

# What Is a Program?

This chapter introduces you to fundamental programming concepts. The task of programming computers has been described as rewarding, challenging, easy, difficult, fast, and slow. Actually, it is a combination of all these descriptions. Writing complex programs to solve advanced problems can be frustrating and time-consuming, but you can have fun along the way, especially with the rich assortment of features that C++ has to offer.

This chapter also describes the concept of programming, from a program's inception to its execution on your computer. The most difficult part of programming is breaking the problem into logical steps that the computer can execute. Before you finish this chapter, you will type and execute your first C++ program.

This chapter introduces you to

- ♦ The concept of programming
- ♦ The program's output
- ♦ Program design
- ♦ Using an editor
- ♦ Using a compiler

- ♦ Typing and running a C++ program
- ♦ Handling errors

After you complete this chapter, you should be ready to learn the C++ programming language elements in greater detail.

## Computer Programs

Before you can make C++ work for you, you must write a C++ program. You have seen the word *program* used several times in this book. The following note defines a program more formally.



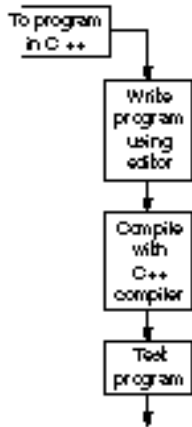
**NOTE:** A *program* is a list of instructions that tells the computer to do things.

Keep in mind that computers are only machines. They're not smart; in fact, they're quite the opposite! They don't do anything until they are given detailed instructions. A word processor, for example, is a program somebody wrote—in a language such as C++—that tells your computer exactly how to behave when you type words into it.

You are familiar with the concept of programming if you have ever followed a recipe, which is a “program,” or a list of instructions, telling you how to prepare a certain dish. A good recipe lists these instructions in their proper order and with enough description so you can carry out the directions successfully, without assuming anything.

If you want your computer to help with your budget, keep track of names and addresses, or compute your gas mileage, it needs a program to tell it how to do those things. You can supply that program in two ways: buy a program somebody else wrote, or write the program yourself.

Writing the program yourself has a big advantage for many applications: The program does exactly what *you* want it to do. If you buy one that is already written, you have to adapt your needs to those of the author of the program. This is where C++ comes into



play. With the C++ programming language (and a little studying), you can make your computer carry out your own tasks precisely.

To give C++ programming instructions to your computer, you need an *editor* and a *C++ compiler*. An editor is similar to a word processor; it is a program that enables you to type a C++ program into memory, make changes (such as moving, copying, inserting, and deleting text), and save the program more permanently in a disk file. After you use the editor to type the program, you must compile it before you can run it.

The C++ programming language is called a *compiled* language. You cannot write a C++ program and run it on your computer unless you have a C++ compiler. This compiler takes your C++ language instructions and translates them into a form that your computer can read. A C++ compiler is the tool your computer uses to understand the C++ language instructions in your programs. Many compilers come with their own built-in editor. If yours does, you probably feel that your C++ programming is more integrated.

To some beginning programmers, the process of compiling a program before running it might seem like an added and meaningless step. If you know the BASIC programming language, you might not have heard of a compiler or understand the need for one. That's because BASIC (also APL and some versions of other computer languages) is not a compiled language, but an *interpreted* language. Instead of translating the entire program into machine-readable form (as a compiler does in one step), an interpreter translates each program instruction—then executes it—before translating the next one. The difference between the two is subtle, but the bottom line is not: Compilers produce *much* more efficient and faster-running programs than interpreters do. This seemingly extra step of compiling is worth the effort (and with today's compilers, there is not much extra effort needed).

Because computers are machines that do not think, the instructions you write in C++ must be detailed. You cannot assume your computer understands what to do if some instruction is not in your program, or if you write an instruction that does not conform to C++ language requirements.

After you write and compile a C++ program, you have to *run*, or *execute*, it. Otherwise, your computer would not know that you

want it to follow the instructions in the program. Just as a cook must follow a recipe's instructions before making the dish, so too your computer must execute a program's instructions before it can accomplish what you want it to do. When you run a program, you are telling the computer to carry out your instructions.

### **The Program and Its Output**

While you are programming, remember the difference between a program and its output. Your program contains only the C++ instructions that you write, but the computer follows your instructions only *after* you run the program.

