C++ for VB Programmers
Copyright © 2000 by Jonathan D. Morrison

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-893115-76-3

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Editorial Directors: Dan Appleman, Gary Cornell, Karen Watterson
Technical Reviewer: Dan Appleman
Editor: Katharine Dvorak
Projects Manager: Grace Wong
Supervising Production Editor: MaryAnn Brickner
Production Services and Page Composition: Impressions Book and Journal Services, Inc.
Artist: Frank Giangreco
Cover: Karl Miyajima

Distributed to the book trade in the United States by Springer-Verlag New York, Inc.,175 Fifth Avenue, New York, NY, 10010
and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany

In the United States, phone 1-800-SPRINGER; orders@springer-ny.com; http://www.springer-ny.com
Outside the United States, contact orders@springer.de; http://www.springer.de; fax +49 6221 345229

For information on translations, please contact Apress directly at 901 Grayson Street, Suite 204, Berkeley, CA, 94710
Phone: 510-549-5931; Fax: 510-549-5939; info@apress.com; http://www.apress.com

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

Morrison_001_224 8/22/00 11:53 AM Page 33

CHAPTER 4

C++ 101

WELL, IT'S INEVITABLE. IF WE ARE GOING TO DO anything useful with C++ we are going to have to learn the syntax of the language. Because you already know Visual Basic, it seems logical to use this knowledge as a crutch for learning C++, which is what we will do throughout this chapter.

As you start to learn C++ it is important to remember that you already know how to program—you just don't know the syntax of C++. Think of the process as learning how to drive a new type of car for the first time. You already know how to drive; you just have to learn where the trunk release is and how to set the clock in the new car and you're as good as you were in your old car.

Data and Variables

Regardless of the language in which it was written, every computer program contains some amount of data. Even if the program is only to write the words "Hello World," the fact that a program runs on a computer is proof that the program contains data. Thus, it is a good idea to start our learning of the C++ language by examining the available C++ data types, their Visual Basic equivalents (where applicable), and their storage capacity, which is summarized in Table 4-1.

You can see that there are many different data types available in C++. Also notice that in most cases they map directly to Visual Basic types.

> **NOTE:** *The* void *data type in C++ is special in that it has no value. It is used to show that there is an absence of value. The void type is usually used as the return type of a function to denote that the function does not return a value, or as the only item in a parameter list for a function that takes no arguments.*

Declaring Variables

Knowing the data types available in C++ is one thing, but we also need to know how to declare variables in order to make use of these data types. The syntax for variable declaration is in the form:

```
DataType VariableName [= InitialValue];
```

Table 4-1. C++ Data Types

NAME	SIZE (BYTES)	VISUAL BASIC EQUIVALENT	RANGE
short	2	Integer	–32,768 to 32,767
unsigned short	2	N/A	0 to 65,535
bool	1	Boolean	**true or false
int	*4	Long	–2,147,483,648 to 2,147,483,647
unsigned int	*4	Long	0 to 4,294,967,295
long	4	Long	–2,147,483,648 to 2,147,483,647
unsigned long	4	Long	0 to 4,294,967,295
char	1	N/A	–128 to 127
unsigned char	1	Byte	0 to 255
enum	*4	Long	Same as int
float	4	Single	3.4E +/- 38 (7 digits)
double	8	Double	1.7E +/- 308 (15 digits)
long double	10	N/A	1.2E +/- 4932 (19 digits)
void	0	N/A	Used to show an absence of value

* Assumes Win32 platform.
** In C++ the value *true* equates to any non-zero value, while in Visual Basic the value *True* equates to 1. In both cases *false* equates to 0.

An example declaration is as follows:

```
int x = 0;
```

This statement assigns the value `0` to the variable `x`. In fact, any of the numeric data types in C++ (which include short, int, long, float and double) can be initialized in this way. Now you may have noticed that the keyword `unsigned` appears before some of the types in our data type chart. This is because C++ supports the idea of unsigned numbers. An *unsigned number* is simply a number whose value cannot be less than zero. Unsigned numbers are useful for several applications, such as representing binary data. (In contrast, Visual Basic has only one unsigned data type, Byte, which covers the range 0 to 255.) For example, if I wanted to assign the number 14 to `x` in my code, I would write the following assignment statement:

```
unsigned int x = 14;
```

> **NOTE:** `unsigned` *can prefix most C++ numeric data types.*

The numeric data types are straight forward, but there are several data types that require a little more explanation as to how they are used. We'll look at those now.

The char Data Type

The char data type is quite flexible. It can be used for its namesake and hold a single character with an assignment such as the following:

```
char myChar = 'a';
```

Notice the use of single quotes surrounding the initializing value of a. When you want to assign a single character to a variable of type char, you must always surround it with single quotes. You may assume that because we told our program to store the character a in memory that it did. Well, actually it didn't. Don't assume that we have a renegade variable here; it is just that a computer is not able to store characters—only numbers.

So what does get stored as the value of `myChar`? The number 61, which just happens to be the hexadecimal equivalent of the number 97, which just happens to be the ASCII (American Standard Code for Information Interchange) value for the character a.

Make sense? Good. Now don't let this confuse you. The key thing to remember is that all of the data in a computer is stored as numbers. It is the context in which we use those numbers that generates meaning. Think of the saying, "One man's trash is another man's treasure." To someone just looking at the memory stored in our program, the number 61 doesn't mean anything. But, because we know that 61 is the ASCII representation of a character, to us it means a lot. Just remember that all data interpretation is relative to its intended use. (This concept will start to make more sense when we cover the `printf()` function later in this chapter.)

The char[] Data Type

The char[] data type is not a unique data type; it is just an array of char, or basically a string of characters. We could use this data type to store any character

Morrison_001_224 8/22/00 11:53 AM Page 36

string that is too large for the type char. (In other words, a string that is more than one character long.) Following is an example usage of type char[]:

```
char myChars[255];
```

This declaration is pretty much identical to the following declaration in Visual Basic:

```
Dim myChars As String * 255;
```

Both of these declarations allocate space for 255 characters. They cannot grow in size later in the program in which they are used, which is limiting in most cases. Where this type of declaration is useful is in a situation where you know the maximum length of a string. However, there is another way to declare a char[ ], shown here:

```
char myChars[] = "Hello";
```

This declaration is identical to the following:

```
char myChars[6] = "Hello";
```

In other words, when a char[] is set to a string with no size specified, the compiler counts the number of characters in the string and automatically sets the char[ ] to that size plus one. Why plus one? The extra space is needed to hold the *NULL terminator* (which I cover in the next chapter). What is interesting about char[ ] is that you can access any of the members of the array directly by using standard array notation. For example, the following statement references the 'e' in "Hello":

```
myChars[1]
```

Why `[1]`? Because unlike the schizophrenic Visual Basic (which has arrays starting at one and zero), arrays in C++ are always 0-based. The first element of an array is at index 0, such that `myChars[0]`,from our previous example, is equal to the letter 'H' in "Hello."

