Chapter 1

Writing Your First C++ Program

In This Chapter

- Finding out about C++
- Installing Dev-CPP from the accompanying CD-ROM
- Creating your first C++ program
- Executing your program

Okay, so here we are: No one here but just you and me. Nothing left to do but get started. Might as well lay out a few fundamental concepts.

A computer is an amazingly fast but incredibly stupid machine. A computer can do anything you tell it (within reason), but it does *exactly* what it's told — nothing more and nothing less.

Perhaps unfortunately for us, computers don't understand any reasonable human language — they don't speak English either. Okay, I know what you're going to say: "I've seen computers that could understand English." What you really saw was a computer executing a *program* that could meaningfully understand English. (I'm still a little unclear on this computer-understanding-language concept, but then I don't know that my son understands my advice, either, so I'll let it slide.)

Computers understand a language variously known as *computer language* or *machine language*. It's possible but extremely difficult for humans to speak machine language. Therefore, computers and humans have agreed to sort of meet in the middle, using intermediate languages such as C++. Humans can speak C++ (sort of), and C++ is converted into machine language for the computer to understand.

Grasping C++ Concepts

In the early 1970s, a consortium of really clever people worked on a computer system called Multix. The goal of Multix was to give all houses inexpensive computer access to graphics, e-mail, stock data, pornography (just kidding), whatever. Of course, this was a completely crazy idea at the time, and the entire concept failed.

A small team of engineers working for Bell Labs decided to save some portion of Multix in a very small, lightweight operating system that they dubbed Unix (*Un*-ix, the single task version of *Mult*-ix, get it?).

Unfortunately for these engineers, they didn't have one large machine but a number of smaller machines, each from a different manufacturer. The standard development tricks of the day were all machine-dependent — they would have to rewrite the same program for each of the available machines. Instead, these engineers invented a small, powerful language named *C*.

C caught on like wildfire. Eventually, however, new programming techniques (most notably object-oriented programming) left the C programming language behind. Not to be outdone, the engineering community added equivalent new features to the C language. The result was called C++.

The C++ language consists of two basic elements:

- **Semantics:** This is a vocabulary of commands that humans can understand and that can be converted into machine language, fairly easily.

  and

- **Syntax:** This is a language structure (or *grammar*) that allows humans to combine these C++ commands into a program that actually does something (well, *maybe* does something).

Think of the semantics as the building blocks of your C++ program and the syntax as the correct way to put them together.

What's a program?

A C++ program is a text file containing a sequence of C++ commands put together according to the laws of C++ grammar. This text file is known as the *source file* (probably because it's the source of all frustration). A C++ source file carries the extension `.CPP` just as a Microsoft Word file ends in `.DOC` or an MS-DOS (remember that?) batch file ends in `.BAT`. The concept extension `.CPP` is just a convention.

The point of programming in C++ is to write a sequence of commands that can be converted into a machine-language program that actually *does* what we want done. The resulting *machine-executable* files carry the extension `.EXE`. The act of creating an executable program from a C++ program is called *compiling* or *building* (the subtle difference between the two is described in Chapter 22).

That sounds easy enough — so what's the big deal? Keep going.

How do I program?

To write a program, you need two specialized computer programs. One (an editor) is what you use to write your code as you build your `.CPP` source file. The other (a compiler) converts your source file into a machine-executable `.EXE` file that carries out your real-world commands (open spreadsheet, make rude noise, deflect incoming asteroid, whatever).

Nowadays, tool developers generally combine compiler and editor into a single package — a development *environment.* After you finish entering the commands that make up your program, you need only click a button to create the executable file.

The most popular of all C++ environments is a Microsoft product, Visual C++.NET (pronounced "Visual-C-plus-plus-DOT-net"). All programs in this book compile and execute with Visual C++.NET; however, many of you may not already own Visual C++.NET — and at $250 bucks a pop, street price, this may be a problem.

Fortunately, there *are* public-domain C++ environments. We use one of them in this book — the Dev-CPP environment. A recent version of Dev-CPP environment is included on CD-ROM enclosed at the back of this book (or you can download the *absolutely most recent* version off the Web at `www.bloodshed.net`).

You can download quite a range of public-domain programs from the Internet. Some of these programs, however, are not free — you're encouraged — or required — to pay some (usually small) fee. You don't *have* to pay to use Dev-C++, but you can contribute to the cause if you like. See the Web site for details.