Throughout this book, you often see a *program listing* (that is, the C++ instructions in the program) followed by the results that occur when you run the program. The results are the output of the program, and they go to an output device such as the screen, the printer, or a disk file.

## **Program Design**

Design your programs before you type them.

You must plan your programs before typing them into your C++ editor. When builders construct houses, for example, they don't immediately grab their lumber and tools and start building! They first find out what the owner of the house wants, then they draw up the plans, order the materials, gather the workers, and finally start building the house.

The hardest part of writing a program is breaking it into logical steps that the computer can follow. Learning the C++ language is a requirement, but it is not the only thing to consider. There is a method of writing programs, a formal procedure you should learn, that makes your programming job easier. To write a program you should:

1. Define the problem to be solved with the computer.
2. Design the program's output (what the user should see).

3. Break the problem into logical steps to achieve this output.
4. Write the program (using the editor).
5. Compile the program.
6. Test the program to assure it performs as you expect.

As you can see from this procedure, the typing of your program occurs toward the end of your programming. This is important, because you first have to plan *how* to tell the computer how to perform each task.

Your computer can perform instructions only step-by-step. You must assume that your computer has no previous knowledge of the problem, so it is up to you to provide that knowledge, which, after all, is what a good recipe does. It would be a useless recipe for a cake if all it said was: “Bake the cake.” Why? Because this *assumes* too much on the part of the baker. Even if you write the recipe in step-by-step fashion, proper care must be taken (through planning) to be sure the steps are in sequence. Wouldn’t it be foolish also to instruct a baker to put the ingredients into the oven before stirring them?

This book adheres to the preceding programming procedure throughout the book, as each program appears. Before you see the actual program, the thought process required to write the program appears. The goals of the program are presented first, then these goals are broken into logical steps, and finally the program is written.

Designing the program in advance guarantees that the entire program structure is more accurate and keeps you from having to make changes later. A builder, for example, knows that a room is much harder to add after the house is built. If you do not properly plan every step, it is going to take you longer to create the final, working program. It is always more difficult to make major changes after you write your program.

Planning and developing according to these six steps becomes much more important as you write longer and more complicated programs. Throughout this book, you learn helpful tips for program design. Now it’s time to launch into C++, so you can experience the satisfaction of typing your own program and seeing it run.

## Using a Program Editor

The instructions in your C++ program are called the *source code*. You type source code into your computer's memory by using your program editor. After you type your C++ source code (your program), you should save it to a disk file before compiling and running the program. Most C++ compilers expect C++ source programs to be stored in files with names ending in .CPP. For example, the following are valid filenames for most C++ compilers:

MYPROG.CPP

SALESACT.CPP

EMPLYEE.CPP

ACCREC.CPP

Many C++ compilers include a built-in editor. Two of the most popular C++ compilers (both conform to the AT&T C++ 2.1 standard and include their own extended language elements) are Borland's C++ and Microsoft's C/C++ 7.0 compilers. These two programs run in fully integrated environments that relieve the programmer from having to worry about finding a separate program editor or learning many compiler-specific commands.

Figure 2.1 shows a Borland C++ screen. Across the top of the screen (as with Microsoft C/C++ 7.0) is a menu that offers pull-down editing, compiling, and running options. The middle of the screen contains the body of the program editor, and this is the area where the program goes. From this screen, you type, edit, compile, and run your C++ source programs. Without an *integrated environment*, you would have to start an editor, type your program, save the program to disk, exit the editor, run the compiler, and only *then* run the compiled program from the operating system. With Borland's C++ and Microsoft C/C++ 7.0, you simply type the program into the editor, then—in one step—you select the proper menu option that compiles and runs the program.

```

File Edit Search Run Compile Debug Project Options Window Help
\\CPP\\C12CNT1.CPP 1
// Filename: C12CNT1.CPP
// Program to print a message 10 times
#include <iostream.h>
main()
{
    int ctr = 0; // Holds the number of times printed

    do
    { cout << "Computers are fun!\n";
      ctr++; // Add one to the count,
             // after each printf()

    } while (ctr < 10); // Print again if fewer
                       // than 10 times

    return 0;
}
1:1

Message 2-[1]
Compiling ...\\CPP\\C12CNT1.CPP:
*Linking C12CNT1.EXE:

F1 Help F10 Menu

```

Figure 2.1. Borland Turbo C++'s integrated environment.

