only to create the first object and set off the chain of events that lead to these objects accomplishing their assigned tasks.

You'll begin by creating a `Game` class that is responsible for keeping track of the user's preferences and getting the game underway.

Creating New Types: Class

Although the built-in types are fine for storing values, they are limited in the complexity of the information they can manage.

Built-in types can be combined, however, into user-defined types that can be far more complex.

For example, suppose you want to store the number of letters from which you'll allow the player to choose and the number of positions (for example, choosing among three numbers in two positions, without duplicates, makes the following codes possible: `ab`, `ba`, `ac`, `ca`, `bc`, and `cb`). You can store these two values in variables, or you can store both of them along with the decision as to whether to allow duplicates--all within a `Game` class.

A class not only has values--it also has capabilities. Just as you know that an `int` can be added, subtracted, multiplied, and divided, a `Game` can be set up, played, restarted, quit, and saved.

Interface Versus Implementation

We draw a sharp distinction between the declaration of a class and its implementation. The declaration of a class tells the compiler about the attributes of the class and what its capabilities are. We often refer to this declaration as the class's *interface*.

Every method that is declared in the interface must be implemented: You must write the code that shows how it works.

---

interface--The declaration of the methods of a class

implementation--The code showing how the class methods work

Clients

Classes provide services to clients of the class. The client of your class is any programmer (even you!) who creates instances of your class in his code.

NOTE: It is regrettable that the generic pronoun in Standard English is masculine, and this is especially exacerbated by the fact that the programming profession is disproportionately male. Please understand that the masculine pronoun is intended to be generic.

Programmers use the term *client* in many ways. For example, if class A calls a method in Class B, we say that A is a client of B. If I write class A and I call code you wrote in class B, I am a client of your code. If my computer calls code running on your computer, my computer is a client of your (server) computer. And so on.

All these share the same essential characteristic: The client receives a service from the server.

The client of your class needs to know what your class does, but not how it works. If you create an `employee` class, your client needs to know that the employee can tell him his hire date, but your client does not need to know how your `employee` class keeps track of that date. You can store it in memory, on disk, or in a central database, but that is not important to your client. He cares about the interface (*can supply date*), not the implementation (*retrieve date from file*). Thus, the client treats your code as a *black box*.

---

client--Any code that makes use of a class

Looking at the Code

Before we discuss classes, let's take a quick look at the new code that is declaring the `Game` class. Once again, there is much here that will be new, but by reading through the code you can get a good idea of how it works--even before we go through it in detail.

Listing 4.1 Game.h

```
0:  class Game
1:  {
2:   public:
3:      Game();
4:      ~Game();
```

```
5:      void   Play();
6:
7:      bool    duplicatesAllowed;
8:      int        howManyLetters;
9:      int        howManyPositions;
10:      int        round;
11:  };
```

Declaring the Class

A class is declared by writing the keyword `class`, followed by the class name and an opening brace (as shown at line 0). The declaration is ended by a closing brace and a semicolon.

NOTE: The keyword `public` is needed as it is shown in line 2. We'll cover what this word does in a later chapter. For now, please be sure to place the keyword `public`, followed by a colon, at the start of every class declaration.

Classes and Objects

A *class* is a type. When you make instances of that type, they are called *objects*. In fact, the action of creating an object is called *instantiation*.

---

class--Defines a new type

object--An instance of the type defined by a class

instantiation--Creating an instance of a class: an object

Novice programmers often confuse classes with objects. The type of something (the class) tells you what it is (cat), what it can do (purr, eat, jump), and what attributes it has (weight and age). Individual objects of that type have specific objects (nine pounds, two years old.)

Member Variables

When this code was in `main()`, you had a number of local variables: `duplicatesAllowed`, `howManyLetters`, `howManyPositions`, and `round`.

These variables are now moved into the class, and they become members of the class itself starting at line 7.

Member variables represent attributes of the objects of that class type. In other words, we are now saying that every `Game` object will keep track of whether duplicates are allowed in that game, how many letters and how many positions are to be used in the game, what the current round is, and that these are the attributes of the class `Game`.

---

member variable--Data that is owned by a particular object of a class, and which represents attributes of that class.

Member variables are different from normal variables only in that they are scoped to a specific class. This is actually a very powerful aspect of object-oriented programming. The details of these values and their management are now delegated to the `Game` class and can be made invisible to the clients of that class.

Member Methods or Functions

The `Game` class has two principal activities:

- **Setup**--Get the user's preferences and choose a secret code.
- **Play**--Ask the user for guesses and score the guesses based on how many are correct and how many are in the correct position.

You provide these capabilities to a class by giving the class member methods, which are also called *member functions*. A member method is a function that is owned by a class--a method that is *scoped* to a particular class.

NOTE: When we say that a member method is *scoped* to a class, we mean that the identifier that is the member method is visible only within the context of the class or an

object of that class.

It is through these member methods that an object of a class achieves its behavior.

The Size of Objects

The size of an object is the sum of the sizes of the member variables that are declared for its class. Thus, if an `int` is 4 bytes and your class declares three integer member variables, each object is 12 bytes. Functions have no size.

Files

You create a class in two steps: First, the interface to the class is declared in a *header file*; second, the member methods are created in a *source code file*.

NOTE: Header file--A text file that contains the class declaration. Traditionally named with the .h extension

Source file--A text file that contains the source code for the member methods of a class. Traiditionally named with the .cpp extension

---

The header file typically has an extension of .h or .hpp, and the source code file has the extension .cpp. So for your `Game` class, you can expect to find the declaration in the file Game.h and the implementation of the class methods in Game.cpp.

Constructors

It is not uncommon for a class to require a bit of setting up before it can be used. In fact, an object of that class might not be considered valid if it hasn't been set up properly. C++ provides a special method to set up and initialize each object, called a *constructor*, as shown at line 3.

In this case, you want the constructor to initialize each of the member variables. For some member variables, you'll *hard wire* a reasonable value; for example, you'll keep track of what round of play you are on, and of course, you'll start with round 1.

NOTE: *Hard wire* is a programming term that means that the value is written into the code and doesn't change each time you run the program.

For other member variables, you must ask the user to choose an appropriate starting value. For example, you'll ask the user to tell you whether duplicates are allowed, how many letters are to be used, and how many positions are to appear in the secret code.

A constructor (line 3) has the same name as the class itself, and never has a return value.

NOTE: The absence of a return value does not, in this case, mean that it returns `void`. Constructors are special: They have no return value. There are only two types of methods for which this is true--constructors and destructors.

Destructors

The job of the destructor (line 4) is to tear down the object. This idea will make more sense after we talk about allocating memory or other resources. For now, the destructor won't do much, but as a matter of form, if I create a constructor, I always create a destructor.

Implementing the Methods

The header file provides the interface. Each of the methods is named, but the actual implementation is not in this file--it is in the implementation file (See Listing 4.2).

Listing 4.2 Game.cpp

```
0:  #include "Game.h"
1:  #include <iostream.h>
2:
3:
4:  Game::Game():
5:  round(1),
6:  howManyPositions(0),
```

```
7:  howManyLetters(0),
8:  duplicatesAllowed(false)
9:  {
10:     enum        BoundedValues
11:     {
12:        minPos = 2,
13:        maxPos = 10,
14:        minLetters = 2,
15:        maxLetters = 26
16:     };
17:     bool valid = false;
18:     while ( ! valid )
19:     {
20:        while ( howManyLetters < minLetters
21:           || howManyLetters > maxLetters )
22:        {
23:           cout << "How many letters? (";
24:           cout << minLetters << "-" << maxLetters << "): ";
25:           cin >> howManyLetters;
26:           if ( howManyLetters < minLetters
27:              || howManyLetters > maxLetters )
28:           {
29:              cout << "please enter a number between ";
30:              cout << minLetters << " and " << maxLetters << endl;
31:           }
32:        }
33:
34:        while ( howManyPositions < minPos
35:           || howManyPositions > maxPos )
36:        {
37:           cout << "How many positions? (";
38:           cout << minPos << "-" << maxPos << "): ";
39:           cin >> howManyPositions;
40:           if ( howManyPositions < minPos
41:              || howManyPositions > maxPos )
42:           {
43:              cout << "please enter a number between ";
44:              cout << minPos <<" and " << maxPos << endl;
45:           }
46:        }
47:
48:        char choice = ' ';
49:        while ( choice != 'y' && choice != 'n' )
```

```
50:        {
51:           cout << "Allow duplicates (y/n)? ";
52:           cin >> choice;
53:        }
54:
55:        duplicatesAllowed = choice == 'y' ? true : false;
56:
57:        if ( ! duplicatesAllowed &&
58:           howManyPositions > howManyLetters )
59:        {
60:         cout << "I can't put " << howManyLetters;
61:         cout << " letters in " << howManyPositions;
62:         cout << " positions without duplicates! Please try again.
\n";
63:         howManyLetters = 0;
64:         howManyPositions = 0;
65:        }
66:        else
67:           valid = true;
68:     }
69:
70:
71:  }
72:
73:  Game::~Game()
74:  {
75:
76:  }
77:
78:  void Game::Play()
79:  {
80:
81:  }
```

Listing 4.3 provides a short driver program that does nothing but instantiate an object of type `Game`.

Listing 4.3 Decryptix.cpp

```
0: #include <iostream >
1: #include "Game.h"
2:
3: int main()
4: {
```

```
5:     Game theGame;
6:     return 0;
8: }
```

Including the Header

The compiler can't know what a `Game` is without the definition, which is in the header file. To tell the compiler what a `Game` object is, the first thing you do in the implementation file is to `#include` the file with the definition of the `Game` class, in this case Game.h (as shown on line 1 of Listing 4.2).

NOTE: It is desirable to minimize the number of header files that are included in other header files. Having many `include` statements within a header file can risk the creation of circular references (a includes b, which includes c, which includes a) that won't compile. This can also introduce *order dependence*, which means that the proper execution of your code depends on files being added in the "correct order." This makes for code that is difficult to maintain.

There is no limit to the number of header files you might want to include in implementation files, but keep the includes in your header file to a minimum.

Implementing the Constructor

A member function definition begins with the name of the class, followed by two colons (the scoping operator), the name of the function, and its parameters. On line 4 in Listing 4.2, you can see the implementation of the constructor.

---

scope operator--The pair of colons between the class name and the method

identifier--Any named thing: object, method, class, variable, and so on

---

Like all methods, the constructor begins with an open brace (`{`) and ends with a closing brace (`}`). The body of the constructor lies between the braces.

Initialization

In the exploration of variables, I talked about the difference between assignment and initialization. Member variables can be initialized as well. In fact, the constructor actually executes in two steps:

- Initialization
- Construction

Construction is accomplished in the body of the constructor. Initialization is accomplished through the syntax that is shown: After the closing parentheses on the constructor, add a colon. For each member variable you want to initialize, write the variable name, followed by the value to which you want to initialize it (enclosed in parentheses). Note also that you can initialize multiple members by separating them with commas. There must be no comma after the last initialized value.

Thus, on line 5 in Listing 4.3, you see `round` initialized to the value `1`, `howManyPositions` to the value `0`, `howManyLetters` to the value `0`, and `duplicatesAllowed` to the value `false`.

> **NOTE:** The new line I've placed between each initialized value is only for the convenience of the programmer. I can just as easily put them all on one line, separated by spaces:

```
Game::Game():
round(1),
howManyPositions(0),
howManyLetters(0),
duplicatesAllowed(false)
{
```

All this initialization occurs before the body of the constructor runs, beginning on line 10 of Listing 4.2 .

> **NOTE:** We talk of methods or functions running, being executed, or being called, depending on context. These all mean the same thing: Program execution branches to the function, beginning at the first line and proceeding from there until it reaches a return statement.

Within the body of the constructor, you see that an enumerated constant, `BoundedValues`, is created, and a *local* variable, `valid`, is created and initialized on line 17.

This local variable, `valid`, will exist only for the duration of the constructor. Because this value is needed only temporarily and is not part of the permanent state of the object (it is not an attribute of the class `Game`), do not make it a member variable.

Just as `valid` is a variable that is local to the constructor, the instance of `Game` that is created in `main()` is local to `main()` (Listing 4.3, line 5). Declare it like you declare any other variable--by declaring its type (`Game`), and then the name of the object itself (`theGame`). You can name the object anything you want, but it is best to name it something meaningful so that the code can be easily understood.

By defining this object, you bring it into existence, and that causes the constructor to be invoked automatically.

Normally, methods are called *explicitly*. The constructor, however, is called *implicitly* when the object is created, and the destructor is called implicitly when the object is destroyed. When a method is called implicitly, the call doesn't appear in your code: It is understood to be the result of another action. Thus, when you create an object, you implicitly call the constructor; when you delete an object, you implicitly call the destructor. Not only do you not have to call these methods explicitly, you are prohibited from doing so.

There are two ways to see this explicitly. One way is to add a temporary output line to the constructor and destructor (as shown in Listing 4.4), and to `main()` (as shown in Listing 4.5).

Listing 4.4 Implicit Call to Constructor and Destructor

```
0:  #include "Game.h"
1:  #include <iostream>
2:
3:  Game::Game():
4:       round(1),
5:       howManyPositions(0),
6:       howManyLetters(0),
7:       duplicatesAllowed(false)
8:  {
9:      cout << "In the Game constructor\n"" << endl;
10:  }
11:
12:  Game::~Game()
```

```
13:  {
14:      cout << "In the Game destructor\n" << endl;
15:  }
16:
17:  void Game::Play()
18:  {
19:
20:  }
```

Listing 4.5 Driver Program for Listing 4.4

```
0:  #include <iostream>
1:  #include "Game.h"
2:
3:  using namespace std;
4:
5:  int main()
6:  {
7:      cout << "Creating the game\n" << endl;
8:      Game theGame;
9:      cout << "Exiting main\n" << endl;
10:      return 0;
11:  }
Creating the game
In the Game constructor
Exiting main
In the Game destructor
```

Here we've stripped the constructor down to do nothing except print an informative message. As you can see, creating the `Game` object causes the constructor to be invoked. Returning from `main()` ends the function and implicitly destroys any local objects. This causes the destructor to be invoked, which prints an equally informative message.

Using the Debugger

Although this works, it is tedious to add these printout messages; in any case, you can only infer the effect because you don't actually see the constructor being invoked. The debugger is a far more powerful tool.

Load Listings 4.1, 4.2, and 4.3 into a project and compile, link, and run it. Now, put a break point on line 5 in Listing 4.3--the creation of the `Game` object (see Figure 4.?). You are ready to see what this does, so step into the function call. You find yourself at the opening brace to the constructor.

The debugger is a powerful tool for learning C++. It can show you explicitly what is happening in your program as it runs. Because you'll be using the debugger throughout this book, you might want to take a few minutes and read the documentation that came with your programming environment to learn more about how to use your debugger.

Examining the Constructor

Careful examination of the constructor reveals that you have, essentially, duplicated the logic you had in `main()` in the preceding chapter. The one exception is that you are now capturing and storing the user's preferences in member variables as shown on lines 25, 39, and 52. These member variables are part of the object and will, therefore, persist and contain these values after the constructor returns.

The Other Methods

The `Game` object has two other methods: a destructor and the `Play()` method. At this time neither of these methods has any action, and you'll note that `Play()` is not yet called. This *stubbed out* function exists only to remind the programmer of his intent--that eventually this class will include a meaningful `Play()` method.

NOTE: When a programmer wants to show a method or function but does not want to do the work of implementing that function, he *stubs it out*: He creates a stub function that does nothing more than return, or at most prints, a message "in myTestMethod" and then returns.

Storing the Pattern

The computer creates a pattern that the human player guesses. How is this pattern to be stored? Clearly, you need the capability to store between `minLetters` and `maxLetters` characters as a pattern against which you can compare the human's guesses.

NOTE: Let me explain the preceding sentence because this is exactly how programmers talk about a problem like this: "The capability to store between `minLetters` and `maxLetters` characters." This sentence is easier to understand if we use sample values: If `minLetters` is `2` and `maxLetters` is `10`, this sentence means that you need the

capability to store between `2` and `10` letters in a pattern.

Programmers become comfortable using variables rather than absolute values in their formulations of a problem. The astute reader might also note that in fact we store these values (`minLetter` and `maxLetter`) as constants, not variables. That is true, but the values can vary from one compiled version to another, so they are variable in the more general sense.

So how *do* you store the computer's secret code? Let's assume that the player chooses seven possible letters with five positions, and the computer generates a secret code of `acbed`. How do you store this string of letters?

You have several options. You can use the built-in array class or the standard library string, you can create your own data structure to hold the letters, or you can use one of the standard library collection classes.

The rest of this chapter examines arrays in some detail; in coming chapters you'll turn to other alternatives.

What Is an Array?

An *array* is a fixed-size collection of data storage locations, each of which holds the same type of data. Each storage location is called an *element* of the array.

Array--A fixed size collection of data

You declare an array by writing the type, followed by the array name and the subscript. The *subscript* is the number of elements in the array, surrounded by square brackets. For example

```
long LongArray[25];
```

declares an array of 25 long integers, named `LongArray`. When the compiler sees this declaration, it sets aside enough memory to hold all 25 elements. Because each long integer requires 4 bytes, this declaration sets aside 100 contiguous bytes of memory, as illustrated in Figure 4.1.

---

subscript--The number of elements in an array

Figure 4.1 *Declaring an array.*

Initializing Arrays

You can initialize a simple array of built-in types, such as integers and characters, when you first declare the array. After the array name, put an equal sign (=) and a list of comma-separated values enclosed in braces. For example

```
int IntegerArray[5] = { 10, 20, 30, 40, 50 };
```

declares `IntegerArray` to be an array of five integers. It assigns `IntegerArray[0]` the value `10`, `IntegerArray[1]` the value `20`, and so on.

If you omit the size of the array, an array that is just big enough to hold the initialization is created. Therefore, if you write

```
int IntegerArray[] = { 10, 20, 30, 40, 50 };
```

you create exactly the same array as you did in the preceding example.

If you need to know the size of the array, you can ask the compiler to compute it for you. For example

```
int IntegerArrayLength = sizeof(IntegerArray)/sizeof(IntegerArray[0]);
```

sets the `int` variable `IntegerArrayLength` to the result that is obtained by dividing the size of the entire array by the size of each individual entry in the array. That quotient is the number of members in the array.

You cannot initialize more elements than you've declared for the array. Therefore,

```
int IntegerArray[5] = { 10, 20, 30, 40, 50, 60};
```

generates a compiler error because you've declared a five-member array and initialized six values. You can, however, write

```
int IntegerArray[5] = { 10, 20};
```

Uninitialized array members have no guaranteed values; therefore, any value might be in an array member if you don't initialize it.

Initializing Character Arrays

You can use a special syntax for initializing character arrays. Rather than writing

```
char alpha[] = { 'a', 'b', 'c' };
```

you can write

```
char alpha[] = "abc";
```

This creates an array of four characters and initializes with the three letters shown, followed by a `NULL` character. It is exactly as if you had written

```
char alpha[4] = {'a','b','c',0}
```

It adds the `NULL` because `NULL`-terminated strings have special meaning in C and C++.

C-Style Strings

C++ inherits from C the capability to create *strings*, which are meaningful groups of characters used to store words, phrases, and other strings of characters. These strings are represented in C and C++ as `NULL`-terminated arrays of characters. The old C library string.h, still a part of C++, provides methods for manipulating these strings: copying them, printing them, and so on.

The new Standard Library now includes a far better alternative: the `string` class. Objects of type `string` offer all the functionality of old C-style `NULL`-terminated arrays of characters, but with all the benefits of being well-defined types. That is, the new libraries are object-oriented, type safe, and well encapsulated.

You'll look at `string` objects in some detail as we go forward, and along the way I'll review some of the details of how C-style strings are used. For now, you're actually using this array of characters as a simple array, and it is not `NULL`-terminated. You are using the array as a collection. The objects that are being collected happen to be characters, but they can just as easily be integers or anything else you can store in an array.

Array Elements

You access each of the array elements by referring to an offset from the array name. Array elements are counted from zero. Therefore, the first array element is `arrayName[0]`.

In the version of the program we'll examine for the rest of this chapter, you'll add a member variable `solution`, which will hold an array of characters that represent the solution to the code generated by the compiler.

The declaration for that array is

```
char solution[maxPos+1];
```

This creates an array of characters that can hold exactly one more than `maxPos` characters. `maxPos` is a symbolic constant defined to `10`, so this defines an array of 11 characters. The 11th character is the `NULL` character.

Because arrays count from offset `0`, the elements in this array are `solution[0], solution[1], solution[2]...solution[9]`.

Writing Past the End of an Array

When you write a value to an element in an array, the compiler computes where to store the value, based on the size of each element and the subscript. Suppose that you ask to write over the value at `solution[5]`, which is the sixth element. The compiler multiplies the offset (`5`) by the size of each element. Since this is a `char` array, each element is 1 byte, so the math is fairly simple. The compiler then moves that many bytes (5) from the beginning of the array and writes the new value at that location.

If you ask to write at `solution[12]`, the compiler ignores the fact that there is no such element. It computes how far past the first element it is to look, and then writes over whatever is at that location. This can be virtually any data, and writing your new value there might have unpredictable results. If you're lucky, your program will crash immediately. If you're unlucky, you'll get strange results elsewhere in your program, and you'll spend weeks trying to find the bug.

Fence Post Errors

It is so common to write to one past the end of an array that this bug has its own name. It is called a *fence post error*. This refers to the problem in counting how many fence posts you need for a 10-foot fence if you need one post for every foot: Most people answer ten, but of course you need 11. Figure 4.2 makes this clear.

Figure 4.2 *Fence post errors.*

This sort of "off by one" counting can be the bane of any programmer's life. Over time, however, you'll get used to the idea that a 25-element array counts only to element 24, and that everything counts from zero.

NOTE: Fence post errors are responsible for one of the great misunderstandings of our time: when the new millennium begins. A millennium is 1,000 years. We are ending the second millennium of the Common Era, and we are about to start the third--but when, exactly?

The first Millennium began with the year 1 and ended with the year 1000. The second Millennium runs from 1001 to 2000. The third will begin on January 1, 2001. (Don't tell the newspapers.)

Of course, to a C++ programmer this is all wrong. We begin counting with 0, and 1K is 1,024, thus the third C++ Millennium begins on January 1, 2048. Call it the Y2K2 problem.

Some programmers refer to `ArrayName[0]` as the *zeroth* element. Getting into this habit is a big mistake. If `ArrayName[0]` is the zeroth element, what is `ArrayName [1],` the *oneth*? If so, when you see `ArrayName[24]`, will you realize that it is not the 24th element, but rather the 25th? It is far better to say that `ArrayName[0]` is at offset zero and is the first element.

Generating the Solution

Now that you have an array to hold the solution, how do you add letters to it? You want to generate letters at random, and you don't want the user to be able to guess the solution. The C++ library provides a method, `rand()`, which generates a pseudo-random number. It is pseudo-random in that it always generates numbers in the same predictable order, depending on where it starts--but they appear to be random.

You can increase the apparent randomness of the numbers that are generated if you start the random number generator with a different *starting* number (which we call a seed number) each time you run the program.