I have tested the programs in this book with Dev-C++ version 4.9.8.0; they should work with other versions as well. You can check out my Web site at `www.stephendavis.com` for a list of any problems that may arise with future versions of Dev-C++ or Windows.

Dev-C++ is not some bug-ridden, limited edition C++ compiler from some fly-by-night group of developers. Dev-C++ is a full-fledged C++ environment. Dev-C++ supports the entire C++ language and executes all the programs in this book (and any other C++ book) just fine, thank you.

Dev-C++ does generate Windows-compatible 32-bit programs, but it does not easily support creating programs that have the classic Windows look. If you want to do that, you'll have to break open the wallet and go for a commercial package like Visual Studio.NET. Having said that, I strongly recommend that you work through the examples in this book first to learn C++ *before* you tackle Windows development. They are two separate things and (for the sake of sanity) should remain so in your mind.

Follow the steps in the next section to install Dev-C++ and build your first C++ program. This program's task is to convert a temperature value entered by the user from degrees Celsius to degrees Fahrenheit.

The programs in this book are compatible with Visual C++.NET and the C++ section of Visual Studio.NET (which are essentially the same thing). Use the documentation in the Visual C++ .NET for instructions on installing C++. True, the error messages generated by Visual C++.NET are different (and often just as difficult to decipher), but the territory will seem mysteriously familiar. Even though you're using a different songbook, you shouldn't have much trouble following the tune.

Installing Dev-C++

The CD-ROM that accompanies this book includes the most recent version of the Dev-C++ environment at the time of this writing.

The Dev-C++ environment comes in an easy-to-install, compressed executable file. This executable file is contained in the `DevCPP` directory on the accompanying CD-ROM. Here's the rundown on installing the environment:

1. **Navigate to and double-click the file `devcpp4980.exe`, or (in Windows) click Start➪Run.**

   - Double-clicking the file installs the environment automatically. (Note that 4.9.8.0, the version number, will be different on any newer version of Dev-C++ you downloaded off the Web.)
   - If you clicked Start➪Run, type ***x*:\devcpp\devcpp4980** in the Run window that appears, where *x* is the letter designation for your CD-ROM drive (normally **D** but perhaps **E** — if one doesn't work, try the other).

     Dev-C++ begins with a warning (shown in Figure 1-1) that you'd better uninstall any older version of Dev-C++ you may have hanging around, and then reboot and start over. (Starting an installation with a threat is an inauspicious way to begin a relationship, but everything gets better from here.)

Figure 1-1: You must uninstall earlier versions of Dev-C++ before you begin the installation process.

2. **If you don't have to uninstall an old version of Dev-C++, skip to Step 4; if you do have to uninstall, abort the current installation process by closing the Run window.**

   Don't get upset if you've never even heard of Dev-C++ and you still get the warning message. It's just a reminder.

3. **Okay, if you're on this step, you're uninstalling: Open the Dev-CPP folder and double-click the `Uninstall.exe` file there.**

   The uninstall program does its thing, preparing the way for the new installation; the End User Legal Agreement (commonly known as the *EULA*) appears.

4. **Read the EULA and then click OK if you can live with its provisions.**

   Nope, the package really won't install itself if you don't accept. Assuming you *do* click OK, Dev-C++ opens the window shown in Figure 1-2 and offers you some installation options. The defaults are innocuous, with two exceptions:

   - You must leave the *Mingw compiler system. . .* option enabled.
   - The *Associate C and C++ Files to Dev-C++* option means that double-clicking a .CPP file automatically opens Dev-C++ rather than some other program (such as Visual C++ .NET, for example). It is possible, but difficult, to undo this association.

Don't check this option if you also have Visual Studio.NET installed. Dev-C++ and Visual Studio.NET coexist peacefully on the same machine, but what Visual Studio has done, let no man cast assunder. You can still open your .CPP files with Dev-C++ by right-clicking on the file and selecting Open With. Personally, I prefer to use this option, even with Visual Studio.NET installed. It doesn't cause any problems, and Dev-C++ starts a *lot* faster than Visual Studio.