The enum Data Type

The enum data type is the identical cousin of the enum construct in Visual Basic. To declare an enumeration we use the following syntax:

```
enum RETURN_VALUES
{
    RET_OK,
    RET_FAILED,
    RET_UNDEFINED
};
```

Now, just as in the Visual Basic version of enum, if no specific value is set for an element, all of the elements are incremented in order of appearance starting at zero. For example, in our previous declaration, `RET_OK` equals 0, `RET_FAILED` equals 1 and `RET_UNDEFINED` equals 2. If any member is set with a value then the next member that does not have a value gets incremented based on that number. Consider the following alteration to our enum:

```
enum RETURN_VALUES
{
    RET_OK,
    RET_FAILED = 100,
    RET_UNDEFINED
};
```

In this version of our enum, `RET_OK` equals 0, `RET_FAILED` equals 100, and `RET_UNDEFINED` equals 101. This works exactly as in Visual Basic.

> **NOTE:** *While* enums *allow for assigning some values and not others, for the sake of readability it is generally better to assign all of the values in an* enum *or assign none at all.*

User Defined Types

Although the pre-defined types in C++ are useful, sometimes you just need to create your own custom type. In Visual Basic we do this with a User Defined Type (UDT) as shown here:

```
Private Type MyType
    lngNum1 As Long
    lngNum2 As Long
End Type
```

We can then use it in code like this:

```
Private Sub Form_Load()

Dim t As MyType

t.lngNum1 = 1
t.lngNum2 = 2

End Sub
```

This creates a flexible way of representing custom data types and data relationships. It should be no surprise that C++ also has the ability to express UDTs. In C++, a struct is used to do this. The syntax for creating a struct is as follows:

```
struct myStruct
{
    int num1;
    int num2;
};
```

We can then use it like so:

```
struct myStruct x;
x.num1 = 1;
x.num2 = 2;
```

Notice that it is similar to the Visual Basic version except that we have to prefix our variable declaration with the keyword `struct` in C++.

Standard Library Functions

Visual Basic makes it easy to become spoiled by the number of things that are done for us by the development environment and the language itself. The fact that all of Visual Basic's language facilities are at our fingertips as soon as we open a new project is not only useful but also subversively amazing. We don't have to worry about where Visual Basic keeps the `MsgBox()` function or where the `InStr()` function lives. We just call them up as we need to and they magically appear.

In C++ however, this is not the case. We get nothing by default; we have to ask for anything we want. The way to ask for things in C++ is to use the `#include` directive. Remember that `#include` is used to bring another file (usually a header file) into our program. If we want to print to the screen from our program, we

Morrison_001_224 8/22/00 11:53 AM Page 39

have to either write a function to do so, or we have to `#include` a header file that links to or contains a function that does screen output. There is no function in the C++ language itself that prints to the screen. Now, anyone with any previous exposure to C++ will probably disagree with me and say that there are in fact functions in C++ that print to the screen. And, they'd be right—sort of. Let me explain.

In C++ there is an entity known as the *Standard C++ Library*, which is a set of header files that link to the C++ runtime library. Every implementation of the C++ compiler ships with these header files and the runtime library. These header files are the C++ programmers interface into the C++ runtime library. The Standard C++ Library is a set of language support files that make using the language easier and more efficient. They are a part of the American National Standards Institute (ANSI) C++ standard but are not an official part of the language.

Whether or not the Standard C++ Runtime Library is part of the C++ language is only a matter of semantics, because the fact of the matter is that this library contains the functions that all C++ programmers use to write C++ programs. You may be wondering why I brought up this whole discussion if it is just about semantics. Here's why: Let's suppose that the C++ runtime has a function named `printToScreen()` and I write C++ source code that uses that function. I would expect that when I compiled my program that the compiler would find that function in the standard C++ library, but it doesn't. Remember from Chapter 3 that the linker only knows about things that are on its "to do" list, which comes in the form of function prototypes and declarations. In my source code I never told the linker anything about the `printToScreen()` function. I just used it in my code. This is no good because the linker has to take my call to that function and convert to a function pointer that points to the real implementation of `printToScreen()`, but in this case there is no declaration for the linker to resolve with. Therefore, if I want to use the `printToScreen()` function, I have to `#include` the header file that contains the declaration of that function.

There are roughly 50 header files in the Standard C++ Library, which correlate directly to the C++ runtime library. These headers are divided into logical opera-

> **NOTE:** *Due to the vastness of the Standard C++ Library, I will not attempt to detail each one here. I will instead explain the use of each function that we use in our example programs. Microsoft has done a good job of documenting and providing example uses of each function in the C++ runtime library. I suggest that you poke around through each header file and look up the usage of the various functions. You can also search by functionality in the help file. It is also worth noting that in Windows programming, many of the C++ Runtime Library functions have more efficient counterparts available through the Win32API. However, it is useful to know the common C++ Runtime Library functions since you will be writing C++ code.*

tional units. For example, the Standard C++ Library file, `<stdio.h>`, defines the standard input/output functions available from the C++ runtime library. Ultimately, if you want to use one of the functions in the runtime, you have to include the proper header file.

Program Structure

Okay, now that we have some understanding of the basic elements of C++, let's try to put them to use in an application.

C++ programs are structured pretty much like Visual Basic programs. They consist of code modules that contain functions, variables, and constants (which should all be familiar terms). There are, however, a few elements of C++ that will be completely foreign to most Visual Basic programmers. Not to worry—by the end of this book they will be as familiar as "Option Explicit" is to you today. I am a firm believer in learning by example, so, let's write a simple C++ program and then examine each part of it in detail in order to learn the different components that make up a C++ program.

To start, open Visual Studio and select File ➔ New. Then select "Win32 console application" as our project type. In the Project Name field of the New Project dialog box, enter *ex01_ch4*. When prompted for the type of project, select "A simple application." as shown in Figure 4-1.

After clicking OK, your development environment should look similar to the environment shown in Figure 4-2.