You provide `rand()` with a seed number by first calling `srand()` and passing in a value. `srand` (*s*eed *rand*om) gives the random number generate a starting point to work from. The seed determines the

first random number that will be generated.

If you don't call `srand()` first, `rand()` behaves as if you have called `srand()` with the seed value `1`.

You want to change the seed value each time you run the program so that you'll invoke one more library function: `time()`. The function `time` returns the system time, expressed as a large integer.

NOTE: Interestingly, it actually provides you with the number of seconds that have elapsed since midnight, January 1, 1970, according to the system clock. This date, 1/1/1970, is known as the *epoch*, the moment in time from which all other computer dates are calculated.

The `time()` function takes a parameter of type `time_t`, but we don't care about this because it is happy taking the `NULL` value instead:

```
srand( (unsigned)time( NULL ) );
```

The sequence, then, is to call `time()`, pass in `NULL`, cast the returned value to unsigned `int`, and pass that result to `srand()`. This provides a reasonably random value to `srand()`, causing it to initialize `rand()` to a nearly-random starting point.

NOTE: Let's talk about casting a value. When you cast a value to unsigned you say to the compiler, "I know you don't think this is an unsigned integer, but I know better, so just treat it like one." In this case, `time()` returns the value of type `time_t`, but you know from the documentation that this can be treated as an unsigned integer--and an unsigned integer is what `srand()` expects. Casting is also called "hitting it with the big hammer." It works great, but you've disconnected the sprinklers and disabled the alarms, so be sure you know what you're doing.

Now that you have a random number, you need to convert it into a letter in the range you need. To do this, you'll use an array of 26 characters, the letters *a-z*. By creating such an array, you can convert the value `0` to `a`, the value `1` to `b`, and so on.

Quick! What is the value of *z*? If you said `25`, pat yourself on the back for not making the fence post error of thinking it would be `26`.

We'll call the character array `alpha`. You want this array to be available from just about anywhere in your program. Earlier we talked about local variables, variables whose scope is limited to a particular method. We also talked about class member variables, which are variables that are scoped to a particular object of a class. A third alternative is a *global variable*.

---

global variable--A variable with no limitation in its scope--visible from anywhere in your program

---

The advantage of global variables is that they are visible and accessible from anywhere in your program. That is also the bad news--and C++ programmers avoid global variables like the plague. The problem is that they can be changed from any part of the program, and it is not uncommon for global variables to create tricky bugs that are terribly difficult to find.

Here's the problem: You're going along in your program and everything is behaving as expected. Suddenly, a global variable has a new and unexpected value. How'd that happen? With global variables, it is difficult to tell because they can be changed from just about anywhere.

In this particular case, although you want `alpha` to be visible throughout the program, you don't want it changed at all. You want to create it once and then leave it around. That is just what constants are for. Instead of creating a global variable, which can be problematic, you'll create a *global constant*. Global constants are just fine:

```
const char alpha[] = "abcdefghijklmnopqrstuvwxyz";
```

---

global constant--A constant with no limitation in its scope--visible from anywhere in your program.

This creates a constant named `alpha` that holds 27 characters (the characters *a*-*z* and the terminating `NULL`). With this in place,

`alpha[0]`

evaluates to *a*, and

`alpha[25]`

evaluates to *z*.

NOTE: We'll include the declaration of `alpha` in a new file called definedValues.h, and we'll `#include` that file in any file that needs to access `alpha`. This way, we create one place for all our global constants (all our defined values), and we can change any or all of them by going to that one file.

Listing 4.5 Adding Characters to the Array

```
0:  for ( i = 0; i < howManyPositions; )
1:  {
2:      int nextValue = rand() % (howManyLetters);
3:      char c = alpha[nextValue];
4:      if ( ! duplicatesAllowed && i > 0 )
5:      {
6:          int count = howMany(solution, c);
7:          if ( count > 0 )
8:              continue;
9:      }
10:      // add to the array
11:      solution[i] = c;
12:      i++;
13:  }
14:  solution[i] = '\0';
15:
16:  }
```

On line 0 you create a `for` loop to run once for each position. Thus, if the user has asked for a code with five positions, you'll create five letters.

On line 2 you call `rand()`, which generates a random value. You use the modulus operator (%) to turn that value into one in the range `0` to `howManyLetters-1`. Thus, if `howManyLetters` is `7`, this

forces the value to be `0`, `1`, `2`, `3`, `4`, `5`, or `6`.

Let's assume for the purpose of this discussion that `rand()` first generates the value `12`, and that `howManyLetters` is `7`. How is the value `12` turned into a value in the range `0` through `6`? To understand this, you must start by examining *integer division*.

Integer division is somewhat different from everyday division. In fact, it is exactly like the division you originally learned in fourth grade. "Class, how much is 12 divided by seven?" The answer, to a fourth grader, is "One, remainder five." That is, seven goes into 12 exactly once, with five "left over."

---

integer division--When the compiler divides two integers, it returns the whole number value and loses the "remainder."

When an adult divides 12 by 7, the result is a real number (1.714285714286). Integers, however, don't have fractions or decimal parts, so when you ask a programming language to divide two integers, it responds like a fourth grader, giving you the whole number value without the remainder. Thus, in integer math, 12/7 returns the value `1`.

Just as you can ask the fourth grader to tell you the remainder, you can use the modulus operator (%) to ask your programming language for the remainder in integer division. To get the remainder, you take 12 modulus 7 (`12 % 7`), and the result is 5. The modulus operator tells you the remainder after an integer division.

This result of a modulus operator is always in the range zero through the operand minus one. In this case, zero through seven minus one (or zero through six). If an array contains seven letters, the offsets are `0-6`, so the modulus operator does exactly what you want: It returns a valid offset into the array of letters.

On line 3 you can use the value that is returned from the modulus operator as an offset into `alpha,` thus returning the appropriate letter. If you set `howManyLetters` to `7`, the result will be that you'll always get a number between zero and six, and, therefore, a letter in the range *a* through *g*--exactly what you want!

Next, on line 4 you check to see whether you're allowing duplicates in this game. If not, enter the body of the `if` statement.

Remember, the bang symbol (`!`) indicates *not*, so

```
if ( ! duplicatesAllowed )
```

evaluates `true` if `duplicatesAllowed` evaluates `false`. Thus, if not, `duplicatesAllowed` means "if we're not allowing duplicates." The second half of the and statement is that *i* is greater than zero. There is no point in worrying about duplicates if this is the first letter you're adding to the array.

On line 6 you assign to the integer variable `count` the result of the member method `howMany()`. This method takes two parameters--a character array and a character--and returns the number of times the character appears in the array. If that value is greater than zero, this character is already in the array and the `continue` statement causes processing to jump immediately to the top of the `for` loop, on line 0. This tests *i*, which is unchanged, so proceed with the body of the `for` loop on line 2, where you'll generate a new value to try out.

If `howMany()` returns zero, processing continues on line 11, where the character is added to `solution` at offset `i`. The net result of this is that only unique values are added to the solution if you're not allowing duplicates. Next, `i` is incremented (`i++`) and processing returns to line 0, where `i` is tested against `howManyPositions`. When `i` is equal to `howManyPositions`, the `for` loop is completed.

Finally, on line 14 you add a `NULL` to the end of the array to indicate the end of the character array. This enables you to pass this array to `cout`, which prints every character up to the `NULL`.

NOTE: To designate a `NULL in a character array`, use the special character `'\0'`. To designate `NULL` otherwise, use the value `0` or the constant `NULL`.

Examining the Defined Values File

Take a look at Listing 4.6, in which we declare our constant array of characters `alpha`.

Listing 4.6 definedValues.h

```
0:  #ifndef DEFINED_VALUES
1:  #define DEFINED_VALUES
2:
3:  #include <iostream>
4:  using namespace std;
5:
```

```
6:  const char alpha[] = "abcdefghijklmnopqrstuvwxyz";
7:
8:  const int minPos = 2;
9:  const int maxPos = 10;
10:  const int minLetters = 2;
11:  const int maxLetters = 26;
12:
13:  #endif
```

This listing introduces several new elements.On line 0 you see the precompiler directive `#ifndef`. This is read "if not defined," and it checks to see whether you've already defined whatever follows (in this case, the string `DEFINED_VALUES`).

If this test fails (if the value `DEFINED_VALUES` is already defined), nothing is processed until the next `#endif` statement, on line 13. Thus, the entire body of this file is skipped if `DEFINED_VALUES` is already defined.

If this is the first time the precompiler reads this file, that value will not yet be defined; processing will continue on line 2, at which point it will be defined. Thus, the net effect is that this file is processed exactly once.

The `#ifndef/#define` combination is called an *inclusion guard*, and it guards against multiple inclusions of the same header file throughout your program. Every header file needs to be guarded in this way.

NOTE: *Inclusion guards* are added to header files to ensure that they are included in the program only once.

We intend to include the definedValues.h header file into all our other files so that it constitutes a global set of definitions and declarations. By including, for example, iostream.h here, we don't need to include it elsewhere in the program.

On line 6 you declare the constant character array that was discussed earlier. On lines 8-11 you declare a number of other constant values that will be available throughout the program.

Contents

C++ From Scratch

Contents

5

Playing The Game

- Predefined Macros
- Through the Program Once, by the Numbers

The declaration of the `Game` object builds on the material we've covered so far, and it adds a few new elements you'll need to build a robust class. Let's start by taking a look at the code (see Listing 5.1) and then discussing it in detail.

Listing 5.1 Game.h

```
1:    #ifndef GAME_H
2:    #define GAME_H
3:
4:    #include "definedValues.h"
5:
6:    class Game
7:    {
8:    public:
9:        Game();
10:        ~Game()        {}
11:        void Display(const char * charArray) const
12:        {
13:            cout << charArray << endl;
14:        }
15:        void Play();
16:        const char * GetSolution() const
17:        {
18:            return solution;
19:        }
20:    void Score(const char * thisGuess, int & correct, int &
position);
21:
22:    private:
23:        int    howMany(const char *, char);
24:        char solution[maxPos];
25:        int    howManyLetters;
26:        int    howManyPositions;
27:        int    round;
28:        bool duplicates;
```

29: };Once again, you see inclusion guards on line 1, and you now see the naming pattern that I'll use throughout this book. The inclusion guard will typically have the name of the class or file, in all uppercase, followed by the underscore (_) and the uppercase letter *H*. By having a standard for the creation of inclusion guards, you can reduce the likelihood of using the same guard name on two different header files.

On line 4, we include definedValues.h. As promised, this file will be included throughout the program.

On line 6, we begin the declaration of the `Game` class. In the public section we see the public interface for this class. Note that the public interface contains only methods, not data; in fact, it contains only those methods that we want to expose to clients of this class.

On line 9, we see the default constructor, as described in Chapter 4, "Creating Classes," and on line 10, we see the destructor. The destructor, as it is shown here, has an *inline* implementation, as do `Display()` (lines 11 to 14) and `GetSolution` (lines 16 to 19).

Inline Implementation

Normally, when a function is called, processing literally jumps from the calling function to the called function.

The processor must stash away information about the current state of the program. It stores this information in an area of memory known as the *stack*, which is also where local variables are created.

NOTE: The *stack* is an area of memory in which local variables and other information about the state of the program are stored.

The processor must also put the parameters to the new function on the stack and adjust the instruction pointer (which keeps track of which instruction will execute next), as illustrated in Figure 5.1. When the function returns, the return value must be popped off the stack, the local variables and other local state from the function must be cleaned up, and the registers must be readjusted to return you to the state you were in before the function call.

Figure 5.1 *The instruction pointer.*

An alternative to a normal function call is to define the function with the keyword `inline`. In this case, the compiler does not create a real function: It copies the code from the inline function directly into the

calling function. No jump is made. It is just as if you had written the statements of the called function right into the calling function.

Note that inline functions can bring a heavy cost. If the function is called 10 times, the inline code is copied into the calling functions each of those 10 times. The tiny improvement in speed you might achieve is more than swamped by the increase in size of the executable program. Even the speed increase might be illusory. First, today's optimizing compilers do a terrific job on their own, and there is almost never a big gain from declaring a function inline. More importantly, though, the increased size brings its own performance cost.

If the function you are calling is very small, it might still make sense to designate that function as inline. There are two ways to do so. One is to put the keyword `inline` into the definition of the function, before the return value:

```
inline int Game::howMany()
```

An alternative syntax for class member methods is just to define the method right in the declaration of the class itself. Thus, on line 19 you might note that the destructor's implementation is defined right in the declaration. It turns out that this destructor does nothing, so the braces are empty. This is a perfect use of inlining: There is no need to branch out to the destructor, just to take no action.

Constant Member Methods

Take a look at `Display` on lines 11-14. After the argument list is the keyword `const`. This use of `const` means, "I promise that this method does not change the object on which you invoke this method," or, in this case, "I promise that `Display()` won't change the `Game` object on which you call `Display()`."

`const` methods attempt to enlist the compiler in helping you to enforce your design decisions. If you believe that a member method ought not change the object (that is, you want the object treated as if it were read-only), declare the method constant. If you then change the object as a result of calling the method, the compiler flags the error, which can save you from having a difficult-to-find bug.

Geek SpeakThis entire section needs to be moved to later in this chapter. Thanks. -jThe Signature

On line 20, we see the *signature* for the member method `Score()`. The signature of a method is the name (`Score`) and the parameter list. The declaration of a method consists of its signature and its return value (in this case, `void`).

---

signature--The name and parameter list of a method.

Before we examine this signature in detail, let's talk about what this method does and what parameters it needs. The responsibility of this method is to examine the player's guess (an array of characters) and to score it on how many letters he correctly found, and of those letters, how many were in the right place.

Let's take a look at how this will be used. Listing 5.2 shows the `Play()` method, which calls `Score()`.

Listing 5.2 The Play Method

```
0:  void Game::Play()
1:  {
2:      char guess[80];
3:      int correct = 0;
4:      int position = 0;
5:
6:      //...
7:          cout << "\nYour guess: ";
8:      Display(guess);
9:
10:      Score(guess,correct,position);
11:      cout << "\t\t" << correct << " correct, " << position
12:       << " in position." << endl;
13:  }
```

I've elided much of this method (indicated by the `//...` marks), but the code with which we are concerned is shown. We ask the user for his guess and store it in the character array `guess`on line 2. We display `guess` by calling `Display()` on line 8 and passing `guess` in as a parameter. We then we score it by calling `Score()` on line 10 and passing in `guess` and two integer variables: `correct` (declared on line 3) and `position` (declared on line 4). Finally, we print the values for `correct` and `position` on lines 11 and 12.

`Score()` adjusts the values of `correct` and `position`. To accomplish this, we must pass in these two variables *by reference*.

Passing by Reference and by Value

When you pass an object into a method as a parameter to that method, you can pass it either by reference

or by value. If you pass it by reference, you are providing that function with access to the object itself. If you pass it by value, you are actually passing in a copy of the object.

---

passing by reference--Passing an object into a function.

passing by value--Passing a copy of an object into a function.

This distinction is critical. If we pass `correct` and `position` by value, `Score()` cannot make changes to these variables back in `Play()`. Listing 5.3 illustrates a very simple program that shows this problem.

Listing 5.3 Illustrating Pass by Value

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  class Game
4:  {
5:  public:
6:      Game(){};
7:      ~Game(){}
8:      void Play();
9:      void Score(int correct, int position);
10:
11:  private:
12:      int howManyLetters;
13:      int howManyPositions;
14:  };
15:
16:  void Game::Score(int  correct, int  position)
17:  {
18:      cout << "\nBeginning score. Correct: ";
19:      cout << correct << " Position: " << position << endl;
20:      correct = 5;
21:      position = 7;
22:      cout << "Departing score. Correct: ";
23:      cout << correct << " Position: " << position << endl;
24:  }
```

```
25:
26:   void Game::Play()
27:  {
28:      int correct = 0;
29:      int position = 0;
30:
31:      cout << "Beginning Play. Correct: ";
32:      cout << correct << " Position: " << position << endl;
33:      correct = 2;
34:      position = 4;
35:      cout << "Play updated values. Correct: " ;
36:      cout << correct << " Position: " << position << endl;
37:      cout << "\nCalling score..." << endl;
38:      Score(correct, position);
39:      cout << "\nBack from Score() in Play. Correct: ";
40:      cout << correct << " Position: " << position << endl;
41:  }
42:
43:  int main()
44:  {
45:
46:      Game theGame;
47:      theGame.Play();
48:      return 0;
49:  }
50: Beginning Play. Correct: 0 Position: 0
51: Play updated values. Correct: 2 Position: 4
52:
53: Calling score...
54:
55: Beginning score. Correct: 2 Position: 4
56: Departing score. Correct: 5 Position: 7
57:
58: Back from Score() in Play. Correct: 2 Position: 4
```

The very first thing to note is that I've moved everything into one file: Decryptix.cpp. This is for convenience only. In a real program, the declaration of `Game` would be in Game.h, the implementation of Game would be in Game.cpp, and so forth.

Let's examine the code. On line 6, you see that we've simplified the constructor to take no action, and we've implemented it inline. For the purpose of this illustration, we don't need to focus on anything except the invocation of `Score()` from `Play()`. On line 9, you might notice that I've simplified the signature of `Score():`, eliminating the array of characters. We'll come back to how to pass an array

into a function later, but for now I want to focus on the two integer variables, `correct` and `position`. Note that in this illustration the ampersand (&) is gone: We're now passing by value, not by reference.

Program flow begins in `main()`, toward the bottom of the file on line 43. On line 46 we create an instance of a `Game` object, and at (16) we invoke (or call) the method `Play()` on that method.

This call to `Play()` causes program flow to jump to the beginning of `Play()` on line 26. We start by initializing both `correct` and `position` to `0`, on line 28. We then print these values on line 32, which is reflected in the output on line 50.

Next, on lines 33 and 34 we change the values of `correct` and `position` to `2` and `4`, respectively, and then on line 36 we print them again, which is shown in the output on line 51.

On line 38 we invoke `Score()`, passing in `correct` and `position`. This causes the program flow to jump to the implementation of `Score()`, which is shown on lines 16-24.

The signature of `Score()` at its implementation matches that of `Score()` in the class declaration, as it must. Thus, `correct` and `position` are passed in by value. This is exactly as if you had declared local variables in this function and initialized them to the values they had in `Play()`.

On line 19 we print `correct` and `position` and, as the output shows on line 55, they match the values they had in `Play()`.

On lines 20 and 21, we change these values to `5` and `7`, and then on line 23 we print them again to prove that the change occurred; this appears in the output at line 56.

`Score()` now returns, and program flow resumes on 39; the values are printed again, as shown in the output on line 58.

Until this moment, everything has proceeded according to plan; however, the values back in `Play()` are not changed, even though you know they were in `Score()`. Step through this in your debugger, and you'll find that the values *are* changed in `Score()`, but when you are back in `Play()`, they are unchanged.

As you have probably already guessed, this is the result of passing the parameters by value. If you make one tiny change to this program and declare the values to be passed by reference, this program works as expected (see Listing 5.4).

Listing 5.4 Passing by Reference

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  class Game
4:  {
5:  public:
6:      Game(){}
7:      ~Game(){}
8:      void Play();
9:      void Score(int & correct, int & position);
10:
11:  private:
12:      int howManyLetters;
13:      int howManyPositions;
14:  };
15:
16:  void Game::Score(int & rCorrect, int & rPosition)
17:  {
18:      cout << "\nBeginning score. Correct: " << rCorrect
18a:      << " Position: " << rPosition << endl;
19:      rCorrect = 5;
20:      rPosition = 7;
21:  cout << "Departing score. Correct: "; << rCorrect;
21a: cout << " Position: " << rPosition << endl;
22:  }
23:
24:  void Game::Play()
25:  {
26:      int correct = 0;
27:      int position = 0;
28:
29:      cout << "Beginning Play. Correct: " << correct;
29a:     cout << " Position: " << position << endl;
30:      correct = 2;
31:      position = 4;
32:      cout << "Play updated values. Correct: " << correct;
32a:     cout << " Position: " << position << endl;
33:      cout << "\nCalling score..." << endl;
34:      Score(correct, position);
35:      cout << "\nBack from Score() in Play. Correct: " << correct;
35a:     cout << " Position: " << position << endl;
36:  }
37:
```

```
38:  int main()
39:  {
40:
41:      Game theGame;
42:      theGame.Play();
43:      return 0;
44:  }
45: Beginning Play. Correct: 0 Position: 0
46: Play updated values. Correct: 2 Position: 4
47:
48: Calling score...
49:
50: Beginning score. Correct: 2 Position: 4
51: Departing score. Correct: 5 Position: 7
52:
53: Back from Score() in Play. Correct: 5 Position: 7
```

The only change in this version is to the signature of `Score()` (on line 9), which is matched in the implementation (on line 16). The parameter names (for example, `rCorrect`) need not match between the declaration and the implementation.

NOTE: The parameter names are actually optional at the declaration. If you leave them off, the program compiles without error. As a general programming practice, however, be sure to include good parameter names even though they are not required. They serve as documentation and make your source code easier to understand.

The invocation of `Score()` on line 34 does not change at all. The client of `Score()` doesn't have to manage the fact that you are now passing `correct` and `position` by reference.

The output illustrates on line 53 that the change to the values in `Score()` did change the values back in `Play()`. This happens because this time no copy was made--you were changing the actual values.

References and Passing by Reference

The change in the signature is a change in type. You have changed `correct` from the integer

```
int correct
```

into a reference to an integer:

```
int & rCorrect
```

A *reference* is a special type that acts as an alias.

NOTE: A *reference* is a type that acts as an alias to an existing object.

The references `rCorrect` and `rPosition` are used within `Score()` exactly as if they were normal integer variables, but the values assigned to them are actually assigned to the original variables--`correct` and `position`--back in `Play()`.

NOTE: The name I've given it, `rCorrect`, is a clue to me that this is a reference. I tend to prepend reference variables with the letter *r* and pointers (discussed later) with the letter *p*, but the language does certainly not require this. You can name the variables in `Score()` and the variables in `Play()` using exactly the same names, but it makes the source code a bit more difficult to understand.

---

prepend--Programmers use the term *prepend* to indicate that you add something to the beginning of a term or variable. Thus, we prepend the letter *p* to variable names for pointers. The Free Online Dictionary of Computing (`http://www.instantweb.com/foldoc/foldoc.cgi?query=prepend`) defines *prepend* as follows: /pree *pend*/ (by analogy with *append*) To prefix or add to the beginning.

It is important to distinguish between passing *by* reference and passing *a* reference. There are two ways to pass by reference. So far we've examined one way: using a reference. Let's take a look at the alternative--pointers.

Pointers

Your throat tightens, your pulse quickens, and a cold, sickening dread grows in the pit of your stomach. Nothing unnerves new C++ programmers as does working with pointers. Well, relax. When you understand that a pointer is nothing more than a variable that holds the *address in memory* of another variable, pointers are a piece of cake.

What is a pointer?

When you create objects in your program, you create them in memory.

When you create the local variable `correct` in the method `Play()`, `correct` is kept in memory. Later you'll examine where in memory variables live, but for now it doesn't matter. What does matter is that `correct` is in memory, and every location in memory has an address. Normally, you don't care about the specific address of an object because you have a label (or name), for example, `correct`. If you need to get to `correct`, you can do so with its label.