If you do not own an integrated environment such as Borland C++ or Microsoft C/C++, you have to find a program editor. Word processors can act as editors, but you have to learn how to save and load files in a true ASCII text format. It is often easier to use an editor than it is to make a word processor work like one.

On PCs, DOS Version 5 comes with a nice, full-screen editor called EDIT. It offers menu-driven commands and full cursor-control capabilities. EDIT is a simple program to use, and is a good beginner's program editor. Refer to your DOS manual or a good book on DOS, such as *MS-DOS 5 QuickStart* (Que), for more information on this program editor.

Another editor, called EDLIN, is available for earlier versions of DOS. EDLIN is a line editor that does not allow full-screen cursor control, and it requires you to learn some cryptic commands. The advantage to learning EDLIN is that it is always included with all PCs that use a release of DOS prior to Version 5.

If you use a computer other than a PC, such as a UNIX-based minicomputer or a mainframe, you have to determine which editors are available. Most UNIX systems include the `vi` editor. If you program on a UNIX operating system, it would be worth your time to learn `vi`. It is to UNIX what EDLIN is to PC operating systems, and is available on almost every UNIX computer in the world.

Mainframe users have other editors available, such as the ISPF editor. You might have to check with your systems department to find an editor accessible from your account.



**NOTE:** Because this book teaches the generic AT&T C++ standard programming language, no attempt is made to tie in editor or compiler commands—there are too many on the market to cover them all in one book. As long as you write programs specific to the AT&T C++, the tools you use to edit, compile, and run those programs are secondary; your goal of good programming is the result of whatever applications you produce.

## Using a C++ Compiler

After you type and edit your C++ program's source code, you have to compile the program. The process you use to compile your program depends on the version of C++ and the computer you are using. Borland C++ and Microsoft C/C++ users need only press Alt-R to compile and run their programs. When you compile programs on most PCs, your compiler eventually produces an *executable* file with a name beginning with the same name as the source code, but ends with an .EXE file extension. For example, if your source program is named GRADEAVG.CPP, the PC would produce a compiled file called GRADEAVG.EXE, which you could execute at the DOS prompt by typing the name `gradeavg`.





**NOTE:** Each program in this book contains a comment that specifies a recommended filename for the source program. You do not have to follow the file-naming conventions used in this book; the filenames are only suggestions. If you use a mainframe, you have to follow the dataset-naming conventions set up by your system administrator. Each program name in the sample disk (see the order form at the back of the book) matches the filenames of the program listings.

UNIX users might have to use the `cfront` compiler. Most `cfront` compilers actually convert C++ code into regular C code. The C code is then compiled by the system's C compiler. This produces an executable file whose name (by default) is `A.OUT`. You can then run the `A.OUT` file from the UNIX prompt. Mainframe users generally have company-standard procedures for compiling C++ source programs and storing their results in a test account.

Unlike many other programming languages, your C++ program must be routed through a *preprocessor* before it is compiled. The preprocessor reads preprocessor directives that you enter in the program to control the program's compilation. Your C++ compiler automatically performs the preprocessor step, so it requires no additional effort or commands to learn on your part.

You might have to refer to your compiler's reference manuals or to your company's system personnel to learn how to compile programs for your programming environment. Again, learning the programming environment is not as critical as learning the C++ language. The compiler is just a way to transform your program from a source code file to an executable file.

Your program must go through one additional stage after compiling and before running. It is called the *linking*, or the *link editing* stage. When your program is linked, a program called the linker supplies needed runtime information to the compiled program. You can also combine several compiled programs into one executable program by linking them. Most of the time, however,

your compiler initiates the link editing stage (this is especially true with integrated compilers such as Borland C++ and Microsoft C/C++) and you do not have to worry about the process.

Figure 2.2 shows the steps that your C++ compiler and link editor perform to produce an executable program.