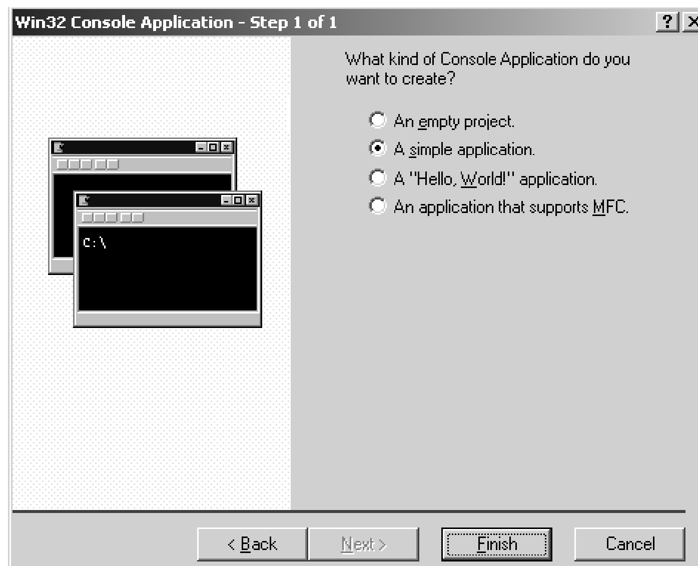

Figure 4-1. The "New" project dialog box

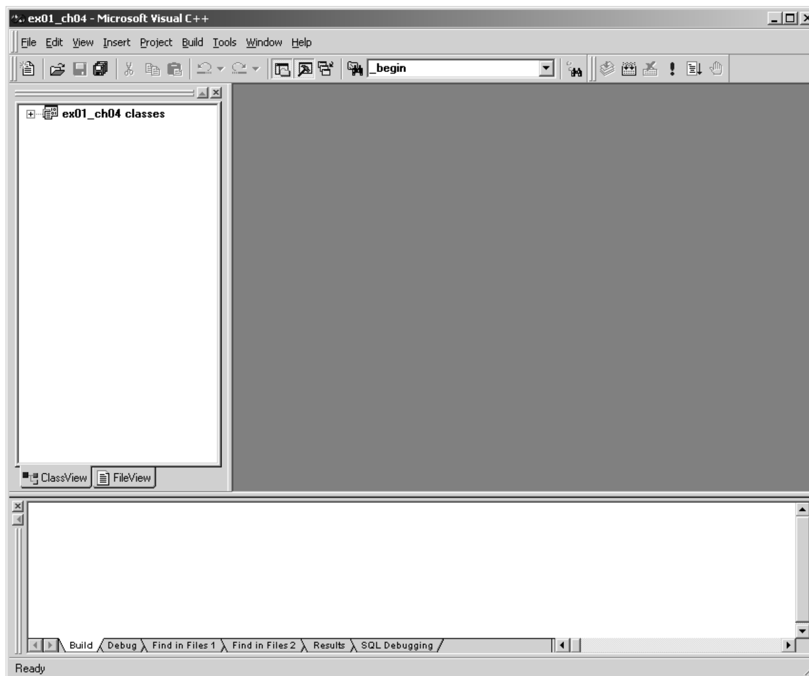

Figure 4-2. The Visual Studio development environment with ex01_ch04 loaded

Notice that there are two views available for the project: Class View and File View. These choices are available by selecting one of the aforementioned tabs at the bottom of the Workspace window on the left side of the development environment (assuming you haven't moved it), as shown in Figure 4-3.

We want to select File View as our view. This enables us to see all of the files in our project grouped by type. Let's open the Source Files node on the treeview control in the File View window. You should see two files in the node: ex01_ch4.cpp and stdafx.cpp. The contents of the file should display in your development environment's code window. Now, add the following code to your open file so that it looks like the code listed here:

```
// ex01_ch4.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
//variable declarations
int iPublicVar = 0;

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

```
    printf("iPublicVar == %d.\n",iPublicVar);
    printf("The sum of 5 and 4 is %d.\n",addNums(5,4));
    changeVar();
    printf("iPublicVar == %d.\n",iPublicVar);
    changeVar();
    printf("iPublicVar == %d.\n",iPublicVar);
    return 0;
}

//function to add numbers
int addNums(int iNum1, int iNum2)
{
    int iSum = 0;
    iSum = iNum1 + iNum2;
    return iSum;
}

//sub to change a variable.
void changeVar()
{
    iPublicVar++;
}
```

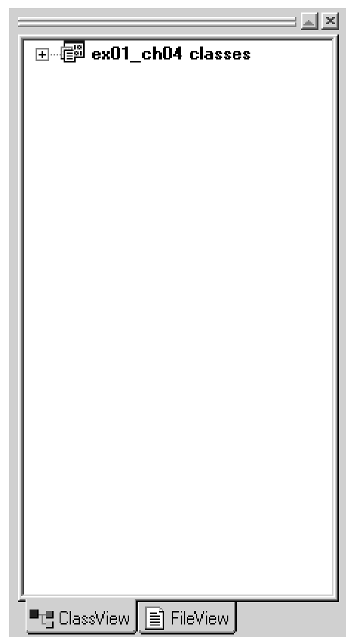

Figure 4-3. The Class View and File View tabs

Morrison_001_224 8/22/00 11:53 AM Page 43

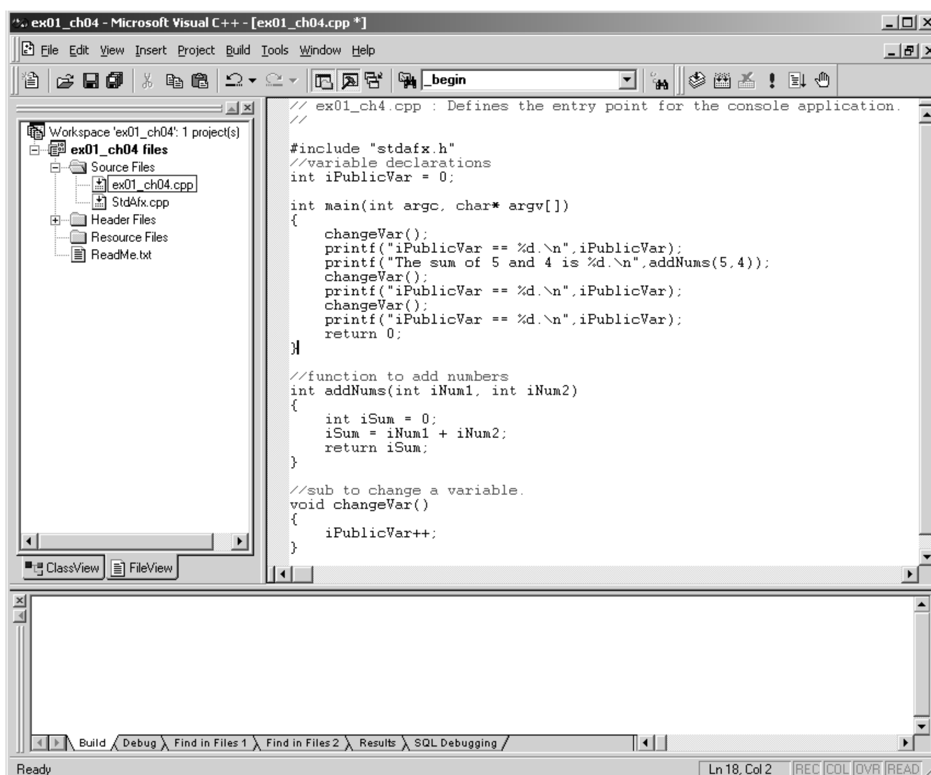

Figure 4-4. Visual Studio in File View with the project ex01_ch04 *loaded*

Your development environment should now look like the environment shown in Figure 4-4.

Remember from Chapter 3 that the `#include` preprocessor directive copies the contents of one file into another at compile time. Well, guess what the first line of code in our program does. Yep, `#include`s `stdafx.h`. The reason we do this is because we want to include the contents of `stdafx.h` in our source code file before it gets compiled. (Makes sense, right?)

Generally, header files contain variable declarations, constant definitions, or preprocessor directives (including `#include`; yes you can have nested `#include`s). One of the most common elements found in a header file is a function prototype. *Function prototyping* is the first one of those "foreign concepts" I mentioned earlier in this chapter, so let me explain. In Visual Basic we always use Option Explicit, which forces us to declare variables in our program before we use them. C++ takes this idea one step further by requiring function prototypes in our program before we use them. But, before we can go too far I must explain how a C++ function is organized. The form of a C++ function is as follows:

Morrison_001_224 8/22/00 11:53 AM Page 44