You can get the address of `correct` by using the address of operator (`&`):

```
&correct;
```

NOTE: The address-of operator (`&`) uses the same ampersand that we used to identify references. The compiler can tell which you want by context.

When you have the address of `correct`, you can stash that address in a pointer. A pointer is a variable that holds the address of some object in memory.

---

pointer--A variable that holds the address of an object.

Memory Addresses

You can imagine that each memory location is ordered sequentially, as an offset from some arbitrary starting point. Picture, if you will, a series of cubbyholes, all aligned and numbered, perhaps as shown in Figure 5.2.

Figure 5.2 *Memory as cubbyholes.*

Every integer, character, or object you create is stored in these addresses. Because each cubby holds one byte, a 4-byte integer such as `correct` takes up four such locations.

`correct`'s address is just the first byte of that storage location. Because the compiler knows that an integer is 4 bytes, if its address is `0x001`, the compiler knows it occupies `0x001`, `0x002`, `0x003`, and `0x004`, and therefore puts the next integer at `0x005-0x008`.

NOTE: These addresses are in hexadecimal, which is a base-16 numbering system. If you are curious about this, please see Appendix A, "Binary and Hexadecimal."

There are two perspectives on what is stored in these locations. One is the bit perspective, which is pretty close to how the compiler "thinks" about memory (see Figure 5.3).

Figure 5.3 *How the compiler thinks about data in memory.*

From this perspective, the four bytes are filled with binary digits. How these values are interpreted is irrelevant to the compiler. The bits might represent an integer, a character, or an address somewhere else in memory. We'll return to that idea in a moment.

The point is that the compiler doesn't know or care how to interpret the bits--it just knows what is stored at a given location. To the programmer, however, this memory is conceived somewhat differently (as shown in Figure 5.4).

Figure 5.4 *How programmers think about data in memory.*

To the programmer, the value `5` is stored at this location like a letter in a mailbox. The programmer doesn't much care how the bits are configured, he just knows that the value is stashed away at a particular location.

Let's return to the idea that you can store a memory address. This is a powerful idea. We show here that the value `5` is stored at memory location `0x001`. What if you take that address, `0x001` (which in binary is `00000000 00000000 00000000 00000001`), and you stash that pattern at another address in memory: `0x1101` (see Figure 5.5).

Figure 5.5 *Storing the address.*

NOTE: There are some simplifying assumptions here that do not distort the point of this discussion. For example, these are not real memory locations, and values are often stored in memory in a slightly different order than is shown. In addition, compilers often store values at even boundaries in memory.

Here you see that at location `1101`, you have stored the value `0x001`: the memory location at which you stored the value `5`.

At that *pointed to* address, you hold the value `5` as illustrated in Figure 5.4. You can now assign this address to a variable that holds an address--a pointer. You declare a pointer by indicating the type of object it points to (in this case, `int`), followed by the pointer operator (`*`), followed by the name of the variable:

```
int * pCorrect;
```

This declares `pCorrect` to be a pointer to an integer. You can then assign the address of any integer, in this case `correct`, to that pointer:

```
pCorrect = &correct;
```

Thus, `pCorrect` now contains the address of the score, as shown in Figure 5.6.

Figure 5.6 *`pCorrect` points to `correct`.*

`pCorrect` is a pointer to an integer. The integer itself, `correct`, is stored at `0x001`, and `pCorrect` stores the address of that integer.

The pointer does not have to be in the same method as the variable. In fact, by passing the address into a method and manipulating it with a pointer, you can achieve the same pass by reference effect you achieved using references. Listing 5.5 illustrates this point by rewriting Listing 5.4 using pointers.

Listing 5.5 Using Pointers

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  class Game
```

```
4:  {
5:  public:
6:      Game(){}
7:      ~Game(){}
8:      void Play();
9:      void Score(int * correct, int * position);
10:
11:  private:
12:      int howManyLetters;
13:      int howManyPositions;
14:  };
15:
16:  void Game::Score(int * pCorrect, int * pPosition)
17:  {
18:      cout << "\nBeginning score. Correct: " << * pCorrect
18a:      << " Position: " << * pPosition << endl;
19:      * pCorrect = 5;
20:      * pPosition = 7;
21:      cout << "Departing score. Correct: " << * pCorrect
21a:      << " Position: " << * pPosition << endl;
22:  }
23:
24:  void Game::Play()
25:  {
26:      int correct = 0;
27:      int position = 0;
28:
29:      cout << "Beginning Play. Correct: " << correct
29a:      << " Position: " << position << endl;
30:      correct = 2;
31:      position = 4;
32:      cout << "Play updated values. Correct: " << correct
32a:      << " Position: " << position << endl;
33:      cout << "\nCalling score..." << endl;
34:      Score(&correct, &position);
35:      cout << "\nBack from Score() in Play. Correct: " << correct
35a:      << " Position: " << position << endl;
36:  }
37:
38:  int main()
39:  {
40:
41:      Game theGame;
```

```
42:      theGame.Play();
43:      return 0;
44:  }
45: Beginning Play. Correct: 0 Position: 0
46: Play updated values. Correct: 2 Position: 4
47:
48: Calling score...
49:
50: Beginning score. Correct: 2 Position: 4
51: Departing score. Correct: 5 Position: 7
52:
53: Back from Score() in Play. Correct: 5 Position: 7
```

The signature to `Score()` has changed again, as shown on lines 9 and 16. This time, `pCorrect` and `pPosition` are declared to be *pointers to* `int`: They hold the address of an integer.

On line 34, `Play()` calls `Score()` and passes in the addresses of `correct` and `position` using the address-of operator (&). There is no reason to declare a pointer here. All you need is the address, and you can get that using the address-of operator.

The compiler puts this address into the pointers that are declared to be the parameters to `Score()`. Thus, on line 34, the variables `pCorrect` and `pPosition` are filled with the addresses of `correct` and `position`, respectively.

Dereferencing

On line 18 you want to print the values of `correct` and `position`. You don't want the values of `pCorrect` and `pPosition` because these are addresses. Rather, you want to print the values at the variables whose addresses these pointers hold.

Similarly, on line 19 and 20 you want to set a new value into the variable whose address is stored in `pCorrect`. You do *not* want to write

```
pCorrect = 5;
```

because that assigns `5` to `pCorrect`, and `pCorrect` needs an address, not a simple integer value.

NOTE: This will compile, but it stores the address `5` to this pointer, which is a disaster waiting to happen.

The *dereference operator* (`*`) is used. Again, this is the pointer operator, but its meaning is understood in context.

---

dereference operator--The dereference operator is used to access the object to which the pointer points.

The dereference operator returns the object whose address is stored in the pointer. Thus

```
*pCorrect
```

returns the variable `correct`. By writing

```
*pCorrect = 5;
```

we store the value `5` in `correct`.

NOTE: I read the statement

```
*pCorrect = 5
```

NOTE: as "set the value at `pCorrect` to `5`." That is, assign `5` to the variable whose address is stored in `pCorrect`.

Getting the Operators Straight

There are two hurdles for a novice programmer--syntax and semantics--and pointers challenge you on both. The syntax is different because the same symbols (`&` and `*`) are used for many different purposes.

The asterisk is used for multiplication, for the declaration of a pointer, and for dereferencing:

```
z = x * y; // z equals x multiplied by y
int * ptr; // declare a pointer
```

*ptr = 7; // assign 7 to the dereferenced pointerSimilarly, the ampersand is used for references, for the address-of operator, and for logical AND:

```
if ( x && y ) // if x and also y
ptr = &x; // address-of operator
```

int & x = y; // initialize a referenceMore important than the confusing syntax is the difficulty with semantics. When I assign the value `5` on line 19, realize that I'm assigning `5` to `correct` in `Play()` indirectly through the pointer that is passed into `Score()`.

Arrays

Listing 5.6 reproduces the excerpt of `Play()` that we were examining in Listing 5.2 when we went off on the discussion of pointers. Remember that on line 10 we call `Score()` with three parameters, the first of which is our array of characters--`guess`. We've considered the two other parameters, `correct` and `position`, which are passed by reference using references. How is `guess` passed?

Listing 5.6 Play

```
            0:  void Game::Play()
1:  {
2:      char guess[80];
3:      int correct = 0;
4:      int position = 0;
5:
6:      //...
7:          cout << "\nYour guess: ";
8:          Display(guess);
9:
10:          Score(guess,correct,position);
11:          cout << "\t\t" << correct << " correct, " << position
11a:             << " in position." << endl;
12:  }
```

You must always pass arrays by reference; this is fairly easy to accomplish because C++ supports a close symmetry between arrays and pointers.

Excursion: Pointers and Constants

Every nonstatic member method has, as a hidden parameter, a pointer called the `this` pointer. The `this` pointer has the address of the object itself. When you write

```
const char * Game::GetSolution()
{
     return solution;
}
```

the compiler invisibly turns it into

```
const char * Game::GetSolution(Game * this)
{
     return this->solution;
}
```

You are free to use the `this` pointer explicitly: you can write your code as follows

```
const char * Game::GetSolution()
{
     return this->solution;
}
```

but there is little point in doing so. That said, there are times when you will use the `this` pointer explicitly to obtain the address of the object itself. We'll discuss this later in the book.

When you declare the member method `const`, the compiler changes the `this` pointer from a pointer to an object into a pointer to a *constant* object. Thus, the compiler turns the following code

```
const char * Game::GetSolution() const
{
     return solution;
}
```

into

```
const char * Game::GetSolution(const Game * this) const
{
     return this->solution;
}
```

The constant `this` pointer enforces the constancy of class `method`.

Arrays as Pointers

The name of an array (in our case, `guess`) is thought of as a pointer to the first element in the array. You can access elements of an array using the offset operator (`[]`), or by using the name of the array and what is called *pointer arithmetic*. Listing 5.7 illustrates this relationship between pointers and arrays.

---

pointer arithmetic--You can determine how many objects are in a range by subtracting the address of one pointer from another.

Listing 5.7 Relationship Between Pointers and Arrays

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  int main()
4:  {
5:      char myString[80];
6:      strcpy(myString,"Hello there");
7:      cout << "myString is " << strlen(myString)
7a:       << " characters long!" << endl;
8:      cout << "myString: " << myString << endl;
9:      char c1 = myString[1];
10:      char c2 = *(myString + 1);
11:      cout << "c1: " << c1 << " c2: " << c2 << endl;
12:      char * p1 = myString;
13:      char * p2 = myString + 1;
14:      cout << "p1: " << p1 << endl;
15:      cout << "p2: " << p2 << endl;
16:      cout << "myString+1: " << myString+1 << endl;
17:      myString[4] = 'a';
18:      cout << "myString: " << myString << endl;
19:      *(myString+4) = 'b';
20:      cout << "myString: " << myString << endl;
21:      p1[4] = 'c';
22:      cout << "myString: " << myString << endl;
```

```
23:      *(p1+4) = 'd';
24:      cout << "myString: " << myString << endl;
25:      myString[4] = 'o';
26:      myString[5] = '\0';
27:      cout << "myString: " << myString << endl;
28:      return 0;
29:  }
30: myString is 11 characters long!
31: myString: Hello there
32: c1: e c2: e
33: p1: Hello there
34: p2: ello there
35: myString+1: ello there
36: myString: Hella there
37: myString: Hellb there
38: myString: Hellc there
39: myString: Helld there
40: myString: Hello
```

On line 5, we create a character array that is large enough to hold the string. On line 6 we use the old-fashioned C-style string library routine `strcpy` to copy into our array a null-terminated string with the words *Hello there*.

On line 7 we use the C-style library routine `strlen` to obtain the length of the string. This measures the number of characters until the first `NULL` and returns that value (11) as an integer, which is printed by `cout` and shown on line 30.

On line 8 we pass the array to `cout`, which treats the name of the array (`myString`) as a pointer to the first byte of the string. `cout` knows that when it is given an array name it is to print every character until the first `NULL`. This is shown on line 30.

On line 9 we create a character variable, `c1`, which is initialized with the character at offset `1`--that is, the second character in the array, `e`.

On line 13 we treat `myString` as a pointer and add one to it. When you add one to a pointer, the compiler looks at the type of the object that is pointed to, which in this case is `char`. It uses the type to determine the size of the object, which in this case is one byte. It then returns the address of the next object of that size. If this were a pointer to `int`, it would return the address of the next `int`, four bytes later in memory.

Take the address that is returned (`myString+1`) and dereference it; this returns the character at that address. Then initialize a new character variable, `c2`, with that character. Note that `c2` is not a pointer;

by dereferencing, you're actually getting a character, and that is what is assigned to `c2`.

We print these two characters on line 11, and the printout is on line 32.

On line 12 we create a pointer to a character and assign it to `myString`. Because the name of the array acts as a pointer to the first byte of the array, `p1` now also points to the first byte. On line 13 we create a second pointer and point it to the second character in the array. These are printed on lines 14 and 15 and shown at lines 33 and 34). This illustrates that in each case `cout` acts as expected, printing the string beginning at the byte that is pointed to and continuing until the first `NULL`. These have the same printout as on line 18, which uses the string offset directly and which is shown on line 35.

On line 21 we use the offset operator to change the character that is stored at offset `4` (the fifth character). This is printed, and the output appears on line 36.

You can accomplish the same thing on line 16 by using pointer arithmetic and dereferencing the address that is returned; see the output on line 37. Because `p1` is pointing to `myString`, you can use the offset operator on the pointer on line 21. Remember that the name of the array is a pointer to the first element--and that is exactly what `p1` is. The effect on line 38 is identical.

Similarly, on line 23 we can use pointer arithmetic on `p1` and then dereference the resulting address, just as we did with the array name. The resulting printout is shown on line 39.

On line 25 we change the value back to `'o'` using the offset operator, and then we insert a null at offset 5. You can do the same thing with pointer arithmetic, but you get the point. As you probably remember, `cout` prints only to the first null, so the string `hello` is printed; nothing further in the array is printed, however, even though the word *there* still remains. This is shown on line 40.

Passing the Array as a Pointer

We said earlier that `guess` is passed by reference, as a pointer. What you see passed in Listing 5.6 is the name of the array, which is a pointer to the first element in the array:

Score(guess,`correct`,`position`);In `Score()` this first parameter must be declared as a pointer to character, which it is. Listing 5.8 reproduces Listing 5.1, the declaration of the `Game` class.

Listing 5.8 Reproducing Listing 5.1

```
0:  #ifndef GAME_H
1:  #define GAME_H
2:
3:  #include "definedValues.h"
```

```
4:
5:  class Game
6:  {
7:  public:
8:      Game();
9:      ~Game(){}
10:      void Display(const char * charArray)const{ cout << charArray
<<
10a:                                              endl;}
11:    void Play();
12:    const char * GetSolution() const { return solution; }
13:    void Score(const char * thisGuess, int & correct, int &
position);
14:
15:  private:
16:      int howMany(const char *, char);
17:      char solution[maxPos];
18:      int howManyLetters;
19:      int howManyPositions;
20:      int round;
21:      bool duplicates;
22:  };
23:
24:  #endif
```

You can see on line 13 that the first parameter to `Score()` is declared as a pointer to `char`, just as we require. Listing 5.9 shows the implementation of the `Score()` method.

Listing 5.9 Implementing Score()

```
0:void Game::Score(const char * thisGuess, int & correct, int &
position)
1:  {
2:      correct = 0;
3:      position = 0;
4:
5:      for ( int i = 0; i < howManyLetters; i++)
6:      {
7:          int howManyInGuess = howMany (thisGuess, alpha[i]);
8:          int howManyInAnswer = howMany (solution, alpha[i]);
9:          correct += howManyInGuess < howManyInAnswer ?
10:                          howManyInGuess : howManyInAnswer;
11:
```

```
12:      }
13:
14:      for ( int j = 0; j < howManyPositions; j++)
15:      {
16:          if ( thisGuess[j] == solution[j] )
17:              position++;
18:      }
19:
20:  }
```

The signature on the implementation agrees, as it must, with the declaration. `thisGuess` is a pointer to `char` and is the same array as `guess` in `Play()`. Because `guess` was passed by reference (as arrays must be), this is the same array, and changes to this array are reflected back in `Play`.

Because you must pass by reference but you do not want to allow `Score()` to change this array (and there is no reason for it to do so), declare the parameter to be a pointer to a constant `char` rather than a pointer to `char`. This keyword `const` says to the compiler, "I don't intend to change the object that is pointed to, so tell me if I do." This way, the compiler taps you on the shoulder if you attempt to make such a change and says, "Excuse me, sir, but you've changed an object when you promised you wouldn't. Not cricket, sir." (Your compiler's error message might vary).

Let's walk through this implementation of `Score()` line by line. On lines 2 and 3, we initialize both integers, `correct` and `position,` to `0`. If we take no other action, the score is zero correct and zero in position.

On line 5 we begin a `for` loop that will run once for each letter in `thisGuess`. The body of the `for` loop consists of three statements.

On line 7 a local variable--`howManyInGuess`--is initialized to store the result of calling the private member method `howMany(). When we call howMany, we pass` in the pointer to the array as the first parameter and the letter at `alpha[i]` as the second parameter.

This is a classic C++ statement, which does at least three things at once. Let's take the statement apart.

The first thing that happens is that `alpha[i]` is returned. The first time through this loop, `alpha[0]` is returned, which is `'a'`. The second time through, `'b'` is returned, and so forth.

This letter becomes the second parameter to the call to `howMany()`. If you look back at the declaration of `Game`, you'll find that `howMany()` is a private method that takes two parameters: a pointer to a constant `char` (the guess from `Play()`) and a character. Listing 5.10 shows the implementation of `howMany()`.

Listing 5.10 Implementing Game::HowMany()

```
0:  inline int Game::howMany(const char * theString, char c)
1:  {
2:      int count = 0;
3:      for ( int i = 0; i < strlen(theString); i++)
4:    {
5:          if ( theString[i] == c )
6:              count ++;
7:    }
8:      return count;
```

9: }The purpose of this method is to return the number of times an individual letter occurs in an array of characters. On line 2 the counter is initialized to zero. On line 3 we begin a `for` loop that iterates through every `position` in the array.

On line 5 we test each character to see whether it matches the character that was sent in to be tested; if so, we increment the counter. Note that the braces at lines 4 and 7 are not technically necessary, but as Donald Xie pointed out when editing this book, they do make the code much easier to read.

Finally, on line 8 we return that value.

In Listing 5.9, on line 7, we now have a value on the right side of the assignment that represents how many times `alpha[i]` occurs in `thisGuess`: that is, in the array that is passed in from `Play()`.

On line 8, we compute the same value for the solution. The value of `correct` is the lesser of these two, which we accomplish on lines 9 and 10 by using the ternary operator to find the smaller value.

An example makes this clearer: If the solution has *aabba* and the guess has *ababc*, we examine the first letter *a*. `howMany()` returns `2` for the guess and `3` for the solution, so the player has the lesser, `2`, correct.

On lines 14-18, we iterate again through the loops, this time testing on line 16 to see whether the character at a specific offset in `thisGuess` is the same as the character at the same offset in the solution. If so, another letter is in the right position.

Because `correct` and `position` were passed in as references, the changes that are made in `Score()` are reflected back in `Play()`.

Using ASSERT

Before moving on, I want to demonstrate how this code can be made both more reliable and more understandable through the use of ASSERT.

The purpose of ASSERT is to test your assumptions when you are debugging your code, but to have no effect at all when you release your final production version.

When you are debugging your code, you signal your compiler to enter *debug mode*. When you are ready to release your program to the paying public, you rebuild in *release mode*. Debug mode brings along a lot of debugging information that you don't want in release mode.

Thus, in debug mode, you can write

```
ASSERT ( position <= correct )
```

Here you are simultaneously documenting your belief that `position` must never be larger than `correct`. (You can never have five in the correct position if you only have four correct letters!) You are also testing that assertion each time the code runs to prove that you are right. In debug mode, if `position` ever is larger than `correct`, this ASSERT statement fails and an error message is written.

When your program is ready to be released, the ASSERT macro magically disappears and has no effect on the efficiency of your code.

How ASSERT Works

ASSERT is typically implemented as a *macro*. Macros are left over from C; they are type-unsafe routines that are processed not by the compiler but by the precompiler, the same beast that handles your `#include` and `#define` statements. In fact, a macro *is* a `#define` statement.

---

macro--A text substitution by the precompiler. Macros can act as small subprograms.

Excursion: Macros

A macro function is a symbol that is created using `#define`, which takes an argument much like a function does, and which replaces the macro and its argument with a *substitution string*. For example, you can define the macro `TWICE` as follows:

```
#define TWICE(x) ( (x) * 2 )
```

Then in your code you write

```
TWICE(4)
```

The entire string `TWICE(4)` is removed and the value `4*2` is substituted. When the precompiler sees TWICE(`4`), it substitutes `( (4) * 2 )`. That is just what you want because 4*2 evaluates to 8, so `TWICE` will have done just the work you expected.

A macro can have more than one parameter, and each parameter can be used repeatedly in the replacement text. Two common macros are `MAX` and `MIN`:

```
#define MAX(x,y) ( (x) > (y) ? (x) : (y) )
#define MIN(x,y) ( (x) < (y) ? (x) : (y) )
```

`MAX` returns the larger of two values (`x` and `y`), and `MIN` returns the lesser. Thus, `MAX(7,5)` is `7`, and `MIN(7,5)` is `5`.

NOTE: In a macro function definition, the opening parenthesis for the parameter list must immediately follow the macro name, with no spaces. The preprocessor is not as forgiving of white space as is the compiler.

Why All the Parentheses?

You might be wondering why there are so many parentheses in these macros. The preprocessor does not demand that parentheses be placed around the arguments in the substitution string, but the parentheses help you avoid unwanted side effects when you pass complicated values to a macro. For example, if you define `MAX` as

```
#define MAX(x,y) x > y ? x : y
```

and pass in the values `5` and `7`, the macro works as intended. If you pass in a more complicated expression, however, you'll get unintended results, as shown in Listing 5.11.

Listing 5.11 Unintended Macro Results

```
0:
1:  #include <iostream.h>
2:
3:  #define CUBE(a) ( (a) * (a) * (a) )
4:  #define THREE(a) a * a * a
5:
6:  int main()
7:  {
8:      long x = 5;
9:      long y = CUBE(x);
10:      long z = THREE(x);
11:
12:      cout << "y: " << y << endl;
13:      cout << "z: " << z << endl;
14:
15:      long a = 5, b = 7;
16:      y = CUBE(a+b);
17:      z = THREE(a+b);
18:
19:      cout << "y: " << y << endl;
20:      cout << "z: " << z << endl;
21:      return 0;
22:  }
***Please Insert Output icon herey: 125
z: 125
y: 1728
z: 82
```

On line 1, we use the old-fashioned iostream.h so that we can avoid using namespaces. This is perfectly legal in C++, and it is common in writing very short demonstration programs.

On line 3, the macro `CUBE` is defined, with the argument `x` put into parentheses each time it is used. On line 4, the macro `THREE` is defined, without the parentheses. It is intended for these macros to do exactly the same thing: to multiply their arguments times themselves, three times.