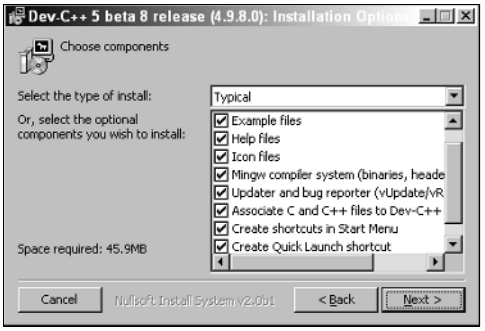

Figure 1-2: The default installation options should be acceptable to most users.

5. **Click the Next button.**

   The installation program asks where you want it to install Dev-C++, using a message like that shown in Figure 1-3.

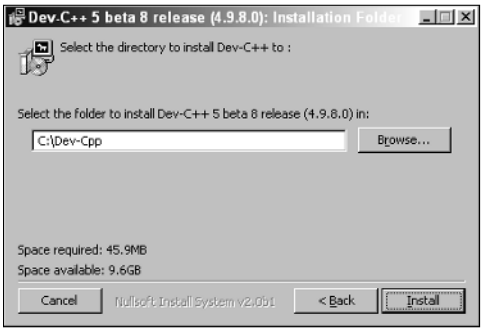

Figure 1-3: The default location for the Dev-C++ environment is provided.

6. **Accept the default directory,** `c:\Dev-CPP`**.**

   Don't install Dev-C++ in the directory `\Program Files` with all the other executables. That's because Dev-C++ doesn't do well with directories that contain spaces in their names. I haven't experimented much along these lines, but I believe you can use any other directory name without any special characters other than '_'. It's safer just to accept the default.

7. **Make sure you have enough room for the program, wherever you decide to put it.**

   The Dev-C++ environment uses only a paltry 45MB, but it's always good practice to check.

8. **Click Install.**

   At first, nothing seems to happen. Then Dev-C++ gets going, copying a whole passel of files to the `Dev-CPP` directory — putting absolutely nothing in the Windows home directory. Figure 1-4 displays the eventual result.

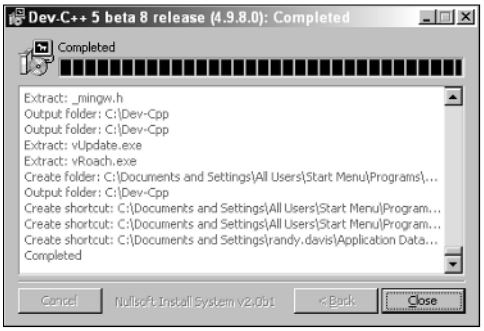

Figure 1-4: The Dev-C++ installation process unzips a large number of mostly small files.

While the installation is going on, Dev-C++ presents a window that asks whether you want to *install for all users* once it's done copying files onto your hard drive. That question boils down to this: If someone else logs on to your computer, do you want her or him to be able to execute Dev-C++? (The answer is "Yes" in my case.)

9. **Choose whether you want to install for all users, and then click the Close button to complete installation of the package.**

   Dev-C++ starts immediately, so you can set its options properly for your needs. (Yep, there's more work to do. But you knew that. Read on.)

Setting the options

As you probably know if you've spent more than a coffee break's worth of time installing software, setting options is a procedure unto itself. In this case, Dev-C++ has two options that must be set before you can use it. Set 'em as follows:

1. **Choose Tools⇨Compiler Options.**

   You can change these settings at any time, but now is as good as any.

2. **Choose the Settings tab.**

3. **Choose Code Generation from the menu on the left.**

   Make sure that the Enable Exception Handling is enabled, as shown in Figure 1-5. (If it isn't, click on the option box to display the two choices and select Yes.)

4. **Choose Linker and make sure the Generate Debugging Information option is enabled.**

   Figure 1-6 shows you what to look for.

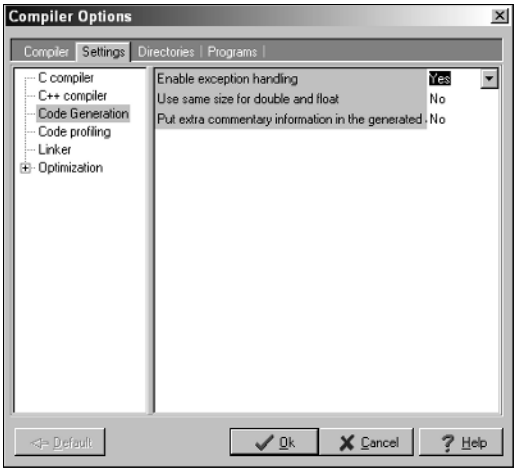

Figure 1-5: The Enable Exception Handling option must be enabled.