```
returnValueType functionName([arguments]){functionBody}
```

This is in contrast to a Visual Basic function, which takes the following form:

```
[Public or Private] Function functionName([arguments]) ReturnValueType
```

Let's look at two examples of a function prototype:

```
int addNums(int iNum1,int iNum2); //first function prototype
void changeVar(); //second function prototype
```

These are both examples of function prototypes. You may notice that the prototypes don't contain any executable code. So then, what is the point of having a function prototype? If you remember from Chapter 3, we said that when the compiler runs, it generates an error if it hits an unrecognized symbol (such as an undefined variable or function). The way we define a symbol is by either defining a variable or writing a function prototype. What a prototype does is tell the compiler that there is a function matching its description somewhere in the program. Once this is done, the compiler is satisfied. The next step is for the linker to go and actually find the executable code that implements the function prototype and put it into the program.

Think of this process as similar to having a credit card. You may have a credit card (function declaration) in your possession, which suggests that you will be able to pay for a purchase. However, when you go to the counter the cashier (the linker) will check with the credit card company to verify that you have available credit for your purchases (function's implementation). If she finds that you do (the implementation is found), you can take your purchases and go (your program gets linked into an .exe). On the other hand, if your credit card is rejected (no function implementation is found), you will not be able to purchase your items (your program will not be linked into an .exe).

Now, let's look at our header file, `stdafx.h`, after you've added the function prototypes for the functions added earlier to the file ex01-ch4.cpp:

```
//    stdafx.h : include file for standard system include files,
//    or project specific include files that are used frequently, but
//    are changed infrequently
//

#if !defined(AFX_STDAFX_H__7AC25963_B4D4_11D3_BC89_901151C10000__INCLUDED_)
#define AFX_STDAFX_H__7AC25963_B4D4_11D3_BC89_901151C10000__INCLUDED_

#if_MSC_VER > 1000
#pragma once
```

```
#endif // _MSC_VER > 1000

#define WIN32_LEAN_AND_MEAN    // Exclude rarely-used stuff from Windows headers

#include <stdio.h>

//function prototypes
int addNums(int iNum1,int iNum2);
void changeVar();

// TODO: reference additional headers your program requires here

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif // !defined(AFX_STDAFX_H__7AC25963_B4D4_11D3_BC89_901151C10000__INCLUDED_)
```

You can see the function prototypes for our two functions, `addNums()` and `changeVar()`. We also defined one public variable, `iPublicVar`, whose initial value is set to `0`. Because we `#included` `stdafx.h` in our source code file, ex01_ch4.cpp, our function prototypes are recognized before the compiler runs into our function calls in our program.

> **NOTE:** *The order of the function prototypes and function calls is important. The function prototype must appear before any call to the function that it represents. It is for this reason that function prototypes usually live in header files that get* `#included` *into a source file where the actual function is called.*

Continuing on with our program, we see the line after our `#include` is the actual entry point of our application. We know this because of the fact that it is the definition of the function `main()`. Let's look at it:

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

We know from our previous definition of a C++ function that the left most symbol, `int`, indicates that our function will have a return value type of `int`. And that the next symbol, reading from right to left, is our function's name; `main`. The next two sets of declarations, `int argc` and `char* argv[]`, are our programs arguments. The first argument tells us the number of arguments that were passed to our program, while the second argument is an array of char* that actually con-

tains the arguments. We will leave this discussion until after we have discussed pointers, but suffice it to say that with these two parameters we can accept any number of arguments from the command line.

Curly Braces { }

In case you're not familiar with the term, *scope* identifies a certain section of code as being a cohesive unit. In other words, anything defined in that particular scope is visible to all other elements of that scope. If a variable is declared inside of a function, it is said to have *Function Scope*. If a variable is declared in a public code module not contained within a function, it is said to have *Global Scope*. When a scope is exited (such as a function returning), all items declared in that scope (such as variables) disappear as well. In C++ you can create scope by using curly braces ({ }).

If you declare a variable after a curly brace, it will only be in scope until the closing curly brace is encountered. In some cases a curly brace is optional, as in an *if* statement, but I find it much more clear to always enclose a related group of statements together in a set of curly braces. (See "The if Statement" later in this chapter for more information on *if* statements.)

Semi-Colon ;

The next line in our program is a call to `changeVar()` followed by a semi-colon. A semi-colon in C++ is used to terminate a statement. This is exactly the same as in Visual Basic, except that Visual Basic uses an *end of line* character to terminate a statement.

So does that mean that every line of code in C++ will be terminated with a semi-colon? No. Consider the following:

```
#define CALL_CHANGE_VAR        changeVar()
```

This line of code sets `CALL_CHANGE_VAR` as an alias for the text "changeVar()." So why doesn't this line need to be terminated with a semi-colon? This line is not an executable statement. Remember from Chapter 3 that `#define` is a preprocessor directive, not a compiler directive. Before our program compiles, the preprocessor will go through our source code and replace all occurrences of `CALL_CHANGE_VAR` with the text "changeVar()." I would use this in code as follows:

```
{
//Other program statements here
CALL_CHANGE_VAR;
//Other program statements here
}
```

Morrison_001_224 8/22/00 11:53 AM Page 47

Now after the preprocessor runs, this code fragment would look like this:

```
{
//Other program statements here
changeVar();
//Other program statements here
}
```

Just as in Visual Basic, in C++ there is no requirement that a statement be on a single line. Consider the following statement:

```
int x = 2 + 2;
```

I could have just as easily (and correctly) written it as:

```
int x
=
2
+
2;
```

An equivalent statement in Visual Basic would look like this:

```
Dim x As Integer
x _
= _
2 _
+ _
2
```

As you can see, C++ does not require a "line continuation character" (like Visual Basic does) to continue a program statement to the next line. Use this technique judiciously, however, as it can make the source code more difficult to read.

printf()

Our next program statement:

```
printf("iPublicVar == %d.\n",iPublicVar);
```

makes use of the Standard C++ Library function, `printf()`. This function is used to print to the standard output device (which is usually a computer monitor's

Morrison_001_224 8/22/00 11:53 AM Page 48

screen). The function prototype for this function is in the stdio.h header file, which we `#included` in our program inside of our header file, stdafx.h. The `printf()` prototype looks like this:

```
int printf(const char *, ...);
```

Now, if we dissect this prototype, we can see that `printf()` returns a value of type `int` (which indicates the number of characters written to the screen, or a negative number if there was an error). You may notice that the arguments section of our function prototype look a bit odd. This is because we have just encountered two new C++ language elements, one of which is the *ellipsis* (the three periods next to each other in the place of the functions second parameter). An ellipsis is simply a matter of notation that tells the compiler that this function will accept any number and type of arguments. The compiler then disables type-checking for the additional parameters (because there is nothing to check against).

This is considered bad programming practice in most cases. However, there are some functions that necessitate using the ellipsis. I will not go into the specifics of implementing a function that uses an ellipsis, but you will probably see them from time to time. For example, `printf()` is one of those functions that makes legitimate use of the ellipsis. To see why, let's continue examining our call to `printf()`:

```
printf("iPublicVar == %d.\n",iPublicVar);
```

The first parameter we pass to `printf()` in the previous code is an ordinary string literal. There is, however, something that looks a little odd contained within that ordinary string literal. First, you may have noticed a `%d` in the place where we want our variable's (`iPublicVar`) value to be displayed. Then at the end of our string literal you may also have seen a `\n`. This brings to the surface two more new C++ language features: format specifiers and escape sequences. First let's examine the area of format specifiers.

Format Specifiers

The `f` in `printf()` stands for *format*, which means `printf()`, prints a formatted string. In fact, `printf()` is somewhat similar to the Format function in Visual Basic. However, to achieve the effect of our call to `printf()` with Visual Basic, we have to use concatenation. Let's see how this looks:

Morrison_001_224 8/22/00 11:53 AM Page 49


```
Dim strMessage As String
Dim lngPublicVar As Long

Public Sub Main
lngPublicVar = 5
    Print "lngPublicVar = " & lngPublicVar & "." & vbCrLf
End Sub
```

NOTE: Concatenation *is the process of joining various string literals and variables together to form one string.*

With these few lines of code we concatenate the variable `lngPublicVar` to the string literal `"lngPublicVar = "`. We also add a `vbCrLf` to the end of the line to ensure that we send a carriage return/line feed to the screen. `printf()` produces the same result, but using *format specifiers* instead. `printf()` actually takes the format specifiers from our string literal and replaces them with the values of the variables passed in to the ellipsis parameter. Remember, because we are not restricted by the number of arguments we can pass to `printf()`, we could actually use `printf()` as follows:

```
printf("iPublicVar == %d.%d%d\n",iPublicVar,iPublicVar,iPublicVar);
```

This would produce the following output (assuming that `iPublicVar` equals 5):

```
"iPublicVar == 5.55"
```

You may be wondering why I put a `%d` as the format specifier. Well, `%d` tells `printf()` to interpret the variable passed to it as a decimal number (hence, `d` for decimal).

NOTE: *There are many other format specifiers available in C++. All of the C++ format specifiers and their respective meanings and uses are available in the online documentation provided with Visual Studio 6.0. We will be seeing some of the additional format specifiers used throughout the rest of the book though.*

Escape Sequences

Much like the format specifiers we just looked at, escape sequences tell C++ functions how to format output. In the above example of printf:

```
printf("iPublicVar == %d.%d%d\n",iPublicVar,iPublicVar,iPublicVar);
```

The `\n` after the three `%d` format specifiers is the "newline" escape sequence. It tells the printf function to force the output device to place a newline (or carriage return) in the place the `\n` escape sequence occupies. There are many escape sequences available in C++. They are all listed with their meanings and uses in the Visual Studio Help File. The next line in our program:

```
printf("The sum of 5 and 4 is %d.\n",addNums(5,4));
```

makes another call to `printf()`, but instead of passing a variable as the second parameter, we pass the results of a function call, `addNums(5,4)`. This may seem strange, but the statements in the second parameter position of a call to `printf()` are always fully evaluated before being passed to the actual call, `printf()`. For example, our function, `addNums()`, evaluates to the sum of the parameters passed to it. For all intents and purposes, `printf()` doesn't know the difference between a function in its parameter list and an actual hard coded value. As far as `printf()` is concerned, the following two calls are identical:

```
printf("The sum of 5 and 4 is %d.\n",addNums(5,4));
printf("The sum of 5 and 4 is %d.\n",9);
```

Most of the rest of the code in our program is just a repeat of earlier calls with one exception:

```
    return 0;
}
```

The keyword *return* is used to exit a function. In this case we tell the program to exit the function `main()` and return a value of `0`. To exit a function that does not return a value, you just call return with no value as such:

```
return;
```

Figure 4-5 shows the output of our program.

Granted, this example is fairly useless as an application, but it explains the structure of a C++ program pretty well.

Operators

In order to benefit from C++ we are going to perform various operations on our data, so it's important that you learn about some of the operators available in C++. I listed the most relevant C++ operators, their Visual Basic equivalents (where applicable), and a description of the functionality they provide in Table 4-2. A complete list of C++ operators is available in the C+ help file.

Morrison_001_224 8/22/00 11:53 AM Page 51


```
"D:\projects\APress\ex01_ch04\Debug\ex01_ch04.exe"
iPublicVar == 1.
The sum of 5 and 4 is 9.
iPublicVar == 2.
iPublicVar == 3.
Press any key to continue
```

Figure 4-5. The output of ex01_ch04.exe

Table 4-2. C++ Operators

C++ OPERATOR	VB EQUIVALENT	DESCRIPTION
+	+	Addition
-	-	Subtraction
*	*	Multiplication
/	/	Division
%	%	Remainder from a division (modulus)
++	N/A	Increment operator
--	N/A	Decrement operator
=	=	Assignment
==	=	Equality
!=	<>	Inequality
>	>	Greater than
<	<	Less than
>=	>=	Greater than or equal to
<=	<=	Less than or equal to
&	And	Bitwise AND
&&	N/A	Logical AND
|	Or	Bitwise Or
||	N/A	Logical Or
~	Not	Bitwise Not (complement)
!	N/A	Logical Not
+=	N/A	Addition and assignment
-=	N/A	Subtraction and assignment
*=	N/A	Multiplication and assignment
/=	N/A	Division and assignment

Morrison_001_224 8/22/00 11:53 AM Page 52

In order to get a better feel for the operators in C++ let's build a small program that will use a variety of the operators as well as some of the logic control structures such as for-loops and select-case. Don't expect to learn too much about good programming practice from this application, just use it to learn about the use of the C++ operators. Again, I am only reviewing a few of the most commonly used operators.

Open a new console application project in Visual Studio and copy the contents of the source file *ex02_ch4* and the header file *stdafx.h*, which are listed here. (This project is also located on the Web site that accompanies this book at `www.apress.com`.)

The contents of the file ex2_ch4.cpp are