In the first use of these macros, on line 16, the value `5` is given as the parameter and both macros work fine. `CUBE(5)` expands to `( (5) * (5) * (5) )`, which evaluates to `125`, and `THREE(5)` expands to `5 * 5 * 5`, which also evaluates to `125`.

In the second use, on line 17, the parameter is `5 + 7`. In this case, `CUBE(5+7)` evaluates to

```
 ( (5+7) * (5+7) * (5+7) )
```

which evaluates to

```
 ( (12) * (12) * (12) )
```

which in turn evaluates to `1,728`. `THREE(5+7)`, however, evaluates to

```
5 + 7 * 5 + 7 * 5 + 7
```

Because multiplication has a higher precedence than addition, this becomes

```
5 + (7 * 5) + (7 * 5) + 7
```

which evaluates to

```
5 + (35) + (35) + 7
```

which finally evaluates to `82`.

Macros Versus Functions

Macros suffer from four problems in the eyes of a C++ programmer. First, because all macros must be defined on one line, they can be confusing if they become large. You can extend that line by using the backslash character (\), but large macros quickly become difficult to manage.

Second, macros are expanded inline each time they are used. This means that if a macro is used a dozen times, the substitution appears 12 times in your program, rather than appearing once as a function call does. On the other hand, they are usually quicker than a function call because the overhead of a function call is avoided.

The fact that they are expanded inline leads to the third problem, which is that the macro does not appear in the intermediate source code that is used by the compiler, and therefore it is not visible in most debuggers. By the time you see it in the debugger, the substitution is already accomplished. This makes debugging macros tricky.

The final problem, however, is the largest: Macros are not type-safe. Although it is convenient that absolutely any argument can be used with a macro, this completely undermines the strong typing of C++ and so is anathema to C++ programmers.

That said, the `ASSERT` macro is a good example of a time when this is not a bug, but a feature: One `ASSERT` macro can test any condition, mathematical or otherwise.

String Manipulation

The preprocessor provides two special operators for manipulating strings in macros. The *stringizing operator* (#) substitutes a quoted string for whatever follows the stringizing operator. The *concatenation operator* (##) bonds two strings together into one.

NOTE: The *stringizing operator* (#) substitutes a quoted string for whatever follows the stringizing operator.

The *concatenation operator* (##) bonds two strings together into one.

Stringizing

The stringizing operator(#) puts quotes around any characters that follow the operator, up to the next white space. Thus, if you write

```
#define WRITESTRING(x) cout << #x
```

and then call

```
WRITESTRING(This is a string);
```

the precompiler turns it into

```
cout << "This is a string";
```

Note that the string `This is a string` is put into quotes, as is required by `cout`.

Concatenation

The concatenation operator (##) enables you to bond together more than one term into a new word. The new word is actually a token that can be used as a class name, a variable name, or an offset into an array--or anywhere else a series of letters might appear.

Assume for a moment that you have five functions named `fOnePrint`, `fTwoPrint`, `fThreePrint`, `fFourPrint`, and `fFivePrint`. You can then declare

```
#define fPRINT(x) f ## x ## Print
```

and then use it with `fPRINT(Two)` to generate `fTwoPrint`, and with `fPRINT(Three)` to generate `fThreePrint`.

Predefined Macros

Many compilers predefine a number of useful macros, including `__DATE__`, `__TIME__`, `__LINE__`, and `__FILE__`. Each of these names is surrounded by two underscore characters to reduce the likelihood that the names will conflict with names you've used in your program.

When the precompiler sees one of these macros, it makes the appropriate substitutes. For `__DATE__`, the current Date is substituted; for `__TIME__`, the current time is substituted. `__LINE__` and `__FILE__` are replaced with the source code line number and filename, respectively. Note that this substitution is made when the source is precompiled, not when the program is run. If you ask the program to print `__DATE__`, you do not get the current date; instead, you get the date the program was compiled. These defined macros are very useful in debugging.

Although many compilers do provide an `ASSERT` macro, it will be instructive to create our own, shown in Listing 5.12.

Listing 5.12 An ASSERT Macro

```
0:  #define DEBUG
1:
2:  #ifndef DEBUG
3:      #define ASSERT(x)
4:  #else
5:      #define ASSERT(x) \
6:              if (! (x)) \
7:              { \
8:                  cout << "ERROR!! Assert " << #x << " failed\n"; \
9:                  cout << " on line " << __LINE__  << "\n"; \
10:                  cout << " in file " << __FILE__ << "\n";  \
11:              }
12:  #endif
```

On line 0, we define the value `DEBUG`, which we test on line 2. In the production version we'll remove the definition of `DEBUG`, and the test on line 2 will fail. When the test fails, this macro defines `ASSERT(x)` to do nothing, as shown on line 3. If the test succeeds, as it will while we are debugging, this macro defines `ASSERT` as shown on line 5.

In a macro, any line ending with \ continues on the next line as if both were on the same line. The entire set of lines from line 5 to line 11 is thus considered a single line of the macro. On line 6, whatever is passed to the macro (x) is tested; if it fails, the body of the `if` statement executes, writing an error message to the screen.

On line 8, we see the stringizing macro at work, and the following lines take advantage of the `__FILE__` and `__LINE__` macros that are supplied by the compiler vendor.

I don't show ASSERT macros everywhere they might appear in this book because they can detract from the point that is being made; at other times, however, they can greatly clarify the program. For example, I'd rewrite `Score()` as shown in Listing 5.13.

Listing 5.13 Rewriting Score with ASSERT

```
0:void Game::Score(const char * thisGuess, int & correct, int &
position)
1:  {
2:       correct = 0;
3:       position = 0;
4:
5:       ASSERT ( strlen(thisGuess) == howManyPositions )
6:       ASSERT ( strlen(solution) == howManyPositions )
7:
8:       for ( int i = 0; i < howManyLetters; i++)
9:       {
10:           int howManyInGuess = howMany (thisGuess, alpha[i]);
11:           int howManyInAnswer = howMany (solution, alpha[i]);
12:           correct += howManyInGuess < howManyInAnswer ?
12a:                    howManyInGuess : howManyInAnswer;
13:       }
14:
15:       for (  i = 0; i < howManyPositions; i++)
16:       {
17:           if ( thisGuess[i] == solution[i] )
18:               position++;
19:       }
20:
21:       ASSERT ( position <= correct )
22:
23:  }
```

The ASSERT on line 5 documents and tests my assumption that the string passed in as `thisGuess` is

exactly `howManyPositions` long. The `ASSERT` on line 6 does the same for the solution. Finally, the `ASSERT` on line 21 documents and tests my assumption that the number in the correct position can never be greater than the number of `correct` letters.

Through the Program Once, by the Numbers

Listing 5.14 provides the complete listing of this program. Let's walk through one round, line by line.

Listing 5.14 Using ASSERT (DefinedValues.h)

```
1:    #ifndef DEFINED
2:    #define DEFINED
3:
4:    #include <iostream>
5:    using namespace std;
6:
7:    const char alpha[] = "abcdefghijklmnopqrstuvwxyz";
8:    const int minPos = 2;
9:    const int maxPos = 10;
10:    const int minLetters = 2;
11:    const int maxLetters = 26;
12:
13:    #define DEBUG
14:
15:    #ifndef DEBUG
16:       #define ASSERT(x)
17:    #else
18:       #define ASSERT(x) \
19:                if (! (x)) \
20:                { \
21:                    cout << "ERROR!! Assert " << #x << " failed
\n"; \
22:                    cout << " on line " << __LINE__  << "\n"; \
23:                    cout << " in file " << __FILE__ << "\n";  \
24:                }
25:    #endif
26:
27:    #endif
```

Game.h

```
28:    #ifndef GAME_H
29:    #define GAME_H
30:
31:    #include "DefinedValues.h"
32:
33:    class Game
34:    {
35:    public:
36:        Game();
37:        ~Game(){}
38:        void Display(const char * charArray) const
39:        {
40:            cout << charArray << endl;
41:        }
42:    void Play();
43:    const char * GetSolution() const { return solution; }
44:    void Score(const char * thisGuess, int & correct, int &
position);
45:
46:    private:
47:        int HowMany(const char *, char);
48:        char solution[maxPos+1];
49:        int howManyLetters;
50:        int howManyPositions;
51:        int round;
52:        bool duplicates;
53:    };
54:
55:    #endif
```

Game.cpp

```
56:    #include <time.h>
57:    #include "Game.h"
58:
59:    void Game::Score(const char * thisGuess, int & > rCorrect, int
& >      rPosition)
60:    {
61:        rCorrect = 0;
62:        rPosition = 0;
63:
64:        ASSERT ( strlen(thisGuess) == howManyPositions)
```

```
65:         ASSERT ( strlen(solution) == howManyPositions)
66:         int i;
67:         for ( i = 0; i < howManyLetters; i++)
68:         {
69:             int howManyInGuess = HowMany (thisGuess, alpha[i]);
70:             int howManyInAnswer = HowMany (solution, alpha[i]);
71:             rCorrect += howManyInGuess < howManyInAnswer ?
72:                             howManyInGuess : howManyInAnswer;
73:         }
74:
75:         for (  i = 0; i < howManyPositions; i++)
76:         {
77:             if ( thisGuess[i] == solution[i] )
78:                 rPosition ++;
79:         }
80:
81:         ASSERT ( rPosition <= rCorrect)
82:
83:     }
84:
85:     Game::Game():
86:           round(1),
87:         howManyPositions(0),
88:         howManyLetters(0),
89:         duplicates(false)
90:     {
91:
92:         bool valid = false;
93:         while ( ! valid )
94:         {
95:             while ( howManyLetters < minLetters ||
96:                     howManyLetters > maxLetters )
97:             {
98:                 cout << "How many letters? (";
99:                 cout << minLetters << "-" << maxLetters << "): ";
100:                cin >> howManyLetters;
101:                if ( howManyLetters < minLetters ||
102:                     howManyLetters > maxLetters )
103:                cout << "please enter a number between ";
104:                cout << minLetters << " and " << maxLetters <<
endl;
105:            }
106:
```

```
107:            while ( howManyPositions < minPos ||
107a:               howManyPositions > maxPos )
108:            {
109:                cout << "How many positions? (";
110:                cout << minPos << "-" << maxPos << "): ";
111:                cin >> howManyPositions;
112:                if ( howManyPositions < minPos ||
112a:                            howManyPositions > maxPos )
113:                cout << "please enter a number between ";
114:                cout << minPos <<" and " << maxPos << endl;
115:            }
116:
117:            char choice = ' ';
118:            while ( choice != 'y' && choice != 'n' )
119:            {
120:                cout << "Allow duplicates (y/n)? ";
121:                cin >> choice;
122:            }
123:
124:            duplicates = choice == 'y' ? true : false;
125:
126:            if ( ! duplicates && howManyPositions >
howManyLetters )
127:            {
128:                cout << "I can't put " << howManyLetters;
128a:               cout << " letters in ";
129:                cout << howManyPositions;
130:                cout << " positions without duplicates! ";
131:                cout << Please try again.\n";
132:                howManyLetters = 0;
133:                howManyPositions = 0;
134:            }
135:            else
136:                valid = true;
137:        }
138:
139:        int i;
140:        for (i = 0; i < maxPos; i++ )
141:            solution[i] = 0;
142:
143:        srand( (unsigned)time( NULL ) );
144:
145:        for ( i = 0; i < howManyPositions; )
```

```
146:        {
147:            int nextValue = rand() % (howManyLetters);
148:            char c = alpha[nextValue];
149:            if ( ! duplicates && i > 0 )
150:            {
151:                int count = HowMany(solution, c);
152:                if ( count > 0 )
153:                    continue;
154:            }
155:            // add to the array
156:            solution[i] = c;
157:            i++;
158:        }
159:        solution[i] = '\0';
160:
161:    }
162:
163:    void Game::Play()
164:    {
165:        char guess[80];
166:        int correct = 0;
167:        int position = 0;
168:        bool quit = false;
169:
170:        while ( position < howManyPositions )
171:        {
172:
173:            cout << "\nRound " << round << ". Enter ";
174:            cout << howManyPositions << " letters between ";
175:            cout << alpha[0] << " and ";
175a:           cout << alpha[howManyLetters-1] << ": ";
176:
177:            cin >> guess;
178:
179:            if ( strlen(guess) != howManyPositions )
180:            {
181:                cout << "\n ** Please enter exactly ";
182:                cout << howManyPositions << " letters. **\n";
183:                continue;
184:            }
185:
186:
187:            round++;
```

```
188:
189:            cout << "\nYour guess: ";
190:            Display(guess);
191:
192:            Score(guess,correct,position);
193:            cout << "\t\t" << correct << " correct, ";
194:            cout << position << " in position." << endl;
195:        }
196:
197:        cout << "\n\nCongratulations! It took you ";
198:
199:        if ( round <= 6 )
200:        cout << "only ";
201:
202:        if ( round-1 == 1 )
203:            cout << "one round!" << endl;
204:        else
205:            cout << round-1 << " rounds." << endl;
206:    }
207:
208:
209:    inline int Game::HowMany(const char * theString, char c)
210:    {
211:        int count = 0;
212:        for ( int i = 0; i < strlen(theString); i++)
213:        if ( theString[i] == c )
214:            count ++;
215:        return count;
216:    }
```

Decryptix.cpp

```
217:    #include "DefinedValues.h"
218:    #include "Game.h"
219:
220:    int main()
221:    {
222:        cout << "Decryptix. Copyright 1999 Liberty ";
223:        cout << "Associates, Inc. Version 0.3\n\n" << endl;
224:        bool playAgain = true;
225:
226:        while ( playAgain )
227:        {
```

```
228:            char choice = ' ';
229:            Game theGame;
230:            theGame.Play();
231:
232:            cout << "\nThe answer: ";
233:            theGame.Display(theGame.GetSolution());
234:            cout << "\n\n" << endl;
235:
236:            while ( choice != 'y' && choice != 'n' )
237:            {
238:                cout << "\nPlay again (y/n): ";
239:                cin >> choice;
240:            }
241:
242:            playAgain = choice == 'y' ? true : false;
243:        }
244:
245:        return 0;
246:    }
```

We begin by loading this program in the debugger and placing a break point on line 222, as illustrated in Figure 5.7. Your particular debugger might look somewhat different, but the essentials are probably the same. Choose Run to break point (in Microsoft Visual Studio, this is F5).

Figure 5.7 *Examining a break point.*

By the time the program has stopped at this break point, it has loaded the two header files, definedValues.h and Game.h.

Loading `definedValues` brings us to line 1, where the inclusion guards are checked and we find that `DEFINED` has not yet been defined. Thus, the body of `definedValues` is read, which brings in `iostream` (line 4) and which declares (line 5) that we are using the standard namespace.

The global constants are defined on line 7, and `ASSERT` is defined on line 18.

Including Game.h brings us to line 31, where we attempt to include `definedValues`. This brings us back to line 1, where the inclusion guards protect us by determining that `DEFINED` has already been defined, so the rest of definedValues.h is ignored. Returning to line 33, we find the declaration of the `Game` class.

The constructor and destructor are declared lines 36 and 37. Lines 38-41 are the `Display` routine, which prints to the screen any character array that is passed in. On line 42 is the heart of the class: the

`Play()` method. On line 43 I've added a new method, `GetSolution()`, which simply returns the solution as a pointer to a constant character--exactly what is needed by `Display()`. Finally, On line 44) we see the `Score()` method, which takes an array of characters (passed in by reference as a pointer to a constant character), and two references to integers.

NOTE: Part of the private interface--not exposed to the public but used by the class's methods to aid in implementation--are several state variables (such as `howManyLetters` on line 49 and `howManyPositions` on line 50) that indicate which round we're playing (on line 51) and whether we're allowing duplicates (on line 52).

In addition, line 48 shows the array that holds the solution to the game; line 47 shows a helper function, which is used by other methods of this class to determine how many instances of a particular character are found within any array of characters. You'll see how all the methods work as we step through the code.

Our break point on line 222 causes the program to stop before it prints to the screen. See your debugger's documentation for how to *step over* a function call; in Microsoft's debugger it is **F10**. Pressing step-over causes the copyright notice to print, and then the Boolean value. `playAgain` is initialized to `true`. This is used in the test on line 226, and of course this test passes because the value was just initialized one line earlier.

This brings us into the body of the `while` statement, where we create an instance of a `Game` object on line 229. This causes program flow to jump to the constructor of the `Game` object on line 85. We see that the member variables are initialized, and we enter the `while` loop on line 93. On lines 95 and 96 we test whether `howManyLetters` (initialized to `0`) is less than `minLetters` (set in definedValues. h to `2`). Because this proves true, the second half of the `OR` statement (`howManyLetters > maxLetters`) is not even evaluated; instead, we enter the `while` loop on line 98.

The user is prompted to enter how many letters he'll be guessing in this instance of the game. We'll choose `4`; that value is stored in the member variable `howManyLetters` on line 100.

On line 101 we test to ensure that we have a legal value; if not, we print a reminder to the player. Program flow loops back up to line 95, where the value is checked; if we have a valid value we proceed on line 107, where the same logic is applied to the number of positions. We'll choose `3`.

On line 121 we prompt the user to tell us whether he wants to allow duplicates. Note that this is not robust code: If the user enters *Y* rather than *y* (that is, uppercase rather than lowercase), the `while` statement continues to prompt him until he gets it right. We'll fix that up in the next version. For now,

we'll enter `n`.

On line 124 we test the value that is received; if it is `'y'`, we set `duplicates` to `true`; otherwise, we set it to `false`. In this case, we set it to `false` because we've entered `'n'`.

Take a look at the member variables, as shown in Figure 5.8.

Figure 5.8 *Examining member variables.*

Notice, in the variables window in the lower-left corner, that `choice` has the value `'n'`, and in the watch window in the lower-right, that you have `howManyLetters 4`, and `howManyLetters 3`. Also note, in the variables window, that `duplicates` is shown as `0`. My debugger cannot handle bools, so it shows `true` as `1` and `false` as `0`. This is legal in C++ (`0` does evaluate `false` and all other integers evaluate `true`), but it might be better if the debugger showed the actual Boolean value.

NOTE: These images are from the Visual C++ debugger. In other environments you may find a different display, but you should be able to see the same values and information.

On line 126 we test the logic of the user's choices. If he asks for three letters in four positions without duplicates, we point out that this is impossible.

On line 140 we iterate through the entire array, setting every member to zero. It is interesting to put `solution` into a watch window in the debugger and step through this loop watching as each offset member of the array is changed to zero. Note that this is zero the numeric value, not *0* the character. In C ++, 0 is the value of `NULL`, so this loop sets our character array to contain nothing but `NULL`s.

At line 143 we use `srand` to seed the random number generator. You might find it interesting to step into the call to `time`, but this is not relevant to our discussion here.

On line 145 we begin the work of populating the `solution` array. First, the local counter variable `i` is initialized to zero.

NOTE: You might find that many C++ programmers use the variables `i`, `j`, `k`, `l`, and `m` as counter variables in `for` loops, and many can't even tell you why. Why not `a`? Why not `counter`?

> This is a perfect example of historical anachronisms living on past the time they make any sense. Back in the ancient days of mainframe computing, early versions of FORTRAN (FORmula TRANslator) used only the variables `i`, `j`, `k`, `l`, and `m` as legal counting variables. My second computing language was FORTRAN IV, which I learned in high school in 1971. ("You had zeros? We had to use *o*s!") Old habits die hard.
>
> Just as an aside, my first programming language was Monrobot machine language (*1*s and *0*s), which we programmed using paper tape. The computer on which we ran this also had an assembler called QuickComp, which was used by the programming students. To use QuickComp, you had to load a machine language "loader" by running the appropriate tape before running your program. A few of us hacked the QuickComp tape so that on loading it printed `go away, I'm sleeping` and then shut down the system. In those days, programming, and my sense of humor, were a lot simpler.

On line 147 we examine the result of applying the modulus operator to the result of calling `rand()` and `howManyLetters`. If you want to see this at work, rewrite this line as follows:

```
// int nextValue = rand() % (howManyLetters);
int randResult = rand();
int nextValue = randResult % (howManyLetters);
```

This way you can see the result from `rand()` (stored in `randResult`), and then the effect of the modulus operator.

The first time I ran this on my machine, `randResult` was 17,516. I note that `howManyLetters` is `4`. 17,516 divided by 4 is equal to exactly 4,379. Thus, there is no remainder, so the value that is returned by the modulus operator is `0`.

On line 148 the character variable `c` is set to the letter at offset 0 in alpha (`'a'`).

On line 149 we test the value of `duplicates` (in this case, `false`) and whether `i` is greater than zero. In this case, `i` is zero, so the `if` statement is skipped. On line 156 `solution[0]` is set to `a`. Then `i` is incremented to `1` and is compared with `howManyPositions` at 41. (Notice that we do not do the increment in the body of the `for` loop.) This is because we only want to increment `i` if we get to line 156. We'll see the alternative in just a moment.

On line 147 we generate `nextValue` again. On my computer this generates a `randResult` of `14846` and a `nextValue` of `2`. Does this make sense? `howManyLetters` is `4`. It turns out that 14,846 divided by 4 is 3,711, with a remainder of 2. (3,711 times 4 is 14,844). Thus the modulus operator returns `2`, and the character `c` is assigned `alpha[2]` or `c`.

This time the `if` statement at line 149 returns `true`, and we enter the body of the `if` statement. On line 151 we assign the result of calling `HowMany` to the variable `count`, passing in the solution array and the letter `c`.

Program flow branches to line 209. The array is now represented as a pointer. On line 211 the local variable count is initialized to zero. On line 212 we iterate through the string that is passed in (the solution), and each time through the loop we test whether the value at the current offset is equal to the character that is passed in .

This time, `strlen(theString)` is `1`. You can test this by inserting a line between, rewriting line 212 as follows:

```
int stringLength = strlen(theString);
for ( int i = 0; i < stringLength; i++ )
```

C++ programmers snicker at this kind of code, with lots of temporary variables, but I've come to believe strongly that this is the right way to do things. By introducing the temporary variable `stringLength`, you can examine this value in the debugger in a way that is not possible with the more compact version of this code (in which `strlen` is used in the `for` loop test).

We see that this first time `stringLength` is `1`, so the `for` loop runs only once. Because `theString [0]` is `'a'` and our character `c` is `'c'`, the `if` statement fails and `count` is not incremented. The `for` loop then ends, and the value `0` is returned. Program flow now resumes at the line immediately following line 151, where the returned value (`count`) is tested. Because it is not greater than zero, the `continue` statement does not execute, and program flow continues on line 156 where the character `'c'` is added to the array and, once again, `i` is incremented.

The third time through the `for` loop at on line 145, my computer generates the value `5,092`, which is also exactly divisible by four, returning a nextValue of `0` and a character of `'a'`. This time, when we enter `howMany`, the character matches, so the counter is incremented and the value `1` is returned from `howMany`. In this case, when the flow resumes at the line just after the call to `howMany` on line 151, the `if` statement returns `true` (1 is greater than 0), so the `continue` statement executes. This causes the program flow to immediately return to line 145, where we will generate and test a new value.