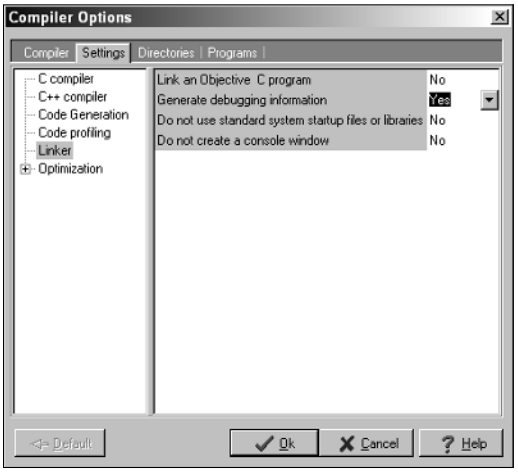

Figure 1-6: The Generate Debugging Information option must be enabled.

5. **Choose OK.**

   Installation is now complete! (Your options are saved automatically.)

Creating Your First C++ Program

In this section, you create your first C++ program. You first enter the C++ code into a file called `CONVERT.CPP`, and then convert the C++ code into an executable program.

Entering the C++ code

The first step to creating any C++ program is to enter C++ instructions using a text editor. The Dev-C++ user interface is built around a program editor specifically designed to create C++ programs.

1. **Click Start➪Programs➪Bloodshed Dev-C++➪Dev-C++ to start up the Dev-C++ tool.**

   The Dev-C++ interface looks fundamentally like that of any other Windows program — perhaps a little clunkier, but a Windows application nonetheless.

   This is a lot of clicking. My personal preference is to create a shortcut on the desktop. To create a shortcut, double-click My Computer. Now double-click the Local Disk (C:). Finally, double-click Dev-CPP — whew! Right-click the file `devcpp.exe` and choose Create Shortcut from the drop down menu. Drag the `Shortcut to devcpp.exe` file onto your desktop (or some other easily accessible place). From now on, you can just double-click the shortcut to start Dev-C++.

2. **Choose File➪New➪Source File.**

   Dev-C++ opens a blank window wherein you get to enter your new code. Don't worry if you find yourself wishing you knew what to enter right now — that's why you bought this book.

3. **Enter the following program exactly as written.**

   Don't worry too much about indentation or spacing — it isn't critical whether a given line is indented two or three spaces, or whether there are one or two spaces between two words. C++ is case sensitive, however, so you need to make sure everything is lowercase.

   You can cheat and copy the `Conversion.cpp` file contained on the enclosed CD-ROM in directory `\CPP_Programs\Chap01`.

```
//
//  Program to convert temperature from Celsius degree
//  units into Fahrenheit degree units:
//  Fahrenheit = Celsius  * (212 - 32)/100 + 32
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
```

```
  // enter the temperature in Celsius
  int celsius;
  cout << "Enter the temperature in Celsius:";
  cin >> celsius;

  // calculate conversion factor for Celsius
  // to Fahrenheit
  int factor;
  factor = 212 - 32;

  // use conversion factor to convert Celsius
  // into Fahrenheit values
  int fahrenheit;
  fahrenheit = factor * celsius/100 + 32;

  // output the results (followed by a NewLine)
  cout << "Fahrenheit value is:";
  cout << fahrenheit << endl;

  // wait until user is ready before terminating program
  // to allow the user to see the program results
  system("PAUSE");
  return 0;
}
```

4. **Choose Save As under the File menu. Then type in the program name and press Enter.**

   I know that it may not seem all that exciting, but you've just created your first C++ program!

   For purposes of this book, I created a folder `CPP_Programs`. Within this, I created `Chap01`. Finally, I saved the program with the name `Conversion.cpp`. Note that Dev-C++ won't work properly with directory names that contain spaces. (It doesn't have a problem with names longer than eight characters in length — thank goodness!)

Building your program

After you've saved your `Conversion.cpp` C++ source file to disk, it's time to generate the executable machine instructions.

To build your `Conversion.cpp` program, you choose Execute➪Compile from the menu or press F9 — or you can even click that cute little icon with four colored squares on the menu bar (use the Tool Tips to see which one I'm talking about). In response, Dev-C++ opens a compiling window. Nothing will happen at first (sshh . . . it's thinking). After a second or two, Dev-C++ seems to take off, compiling your program with gusto. If all goes well, a window like that shown in Figure 1-7 appears.