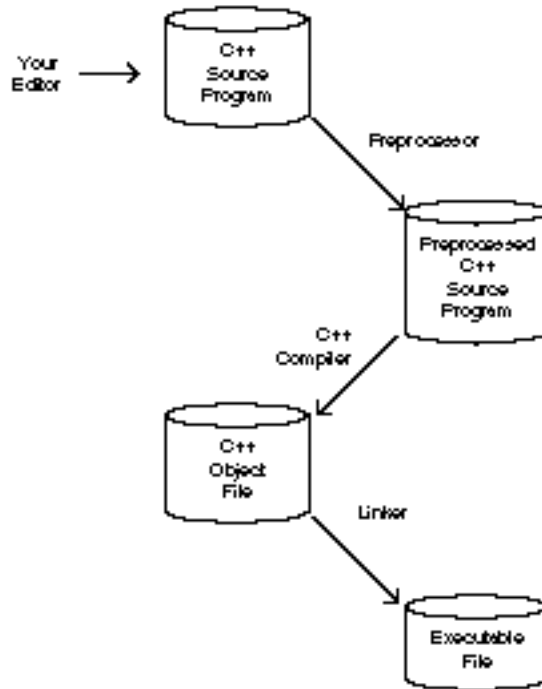


Figure 2.2. Compiling C++ source code into an executable program.

## Running a Sample Program

Before delving into the specifics of the C++ language, you should take a few moments to become familiar with your editor and C++ compiler. Starting with the next chapter, “Your First C++ Program,” you should put all your concentration into the C++ programming language and not worry about using a specific editor or compiling environment.

Therefore, start your editor of choice and type Listing 2.1, which follows, into your computer. Be as accurate as possible—a single typing mistake could cause the C++ compiler to generate a series of errors. You do not have to understand the program's content at this point; the goal is to give you practice in using your editor and compiler.

### **Listing 2.1. Practicing with the editor.**



*Comment the program with the program name.*  
*Include the header file `iostream.h` so the output properly works.*  
*Start of the `main()` function.*  
*Define the `BELL` constant, which is the computer's beep.*  
*Initialize the integer variable `ctr` to 0.*  
*Define the character array `fname` to hold 20 elements.*  
*Print to the screen What is your first name?.*  
*Accept a string from the keyboard.*  
*Process a loop while the variable `ctr` is less than five.*  
*Print the string accepted from the keyboard.*  
*Increment the variable `ctr` by 1.*  
*Print to the screen the character code that sounds the beep.*  
*Return to the operating system.*

```
// Filename: C2FIRST.CPP
// Requests a name, prints the name five times, and rings a bell.

#include <iostream.h>

main()
{
    const char BELL='\a';           // Constant that rings the bell
    int ctr=0;                      // Integer variable to count through loop
    char fname[20];                 // Define character array to hold name

    cout << "What is your first name? ";    // Prompt the user
    cin >> fname;                          // Get the name from the keyboard
    while (ctr < 5)                       // Loop to print the name
```

```
{                                     // exactly five times.
    cout << fname << "\n";
    ctr++;
}
cout << BELL;                        // Ring the terminal's bell
return 0;
}
```

---

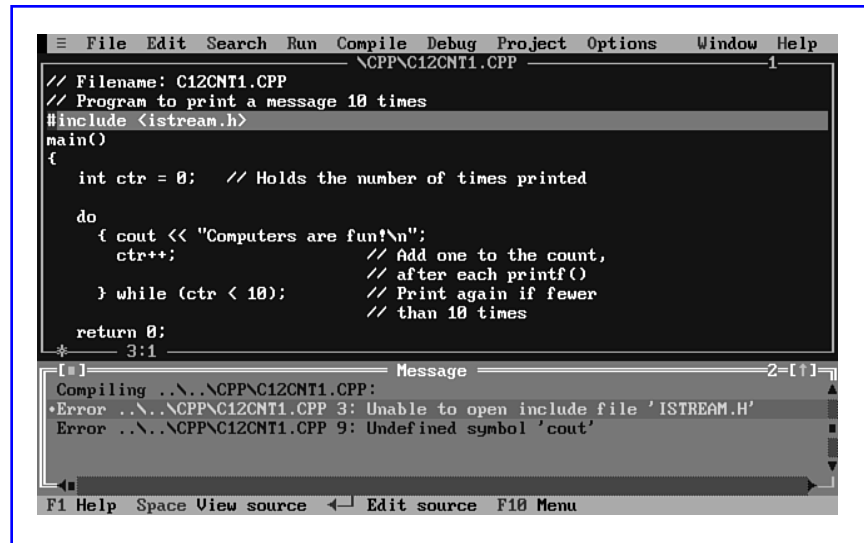
Be as accurate as possible. In most programming languages—and especially in C++—the characters you type into a program must be very accurate. In this sample C++ program, for instance, you see parentheses, `()`, brackets, `[]`, and braces, `{}`, but you cannot use them interchangeably.