This is why you don't want to increment `i`: After all, you have not yet put a value into `solution[2]`. Thus, `i` remains at 2, but the call on line 147 generates a different value; this time, the value is `1,369`, which sets `nextValue` to `1` and the character value `c` to `'b'`. Because `'b'` does not yet appear in our array, it is added, and `i` is incremented. `i` is now `3`, the test on line 145 fails (3 is not less than 3), and we fall through to line 159 where `solution[3]` is set to null.

The result of all this is that `solution` looks like this:

```
solution[0]: 'a'
solution[1]: 'c'
solution[2]: 'b'
solution[3]: 0
```

The constructor now ends, and we resume on line 230 back in `main()`. This immediately calls `Play()`, so processing branches to the implementation of `Play()` on line 163. The local variables `correct` and `position` are initialized to `0` (check your local variables window in your debugger), and the user is prompted to enter a guess on line 173. That guess is stored on line 177 in the array you created on line 165.

We'll guess `abc`. This fails the test on line 179 because the string length of `guess` is 3, which is equal to `howManyPositions` and thus fails the test of not being equal. Processing skips the body of the `if` statement and continues on line 187, where the member variable `round` is incremented from zero to 1. On line 190 this guess is passed to `Display()`, where it is shown to the user; then, on line 192, it is passed into `Score()`.

You can step over `Display` (in Visual C++, press **F10**) and then into Score (in Visual C++, press **F11**) to follow those parts of the program that are of immediate interest. Stepping into `Score()` causes program flow to branch to line 59.

We immediately set the values that are referenced by `rCorrect` and `rPosition` to `0`. We then assert that our assumptions about the sizes of these arrays are correct. On line 67, we enter a `for` loop in which we'll iterate through each array, testing every possible letter and creating a count of how many are correct.

The first time in this loop, `i` is `0` and therefore passes the test of being less than `howManyLetters` (which is 3). The first call to `HowMany()` passes in the current guess (`abc`), and the letter *a* (`alpha[0]`) and returns the value `1`. The second call passes in the solution (`acb`) and the character `'a'` and also returns `1`.

The next line tests whether `howManyInGuess` (which is `1`) is less than `howManyInAnswer` (also `1`). This is false, so it returns the third part of this ternary operator: `howManyInAnswer` (which, again, is `1`). This value of `1` is added to `rCorrect`, incrementing it from `0` to `1`.

We repeat this for all three letters in the two arrays. Next, on line 75, we reset `i` to `0` and test whether `thisGuess[0]` is equal to `solution[0]`. `thisGuess[0]` is `'a'`, and `solution[0]` is also `'a'`, so `rPosition` is incremented from `0` to `1`. On the second time through the loop, `thisGuess[1]` is `'b'`, but `solution[1]` is `'c'`, so `rPosition` is not incremented. On the third time through, `thisGuess[2]` is `'c'` and `solution [1]` is `'b'`, so again, `rPosition` is not incremented.

There is no need to return a value from this method (which is why it is marked `void`) because the variables `rPosition` and `rCounter` are references to the local variables back in `Play()`. When we return from `Score()`, these values are printed and we see that correct is `3` and position is `1`.

This returns us to the top of the `while` loop on line 170; `position` (now `1`) is compared with `howManyPositions` (currently `3`). Because it is still smaller, we're not yet done, and we repeat the loop, offering the user a second guess.

This time let's guess `acb`. The score we receive is three correct and three in position, and this `while` loop ends. Program flow resumes on line 197, where we print a congratulatory message. `Play()` then returns, dropping us on line 232 in `main()`, where the answer is displayed and you are offered (on line 236) the opportunity to play again.

If you decide *not* to play again, the value `0` is returned to the operating system on line 245 and the program ends.

On line 230 we invoke the `Play()` method.

This causes program flow to jump to line 163. To see this, step into this method from line 230. Your debugger brings you to line 163. A few local variables are created and initialized, and then on line 170 we check the value of `position` (which is zero) to see if it is less than `howManyPositions`.

Contents

C++ From Scratch

Contents

6

Using Linked Lists

The problem with using arrays is that you must declare their size at compile time rather than at runtime.

---

compile time--When you compile the program

runtime--When you run the program

This means that you must guess, in advance, how much memory you need to allocate. If you guess wrong and you allocate too little, you run out of room and your program breaks. If, on the other hand, you allocate more than you need, you waste memory.

In Decryptix! this isn't a very big problem because we create only two arrays: one to hold the solution and one to hold the guess. We can just create a pair of arrays large enough to hold the biggest legal solutions and the largest possible guess, and let it go at that.

In other programs, however, fixed size arrays are so wasteful of memory as to be unusable. Software designers are often asked to consider how their program will *scale:* How will they perform as they become larger and handle more data? Programs that use fixed size arrays rarely scale well.

Scaling a program refers to the capability to do more: to handle larger and more complex data sets, more users, or more frequent access. When a program scales, it becomes bigger and typically more complex, and all the weaknesses in performance and maintainability surface.

To solve the problem of fixed size arrays, we need the capability to store data in some form of data structure or collection that grows *dynamically*, which means that it grows as it needs to while the program runs.

Dynamic Data Structures

Over the years, computer scientists have struggled with this issue. In the past, procedural programmers created complex data structures to hold data efficiently. Object-oriented programmers talk about *collection classes*, classes that are designed to hold collections of other objects.

---

collection class--A class designed to hold a collection of other objects

---

Collection classes are built on top of traditional data structures as higher-level abstractions, but the problem they are solving is the same: How do we efficiently deal with large sets of data or objects?

We need a variety of collection classes because our needs and priorities differ from program to program. Sometimes we care about adding objects to the collection quickly. Other times, we don't mind if there is a slight delay adding objects, but we want the capability to find objects quickly. In other programs, the emphasis is on using little memory or little disk space.

The Standard Template Library

The C++ Standard Library now offers a suite of *collection classes* called the Standard Template Library (STL), which is described in coming chapters. The STL classes are designed to hold collections of objects, including built-in objects such as characters and more complex (and dramatically larger) user-defined objects. Most importantly, the STL code has been optimized, debugged, and tested so that you don't have to do this work yourself.

Before considering the STL in detail, however, it is helpful to create our own rather simple collection class, at least once, to see what is involved.

We'll rewrite Decryptix! to use a *linked list* rather than an array. A linked list is a very simple data structure that consists of small containers that can be linked together as needed, and each of which is designed to hold one object.

---

linked list--A simple data structure in which each element in the list points to data and to the next element in the list

---

Each individual container is called a *node*. The first node in the list is called the *head*, and the last node in the list is called the *tail*.

---

node--An element in a data structure

head--The first node in a linked list

tail--The last node in a linked list

Lists come in three fundamental forms. From simplest to most complex, they are

- Singly linked
- Doubly linked
- Trees

In a singly linked list, each node points forward to the next one, but not backward. To find a particular node, start at the top and go from node to node, as in a treasure hunt ("The next node is under the sofa"). A doubly linked list enables you to move backward and forward in the chain. A tree is a complex structure built from nodes, each of which can point in two or three directions. Figure 6.1 shows these three fundamental structures.

Figure 6.1 *Singly linked, doubly linked, and tree structures.*

Linked Lists

We'll build the simplest form of linked list, a singly linked list that is not sorted. Characters are added in the order in which they are received (just as they are in an array).

We'll actually create the linked list three times. The first time we'll take a rather simplistic, traditional approach just to get a good sense of how a linked list works. The second time we'll design a somewhat more object-oriented linked list and see whether we can add some solid object-oriented design heuristics to our solution. Finally, we'll use the linked list to illustrate the concept of abstract data types.

Understanding Linked Lists

Our simplest linked list consists of nothing but nodes. A node is a tiny object with two members. The first member is a pointer to the thing we're storing, in our case a single character. The second member is a pointer to another node. By stringing nodes together, we create a linked list.

When there are no more nodes in the list, the last node points to `NULL`. Figure 6.2 shows what our linked list looks like. The first node in the list (the head node) points to its data (A) and also to the second node

in the list. This second node in turn points to its data and also to the third node. The third node is the tail node, and it points to its data and to null, signifying that there are no more nodes in the list.

Figure 6.2 *Simple linked list.*

Let's implement this linked list and then see how we might use it, instead of an array, to hold our solution. To get started, however, we need only create the `Node` class and fill it with a list of characters. Listing 6.1 has the declaration for our `Node` class.

NOTE: During the development of a program, I'm often confronted with a new technology, in this case the linked list. Rather than trying to figure out how to use it in my program while also figuring out how to implement it, I usually first implement the technology with a simple *driver program*. That is, I'll take it out of context and create a very simple program that does nothing but exercise the new technology. After it is working, I'll go back and integrate it into the real program.

Listing 6.1 The Node Class Declaration

```
0:  class Node
1:  {
2:  public:
3:      Node(char c);
4:      ~Node();
5:      void      Display        ()         const;
6:      int        HowMany        (char c)    const;
7:      void    Insert        (char c);
8:
9:  private:
10:      char    GetChar        ();
11:      Node *    GetNext        ();
12:      char myChar;
13:      Node *    nextNode;
14:  };
```

Let's start by looking at the constructor on line 3. A node is created by passing in a character by value. Rather than keeping a pointer to the character, our `Node` class keeps a copy of the character on line 12. With a tiny one-byte object, this is sensible. With larger objects, you'll want to keep a pointer to avoid making a copy.

NOTE: In C++, pointers are typically 4 bytes. With a 1-byte object, it is cheaper to keep a copy (one byte of memory used) than to keep a pointer (4 bytes of memory used). With large user-defined types, it can be far more expensive to make the copy, in which case a pointer or reference is used.

`Node` provides three methods in addition to its constructor and destructor. On line 5 we see `Display()`, whose job it is to print the characters that are stored in the list. The method `HowMany()` also takes a character and returns the number of times that character appears in the list. Finally, `Insert()` takes a character and inserts it into the list. Listing 6.2 shows the implementation of these simple methods.

Listing 6.2 Implementing Node

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  #include "Node.h"
4:
5:  Node::Node(char c):
6:   myChar,nextNode(0)
7:  {
8:  }
9:
10:  Node::~Node()
11:  {
12:      if ( nextNode )
13:          delete nextNode;
14:  }
15:
16:
17:  void Node::Display() const
18:  {
19:      cout << myChar;
20:      if ( nextNode )
21:          nextNode->Display();
22:  }
23:
24:
25:  int Node::HowMany(char theChar) const
```

```
26:  {
27:      int myCount = 0;
28:      if ( myChar == theChar )
29:          myCount++;
30:      if ( nextNode )
31:          return myCount + nextNode->HowMany(theChar);
32:      else
33:          return myCount;
34:  }
35:
36:  void Node::Insert(char theChar)
37:  {
38:      if ( ! nextNode )
39:          nextNode = new Node(theChar);
40:      else
41:          nextNode->Insert(theChar);
42:  }
```

The constructor on line 5 receives a character and initializes its `myChar` member variable on line 6. The constructor also initializes its `nextNode` pointer to zero--that is, to null. When the `Node` is created, it points to nothing further along in the list.

The destructor on line 10 tests the pointer on line 12, and if the pointer is not `NULL`, the destructor deletes it.

NOTE: The destructor takes advantage of the C++ idiom that any nonzero value is considered `true`. Thus, if `nextNode` is null, its value is `0` and, therefore, `false`, and the `if` statement does not execute. If `nextNode` does point to another node, its value is nonzero and thus `true`, and that object is deleted.

`Display()`, on line 17, prints the character that is held by the current node on line 19, and then calls `Display()` on the `nextNode` in the list, if any (on line 20). In this way, by telling the first node to display itself, you cause every node in the list to display itself.

A Simple Driver Program

On line 25, `HowMany()` takes a character and returns the number of times that character exists in the list. The implementation of this is tricky and instructive because this type of implementation is common

in C++. Explaining how this works in words is much less effective than tracing it in the debugger. To do that, we need a driver program, shown in Listing 6.3.

Listing 6.3 Driver Program

```
0:  #include "DefinedValues.h"
1:  #include "List0601_Node.h"
2:
3:  int main()
4:  {
5:      Node head('a');
6:      head.Insert('b');
7:      int count = head.HowMany('a');
8:      cout << "There are " << count << " instances of a\n";
9:      count = head.HowMany('b');
10:      cout << "There are " << count << " instances of b\n";
11:      cout << "\n\nHere's the entire list: ";
12:      head.Display();
13:      cout << endl;
14:
15:      return 0;
16:  }
There are 1 instances of a
There are 1 instances of b
Here's the entire list: ab
```

Before we examine `HowMany`, let's look at the driver. Its job is to generate two letters and add them to the list. To do this, it creates a first node, called the *`head` node*, on line 5, and initializes it with the value `'a'`. It then tells the `head` node to insert one more letter (`'b'`), starting on line 6.

Our linked list now looks like Figure 6.3.

Figure 6.3 *With two nodes.*

On line 0 we `#include` DefinedValues.h, shown in Listing 6.4.

Listing 6.4 DefinedValues.h

```
1:    #ifndef DEFINED
2:    #define DEFINED
3:
4:    #include <iostream>
```

```
5:    #include <vector>
6:    #include <iterator>
7:    #include <algorithm>
8:    #include <time.h>
9:    #include <utility>
10:
11:    using namespace std;
12:
13:    const char alpha[27] = "abcdefghijklmnopqrstuvwxyz";
14:
15:    const int minPos = 2;
16:    const int maxPos = 10;
17:    const int minLetters = 2;
18:    const int maxLetters = 26;
19:    const int SecondsInMinute = 60;
20:    const int SecondsInHour = SecondsInMinute * 60;
21:    const int SecondsInDay = SecondsInHour * 24;
22:    const int GUESSES_PER_SECOND = 10000;
23:
24:    const int szInt =  sizeof(int);
25:    const int szChar = sizeof(char);
26:    const int szBool = sizeof(bool);
27:
28:    #endif
```

This file serves to include the necessary STL header files, and also to define constants we'll need in this program. We will use this same defineValues file throughout all the sample code for the rest of the book.

Let's not examine how `Insert()` works just yet, but rather assume that the letter *b* is in fact inserted into the list. We'll return to how this works in just a moment, but let's first focus on `HowMany()` works.

The HowMany() Method

On line 7 we ask how many instances of `'a'` there are, and on line 9 we ask the same about how many instances of `'b'` there are. Let's walk through the call to `howMany` on line 9. Put a break point on this line, and run the program to the break point.

The program runs as expected and stops at the following line:

```
count = head.HowMany('b');
```

Stepping into this line of code brings you to the top of `HowMany()`:

```
int Node::HowMany(char theChar) const
{
```

Let's step line by line. The first step initializes `myCount` to `0`, which you can probably see in the local variables window of your debugger.

Which node are we looking at? We'll be entering the `HowMany()` method once for each node. How can we tell where we are? Remember that every nonstatic member method has a pointer called the `this` pointer, which holds the address of the object itself.

You can examine the value of the `this` pointer in your debugger. Take note of the address of the `this` pointer while you are here in `HowMany()`. On my machine, it is `0x0012ff6c`, but yours might be different. The particular value doesn't matter--just write down whatever you have. This is the address of the node we're examining.

Step to the next line, where `myChar` is compared with `theChar`. Examine the `myChar` member variable (`'a'`) and the local variable `theChar` (`'b'`), again in your local variables window.

NOTE: You might need to expand your `this` pointer to see the member variables, or you might need to click on a different debugger window to find them.

Clearly, these values are not the same, so the `if` statement fails. `myCount` remains at `0`.

Step again to the next `if` statement. The `nextNode` pointer should be nonzero. On my machine, it is `0x004800a0`. Your value will differ; again, although the absolute value doesn't matter, write down whatever you get.

Because `nextNode` is nonzero, the `if` statement evaluates `true`, and you step to the following line:

```
        return myCount + nextNode->HowMany(theChar);
```

What do you expect to happen if you step *into* this line? The first thing to be evaluated is

```
nextNode->HowMany(theChar);
```

This calls the `howMany()` method through the `nextNode` pointer. This, in fact, calls `howMany()` on

the object to which `nextNode` points. Remember that `nextNode` had a value, the address of the next node in the list. Let's step in.

The debugger appears to go to the top of the same method. Where are we? Examine the `this` pointer in your debugging window (you might first have to step to the first line of the method). On my machine, the `this` pointer has changed to `0x004800a0`, which was exactly the value previously held in the `nextNode` pointer. Aha! We're now in the second node in the list. We can imagine that our list looks like Figure 6.4.

Figure 6.4 *Nodes with addresses.*

We are running `HowMany` in the *second* node. Once again, `HowMany()` begins by initializing the local variable `myCount`, on line 27, to `0`. Be careful here, the `myCount` we're looking at now is local to this iteration of `HowMany()`. The `myCount` back in the first node has its own local value.

`HowMany()` then tests the character that is passed in against the character it is storing on line 28; if they match, it increments the counter. In this case, they do match, so we compare `myChar` (`'b'`) with `theChar` (also `'b'`). They match, so `myCount` is incremented.

Stepping again brings us to the next `if` statement:

```
30:     if ( nextNode )
```

This time `nextNode` is `NULL` (you should see all zeros in its value in your local variables window). As expected, the second node's `nextNode` points to `NULL`. The `if` statement fails and the `else` statement executes, returning `myCount`, which has the value `1`.

We step into this line and appear to be right back at the `return` statement. Examine the `this` pointer, however, and you'll find that we're back in the first node. The value that is returned (`1`) is added to the value in `myCounter` (now `0`), and it is this combined value that is returned to the calling function, `main()`.

As an exercise, try revising `main()` to insert the values `a`, `b`, `c`, `b`, and `b`. This produces the linked list that is shown in Figure 6.5. Make sure you understand why `HowMany()` returns the value `3` when passed in `'b'`.

Figure 6.5 *Linked list with `abcbb`.*

Insert() in Detail

Now is the time to examine the implementation of `Insert()`, as shown on line 36 in Listing 6.2 and

reproduced here for your convenience:

```
36:  void Node::Insert(char theChar)
37:  {
38:      if ( ! nextNode )
39:          nextNode = new Node(theChar);
40:      else
41:          nextNode->Insert(theChar);
42:  }
```

The goal of this method is to insert a new character (`theChar`) into the list.

Note that on line 39 we use the keyword `new` to create a new `Node` object. This is explained in full in just a few pages; for now, all you need to know is that this creates a new object of type `Node`.

Let's start over, creating the linked list from Figure 6.5, using the code shown in Listing 6.5.

Listing 6.5 Decryptix Driver Program

```
0:  #include "DefinedValues.h"
1:  #include "List0601_Node.h"
2:
3:  int main()
4:  {
5:      Node head('a');
6:      head.Insert('b');
7:      head.Insert('c');
8:      head.Insert('b');
9:      head.Insert('b');
10:      int count = head.HowMany('a');
11:      cout << "There are " << count << " instances of a\n";
12:      count = head.HowMany('b');
13:      cout << "There are " << count << " instances of b\n";
14:      cout << "\n\nHere's the entire list: ";
15:      head.Display();
16:      cout << endl;
17:
18:      return 0;
19:  }
There are 1 instances of a
There are 3 instances of b
Here's the entire list: abcbb
```

On line 5 we create the first `Node` object, which we call `head`. Set a break point on that line and run to the break point. Stepping in takes you to the constructor of the `Node` object:

```
 Node::Node(char c):
myChar,nextNode(0)
{
}
```

This does nothing but initialize the member variables. We now have a node whose `myChar` character variable contains `'a'` and whose `nextNode` pointer is `NULL`.

Returning to `main()`, we step into the call to

```
    head.Insert('b');
```

Step into this code from Listing 6.2, which is once again reproduced for your convenience:

```
36:  void Node::Insert(char theChar)
37:  {
38:      if ( ! nextNode )
39:          nextNode = new Node(theChar);
40:      else
41:          nextNode->Insert(theChar);
42:  }
```

On line 38 we test to see whether `nextNode` is `NULL`. In this case it is, so we must create a new node. The last time we created a node, we simply declared it and passed in the value to store (`'a'`). This time we do something different, calling the `new` operator. Why?

Until now, all the objects you've created were created on the stack. The stack, you'll remember, is where all local variables are stored, along with the parameters to function calls. To understand why creating your new node object on the stack won't work, we need to talk a bit more about what the stack is and how it works.

Excursion: Understanding the Stack

The stack is a special area of memory that is allocated for your program to hold the data required by each of the functions in your program. It is called a stack because it is a *last-in, first-out* (LIFO) queue, much like a stack of dishes at a cafeteria (see Figure 6.6).

Figure 6.6 *A LIFO stack.*

LIFO means that whatever is added to the stack last will be the first thing that is taken off. Other queues are more like a line at a theater, which is called *first in, first out* (*FIFO*): The first one on line is the first one off.

LIFO--Last in, first out, like plates on a stack

FIFO--First in, first out, like people on line to buy tickets at a theater

Interestingly, most airplanes board and unboard coach like a FIFO stack. The people at the rear of the plane are the first to board and the last to get off. Of course, first class is a FIFO structure--first class passengers are the first ones in and the first ones out.

When data is pushed onto the stack, the stack grows; as data is popped off the stack, the stack shrinks. It isn't possible to pop a dish off the stack without first popping off all the dishes placed on after that dish, and it isn't possible to pop data off a stack without first popping all the data added above your data.

A stack of dishes is a fine analogy as far as it goes, but it is fundamentally wrong. A more accurate mental picture is of a series of cubbyholes, aligned top to bottom. The top of the stack is whatever cubby the stack pointer happens to be pointing to. The *stack pointer* is just a pointer whose job is to keep track of the top of the stack.

---

stack pointer--A pointer that keeps track of the top of the stack

---

Each of the cubbies has a sequential address, and one of those addresses is kept in the stack pointer register. Everything below that magic address, known as the top of the stack, is considered to be on the stack. Everything above the top of the stack is considered to be off the stack, and therefore invalid. Figure 6.7 illustrates this idea.

Figure 6.7 *The instruction pointer*.

When data is put on the stack, it is placed into a cubby above the stack pointer, and then the stack pointer is moved up to indicate that the new data is now on the stack.

When data is popped off the stack, all that really happens is that the address of the stack pointer is changed because it moves down the stack. Figure 6.8 makes this rule clear.

Figure 6.8 *Moving the stack pointer*.

The Stack and Functions

Here's what happens when a program that is running on a PC under DOS branches to a function:

1. The address in the instruction pointer is incremented to the next instruction past the function call. That address is then placed on the stack, and it will be the return address when the function returns.

2. Room is made on the stack for the return type you've declared. On a system with two-byte integers, if the return type is declared to be `int`, another two bytes are added to the stack, but no value is placed in these bytes.

3. The address of the called function, which is kept in a special area of memory that is set aside for that purpose, is loaded into the instruction pointer, so the next instruction executed will be in the called function.

4. The current top of the stack is noted and is held in a special pointer called the *stack frame*. Everything that is added to the stack from now until the function returns is considered local to the function.

5. All the arguments to the function are placed on the stack.

6. The instruction that is now in the instruction pointer executes, thus executing the first instruction in the function.

7. Local variables are pushed onto the stack as they are defined.

8. When the function is ready to return, the return value is placed in the area of the stack that is reserved at step 2.

9. The stack is then popped all the way up to the stack frame pointer, which effectively throws away all the local variables and the arguments to the function.

10. The return value is popped off the stack and assigned as the value of the function call itself.

11. The address that is stashed away in step 1 is retrieved and put into the instruction pointer.

12. The program resumes immediately after the function call, with the value of the function retrieved.

Some of the details of this process change from compiler to compiler, or between computers, but the essential ideas are consistent across environments. In general, when you call a function, the return address and parameters are put on the stack. During the life of the function, local variables are added to the stack. When the function returns, these are all removed by popping the stack.

For our purposes, the most important thing to note about this process is that when a function returns, all the local variables are popped off the stack and destroyed.

As was described previously, if we create the new node in `InsertNode` on the stack, when the function returns, that node is destroyed. Let's try it. We'll just change `Insert` to create a local node, and we'll stash away the address of that local `Node` in `nextNode`. Listing 6.6 illustrates the change.

WARNING: These changes compile and link, but will crash when you run the program.

Listing 6.6 Local Nodes