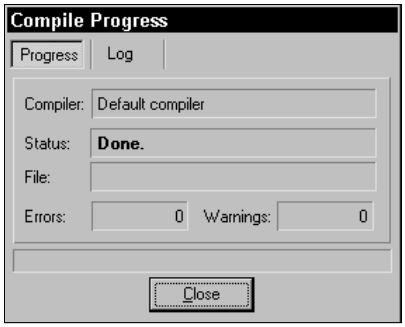

Figure 1-7: The user is rewarded with a simple Done message if his program is error free.

Dev-C++ generates a message if it finds any type of error in your C++ program — and coding errors are about as common as snow in Alaska. You'll undoubtedly encounter numerous warnings and error messages, probably even when entering the simple `Conversion.cpp`. To demonstrate the error-reporting process, let's change Line 17 from `cin >> celsius;` to `cin >>> celsius;`.

This seems an innocent enough offense — forgivable to you and me perhaps, but not to C++. Dev-C++ opens a Compiler tab, as shown in Figure 1-8. The message `parse error before '>` is perhaps a little terse, but descriptive. To get rid of the message, remove the extra `>` and recompile.

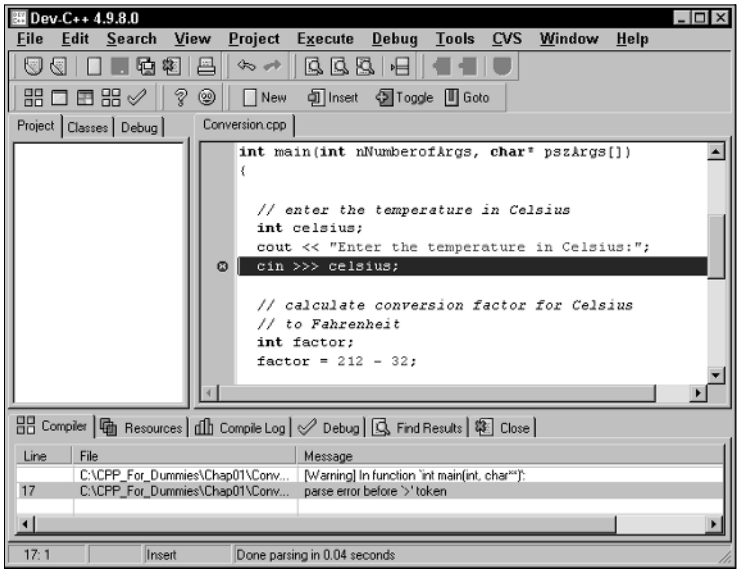

Figure 1-8: Bad little programs generate error messages in the Compiler window.

Why is C++ so picky?

In the example given here, C++ could tell right away — and without a doubt — that I had screwed up. However, if C++ can figure out what I did wrong, why doesn't it just fix the problem and go on?

The answer is simple but profound. C++ *thinks* that I mistyped the >> symbol, but it may be mistaken. What could have been a mistyped command may actually be some other, completely unrelated error. Had the compiler simply corrected the problem, C++ would have masked the real problem.

Finding an error buried in a program that builds without complaining is difficult and time-consuming. It's far better to let the compiler find the error for you if at all possible. Generating a compiler error is a waste of the computer's time — forcing me to find a mistake that C++ could have caught is a waste of *my* time. Guess which one I vote for?

The term *parse* means to convert the C++ commands into something that the machine-code-generating part of the process can work with.

There was once a language that tried to fix simple mistakes like this for you. From my personal experience, I can tell you it was a waste of time — because (except for very simple cases) the compiler was almost always wrong. At least it warned me of the problem so I could fix it myself.

Executing Your Program

It's now time to execute your new creation . . . that is, to run your program. You will run the `CONVERT.EXE` program file and give it input to see how well it works.

To execute the Conversion program, click Execute➪Run or press Ctrl+F10. (I have no idea how they selected function keys. I would think that an action as common as executing a program would warrant its own function key — something without a Control or Shift key to hold down — but maybe that's just me.)