The *comments* (words following the two slashes, `//`) to the right of some lines do not have to end in the same place that you see in the listing. They can be as long or short as you need them to be. However, you should familiarize yourself with your editor and learn to space characters accurately so you can type this program exactly as shown.

Compile the program and execute it. Granted, the first time you do this you might have to check your reference manuals or contact someone who already knows your C++ compiler. Do not worry about damaging your computer: Nothing you do from the keyboard can harm the physical computer. The worst thing you can do at this point is erase portions of your compiler software or change the compiler's options—all of which can be easily corrected by reloading the compiler from its original source. (It is only remotely likely that you would do anything like this, even if you are a beginner.)

## Handling Errors

Because you are typing instructions for a machine, you must be very accurate. If you misspell a word, leave out a quotation mark, or make another mistake, your C++ compiler informs you with an error message. In Borland C++ and Microsoft C/C++, the error probably appears in a separate window, as shown in Figure 2.3. The most common error is a *syntax error*, and this usually implies a misspelled word.

A screenshot of a C++ IDE window titled 'C12CNT1.CPP'. The menu bar includes File, Edit, Search, Run, Compile, Debug, Project, Options, Window, and Help. The code editor shows a program intended to print 'Computers are fun!\n' 10 times using a loop. The code is as follows:

```
// Filename: C12CNT1.CPP
// Program to print a message 10 times
#include <iostream.h>
main()
{
    int ctr = 0; // Holds the number of times printed

    do
    { cout << "Computers are fun!\n";
      ctr++; // Add one to the count,
             // after each printf()

    } while (ctr < 10); // Print again if fewer
                       // than 10 times

    return 0;
}
```

The status bar at the bottom shows 'F1 Help Space View source Edit source F10 Menu'. Below the code editor, a 'Message' window displays the following error messages:

```
Compiling ..\..\CPP\C12CNT1.CPP:
Error ..\..\CPP\C12CNT1.CPP 3: Unable to open include file 'ISTREAM.H'
Error ..\..\CPP\C12CNT1.CPP 9: Undefined symbol 'cout'
```

Figure 2.3. The compiler reporting a program error.

When you get an error message (or more than one), you must return to the program editor and correct the error. If you don't understand the error, you might have to check your reference manual or scour your program's source code until you find the offending code line.

### Getting the Bugs Out

One of the first computers, owned by the military, refused to print some important data one day. After its programmers tried for many hours to find the problem in the program, a programmer by the name of Grace Hopper decided to check the printer.

She found a small moth lodged between two important wires. When she removed the moth, the printer started working perfectly (although the moth did not have the same luck).

Grace Hopper was an admiral from the Navy and, although she was responsible for developing many important computer concepts (she was the author of the original COBOL language), she might be best known for discovering the first computer bug.

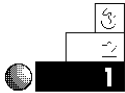
Ever since Admiral Hopper discovered that moth, errors in computer programs have been known as *computer bugs*. When you test your programs, you might have to *debug* them—get the bugs (errors) out by correcting your typing errors or changing the logic so your program does exactly what you want it to do.

After you have typed your program correctly using the editor (and you get no compile errors), the program should run properly by asking for your first name, then printing it on-screen five times. After it prints your name for the fifth time, you hear the computer's bell ring.

This example helps to illustrate the difference between a program and its output. You must type the program (or load one from disk), then run the program to see its output.

## Review Questions

The answers to the review questions are in Appendix B, “Answers to Review Questions.”



1. What is a program?
2. What are the two ways to obtain a program that does what you want?
3. True or false: Computers can think.
4. What is the difference between a program and its output?
5. What do you use for typing C++ programs into the computer?



6. What filename extension do all C++ programs have?
7. Why is typing the program one of the *last* steps in the programming process?
8. What does the term *debug* mean?
9. Why is it important to write programs that are compatible with the AT&T C++?
10. True or false: You must link a program before compiling it.

## Summary

After reading this chapter, you should understand the steps necessary to write a C++ program. You know that planning makes writing the program much easier, and that your program's instructions produce the output only after you run the program.

You also learned how to use your program editor and compiler. Some program editors are as powerful as word processors. Now that you know how to run C++ programs, it is time to start learning the C++ programming language.