```
0:  void Node::Insert(char theChar)
1:  {
2:      if ( ! nextNode )
3:      {
4:          Node localNode(theChar);
5:          nextNode = &localNode;
6:      }
7:      else
8:          nextNode->Insert(theChar);
9:
```

When I run this program, it quickly crashes. Here's what happens: When we create the head `Node`, its `nextNode` pointer is null. When we call `Insert()` on the head node, the `if` statement returns `true`, and we enter the body of the `if` statement on line 4. We create a `localNode` object and assign its address to `nextNode`. We then return from `Insert`. At that moment, the stack unwinds, and the local node we created is destroyed.

Now, all that happens when that local node is destroyed is that its destructor runs and the memory is

marked as reusable. Sometime later, we might assign that memory to a different object. Still later, we might use the `nextNode` pointer and *bang!* the program crashes.

Using new

This is a classic example of when you need to create an object on the heap. Objects that are created on the heap are *not* destroyed when the function returns. They live on until you delete them explicitly, which is just what we need.

Unlike objects on the stack, objects on the heap are unnamed. You create an object on the heap using the `new` operator, and what you get back from `new` is an address, which you must assign to a pointer so that you can manipulate (and later delete) that object.

Let's look again at `Insert()` from Listing 6.2:

```
36:  void Node::Insert(char theChar)
37:  {
38:      if ( ! nextNode )
39:          nextNode = new Node(theChar);
40:      else
41:          nextNode->Insert(theChar);
42:  }
```

The logic of this code is that we test to see whether the `nextNode` pointer is pointing to an object on line 38; if it is not, we create a new object on the heap and assign its address to `nextNode`.

When we create the new object, we call `new`, followed by the type of the object we are creating (`Node`) and any parameters we need to send to the constructor (in this case, `theChar`).

If this object does point to another node, we invoke `Insert` on that object, passing along the character we're trying to solve. Eventually we reach the end of the list--a node that does not point to any other node--and we can create a new node and tag it to the end of the list.

new and delete

Many details are involved in using `new` effectively, which we'll discuss as we come to them. There is one, however, that I want to discuss immediately. When you create an object on the heap using `new`, you own that object, and you must clean it up when you are done with it. If you create an object on the heap and then lose the pointer, that object continues to use up memory until your program ends, but you can't access that memory.

When you have an object that you can't access anymore but that continues to consume memory, we say it has *leaked out* of the program. Memory leaks are of significant concern to C++ programmers. You solve memory leaks by the judicious application of `delete()`. We see this in the destructor in Listing 6.2, copied here:

```
10:  Node::~Node()
11:  {
12:      if ( nextNode )
13:          delete nextNode;
14:  }
```

When we are ready to destroy the linked list, we call `delete` on the head node (implicitly by returning from the function in which the head node was created on the stack, or explicitly if the head node was created on the heap). The destructor examines its own `nextNode`, and if the `nextNode` pointer is not null, the destructor deletes the node to which it points. This mechanism knocks down all the dominoes, each object deleting the next object as part of its own sequence of destruction.

Let's modify `main()` to create the head node on the heap, and we'll delete it explicitly when we're done with the list. To make all this clear, we'll add printouts to the constructors and destructors to see our progress. Listing 6.7 is the entire program, which we'll walk through in some detail.

Listing 6.7a Node.h

```
1:    class Node
2:    {
3:    public:
4:        Node(char c);
5:        ~Node();
6:
7:        void    Display        ()            const;
8:        int        HowMany        (char c)    const;
9:        void    Insert        (char c);
10:
11:    private:
12:        char    myChar;
13:        Node *    nextNode;
14:    };
```

Listing 6.7b[em]Node.cpp

```
15:    #include <iostream.h>
16:    #include "List0603a_Node.h"
```

```
17:
18:    Node::Node(char theChar):
19:    myChar(theChar),nextNode(0)
20:    {
21:        cout << "In constructor of Node(" << this << ")\n";
22:    }
23:
24:    Node::~Node()
25:    {
26:        cout << "In destructor of Node(" << this << ")\n";;
27:        if ( nextNode )
28:            delete nextNode;
29:    }
30:
31:
32:    void Node::Display() const
33:    {
34:        cout << this << ": " << myChar << endl;
35:        if ( nextNode )
36:            nextNode->Display();
37:    }
38:
39:
40:    int Node::HowMany(char theChar) const
41:    {
42:        int myCount = 0;
43:        if ( myChar == theChar)
44:            myCount++;
45:        if ( nextNode )
46:            return myCount + nextNode->HowMany(theChar);
47:        else
48:            return myCount;
49:    }
50:
51:    void Node::Insert(char theChar)
52:    {
53:        if ( ! nextNode )
54:            nextNode = new Node(theChar);
55:        else
56:            nextNode->Insert(theChar);
57:    }
```

Listing 6.7c Driver.cpp

```
58:    #include <iostream.h>
59:    #include "List0603a_Node.h"
60:
61:    int main()
62:    {
63:        Node * pHead = new Node('a');
64:        pHead->Insert('b');
65:        pHead->Insert('c');
66:        pHead->Insert('b');
67:        pHead->Insert('b');
68:        int count = pHead->HowMany('a');
69:        cout << "There are " << count << " instances of a\n";
70:        count = pHead->HowMany('b');
71:        cout << "There are " << count << " instances of b\n";
72:        count = pHead->HowMany('c');
73:        cout << "There are " << count << " instances of c\n";
74:        cout << "\n\nHere's the entire list:\n";
75:        pHead->Display();
76:        cout << "Deleting pHead..." << endl;
77:        delete pHead;
78:        cout << "Exiting main()..." << endl;
79:        return 0;
80:    }
81:    In constructor of Node(0x00430060)
82:    In constructor of Node(0x00431DB0)
83:    In constructor of Node(0x00431D70)
84:    In constructor of Node(0x00431D30)
85:    In constructor of Node(0x00431CF0)
86:    There are 1 instances of a
87:    There are 3 instances of b
88:    There are 1 instances of c
89:
90:
91:    Here's the entire list:
92:    0x00430060: a
93:    0x00431DB0: b
94:    0x00431D70: c
95:    0x00431D30: b
96:    0x00431CF0: b
97:    Deleting pHead...
98:    In destructor of Node(0x00430060)
99:    In destructor of Node(0x00431DB0)
```

```
100:    In destructor of Node(0x00431D70)
101:    In destructor of Node(0x00431D30)
102:    In destructor of Node(0x00431CF0)
103:    Exiting main()...
```

The best way to follow the progress of this code is to put the code in the debugger and set a break point in `main()` at line 63. Run the program to the break point and note that we are going to create a new node and initialize it to hold the letter `'a'`. The address of this new node is stashed away in the `Node` pointer `pHead`. Now, step in at line 63. You might step into the `new` operator. If so, just step back out and step in again, which brings you to the constructor for `Node` on line 18.

Sure enough, the character `'a'` was passed in (now designated `theChar`), and this character is used to initialize the member variable `myChar`. The node's second member variable, `nextNode`, which is a pointer to a `Node` object, is also initialized with the value `0` or `NULL`. Finally, as you step through the constructor, a message is printed on line 21, the effect of which is shown on line 81.

Notice that the `this` pointer is not dereferenced, so its actual value is printed: That is, the address of the `Node` object on the heap whose constructor we are now discussing.

If you continue stepping, you'll return from the constructor back to `main()` on line 64, where we intend to call the `Insert()` method on that `Node`. We do so indirectly, using the pointer that holds its address, and we pass `'b'` to the `Insert` method in the hope that a new `Node` will be created and appended to the list to hold this new value.

Step in at on line 30 and you enter the `Insert()` method of `Node` on line 51, where the parameter `theChar` holds the value `'b'`. On line 53 you test the `Node`'s `nextNode` pointer, which is `NULL` (or `0`), the value to which you initialized it just a moment ago. The `if` statement returns `true`. Take a moment and reflect on why.

If a pointer has a nonzero value, it evaluates `true`. With a `0` value, it evaluates `false`. Thus, asking if a pointer is false is the same as asking if it is null.

The `not` operator turns false to true. Thus, `(! nextNode)` will evaluate `true` if `nextNode` is zero (`false`). Thus

```
if ( ! nextNode )
```

will evaluate `true` and the `if` statement will execute as long as `nextNode` points only to `NULL` (zero).

To most programmers, this is so idiomatic that

`if ( ! nextNode )`

really means "`if nextNode` doesn't yet point to anything..." and we don't much think through all the convoluted logic that makes that work.

In any case, the `if` statement executes by calling `newNode`, passing in `theChar`, and assigning the address that results to `nextNode`. Calling `new` immediately invokes the constructor, so stepping into this line brings us to the `Node` constructor on line 18. Once again, a message is printed on line 82), and we return from the constructor, assigning the address that is returned from `new` to the `nextNode` pointer of the first node.

Before leaving `Insert`, let's examine the `this` and the `nextNode` pointers. You should find that `this` has an address that is equal to the first address printed because we are now back in that first node. You should find that the `nextNode` pointer has the address of the object we just created. Sure enough, we now have a linked list with two objects in it.

Continuing causes us to return from `Insert()` back to `main()`, where the same logic is repeated to insert `'c'`, `'b'`, and once again `'b'`.

If you don't want to work your way through the logic repeatedly, continue to step over these lines until you reach line 70. We are now ready to determine how many instances of `'b'` exist in the program.

Step into this line of code. This takes you into `Node::HowMany() on line 40`, in which the parameter `theChar` has the value `'b'`. On line 42 we'll initialize `myCount` to `0`. On line 43 we test `myChar`, which has the value `'a'`, against `theChar`, which has the value `'b'`. This test fails, so we fall through to line 45, where we test to see whether `nextNode` is nonzero; it is. This causes us to execute the body of the `if` statement:

```
return myCount + nextNode->HowMany(theChar);
```

Step into this line. You find yourself in `HowMany()` for the second node. Continue stepping. `myChar` is `'b'` this time, and it thus matches `theChar`. We increment `myCount` and test `nextNode`. Again it is nonzero, so again we step in, this time to the third node in the list.

In the third node, `myChar` is `'c'`, so it does not match `myChar`; but `nextNode` is nonzero, so we step into the fourth node.

In the fourth node, `mychar` is `'b'`, and we increment `myCount` to `1`. Why is it set to `1` and not to `2`, given that this is the second node with `'b'`? The answer is that `myCount` is local to this invocation of `HowMany()` and therefore can't know about the previous values. Again, `nextNode` is nonzero, so we now step into the fifth node.

Take a look at the `this` pointer and expand it in your local variables window. You are now looking at the local member variables for the fifth node object. `myChar` is `'b'`, and `nextPointer` is `0`. Thus, we increment `myCount`; then the test for `nextPointer` fails, so we return `myCount`.

We thus return the value `1` to the call from the fourth node. This value is added to the `myCount` variable (also `1`), summing to `2`, and this value is now returned to the third node. The third node's `myCount` is `0`, so the value `2` is now returned to the second node. Its `myCount` variable is `1`, so `3` is returned to the first node. The first node's `myCount` is `0`, so `3` is returned to `main()`.

It is very important that you understand how this was accomplished. You might find that using pencil and paper and drawing a picture (see Figure 6.9) makes this easier to understand.

Figure 6.9 *Walking the list to get the answer.*

As you continue to step out of the function calls, you'll find yourself popping up through the calls to `HowMany()`, unwinding your way from Node 5 to 4, 3, 2, and back to Node 1. Step into and out of this set of calls repeatedly until you can match what is happening in your debugger to the diagram in Figure 6.9.

When this is comfortable, continue stepping over code lines until you get to the call to `Display` on line 74. Here you are calling `Display()` on `pHead`. Step into this method call and you'll be in `Display()` for your first node on line 32. Step into the method and note the address of the `this` pointer, which confirms that you are in the first `Node` in the list. `myChar` is printed on line 34 (printing `'a'`), and the `nextNode` pointer is checked on line 45. It returns `true`, so `Display()` is called on the second node in the list.

Step into this call, and you are back at line 32. Step in and notice that the `this` pointer now reflects the address of the second node, as you'd expect. On line 34, the member variable `myChar` is printed (`'b'`), and once again we call `Display()`, this time on the third node.

This continues until the fifth node prints its value. Because the fifth node does not point to another node, the `if` statement fails, and we return through the various `Display()` method invocations, back to `main()`.

On line 77 we call `delete` on `pHead`. To see what this does, place a break point on line 24 and go until the break point. You find yourself in the destructor of the head node. On line 26 we print the address of the first (head) node, and then on line 27 we test `nextNode`, which points to the second node. We delete that object, causing us to come to the destructor for the second node, where the logic is repeated. The second node deletes the third node, the third Node deletes the fourth Node, and the fourth node deletes the fifth.

The client of the linked list, in this case `main()`, never had to call `HowMany()` or `Display()` on

any node except the head node, and it doesn't have to delete any node except the head node. The maintenance of the list is managed by the list itself. Commands such as `Display()` or `delete` are passed along the list as required, each node falling like a domino into the next in the list.

Using Our Simple Linked List in Decryptix!

We are just about ready to change the type of the member variable `solution` in the `Game` class. Until now, it has been an array; we want it to be a linked list.

Let's examine all the places we use `Solution` to see what our linked list must be able to accomplish, and whether our list of Nodes is up to the task.

Following are the lines in `Game` in which we refer to the solution:

```
theGame.Display(theGame.GetSolution());
int howManyInAnswer = howMany (solution, alpha[i]);
if ( thisGuess[i] == solution[i] )
```

That is, we must have the capability to retrieve the solution and display it, count the instances of a particular character in the solution, and retrieve the character at a given offset.

Rather than expose the workings of the `Node` object to the `Game`, I've chosen to create a small class that will serve as an interface to the nodes, which I'll call `LinkedList`. Listing 6.8 shows the declaration of the `LinkedList` class.

Listing 6.8 LinkedList Declared

```
1:
2:  class LinkedList
3:  {
4:  public:
5:      LinkedList();
6:      ~LinkedList();
7:      bool    Add             (char c, bool dupes = true);
8:      void    Display         ()        const;
9:      int        HowMany         (char c)    const;
10:      char    operator[]    (int offset);
11: private:
12:      Node *    headNode;
13:  };
```

To the client (in this case, `Game`), `LinkedList` *is* the linked list structure. The client is oblivious to the existence of nodes (note that `headNode` is private).

The best way to understand the implementation of these methods is to see them in action. Let's change `Game` to use a `LinkedList` as its `solution`, as shown in Listing 6.9.

Listing 6.9 The Game Class Declaration

```
0:  #include "List0606_LL.h"
1:
2:  class Game
3:  {
4:  public:
5:      Game            ();
6:      ~Game            ()        {}
7:      void    Display    (const LinkedList * pList) const
7a:      { pList->Display(); }
8:      const LinkedList &  GetSolution    () const { return
solution; }
9:      void    Play    ();
10:      void    Score    (const char * thisGuess, int &
10a:            correct, int & position);
11:
12:  private:
13:      int        HowMany    (const char * theString, char theChar);
14:
15:      bool        duplicates;
16:      int        howManyLetters;
17:      int        howManyPositions;
18:      int        round;
19:      LinkedList        solution;
20:  };
```

`Game` is unchanged except for the last line, where the `solution` member variable is changed to type `LinkedList`.

NOTE: When one class contains another, as `Game` contains `LinkedList`, it can do so by value or by reference. `LinkedList` contains `Node` by reference (see Figure 6.10).

Figure 6.10 *Containing the node by reference.*

NOTE: `Game`, on the other hand, contains `LinkedList` by value and is diagrammed in the UML as shown in Figure 6.11. The filled in diamond indicates *by value*.

Figure 6.11 *Containing linked list by value.*

Run it!

Let's run through one play of Decryptix! and see how the `LinkedList` is used. Our driver program is unchanged, as shown in Listing 6.10. We begin by instantiating a `Game` object, which brings us into `Game`'s constructor as shown in Listing 6.11.

When you make an instance of an object, you are said to *instantiate* it.

NOTE: To save room, I've left out the beginning of `Game`'s constructor, in which the member variables `howManyLetters`, `howManyPositions`, and `duplicates` are set because this logic is unchanged.

Listing 6.10 Decryptix!.cpp

```
{
while ( choice != 'y' && choice != 'n' )
0:  #include "DefinedValues.h"
1:  #include "List0607_Game.h"
2:
3:  int main()
4:  {
5:      cout << "Decryptix. Copyright 1999 Liberty";
```

```
5a:        cout << Associates, Inc. Version 0.4\n\n" << endl;
6:      bool playAgain = true;
7:
8:      while ( playAgain )
9:      {
10:          char choice = ' ';
11:          Game theGame;
12:          theGame.Play();
13:
14:          cout << "\nThe answer: ";
15:          theGame.GetSolution().Display();
16:          cout << "\n\n" << endl;
17:
18:          while ( choice != 'y' && choice != 'n' )
19:          {
20:              cout << "\nPlay again (y/n): ";
21:              cin >> choice;
22:          }
23:
24:          playAgain = choice == 'y' ? true : false;
25:      }
26:
27:      return 0;
28:  }
```

Listing 6.11 Implementing Game

```
1:    Game::Game():
2:        round(1),
3:        howManyPositions(0),
4:        howManyLetters(0),
5:        duplicates(false)
6:    {
7:
8:    //...
9:
10:        srand( (unsigned)time( NULL ) );
11:
12:        for ( int i = 0; i < howManyPositions; )
13:        {
14:            int nextValue = rand() % (howManyLetters);
15:            char c = alpha[nextValue];
16:            if ( solution.Add(c, duplicates) )
```

```
17:                 i++;
18:         }
19:
20:         cout << "Exiting constructor. List: ";
21:         solution.Display();
22:
23:     }
```

We pick up the logic on line 14, within the `for` loop in which we create our random numbers, turn them into characters on line 20, and then add them to `solution` on line 21. It is here, when we call `solution.add()`, that the logic of the `LinkedList` comes into play. This invokes `LinkedList::Add()`, as shown in Listing 6.12.

Listing 6.12 Implementing LinkedList

```
1:
2:  bool LinkedList::Add(char theChar, bool dupes)
3:  {
4:      bool inserted = false;
5:
6:      if ( ! headNode )
7:      {
8:          headNode = new Node(theChar);
9:          inserted = true;
10:      }
11:      else if ( dupes || HowMany(theChar) == 0 )
12:      {
13:          headNode->Insert(theChar);
14:          inserted = true;
15:      }
16:
17:      return inserted;
18:  }
19:
20:  int LinkedList::HowMany(char theChar) const
21:  {
22:      return headNode->HowMany(theChar);
23:  }
24:
25:  char LinkedList::offset(int offSetValue)
26:  {
27:      Node * pNode = headNode;
28:      for ( int i = 0; i < offSetValue && pNode; i++ )
```

```
29:          pNode = pNode->GetNext();
30:
31:      ASSERT ( pNode );
32:      char c =  pNode->GetChar();
33:      return c;
34:  }
35:
36:  char LinkedList::operator[](int offSetValue)
37:  {
38:      Node * pNode = headNode;
39:      for ( int i = 0; i < offSetValue && pNode; i++ )
40:          pNode = pNode->GetNext();
41:
42:      ASSERT ( pNode );
43:      char c =  pNode->GetChar();
44:      return c;
45:  }
```

On line 6, `LinkedList` checks to see whether it already has a `headNode`. To do this, it checks its `headNode` pointer, which was initialized to `NULL` in `LinkedList`'s constructor. If that is pointer is still `NULL`, no `headNode` exists, so one is now created, passing in the character to be stored.

The constructor to `Node` was considered earlier (refer to Listing 6.3). Remember that the `Node`'s member variable `myChar` is initialized with the character passed in (`theChar`), and its member variable `nextNode` is initialized with `NULL` as shown in Listing 6.13.

Listing 6.13 Node Constructor

```
0:  Node::Node(char theChar):
1:  myChar(theChar),nextNode(0)
2:  {
3:  }
```

This effectively adds the character to the linked list, storing it in the head node.

If there already is a `HeadNode`, (that is, the pointer is not `NULL`), we have a list already and must decide whether we want to add the character. If we are accepting duplicates or if the character does not appear in the list already (line 11), we add it by calling `Insert` on the `headNode` (line 13). I already described `HeadNode()::Insert` in Listing 6.3.

We determine whether the character is in the list on line 11 by calling `LinkedList::HowMany()`, as shown in Listing 6.14.

Listing 6.14 LinkedList's HowMany Method

```
0:  int LinkedList::HowMany(char theChar) const
1:  {
2:      return headNode->HowMany(theChar);
3:  }
```

As you can see, `LinkedList` does nothing but pass along the character to the `headNode`, calling the logic that was considered earlier (in Listing 6.3). The `LinkedList` method `HowMany()` is considered a *wrapper* method: It wraps around the `Node::HowMany()` method, encapsulating its interface, but delegating all the work to the encapsulated method.

---

wrapper--A class is a wrapper for another when it provides a public interface but delegates all the work to the contained class.

method--A method is a wrapper for another method when it provides an encapsulating interface to that method but delegates all the work to the wrapped method.

Playing the Game

After the `solution` member linked list is populated, the `Game` object is fully constructed and the next line in `main()` is a call to the `Game`'s `Play()` method. This was considered earlier, and you probably remember that `Play()` solicits a guess from the player and then calls `Game::Score()`.

`Game::Score()` was also considered earlier, but because solution is now a linked list, this is worth another look. I've reproduced `Game::Score()` in Listing 6.15 for your convenience.

Listing 6.15 The Score Method

```
0:void Game::Score(const char * thisGuess, int & correct, int &
position)
1:  {
2:      correct = 0;
3:      position = 0;
4:
5:
6:      for ( int i = 0; i < howManyLetters; i++)
```