```
// ex02_ch4.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

int i1 = 0;

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

    //increment i1 by 5
    printf("Begin for-loop with (i1 == %d)...\n",i1);
    for (int a = 0; a < 5; a++)
    {
        printf("i1 == %d.\n",i1++);
    }
    printf("End for-loop with (i1 == %d)...\n\n",i1);


    //decrement it back down to 0
    printf("Begin do-while-loop with (i1 == %d)...\n",i1);
    do
    {
        printf("i1 == %d.\n",i1--);
    }while (i1 > 0);
    printf("End do-while-loop with (i1 == %d)...\n\n",i1);

    //increment i1 by 5
    printf("Begin for-loop with (i1 == %d)...\n",i1);
    for (int b = 0; b < 5; b++)
    {
        printf("i1 == %d.\n",++i1);
```

Morrison_001_224 8/22/00 11:53 AM Page 53


```
    }
    printf("End for-loop with (i1 == %d)...\n\n",i1);

    //increment i1 by multiples of 5
    printf("Begin for-loop with (i1 == %d)...\n",i1);
    for (int c = 0; c < 5; c++)
    {
        printf("(i1 += 5) == %d.\n",(i1 += 5));
    }
    printf("End for-loop with (i1 == %d)...\n\n",i1);

    //bitwise and logical operations
    printf("Begin logical and bitwise operations with (i1 == %d)...\n",i1);
    printf("(i1 & 2) == %d\n",i1 & 2); //bitwise
    printf("(i1 && 2) == %d\n",i1 && 2); //logical
    printf("(i1 | 2) == %d\n",i1 | 2); //bitwise
    printf("(i1 || 2) == %d\n",i1 || 2); //logical
    printf("(!i1) == %d\n",!i1); //logical
    printf("(~i1) == %d\n",~i1); //bitwise
    printf("End logical and bitwise operations with (i1 == %d)...\n",i1);

    return 0;
}
```

The contents of the file stdafx.h are

```
// stdafx.h : include file for standard system include files,
//    or project specific include files that are used frequently, but
//    are changed infrequently
//

#if!defined(AFX_STDAFX_H__ED10E577_1CBD_4B53_8CB3_FC5D51278B80__INCLUDED_)
#define AFX_STDAFX_H__ED10E577_1CBD_4B53_8CB3_FC5D51278B80__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define WIN32_LEAN_AND_MEAN    // Exclude rarely-used stuff from
#include <stdio.h>

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.
#endif // !defined(AFX_STDAFX_H__ED10E577_1CBD_4B53_8CB3_FC5D51278B80__INCLUDED_)
```

Morrison_001_224 8/22/00 11:53 AM Page 54


```
"D:\projects\APress\ex02_ch4\Debug\ex02_ch4.exe"
Begin for-loop with (i1 == 0)...
i1 == 0.
i1 == 1.
i1 == 2.
i1 == 3.
i1 == 4.
End for-loop with (i1 == 5)...

Begin do-while-loop with (i1 == 5)...
i1 == 5.
i1 == 4.
i1 == 3.
i1 == 2.
i1 == 1.
End do-while-loop with (i1 == 0)...

Begin for-loop with (i1 == 0)...
i1 == 1.
i1 == 2.
i1 == 3.
i1 == 4.
i1 == 5.
End for-loop with (i1 == 5)...

Begin for-loop with (i1 == 5)...
(i1 += 5) == 10.
(i1 += 5) == 15.
(i1 += 5) == 20.
(i1 += 5) == 25.
(i1 += 5) == 30.
End for-loop with (i1 == 30)...

Begin logical and bitwise operations with (i1 == 30)...
(i1 & 2) == 2
(i1 && 2) == 1
(i1 | 2) == 30
(i1 || 2) == 1
(!i1) == 0
(~i1) == -31
End logical and bitwise operations with (i1 == 30)...
Press any key to continue
```

Figure 4-6. ex02_ch4.exe

Figure 4-6 displays the output from the program when it is run.

There certainly is a lot going on in this little program. First and foremost there are a lot of mathematical operations. Now I won't spend time going over the obvious operations such as addition and subtraction, but I will briefly explain the operators that do not have a Visual Basic equivalent.

The ++ and -- Operators

The origin of the name for the C++ programming language comes from the ++ operator. The ++ operator is a unary operator that increments the variable that it precedes or follows. The C programming language was an improvement to the proprietary Bell Labs programming language "B". (*B* presumably stands for *Bell.*) And C++ was an improvement on the C programming language. Therefore, C++ should theoretically be called "D." However, the inventors of C++, being the jokesters they are, decided to increment the C programming language by calling it C++.

Morrison_001_224 8/22/00 11:53 AM Page 55

There is an important point to be made about the placement of the ++ operator. If the ++ operator comes before the operand, such as:

```
x = ++i;
```

the value `i` will be incremented before its current value is assigned to `x`. This is called a *prefix operator*. The postfix version of the ++ operator (or one that comes after the operand) is used as in the following:

```
x = i++;
```

In this case, `x` will be assigned the current value of `i` before it is incremented. This is an important difference to note, as you can see from the example. Our first for-loop (we will discuss the for-loop statement later in this chapter) starts at 0 and ends with 4 by using the postfix version of the ++ operator. However, the second for-loop using the same exact logic and data, prints the values 1 through 5 because the ++ operator is a prefix.

Following is the C++ operation followed by the equivalent Visual Basic code necessary to accomplish the same output.

The C++ version (postfix) is

```
x = i++;
```

The Visual Basic version (postfix) is

```
x = i
i = i + 1
```

The C++ version (prefix) is

```
x = ++i;
```

The Visual Basic version (prefix) is

```
i = i + 1
x = i
```

The - and -- operators follow the same exact principles as the + and ++ operators, but for subtraction.

Binary Operators

You may not realize that Visual Basic does not have any boolean operators. Daft you say? Well it's true (no pun intended). To prove my case let's do a test.

In Boolean logic, "Not 1" should equal 0 (or false). However, if I run this statement in Visual Basic, I get the result of –2. The reason is that Visual Basic does only bitwise operations. In most cases this doesn't make a difference, but you do need to be careful. Consider the following Visual Basic code:

```
Option Explicit

Private Sub Form_Load()

'Check the return value of the function
If Not GetVal Then
    MsgBox "Function failed."
Else
    MsgBox "Function succeded."
End If

End Sub

Public Function GetVal()

'Return what most API functions will return as 'true'
GetVal = 1

End Function
```

Following the logic of this code, you can see that we are checking the return value from our call to `GetVal()` and continuing or terminating our program based on what that return value is. Now, because our version of `GetVal()` always returns 1 (or true, as any non-zero value is true according to Boolean logic), we would expect the following statement:

```
If Not GetVal()
```

to always evaluate to true (or "not false"). However, in this code example it does not. If you run this code, you will see that our *if* statement evaluates to false (or "Not true"). Let's examine why this happens.

Our `GetVal()` function returns 1, which is binary `00000001`. When we apply the *Not* operator to the return value of 1, we are not doing a logical Not, but a bitwise Not operation. The bitwise Not of the binary value `00000001` equates to

Morrison_001_224 8/22/00 11:53 AM Page 57

`11111110`, which equates to the decimal value (-2). The value of *True* in Visual Basic is equal to decimal (-1). And because (-1) and (-2) are not equal, our *if* statement will always fail.

> **TIP:** *If you don't understand bitwise operations, run, don't walk, and learn basic binary mathematics. It will prove invaluable in your understanding of computer programming and computers in general.*

The reason that I point this out is that most application program interface (API) functions use 1 as their return value to indicate *True* and 0 to indicate *False*. Imagine what would happen if `GetVal()` was an API function. This could lead to some very hard to find bugs. Let's look at a more realistic example of how this could happen:

```
Option Explicit

Public Declare Function SetWindowText Lib "user32" Alias "SetWindowTextA" (ByVal
hwnd As Long, ByVal lpString As String) As Long

Public Sub main()

'Variable to catch the return value of the API call
Dim lngRetVal As Long

Load Form1

'Call SetWindowText to change the text in our window
lngRetVal = SetWindowText(Form1.hwnd, "My Title has Been Changed")

'Check the return value of SetWindowText
If Not lngRetVal Then
    'SetWindowText returned false
    MsgBox "Error: Could not change the window's title."
Else
    'SetWindowText returned true so....
    'Continue with the program
End If

Form1.Show

End Sub
```

Morrison_001_224 8/22/00 11:53 AM Page 58

This simple example shows a very real bug. The return value of `SetWindowText` is documented to be "non-zero on success," which means that `SetWindowText` is successful if it returns any value other than zero. Now when Visual Basic evaluates an expression such as:

```
Dim x As Integer

x = 1

If x Then
```

it will evaluate the *if* statement as true. This is because in this grammar, Visual Basic is only checking for the absence of zero. However, the following statements would evaluate as false:

```
Dim x As Integer

x = 1

If x = True Then
```

This is because Visual Basic will now compare x (1) and *True* (-1) and (correctly) return the fact that they are not equal; such that:

```
(1 = (1 = True)) = False
```

Fear not. There is a simple solution to this problem. Always assign values that are logically either true or false to a Boolean variable. Take our previous code for example. If we had made the small change noted here in bold, our program would run correctly:

```
Option Explicit

Public Declare Function SetWindowText Lib "user32" Alias "SetWindowTextA" (ByVal
hwnd As Long, ByVal lpString As String) As Long

Public Sub Main()

'Variable to catch the return value of the API call
Dim lngRetVal As Boolean 'Changed from Long to Boolean

Load Form1

'Call SetWindowText to change the text in our window
```

Morrison_001_224 8/22/00 11:53 AM Page 59


```
lngRetVal = SetWindowText(Form1.hwnd, "My Title has Been Changed")

'Check the return value of SetWindowText
If Not lngRetVal Then
    'SetWindowText returned false
    MsgBox "Error: Could not change the window's title."
Else
    'SetWindowText returned true so....
    'Continue with the program
End If

Form1.Show

End Sub
```

The reason is that Visual Basic coerces any non-zero value to true upon assignment to a Boolean type variable. We could accomplish the same thing by putting the `CBool` function around our `lngRetVal` variable when it was dim-ed as a Long. Here's the bottom line: in C++ you should always use the Boolean operators for comparison unless you actually need to perform a bitwise operation. For instance, if I wanted to compare the return value of a function for success I would do the following:

```
if (!SetWindowLong(hWnd, "My App"));
```

This statement uses the logical *Not* operator, which will give us the desired effect of checking for success or failure. However, the following would not:

```
if (~SetWindowLong(hWnd, "My App"));
```

Using this form of the *Not* operator would put us in the same boat that we were in with Visual Basic.

Assignment and Equality Operators

The last group of operators that I discuss is the assignment and equality operators. One of the most frequent bugs I used to create in my early C++ programs was a very innocent looking one:

```
if (var = condition)
{
    //Do something interesting
}
```

Morrison_001_224 8/22/00 11:53 AM Page 60

Looks pretty innocent, right? (Forget that we don't know the exact syntax of an *if* statement yet.)

Here is the equivalent code in Visual Basic:

```
If var = condition Then
'Do something interesting...
End If
```

Do you see the problem yet? No? Let's look at the C++ version a little closer. It appears at first glance that I am asking if the value `var` is equal to the value `condition`. However, what I am really asking is, if the return value of the statement `var = condition` is true or false. You see, in Visual Basic the equals sign means both assignment and equality. It uses the context of the call to decide which operation to use. In C++, however, the symbol "=" means to assign the value on the right side of the equals sign to the storage location on the left.

So the question still remains, "What does `var = condition` really equal?" Let's look at the following example:

```
printf("%d\n",i1 = 10);
printf("%d\n",i1 = 5);
printf("%d\n",i1 = 3);
```

This program will produce the values 10, 5, and 3, respectively. The answer to our question is: the return value of an assignment operation is always the value on the right hand side. Assuming that `i1` is equal to zero at this point in time, the following statement will return 0 (or false):

```
i1 = 0;
```

You should be able to see the possibility for hard to find bugs here. If you had built a conditional statement based on this statement being true, you'd be in big trouble.

Here's a simple trick to alleviate this kind of bug when you are checking for a constant value. Always put your constant value on the left side of the equals sign as such:

```
0 = i1;
```

This way, if you accidentally put in the assignment operator by mistake, your program won't even compile. The reason is that you can't assign a value to a constant.

Okay, so how do we check for equality? With the == operator. Here is the correct version of our previous statement:

Morrison_001_224 8/22/00 11:53 AM Page 61


```
0 == i1;
```

That's it. Just put an extra equal sign next to the old one and we're done.

Assignment and … Operators

The other types of operators that are new in C++ are the "two-for-one" operators, such as +=. These operators do two operations in one shot. Consider the following statement:

```
x += 10;
```

To see what this statement is doing, let's look at an equivalent Visual Basic statement:

```
x = x + 10
```

Pretty simple, right? All that these types of operators do is combine two operations into one. Refer to Table 4-2 and the sample program, ex2_ch4, for more uses of these operators.

Loops and Control Statements

C++ has loop and control facilities just like Visual Basic, although the syntax is different. We'll go through each type of these logic control structures and their respective use.

The if Statement

The *if* statement is probably the most widely used control statement of any programming language. Regardless of the programming language implementing it, they all work pretty much the same. In Visual Basic, an *if* statement looks like this:

```
If x = y Then
    'Do something
End If
```

All of the statements between `If` and `End If` are executed if the conditional statement is True. The same *if* statement in C++ looks like this:

```
If (x == y)
    //do something;
```

Morrison_001_224 8/22/00 11:53 AM Page 62

Now this is where the stylistic approach to programming comes into play. I feel that the previous line is a little unclear. I would, and always do, write my if statements with a set of curly braces surrounding the conditional statements as follows:

```
if (x == y)
{
    //do something;
}
```

By using the curly braces I am creating a `scope` for the *if* statement. I feel that this makes the code easier to read.

For Loop

The C++ for-loop construct is pretty much identical in function to the Visual Basic version. To get a feel for the similarities and differences in the two let's look at an example.

The Visual Basic version is

```
Dim i As Long
For i = 0 To 100
    'something interesting here
Next i
```

The C++ version is

```
for (int i = 0;i < 100;i++)
{
    //Something interesting here.
}
```

You already know what the Visual Basic version does so I'll jump right into the description of the C++ section. The first line starts with our declaration of the for loop. We do this by using the keyword *for*. Now just as in the Visual Basic for-loop, we must give a starting value for the loop. We use the statement `int i = 0` to accomplish this. Notice that unlike in Visual Basic, in the C++ version we can actually declare and initialize the iterating variable (in this case *i*) right inside the first parameter of the for-loop. Consequently, the variable `i` will have scope only in the for loop. Now all of the arguments of the for loop are separated by a semi-colon. The actual definition of the for-loop arguments are as follows:

```
for (iterting variable;conditional statement;action[s])
```

We satisfied the first parameter with our declaration and initialization of the variable `i`. Our *conditional* statement is

```
i < 100
```

which means "while i is less than 100," but I am sure you have already figured that out. Our final parameter is our *action* clause, which we have denoted as

```
i++
```

This simply means that each time the loop iterates the value of `i` will be incremented by one. Now we aren't tied to just iterating based on `i` increasing by one. I could have just as easily written a loop as follows:

```
for (int i = 0;i < 100;i+=5)
{
    //Loop statements here
}
```

This is identical to the following for-loop in Visual Basic:

```
For I = 0 To 100 Step 5
    'Loop statements
Next I
```

Do-While Loop

The do-while-loop is similar to the Visual Basic version, just as was the for-loop construct. Let's look at an example of these two constructs in the aforementioned languages.

The Visual Basic version is

```
Do
    'Statements Here
    If x > 10 Then
        Exit Do
End If
Loop While True
```

Morrison_001_224 8/22/00 11:53 AM Page 64

The C++ version is

```
do
{
    if (x > 10)
{
    break;
    }
}while (true)
```

Notice the use of the keyword, *break*. This keyword essentially tells the program to "break"-out of the loop. There is also a *continue* keyword that could be used in the place of *break*. The difference in the two is that *break* actually starts executing the next line after the loop statement, but *continue* causes the program to ignore the statements after the continue keyword and begin the next iteration of the loop immediately. Break is the equivalent of the *Exit* keyword in Visual Basic. Consider the following Visual Basic code:

```
Dim x As Integer

Do
    If x > 10 Then
    Exit Do
    End If
    x = x + 1
Loop

For x = 0 To 100
    If x > 10 Then
    Exit For
    End If
    x = x + 1
Next
```

Both the *Exit Do* and *Exit For* are used to exit loops. In C++, it is not necessary to explicitly state which structure you are breaking out of.

Switch-Case

Although the *switch* keyword may not look familiar to you, it is the first cousin of the *Select-Case* construct in Visual Basic. The switch-case construct takes the following form:

Morrison_001_224 8/22/00 11:53 AM Page 65


```
switch (condition)
{
    case case[n]:
        [break];
    case case[n + 1]:
        [break];
}
```

To continue our previous compare and contrast format, let's look at the control structure in both languages.

The Visual Basic version is

```
Dim I As Long

I = 10

Select Case I
    Case I = 1
        'Do something
    Case I = 3,4,5
        'Do something
    Case 10
        ' Do something great
    Case Else
        ' Default Action
End Select
```

The C++ version is

```
int I = 0;

switch (I)
{
    case 1:
        //Do something
        break;
    case 3:
    case 4:
    case 5:
        break;
    case 10:
        //Do something great
    default:
        //default action
}
```

Notice that the structure of the statement is pretty much the same as the Visual Basic version, with a couple of exceptions. The first exception is that unlike in Visual Basic, in C++ we must explicitly break out of the *switch* statement by using the *break* keyword. Second, we can check several conditions by simply listing them with no `break` between them as we did with 3, 4, and 5. The third and last exception is that we use the keyword *default* as our “catch-all” condition as opposed to the *Case Else* that is used in Visual Basic.

Comments

I saved this topic for last in this chapter because it is an important one. In C++ it is extremely important to write comments in your code, because there are so many ways to represent the same operations and manipulations of a program’s data. Believe me, I have spent hours and hours trying to understand other programmers’ uncommented, cryptic code only to find out that the time spent could have been reduced to nothing had the code included even one comment. (I’ll leave you and your conscience to deal with the matter from here.)

Following are two ways to specify comments in C++:

```
//This is a comment line. It works just like VB.

/* This is the C-style comment block. Unlike VB
even the comments here are blocked. Nothing will be
interpreted as code until the closing tag is encountered
like this */
```

And that’s it. Either start a line with the `//` symbol to comment only that line (this works just like the apostrophe in Visual Basic), or use the `/* */` pair to enclose a group of comments (I wish that Visual Basic had an equivalent for this one!).

Conclusion

Although far from all-inclusive, this chapter should have given you a good induction into the basics of C++. I suggest supplementing this chapter with the first few chapters of the definitive C++ guide, *The C++ Programming Language* by Bjarne Stroustrup, the inventor of C++ (2000, Addison-Wesley). It is available from any good technical book source. Other than that, get ready to attack the C++ language.

In the next few chapters we will continue our journey into the world of C++ by learning about pointers, classes, and templates. We will also pick up additional techniques and insights regarding the things we have already learned as we go along. Are we having fun yet?!