A window opens immediately, requesting a temperature in Celsius. Enter a known temperature, such as 100 degrees. After you press Enter, the program returns with the equivalent temperature of 212 degrees Fahrenheit as follows:

```
Enter the temperature in Celsius:100
Fahrenheit value is:212
Press any key to continue . . .
```

The message `Press any key` gives you the opportunity to read what you've entered before it goes away. Press Enter, and the window (along with its contents) disappears. Congratulations! You just entered, built, and executed your first C++ program.

Dev-C++ is not Windows

Notice that Dev-C++ is not truly intended for developing Windows programs. In theory, you can write a Windows application by using Dev-C++, but it isn't easy. (That's *so* much easier in Visual Studio.NET.)

Windows programs show the user a very visually oriented output, all nicely arranged in onscreen windows. Convesion.exe is a 32-bit program that executes *under* Windows, but it's not a "Windows" program in the visual sense.

If you don't know what *32-bit program* means, don't worry about it. As I said earlier, this book isn't about writing Windows programs. The C++ programs you write in this book have a *command line interface* executing within an MS-DOS box.

Budding Windows programmers shouldn't despair — you didn't waste your money. Learning C++ is a prerequisite to writing Windows programs. I think that they should be mastered separately: C++ first, Windows second.

Dev-C++ help

Dev-C++ provides a Help menu item. Choose `Help` followed by `Help on Dev C++` to open up a typical Windows help box. Help is provided on various aspects of the Dev-C++ development package but not much else. Noticeably lacking is help on the C++ language itself. Click a topic of interest to display help.

Reviewing the Annotated Program

Entering data in someone else's program is about as exciting as watching someone else drive a car. You really need to get behind the wheel itself. Programs are a bit like cars as well. All cars are basically the same with small differences and additions — OK, French cars are a lot different than other cars, but the point is still valid. Cars follow the same basic pattern — steering wheel in front of you, seat below you, roof above you and stuff like that.

Similarly, all C++ programs follow a common pattern. This pattern is already present in this very first program. We can review the Conversion program by looking for the elements that are common to all programs.

Examining the framework for all C++ programs

Every C++ program you write for this book uses the same basic framework, which looks a lot like this:

```
//
//  Template - provides a template to be used as the starting
//             point
//
// the following include files define the majority of
// functions that any given program will need
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // your C++ code starts here

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

Without going into all the boring details, execution begins with the code contained in the open and closed braces immediately following the line beginning `main()`.

I have copied this code into a file called `Template.cpp` located in the main `CPP_Programs` folder on the enclosed CD-ROM.

Clarifying source code with comments

The first few lines in `Conversion.cpp` appear to be freeform text. Either this code was meant for human eyes or C++ is a lot smarter than I give it credit for. These first six lines are known as comments. *Comments* are the programmer's

explanation of what he or she is doing or thinking when writing a particular code segment. The compiler ignores comments. Programmers (*good* programmers, anyway) don't.

A C++ comment begins with a double slash (//) and ends with a newline. You can put any character you want in a comment. A comment may be as long as you want, but it's customary to keep comment lines to no more than 80 characters across. Back in the old days — "old" is relative here — screens were limited to 80 characters in width. Some printers still default to 80 characters across when printing text. These days, keeping a single line to under 80 characters is just a good practical idea (easier to read, less likely to cause eyestrain, the usual).

A newline was known as a *carriage return* back in the days of typewriters — when the act of entering characters into a machine was called *typing* and not *keyboarding*. A *newline* is the character that terminates a command line.

C++ allows a second form of comment in which everything appearing after a /* and before a */ is ignored; however, this form of comment isn't normally used in C++ anymore. (Later in this book, I describe the one case in which this type of comment is applied.)

It may seem odd to have a command in C++ (or any other programming language) that's specifically ignored by the computer. However, all computer languages have some version of the comment. It's critical that the programmer explain what was going through her mind when she wrote the code. A programmer's thoughts may not be obvious to the next colleague who picks up her program and tries to use it or modify it. In fact, the programmer herself may forget what her program meant if she looks at it months after writing the original code and has left no clue.

Basing programs on C++ statements

All C++ programs are based on what are known as C++ *statements*. This section reviews the statements that make up the program framework used by the `Conversion.cpp` program.

A *statement* is a single set of commands. All statements other than comments end with a semicolon. (There's a reason that comments don't end with a semicolon, but it's obscure. To my mind, comments *should* end in semicolons as well, for consistency's sake. Why nobody asked me about that remains a mystery.)

Program execution begins with the first C++ statement after the open brace and continues through the listing, one statement at a time.

As you look through the program, you can see that spaces, tabs, and newlines appear throughout the program. In fact, I place a newline after every statement in this program. These characters are collectively known as *white space* because you can't see them on the monitor.

You may add white space anywhere you like in your program to enhance readability — except in the middle of a word:

```
See wha

t I mean?
```

Although C++ may ignore white space, it doesn't ignore case. In fact, it's case sensitive to the point of obsession. The variable `fullspeed` and the variable `FullSpeed` have nothing to do with each other. While the command `int` may be understood completely, C++ has no idea what `INT` means.

Writing declarations

The line `int nCelsius;` is a declaration statement. A *declaration* is a statement that defines a variable. A *variable* is a "holding tank" for a value of some type. A variable contains a *value,* such as a number or a character.

The term variable stems from algebra formulae of the following type:

```
x = 10
y = 3 * x
```

In the second expression, `y` is set equal to 3 times x, but what is `x`? The variable `x` acts as a holding tank for a value. In this case, the value of `x` is 10, but we could have just as well set the value of `x` to 20 or 30 or –1. The second formula makes sense no matter what the value of `x`.

In algebra, you're allowed to begin with a statement, such as `x = 10`. In C++, the programmer must first define the variable `x` before she can use it.

In C++, a variable has a type and a name. The variable defined on Line 11 is called `celsius` and declared to hold an integer. (Why they couldn't have just said *integer* instead of *int,* I'll never know. It's just one of those things you learn to live with.)

The name of a variable has no particular significance to C++. A variable must begin with the letters A through Z or a through z. All subsequent characters must be a letter, a digit 0 through 9 or an underscore (_). Variable names can be as long as you want to make them.

It's convention that variable names begin with a lowercase letter. Each new word *within* a variable begins with a capital letter, as in `myVariable`.

Try to make variable names short but descriptive. Avoid names such as `x` because x has no particular meaning. A variable name such as `lengthOfLine Segment` is much more descriptive.

Generating output

The lines beginning with `cout` and `cin` are known as input/output statements, often contracted to I/O statements. (Like all engineers, programmers love contractions and acronyms.)

The first I/O statement says output the phrase *Enter the temperature in Celsius* to *cout* (pronounced "see-out"). `cout` is the name of the standard C++ output device. In this case, the standard C++ output device is your monitor.

The next line is exactly the opposite. It says, in effect, *Extract a value from the C++ input device and store it in the integer variable* `celsius`. The C++ input device is normally the keyboard. What we've got here is the C++ analog to the algebra formula *x = 10* just mentioned. For the remainder of the program, the value of `celsius` is whatever the user enters there.

Calculating Expressions

All but the most basic programs perform calculations of one type or another. In C++, an *expression* is a statement that performs a calculation. Said another way, an expression is a statement that *has a value.* An *operator* is a command that generates a value.

For example, in the Conversion example program — specifically in the two lines marked as a `calculation expression` — the program declares a variable *factor* and then assigns it the value resulting from a calculation. This particular command calculates the difference of 212 and 32; the operator is the minus sign (`-`), and the expression is `212-32`.

Storing the results of expression

The spoken language can be very ambiguous. The term *equals* is one of those ambiguities. The word *equals* can mean that two things have the same value as in "5 cents equals a nickel." Equals can also imply assignment, as in math when you say that "y equals 3 times *x*."

To avoid ambiguity, C++ programmers call the *assignment operator,* which says (in effect), *Store the results of the expression to the right of the* `equal` *sign in the variable to the left.* Programmers say that "`factor` is *assigned the value* 212 minus 32."

Never say "`factor` is *equal to* 212 minus 32." You'll hear this from some lazy types, but you and I know better.

Examining the remainder of Conversion.cpp

The second expression in `Conversion.cpp` presents a slightly more complicated expression than the first. This expression uses the same mathematical symbols: * for multiplication, / for division and, + for addition. In this case, however, the calculation is performed on variables and not simply on constants.

The value contained in the variable called `factor` (calculated immediately prior, by the way) is multiplied by the value contained in `celsius` (which was input from the keyboard). The result is divided by 100 and summed with 32. The result of the total expression is assigned to the integer variable `fahrenheit`.

The final two commands output the string `Fahrenheit value is:` to the display, followed by the value of `fahrenheit` — and all so fast that the user scarcely knows it's going on.