```
7:      {
8:          int howManyInGuess = HowMany(thisGuess, alpha[i]);
9:          int howManyInAnswer = solution.HowMany(alpha[i]);
10:    correct += howManyInGuess < howManyInAnswer ?
10a:            howManyInGuess : howManyInAnswer;
11:     }
12:
13:     for (  i = 0; i < howManyPositions; i++)
14:     {
15:         if ( thisGuess[i] == solution[i] )
16:             position++;
17:     }
18:
19:     ASSERT ( position <= correct )
20:
21:  }
```

On line 9, we must determine how many times each character appears in `solution`. We do this by calling `HowMany()` on `solution`, passing in each character in turn. This calls `LinkedList::HowMany()`, which, as we just saw, calls `Node::HowMany()`.

On line 15, we compare the letter at each offset into `thisGuess` with the letter at the same offset in `solution`. The `Node` class does not provide an offset operator, but the `LinkedList` class must if we are to make this comparison.

Solving the Problem with a Member Method

You can solve this problem by implementing an `offset` method and calling that method by changing line 15 to read

```
15:        if ( thisGuess[i] == solution.offset(i) )
```

The implementation of the `offset` method is shown in Listing 6.16.

Listing 6.16[em]The offset Operator

```
0:  char LinkedList::offset(int offSetValue)
1:  {
2:      Node * pNode = headNode;
3:      for ( int i = 0; i < offSetValue && pNode; i++ )
4:          pNode = pNode->GetNext();
5:
```

```
6:      ASSERT ( pNode );
7:      char c =  pNode->GetChar();
8:      return c;
9:  }
```

The offset is passed into this method as a parameter. We make a new, local pointer, `pNode`, and we assign to that pointer the address that is currently held in `headNode`. Thus, both pointers now point to the same object on the heap, as shown in Figure 6.12.

The goal of the `for` loop on line 3 in Listing 6.16 is to tick through the linked list, starting at the head node, and find the node that corresponds to `offSetValue`.

As we tick through each node, we must also check that `pNode` continues to point to a valid node (that is, we have not reached the end of the list). This protects us from trying to get an offset that is too large for the list.

We put the test for whether the offset continues to point to a valid Node right into the `for` loop by adding it to the test condition:

```
i < offSetValue && pNode;
```

This tests that the counter `i` is not greater than the offset value that was passed in (that is, we're still ticking through the list) and that `pNode` has a nonzero (that is, non-`NULL`) value.

On line 4 we assign to `pNode` the address returned by `GetNext()`. The call to `Node::GetNext()` simply returns the address that is stored in that node's `nextNode` pointer. The net effect of this is that `pNode` now points to the next node in the list.

This `for` loop continues until `i` is no longer less than `offSetValue`. Thus, if `offSetValue` is `5`, this `for` loop causes `pNode` to point to the sixth node in the list, just as you want it to.

On line 6, we assert that we are still pointing to a valid object (belt and suspenders!), on line 7 we extract from that node the character it holds, and on line 8 we return that character.

The net effect of all this is that if you call `offset(5)`, you get back the character at the fifth offset: that is, the sixth character in the list.

This works great, but it isn't how arrays work. You never write

```
myArray.offset(5);
```

You write

```
myArray[5];
```

It would be nice if our linked list supported the same syntax.

Operator Overloading

C++ provides the capability for the designer of a class to give that class the same operators that are available in the built-in class, such as `+`, `-`, `==`, `<`, `>`, and so forth. This is called *operator overloading*. The goal of operator overloading is to allow your class to behave as if it were a built-in type.

---

operator overloading--The ability to program operators such as plus (+) or assignment (=) for classes

---

How You Accomplish Operator Overloading

C++ has a specific syntax dedicated to creating operators for your classes. We'll examine how the offset operator (`[]`) is overloaded because that is what we need right now in the code; we'll return to operator overloading throughout the book, however, because it is a powerful technique with many subtleties.

To create the offset operator, you use the following syntax:

```
char operator[](int offSetValue);
```

When you write `solution[5]`, the compiler automatically converts this to `solution.operator[]`(5).

The implementation for this overloaded operator is identical to the offset method shown in Listing 6.14, as illustrated in Listing 6.17. The only difference is in the signature of the method.

Listing 6.17 LinkedList's offset Operator

```
0:  char LinkedList::operator[](int offSetValue)
1:  {
2:      Node * pNode = headNode;
3:      for ( int i = 0; i < offSetValue && pNode; i++ )
```

```
4:          pNode = pNode->GetNext();
5:
6:      ASSERT ( pNode );
7:      char c =  pNode->GetChar();
8:      return c;
9:  }
```

As you can see, the body is identical; it is just the signature on line 0 that is different:

```
0:  char LinkedList::operator[](int offSetValue)
```

Once again, the return value is a `char`, but this time we see the keyword `operator`, followed by the operator we're overloading, and then the parameter that is needed by that operator.

You invoke this method by writing

```
solution[5];
```

which the compiler translates into

```
solution.operator[](5);
```

setting the parameter `offset` to the value passed in (5).

With this implementation in place, the `Play()` method can test the value of `solution[i],` and thus the `Play()` method remains unchanged from when we were using arrays.

Passing Objects by Value

After the `Game` is fully constructed, we return to `main()`, where the `Play()` method is invoked. When we return from `Play()`, we see this line:

```
theGame.GetSolution().Display();
```

This causes the `Game`'s `GetSolution()` method to be called, returning a reference to a `LinkedList`; then the method `Display()` is called on that reference to a `LinkedList` object.

It is clearer to write

```
const LinkedList & linkedListConstantReference =
    theGame.GetSolution();
```

```
        linkedListConstantReference.Display();
```

This compiles equally well and makes a bit more explicit what we're up to.

We declare `linkedListConstantReference` to be a reference to a constant `LinkedList` object because that is what `GetSolution()` is declared to return in Game.h:

```
class Game
{
//...
     const LinkedList &  GetSolution () const { return solution; }
//...
}
```

Let's pick this apart:

- `const`, `LinkedList`, and `&` together represent the return value: a reference to a constant `LinkedList` object.
- `GetSolution` is the name of the method, `GetSolution`.
- `()` is the parameter list, which in this case is empty.
- `const` declares this member method to be constant.
- `return solution` is the inline implementation, in which we return the member variable `solution`.

Because the return value is declared to be a constant reference to a `LinkedList`, we actually don't return `solution` itself. Instead, we return a constant reference to `solution`.

NOTE: The fact that we return a reference to a constant object limits what you can do with that object. For example, you can only call constant member methods using that reference. If the reference is to a constant object, you can't change that object, and calling a method that is not constant risks changing the object. The compiler enforces this constraint. We're fine here because the only method we call with this reference is `Display()`, which is a constant member method of `LinkedList`.

Why Is it a Reference?

Why bother returning the `LinkedList` object by reference at all? Why not just return it by value?

```
LinkedList GetSolution () const { return solution; }
```

This has the advantage of not being constant: You can do anything you want with this object. Because it is a copy of the original, you won't affect `solution` at all. The net effect in this case is the same: You can still call `Display`, only this time you'll call it on the copy.

The answer is that it is more expensive to make a copy, but to understand why, we must examine what happens when you pass an object by value, this is the subject of the next chapter, "Creating Object-Oriented Linked Lists."

Contents

C++ From Scratch

Contents

8

Using Polymorphism

In this Chapter

- Specialization
- Overriding Functions

- Virtual Methods
- Implementing Polymorphism

The linked list class works, but it cries out for a bit of improvement. Each node in the list is forever checking to see whether there is a next node in the list before taking action:

```
void Node::Insert(char theChar)
{
     if ( ! nextNode )
          nextNode = new Node(theChar);
     else
          nextNode->Insert(theChar);
}
```

This creates code that is a bit more difficult to maintain. As an object-oriented designer, I notice that I'm asking each node to be capable of being the head node in the list, an internal node in the list, or the tail of the list. There is nothing special about these positions--they are just nodes.

Because any node can be internal or the tail, it can't know whether it has a next node, so it must test. If a node knew that it was an internal node, it would always know that there was a next node ("If I'm not the tail, there must be at least one after me"). If it knew that it was the tail, it would know there was no next node (such is the meaning of being the tail node; objects live a very existential existence, and they often major in epistemology).

Specialization

This leads me to a redesign. In it, I have three types of nodes: One is the head node, one is the tail, and the third is the internal node.

You've already seen that the `LinkedList` class you created can mark the head node position, and that this class has special responsibilities such as supporting an offset operator. Let's break out the `InternalNode` from the `TailNode`.

When we say that the `LinkedList`, `InternalNode,` and `TailNode` are all nodes, this tells us that we are thinking about a specialization/generalization relationship. The `LinkedList`, `InternalNode`, and `TailNode` are specializations of `Node`. The `LinkedList` has the special requirement that it act as an interface to the list, which leads me to the design shown in Figure 8.1

Figure 8.1 *Node Specialization.*

In this design, shown here in UML notation, we indicate that the `LinkedList`, `InternalNode`, and `TailNode` are all kinds of `Node`. This brings us back to the conversation about specialization/generalization in Chapter 1, "Introduction."

The specialization relationship establishes an *is-a* relationship: That is, a `TailNode` *is-a* `Node`. Furthermore, the specialization relationship indicates that `TailNode` adds to the definition of `Node`. It says that this is a special kind of `Node`--one that marks the end of a list.

Similarly, `InternalNode` specializes `Node` to mean a node that manages associated data and that, by implication, does not mark the tail of the list.

Finally, `LinkedList` is-a node, a very special node that marks the head of the list and that provides an interface to users of the list. We could have called this the *head node*, but we want to focus on the user's perception: To the user, the head node *is* the linked list. Thus, we bridge the gap between the architect's view (in which this is the head node in a linked list) and the user's view (in which this *is* the linked list) by having it inherit from `Node` but calling it `LinkedList`.

NOTE: The specialization/generalization relationship is reciprocal. Because `LinkedList`, `TailNode`, and `InternalNode` specialize `Node`, `Node` in turn becomes a generalization of the commonality between all three of these classes. Programmers talk about *factoring out* common features from the derived classes up into the base classes. Thus, you can factor out all the common responsibilities of the three classes up into `Node`.

Benefits from Specialization

The first benefit you receive from this design is that `Node` can serve to hold the common characteristics of `LinkedList`, `InternalNode`, and `TailNode`. The things they have in common in this design are that any `Node` can exist in a list, that you can tell any `Node` to insert a new data object, and that you can tell a `Node` to display itself.

In addition, by specializing `Node`, this design of `TailNode` maintains the `Node` features but adds the special capability to mark the end of the list. This specialization is manifest in the differences in how `TailNode` responds to a request to `Insert` data, which it handles differently than, for example, an `InternalNod e` does.

Polymorphism

The need to handle `Insert()` in a special way might be a good reason to create a new class, but you can imagine that it would make your code much more complicated. Each Node would have to know what it pointed to: If it pointed to an `InternalNode`, it would call `InternalNodeInsert`, and if it pointed to a `TailNode`, it would call `TailNodeInsert`. What a bother.

We want to say, "I have a Node, I don't know what kind. It might be an `InternalNode` or it might be a `TailNode`. When I call `Insert()`, I want my `Node` to act one way if it is an `InternalNode`, and in a different way if it is a `TailNode`."

This is called polymorphism: *poly* means *many* and *morph* means *form*. We want the Node to take on many forms. C++ supports polymorphism, which means that the client can call `Insert` on the `Node` and the compiler will take care of making the right thing happen. In this case, the right thing means that if the Node is really an `InternalNode`, `InternalNode::Insert()` will be called; if the Node is really a `TailNode`, `TailNode::Insert()` will be called instead.

Abstract Data Types

You want to create `InternalNode` objects to hold your data, and you want to create a single `TailNode` object and a single `LinkedList` object. You will never instantiate a `Node` object, though. The `Node` object exists only as an abstraction so that you can say "I will call the next node," and not worry about which kind of node it is. The `Node` class is called an *abstract data type (ADT)* because it exists only to provide an abstraction for the classes that inherit from it.

The classes that are derived from your ADT (in this case, `LinkedList`, `InternalNode`, and `TailNode`) can be *concrete*, and thus can have objects instantiated. Alternatively, you can derive ADTs from other ADTs. Ultimately, however, you must derive a concrete class so that you can create objects.

Abstract Data Type--A class that provides a common interface to a number of classes that derive from it. You can never instantiate an Abstract Data Type.

concrete class--A class that is not abstract and that can therefore be instantiated.

How This Is Implemented in C++

Until now, we've not discussed a word about how all this is implemented in C++. That is because we have rightly been focused on design, not implementation.

The *design* calls for all three of these classes to specialize `Node`. You *implement* that design concept of specialization by using inheritance. Thus, you will have `LinkedList`, `TailNode`, and `InternalNode` inherit from `Node`.

The Syntax of Inheritance

When you declare a class, you can indicate the class from which it derives by writing a colon after the class name, the type of derivation (public or otherwise), and the class from which it derives. For now, focus only on public inheritance because that is what implements the design concept of specialization/generalization.

Thus, to indicate that `InternalNode` is a specialization of `Node`, or that `InternalNode` *derives* from `Node`, you write

```
class InternalNode : public Node
```

NOTE: When one class specializes another, we say that the specialized class is *derived* from the more general class, and that the more general class is the *base class*.

The class from which you derive must have been declared already, or you receive a compiler error.

Overriding Functions

A `LinkedList` object has access to all the member functions in class `Node`, as well as to any member functions the declaration of the `LinkedList` class might add. It can also *override* a base class function. Overriding a function means changing the implementation of a base class function in a derived class. When you instantiate an object of the derived class and call an overridden method, the right thing happens.

NOTE: When a derived class creates a function with the same return type and signature as a member function in the base class, but with a new implementation, it is said to *override* that method.

This is very handy because it allows an `InternalNode` to specialize how it handles some methods (such as `Insert()`) and simply inherit the implementation of other methods.

When you override a function, its signature must be identical to the signature of the function in the base class. The signature is the function prototype, other than the return type: the name, the parameter list, and the keyword `const` (if it is used).

Virtual Methods

I have emphasized the fact that an `InternalNode` object is-a `Node` object. So far that has meant only that the `InternalNode` object has inherited the attributes (data) and capabilities (methods) of its base class. In C++, however, the is-a relationship runs deeper than that.

C++ extends its polymorphism, allowing pointers to base classes to be assigned to derived class objects. Thus, you can write

```
Node * pNode = new InternalNode;
```

This creates a new `InternalNode` object on the heap and returns a pointer to that object, which it assigns to a pointer to `Node`. This is fine because an `InternalNode` is-a `Node`.

In fact, this is the key to polymorphism. You can create all kinds of Windows--they can each have a `draw()` method that does something different (the list box draws a rectangle, the radio button draws a circle). You can create a pointer to a Window without regard to what type of Window you have, and when you call

```
pWindow->Draw();
```

the Window is drawn properly.

Similarly, you can have a pointer to any kind of `Node`, a `LinkedList`, an `InternalNode,` or a `TailNode`, and you can call `Insert()` on that node without regard to what kind of `Node` it is. The right thing will happen.

Here's how it works: You use the pointer to invoke a method on `Node`, for example `Insert()`. If the pointer is really pointing to a `TailNode`, and if `TailNode` has overridden `Insert()`, the overridden version of `Insert` is called. If `TailNode` does not override `Insert()`, it inherits this method from its base class, `Node`, and `Node::Insert()` is called.

This is accomplished through the magic of virtual functions.

NOTE: C++ programmers use the terms *method* and *function* interchangeably. This confusion comes from the fact that C++ has two parents: the object-oriented languages such as SmallTalk, which use the term *method*, and C, which uses the term *function*.

How Virtual Functions Work

When a derived object, such as an `InternalNode` object, is created, first the constructor for the base class is called, and then the constructor for the derived class is called. Figure 8.2 shows how the `InternalNode` object looks after it is created. Note that the `Node` part of the object is contiguous in memory with the `InternalNode` part.

Figure 8.2 *The `InternalNode` object after it is created.*

When a virtual function is created in an object, the object must keep track of that function. Many compilers build a virtual function table, called a *v-table*. One of these tables is kept for each type, and each object of that type keeps a virtual table pointer (called a *vptr* or *v-pointer*) that points to that table. (Although implementations vary, all compilers must accomplish the same thing, so you won't be too wrong with this description.)

---

v-table--The virtual function table used to achieve polymorphism

vptr or v-pointer--The Virtual Function Table Pointer, which is provided for every object from a class with at least one virtual method

---

Each object's vptr points to the v-table that, in turn, has a pointer to each of the virtual functions. When the `Node` part of the `InternalNode` is created, the vptr is initialized to point to the correct part of the v-table, as shown in Figure 8.3.

Figure 8.3 *The v-table of a node.*

When the `InternalNode` constructor is called and the `InternalNode` part of this object is added, the vptr is adjusted to point to the virtual function overrides (if any) in the `InternalNode` object (see Figure 8.4).

Figure 8.4 *The v-table of a* `InternalNode`.

When a pointer to a `Node` is used, the vptr continues to point to the correct function, depending on the "real" type of the object. Thus, when `Insert()` is invoked, the correct (`InternalNode`) version of the function is invoked.

Virtual Destructors

It is legal and common to pass a pointer to a derived object when a pointer to a base object is expected. What happens when that pointer to a derived subject is deleted? If the destructor is virtual, as it should be, the right thing happens: The derived class's destructor is called. Because the derived class's destructor automatically invokes the base class's destructor, the entire object is properly destroyed.

The rule of thumb is this: If any of the functions in your class are virtual, the destructor needs to be virtual as well.

The ANSI/ISO standard dictates that you *can* vary the return type, but few compilers support this. If you change the signature--the name, number, or type of parameters or whether the method is `const`--you are not overriding the method, you are adding a new method.

It is important to note that if you add a new method with the same name as another method in the base class, you hide the base class method and the client can't get to it. Take a look at Listing 8.1, which illustrates this point.

Listing 8.1 Hiding the Base Class Method

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  class Base
4:  {
5:  public:
6:      Base() { cout << "Base constructor\n"; }
7:      virtual ~Base() { cout << "Base destructor\n"; }
8:      virtual void MethodOne() { cout << "Base MethodOne\n"; }
9:      virtual void MethodTwo() { cout << "Base MethodTwo\n"; }
10:      virtual void MethodThree()
11:          { cout << "Base MethodThree\n"; }
12:  private:
13:  };
14:
```

```
15:  class Derived : public Base
16:  {
17:  public:
18:      Derived() { cout << "Derived constructor\n"; }
19:      virtual ~Derived() { cout << "Derived destructor\n"; }
20:      virtual void MethodOne() { cout << "Derived MethodOne\n"; }
21:      virtual void MethodTwo() { cout << "Derived MethodTwo\n"; }
22:      virtual void MethodThree(int myParam)
23:          { cout << "Derived MethodThree\n"; }
24:  private:
25:  };
26:
27:  int main()
28:  {
29:
30:      Base * pb = new Base;
31:      pb->MethodOne();
32:      pb->MethodTwo();
33:      pb->MethodThree();
34:  //    pb->MethodThree(5);
35:      delete pb;
36:      cout << endl;
37:
38:      Base * pbd = new Derived;
39:      pbd->MethodOne();
40:      pbd->MethodTwo();
41:      pbd->MethodThree();
42:  //    pbd->MethodThree(5);
43:      delete pbd;
44:      cout << endl;
45:
46:      Derived * pd = new Derived;
47:      pd->MethodOne();
48:      pd->MethodTwo();
49:  //    pd->MethodThree();
50:      pd->MethodThree(5);
51:      delete pd;
52:
53:      return 0;
54:  }
Base constructor
Base MethodOne
Base MethodTwo
```

```
Base MethodThree
Base destructor
Base constructor
Derived constructor
Derived MethodOne
Derived MethodTwo
Base MethodThree
Derived destructor
Base destructor
Base constructor
Derived constructor
Derived MethodOne
Derived MethodTwo
Derived MethodThree
Derived destructor
Base destructor
```

In this example, we create a `Base` class and a `Derived` class. The `Base` class declares three methods virtual on lines 8-10, which will be overridden in the `Derived` class.

The overridden `MethodThree` differs from the base class's version in that it takes an extra parameter. This overloads the method (rather than overrides it) and *hides* the base class method. What is the effect? Let's see.

On line 30 we declare a pointer to a `Base` object, and we use this pointer to invoke `MethodOne`, `MethodTwo,` and both versions of `MethodThree`. The second, shown on line 34, won't compile and so is commented out. It won't compile because `Base` objects know nothing about this overload version.

The output shows that a `Base` constructor is called, followed by the three `Base` methods, ending with a call to the `Base` destructor. Pretty much as we might expect.

On line 38 we declare a pointer to a `Base` and initialize it with a `Derived`. This is the polymorphic form: We assign a specialized object to a pointer to a more general object. We call `MethodOne`, `MethodTwo`, and `MethodThree,` and they compile fine until we try to compile the version that takes an `int`. Because this is a pointer to a `Base`, it can't find this method and won't compile.

Let's look at the output. We see the `Base` constructor, and then the `Derived` constructor. That is how derived objects are constructed: base first. We then see a call to the `Derived MethodOne`. Polymorphism works! Here we have a `Base` pointer, but because we assigned a `Derived` object to it, when we call a virtual method, the right method is called. This is followed by a call to `Derived MethodTwo`, and then to `Base MethodThree`! As far as the `Base` pointer is concerned, `Base MethodThree` has not been overridden, so `Derived` inherits the `Base` method. Finally, the object is

destroyed in the reverse order in which it was created.

The final set of code begins on line 46, where we create a `Derived Pointer` and initialize it with a `Derived` object. This time we can't call the version of `MethodThree` that was declared in `Base` because our new `Derived` object hides it. This proves the rule: If you overload a base class in the derived class, you must explicitly implement every version of the method (in this case you'd have to implement the version with no parameters) in the derived class, or it is hidden from your derived objects.

The output reflects the fact that this `Derived` object calls (of course) only derived methods.

To avoid the hiding, we can move the version of the method that takes an integer up into `Base`, or at a minimum we can implement the version with no parameters in the derived class. If we choose the first option, we can use both methods polymorphically. If we choose the latter, at least we can access the base method using a derived object.

We can, actually, overcome many of these problems with a bit more magic. Let's take them in turn. To solve the first problem (shown on line 34), we have only to overload `MethodThree` in `Base`:

```
3:  class Base
4:  {
5:  public:
6:      Base() { cout << "Base constructor\n"; }
7:      virtual ~Base() { cout << "Base destructor\n"; }
8:      virtual void MethodOne() { cout << "Base MethodOne\n"; }
9:      virtual void MethodTwo() { cout << "Base MethodTwo\n"; }
10:      virtual void MethodThree()
11:          { cout << "Base MethodThree\n"; }
        virtual void MethodThree(int param)
               { cout << "Base MethodThree(int)\n"; }
12:  private:
13:  };
```

To solve the problem on lines 42 and 49, we need only invoke the base class's method directly:

```
42:  //    pbd->Base::MethodThree(5);
```

Here we use the `Base` class name, followed by the scoping operator (`::`), to invoke the `Base` version of this method. Hey! Presto! Now we've "unhidden" it, and we can call it.

Implementing Polymorphism

All this is fine in theory, but it remains highly abstract until you see it implemented in code. Let's take a look at the implementation of the newly object-oriented `LinkedList` (see Listing 8.2).

Listing 8.2 LinkedList

```
0:  #ifndef LINKEDLIST_H
1:  #define LINKEDLIST_H
2:
3:  #include "DefinedValues.h"
4:
5:  class Node  // abstract data type
6:  {
7:  public:
8:       Node(){}
9:       virtual ~Node() {}
10:       virtual void Display() const { }
11:       virtual int HowMany(char c) const = 0;
12:       virtual Node * Insert(char theCharacter) = 0;
13:       virtual char operator[](int offset) = 0;
14:  private:
15:  };
16:
17:  class InternalNode: public Node
18:  {
19:  public:
20:       InternalNode(char theCharacter, Node * next);
21:       virtual ~InternalNode();
22:       virtual void Display() const;
23:       virtual int HowMany(char c) const;
24:       virtual Node * Insert(char theCharacter);
25:       virtual char operator[](int offset);
26:
27:  private:
28:       char myChar;
29:       Node * nextNode;
30:  };
31:
32:  class TailNode : public Node
33:  {
34:  public:
35:       TailNode(){}
36:       virtual ~TailNode(){}
37:       virtual int HowMany(char c) const;
```

```
38:       virtual Node * Insert(char theCharacter);
39:       virtual char operator[](int offset);
40:
41:  private:
42:
43:  };
44:
45:  class LinkedList : public Node
46:  {
47:  public:
48:       LinkedList();
49:       virtual ~LinkedList();
50:       virtual void Display() const;
51:       virtual int HowMany(char c) const;
52:       virtual char operator[](int offset);
53:
54:       bool Add(char c);
55:       void SetDuplicates(bool dupes);
56:
57:  private:
58:       Node * Insert(char c);
59:       bool duplicates;
60:       Node * nextNode;
61:  };
62:
63: #endif
```

This analysis begins on line 5 with the declaration of the `Node` class. Note that all the methods, with the exception of the constructor, are virtual.

Constructors cannot be virtual; destructors need to be virtual if any method is virtual; and I've made the rest of the methods virtual because I expect that they can be overridden in at least some of the derived classes.

Note that the first three methods--the constructor, the destructor, and `Display()`--all have inline implementations that do nothing. `HowMany()` (line 11), however, does not have an inline implementation. If you check Listing 8.2, which has the implementation for the classes that are declared in Listing 8.1, you will not find an implementation for `Node::HowMany()`.

That is because `Node::HowMany()` is declared to be a *pure* virtual function by virtue of the designation at the end of the declaration, `= 0`. This indicates to the compiler that this method *must* be overridden in the derived class. In fact, it is the presence of one or more pure virtual functions in your class declaration that creates an ADT. To recap: In C++, an abstract data type is created by declaring one

or more pure virtual functions in the class.

Pure Virtual Function--A member function that *must* be overridden in the derived class, and which makes the class in which it is declared an Abstract Data Type. You create a pure virtual function by adding `= 0` to the function declaration.

The `Node` class is, therefore, an ADT from which you derive the concrete nodes you'll instantiate in the program. Every one of these concrete types must override the `HowMany()` and `Insert()` methods, as well as the offset operator. If a derived type fails to override even one of these methods, it too is abstract, and no objects can be instantiated from it.

On line 17, you see the declaration of the `InternalNode` class, which derives from `Node`. As you can see on lines 23-25, this class does override the three pure virtual functions of `Node`; thus, this is a concrete class from which you can instantiate objects.

Although we put the keyword `virtual` on lines 22-25, it is not necessary. When a method is virtual, it remains virtual all the way down the hierarchy of derived classes. So we could have left this designation out here, as we did on line 21.

`InternalNode` adds two private variables: `myChar` and `nextNode`. It is clear why `myChar` can't be in `Node:` Only `InternalNode` classes have a character for which they are responsible. Why not put `nextNode` up in the base class, then? After all, nodes exist in a linked list. You'd expect all nodes to have a `nextNode`.

This is true of all nodes *except* for `TailNode`. Because `TailNode` does not have a `nextNode`, it doesn't make sense for this attribute to be in the base class.

You can put this pointer in the base and then give `TailNode` a null `nextNode` pointer. That might work, but it doesn't map to the semantics of a `Node`. A `Node` is an object that lives in a linked list. It is not part of our definition that a `Node` must point to another `Node`, only that it must be in the list. `TailNodes` are in the list, but they don't point to a `nextNode`. Thus, this pointer is not an intrinsic aspect of a `Node`, so I've left it out of the base class.

The declaration for `TailNode` begins on line 32, and once again you can see that the pure virtual functions are overridden. The distinguishing characteristics of `TailNode` are shown in the implementation of these methods, which we'll consider in a moment.

Finally, on line 45 you see `LinkedList` declared. Again, the pure virtual methods are overridden, but

this time, on lines 54 and 55, you see new public methods: `Add` and `SetDuplicates`. These methods support the functionality of this class, to provide an interface for the `LinkedList` to the client classes.

Note also that on line 58 we've moved `Insert()` to the private section of `LinkedList`. The only `Node` class that any non-`Node` interacts with is this one, and `Insert` is not part of the `LinkedList` class's public interface. When a client wants to add a character to the list, it calls `LinkedList::Add()`, which is shown on line 54.

`Node`'s `Insert()` method is public, however--when nodes interact with one another (for example, when `LinkedList` is adding objects to an `InternalNode`), the `InsertMethod` is used.

Let's look at the implementation of `LinkedList` in Listing 8.3, adding `Game` in Listings 8.4 and 8.5, and the driver program Decryptix! in Listing 8.6. This enables us to step through a few methods using the new object-oriented linked list.

Listing 8.3 Linked List Implementation

```
0:  #include "LinkedList.h"
1:
2:  InternalNode::InternalNode(char theCharacter, Node * next):
3:  myChar(theCharacter),nextNode(next)
4:  {
5:  }
6:
7:  InternalNode::~InternalNode()
8:  {
9:       delete nextNode;
10:  }
11:
12:  void InternalNode::Display() const
13:  {
14:       cout << myChar; nextNode->Display();
15:  }
16:
17:  int InternalNode::HowMany(char theChar) const
18:  {
19:       int myCount = 0;
20:       if ( myChar == theChar )
21:            myCount++;
22:       return myCount + nextNode->HowMany(theChar);
23:  }
24:
```

```
25:  Node * InternalNode::Insert(char theCharacter)
26:  {
27:       nextNode = nextNode->Insert(theCharacter);
28:       return this;
29:  }
30:
31:  char InternalNode::operator[](int offSet)
32:  {
33:       if ( offSet == 0 )
34:            return myChar;
35:       else
36:            return (*nextNode)[--offSet];
37:  }
38:
39:
40:  int TailNode::HowMany(char theChar) const
41:  {
42:       return 0;
43:  }
44:
45:  Node * TailNode::Insert(char theChar)
46:  {
47:       return new InternalNode(theChar, this);
48:  }
49:
50:  char TailNode::operator[](int offset)
51:  {
52:       ASSERT(false);
53:       return ' ';
54:  }
55:
56:
57:
58:  LinkedList::LinkedList():
59:  duplicates(true)
60:  {
61:       nextNode = new TailNode;
62:  }
63:
64:  LinkedList::~LinkedList()
65:  {
66:       delete nextNode;
67:  }
```

```
68:
69:  void LinkedList::Display() const
70:  {
71:       nextNode->Display();
72:  }
73:
74:  int LinkedList::HowMany(char theChar) const
75:  {
76:       return nextNode->HowMany(theChar);
77:  }
78:
79:  Node * LinkedList::Insert(char theChar)
80:  {
81:            nextNode = nextNode->Insert(theChar);
82:            return nextNode;
83:  }
84:
85:  char LinkedList::operator[](int offSet)
86:  {
87:       return (*nextNode)[offSet];
88:  }
89:
90:
91:
92:  bool LinkedList::Add(char theChar)
93:  {
94:       if ( duplicates || HowMany(theChar) == 0 )
95:       {
96:            Insert(theChar);
97:            return true;
98:       }
99:       else
100:           return false;
101:  }
102:
103:
104:  void LinkedList::SetDuplicates(bool dupes)
105:  {
106:       duplicates = dupes;
107:  }
```

Listing 8.4 Game Header

```
0:  #ifndef GAME_H
1:  #define GAME_H
2:
3:  #include "definedValues.h"
4:  #include "LinkedList.h"
5:
6:  class Game
7:  {
8:  public:
9:       Game();
10:       ~Game(){}
11:       void Display(const LinkedList * pList)const
12:       {
13:            pList->Display();
14:       }
15:
16:       void Play();
17:       const LinkedList &  GetSolution() const
18:       {
19:            return solution;
20:       }
21:
22:       void Score(
23:            const char * thisGuess,
24:            int & correct,
25:            int & position
26:            );
27:
28:  private:
29:            int HowMany(const char * theString, char theChar);
30:            LinkedList solution;
31:            int howManyLetters;
32:            int howManyPositions;
33:            int round;
34:            bool duplicates;
35:  };
36:
37:  #endif
```

Listing 8.5 Game Implementation

```
0:  #include <time.h>
1:  #include "game.h"
```

```
2:  #include "definedvalues.h"
3:
4:  Game::Game():
5:       round(1),
6:       howManyPositions(0),
7:       howManyLetters(0),
8:       duplicates(false)
9:  {
10:
11:       bool valid = false;
12:       while ( ! valid )
13:       {
14:            while ( howManyLetters < minLetters
15:                 || howManyLetters > maxLetters )
16:            {
17:                 cout << "How many letters? (" ;
18:                 cout << minLetters << "-" << maxLetters << "): ";
19:                 cin >> howManyLetters;
20:                 if ( howManyLetters < minLetters
21:                      || howManyLetters > maxLetters )
22:                 {
23:                 cout << "please enter a number between ";
24:                 cout << minLetters << " and " << maxLetters <<
endl;
25:                 }
26:            }
27:
28:            while ( howManyPositions < minPos ||
29:                 howManyPositions > maxPos )
30:            {
31:                 cout << "How many positions? (";
32:                 cout << minPos << "-" << maxPos << "): ";
33:                 cin >> howManyPositions;
34:                 if ( howManyPositions < minPos ||
35:                      howManyPositions > maxPos )
36:                 {
37:                      cout << "please enter a number between ";
38:                      cout << minPos <<" and " << maxPos << endl;
39:                 }
40:            }
41:
42:            char choice = ' ';
43:            while ( choice != 'y' && choice != 'n' )
```

```
44:            {
45:                 cout << "Allow duplicates (y/n)? ";
46:                 cin >> choice;
47:            }
48:
49:            duplicates = choice == 'y' ? true : false;
50:            solution.SetDuplicates(duplicates);
51:
52:            if ( ! duplicates && howManyPositions >
howManyLetters )
53:            {
54:          cout << "I can't put " << howManyLetters;
55:          cout << " letters in ";
56:          cout << howManyPositions ;
57:          cout << "positions without duplicates! Please try again.
\n";
58:          howManyLetters = 0;
59:          howManyPositions = 0;
60:            }
61:            else
62:                 valid = true;
63:       }
64:
65:
66:       srand( (unsigned)time( NULL ) );
67:
68:       for ( int i = 0; i < howManyPositions; )
69:       {
70:            int nextValue = rand() % (howManyLetters);
71:            char theChar = alpha[nextValue];
72:            if ( solution.Add(theChar) )
73:                 i++;
74:       }
75:
76:       cout << "Exiting constructor. List: ";
77:       solution.Display();
78:
79:  }
80:
81:  inline int Game::HowMany(const char * theString, char theChar)
82:  {
83:       int count = 0;
84:       for ( int i = 0; i < strlen(theString); i++)
```

```
85:            if ( theString[i] == theChar )
86:                 count ++;
87:       return count;
88:  }
89:
90:  void Game::Play()
91:  {
92:       char guess[80];
93:       int correct = 0;
94:       int position = 0;
95:       bool quit = false;
96:
97:       while ( position < howManyPositions )
98:       {
99:
100:           cout << "\nRound " << round;
101:           cout << ". Enter " << howManyPositions;
102:           cout << " letters between ";
103:           cout << alpha[0] << " and ";
104:           cout << alpha[howManyLetters-1] << ": ";
105:
106:           cin >> guess;
107:
108:           if ( strlen(guess) != howManyPositions )
109:           {
110:                cout << "\n ** Please enter exactly ";
111:                cout << howManyPositions << " letters. **\n";
112:                continue;
113:           }
114:
115:
116:           round++;
117:
118:           cout << "\nYour guess: " << guess << endl;
119:
120:           Score(guess,correct,position);
121:           cout << "\t\t" << correct << " correct, ";
122:           cout << position << " in position." << endl;
123:       }
124:
125:           cout << "\n\nCongratulations! It took you ";
126:
127:           if ( round <= 6 )
```

```
128:                cout << "only ";
129:
130:           if ( round-1 == 1 )
131:                cout << "one round!" << endl;
132:           else
133:                cout << round-1 << " rounds." << endl;
134:
135:  }
136:
137:
138:  void Game::Score(
139:                       const char * thisGuess,
140:                       int & correct,
141:                       int & position
142:                       )
143:  {
144:       correct = 0;
145:       position = 0;
146:
147:
148:       for ( int i = 0; i < howManyLetters; i++)
149:       {
150:            int howManyInGuess = HowMany(thisGuess, alpha[i]);
151:            int howManyInAnswer = solution.HowMany(alpha[i]);
152:            correct += howManyInGuess < howManyInAnswer ?
153:                 howManyInGuess : howManyInAnswer;
154:       }
155:
156:       for (  i = 0; i < howManyPositions; i++)
157:       {
158:            if ( thisGuess[i] == solution[i] )
159:                 position++;
160:       }
161:
162:       ASSERT ( position <= correct );
163:
164:  }
```

Listing 8.6 Decryptix! Driver Program

```
0:  #include "definedValues.h"
1:  #include "game.h"
2:
```

```
3:  int main()
4:  {
5:
6:       cout << "Decryptix. Copyright 1999 Liberty Associates,";
7:       cout << " Inc. Version 0.3\n\n" << endl;
8:       bool playAgain = true;
9:
10:       while ( playAgain )
11:       {
12:            char choice = ' ';
13:            Game theGame;
14:            theGame.Play();
15:
16:            cout << "\nThe answer: ";
17:            theGame.GetSolution().Display();
18:            cout << "\n\n" << endl;
19:
20:            while ( choice != 'y' && choice != 'n' )
21:            {
22:                 cout << "\nPlay again (y/n): ";
23:                 cin >> choice;
24:            }
25:
26:            playAgain = choice == 'y' ? true : false;
27:       }
28:
29:       return 0;
30:  }
```

NOTE: The `Game` class declaration is unchanged from the previous version, and is reproduced here only as a convenience.

Let's start the analysis with the construction of the solution member variable of `Game`. Put a break point on line 58 of Listing 8.2. When the break point is hit, the first thing you do is check the call stack to see when in the execution of the program this constructor was called:

```
LinkedList::LinkedList() line 62
Game::Game() line 10 + 58 bytes
main() line 14 + 8 bytes
```

As you can see, `main()` called the `Game` constructor, which in turn called the `LinkedList` constructor. Line 14 of `main()` looks like this:

```
Game theGame;
```

It's just as you expected--the construction of a `Game` object. Line 10 of `Game` is the opening brace, shown in the code as line 9 of Listing 8.4. Because the `LinkedList` member variable (`solution`) was not initialized, the compiler calls its constructor just before entering the body of `Game`'s constructor.

Returning to Listing 8.2, line 58, note that the `LinkedList` initializes its `duplicates` member variable to `true` (line 59); then, on line 61, it creates a new `TailNode` and sets its own `nextNode` to point to the `Tail`. This creates an empty linked list, as shown in Figure 8.5.

Figure 8.5 *An empty linked list.*

`LinkedList` is thus automatically initialized to a first `Node` (`LinkedList`) and a last `Node` (`TailNode`); neither of them contains any data, but together they create the structure of the list.

If you continue stepping through the code, you find yourself in the body of the `Game` constructor (line 11 of Listing 8.4). Set a break point on line 66 and run to that break point so that you skip examining the initial user interface code that has not changed from previous chapters.

When you are prompted, choose five letters in five positions. The break point is hit, and we generate a seed for the random number generator, based on the time (as discussed in previous chapters). On line 70, we generate a random number and use that as an offset into the `alpha` array to generate our first character. By line 72 we have that first character, which in my case is `'d'`.

Stepping into the `Add` method causes us to jump to line 92 of Listing 8.2. On line 94, `duplicates` is tested and fails (we're not allowing dupes); therefore, the call to `HowMany` is made, passing in the parameter.

Stepping in here brings us to line 74. The `LinkedList` implementation of this is to return the value that is generated by calling `HowMany` on whatever the `LinkedList` points to. Step in, and you'll find yourself in the `HowMany()` method of `TailNode`. This makes sense; right now, `LinkedList` points to `TailNode`.

Because `TailNode` holds no data, it returns `0` regardless of what character it is given. This is returned to `LinkedList::HowMany()`, which in turn returns it to `LinkedList::Add()` on line 94. Because this satisfies the second condition in the `OR` statement, enter the body of the `if` statement on line 96.

Stepping into the `Call` to `Insert` jumps to line 80, the implementation of `LinkedList::Insert()`. `LinkedList`'s strategy is to pass this request on to whatever it points to. Stepping in brings us to line 47, the implementation of `TailNode::Insert()`.

`TailNode` always inserts a node when it is asked to. It knows that it is the tail, so it doesn't have to check--it can just make the insertion. This is the critical difference from the previous version. You'll remember that in Chapter 6, "Using Linked Lists," `Node` responded to `Insert` as follows:

```
void Node::Insert(char theChar)
{
     if ( ! nextNode )
          nextNode = new Node(theChar);
     else
          nextNode->Insert(theChar);
}
```

That is, it was necessary to see whether there were any more `Nodes` in the list. If not (if the current `Node` was the `Tail`), a new `Node` could be inserted. On the other hand, if the current `Node` was not the `Tail`, and therefore was an `InternalNode`, the `Insert` request would be passed down the list.

The new design obviates the need for the test: The `TailNode` knows that it is the tail, and it can just make the insertion. This simple division of responsibility is, in a small way, the very heart of object-oriented software development.

Let's examine the implementation in some detail. Line 47 returns the address of the new `InternalNode` that is created by passing in the character that is received as a parameter and the `this` pointer of the `TailNode`.

This jumps to the constructor for `InternalNode`, shown on line 2. Here you see the creation of the `InternalNode`. The character is inserted into the `InternalNode`'s `myChar` member variable, and the `this` pointer from the `TailNode` is stored in the `nextNode` pointer of `InternalNode`.

The address of this `InternalNode` is then passed back to the caller of `TailNode::Insert()`, and in this case is assigned to the `nextNode` member variable of `LinkedList` (as shown on line 81). Finally, this address is returned by `LinkedList::Insert`, but the calling function--`LinkedList::Add()` on line 96--makes no use of it, and it is thrown on the floor. On line 97, we return `true` to the calling function on line 72 of Listing 8.4

We have now added a first letter to the linked list, and it worked great. It takes a lot longer to explain the process than to perform it. Next, let's track the second letter, now that you have an `InternalNode` in the linked list.

Adding a Second Letter

Return to line 92 of Listing 8.2. Once again, this steps you into `LinkedList::HowMany()` (line 74), this time passing in `'b'`.

This time, `LinkedList`'s `nextNode` points to an `InternalNode` (the one holding `'d'`), so we now jump to line 17. Here a local variable `myCount` is initialized to zero. The `InternalNode`'s member variable `myCount` (`'d'`) is compared to the parameter `theChar` (`'b'`). Because they are not the same, `myCount` remains zero.

We now invoke `HowMany()` on the node that is pointed to by this `InternalNode`'s `nextNode` pointer. Right now, the pointer points to `TailNode`, which we examined previously; it simply returns zero. That zero is added to `myCount` (also zero) for a total of zero, which is the value that is returned to `LinkedList::Add()`.

This causes the second half of the `OR` statement on line 94 to return `true` (`HowMany('b')` equals zero), so the body of the `if` statement executes. This causes the call on `Insert()` to execute, with a jump to line 80, which is the implementation of `LinkedList::Insert()`. Again, `LinkedList`'s strategy is to pass this request on to whatever it points to, which in this case is `InternalNode`'s `Insert()` method (as shown on line 27). `InternalNode`'s strategy is to call `Insert()` on the object to which its `nextNode` pointer points (in this case `TailNode`), and then to set its `nextNode` pointer to whatever value is returned.

Because the `nextNode` is the `TailNode`, `InternalNode::Insert` is now called. As you saw just a moment ago, it creates a new `InternalNode` for `'b'` and tells that new node to point to the tail. It then returns the address of the new node, which is now assigned to the `nextNode` pointer of the node that holds `'d'`. Thus, `'b'` is appended to the list, as shown in Figure 8.6.

Figure 8.6

Appending 'b'.Examining operator[]

Let's take a look at `Game::Score`, beginning on line 138 of Listing 8.4. You've examined the fundamental logic in detail in previous chapters. (We're particularly interested in the letter by letter comparisons.) You've seen how `LinkedList::HowMany()` works; now take a look at the offset operator as it is used on line 158.

Stepping in to this code steps into `Linkedlist::Operator[]` on line 87 of Listing 8.2.

NOTE: Along the way, I've added test code at line 77 to print out the answer so that I can examine the behavior of the system as I test it. You want to remove both lines 76 and 77 before releasing this code.

Not surprisingly, all `LinkedList` does here is invoke this same operator on the node to which it points. Stepping in from line 87 brings you to line 31, `InternalNode::operator[]`. Take a look at your auto member variables, and you'll find that `myChar` is `'d'`, just as you might expect. The offset that was passed in is now tested. If it is `0`, the call was for the first letter in the list. You will be at the first letter in the list, and you can return `myChar`, which is the case now. The letter `'d'` is returned to `LinkedList`; `LinkedList` returns it to the calling method, `score()`, which--on line 158 of Listing 8.4--is compared with the value of the first character in the guess.

This is repeated, but with `i` set to `1`. A call to `solution[1]` is invoked, bringing us back into `LinkedList::operator[]` on line 87 of Listing 8.2.

Stepping in from line 87 brings you back to `InternalNode::operator[]` on line 31. Take a look at your auto member variables, and you'll find that `myChar` is again `'d'`--you're back at the first `InternalNode` in the list. The offset that was passed in (`1`) is now tested. Because it is not zero, the `if` statement on line 33 fails and the body of the `else` statement on line 36 executes:

```
36:             return (*nextNode)[--offSet];
```

This invokes the offset operator on the next `Node` in the list, passing in the decremented `offSet` value. Stepping in appears to bring us back to the top of the same method, but check your variables--`myChar` is now `b`, and `offset` is now zero. Perfect: You'll return the second letter in the list, exactly as you wanted.

The linked list works as you want, and by using inheritance, you've delegated responsibility for monitoring the head and tail of the list to specialized nodes. This simplifies the code and makes it easier to maintain.

The problem with this linked list, however, is that it can only be used by nodes that hold single characters. What if you have other data that you want to insert into your linked list? Must you really rewrite the linked list each time you change the kind of object that is contained? The next chapter, "Implementing Templates," takes a look at modifying your linked list so that it can handle any kind of data.
