Program Development

This chapter describes the essentials of program development. It explains the steps in the process of creating a program and provides a general description of the available computer programming languages. It gives an example of a simple C++ program and walks you through the process of writing source code to make an executable program. The chapter also explains the use of the shell redirection operator to redirect the output and error messages of programs.

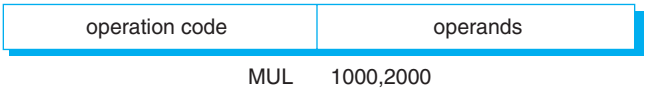
In This Chapter

11.1 PROGRAM DEVELOPMENT

Chapter 1 discussed software and computer programming in general. You learned the important role of software in making computers do all these wonderful things (open to discussion!). You also learned that there are two categories of software: application software and system software.

A *program* consists of a set of instructions that guide the computer in performing its basic arithmetic and logical operations. Each instruction tells the machine to perform one of its basic functions, and usually consists of the operation code and one or more operands. The *operation code* specifies the function to be performed, and the *operands* specify the location or data elements to be manipulated. Figure 11.1 shows a typical instruction.

Figure 11.1
The Format of a Simple Instruction



A computer is controlled by programs that are stored in the computer's memory. Memory storage is capable of storing only zeros and ones, so programs in memory must be in *binary form.* That means programs must be written in zeros and ones or converted to zeros and ones. Do programmers really write programs in zeros and ones? Fortunately, they do not have to anymore. Early programmers did not have a choice; they had to code programs in zeros and ones, for which they deserve a lot of respect. Programmers write programs to create both categories of software. To write a program you need a programming language, and there are quite a few you can choose from.

11.2 PROGRAMMING LANGUAGES

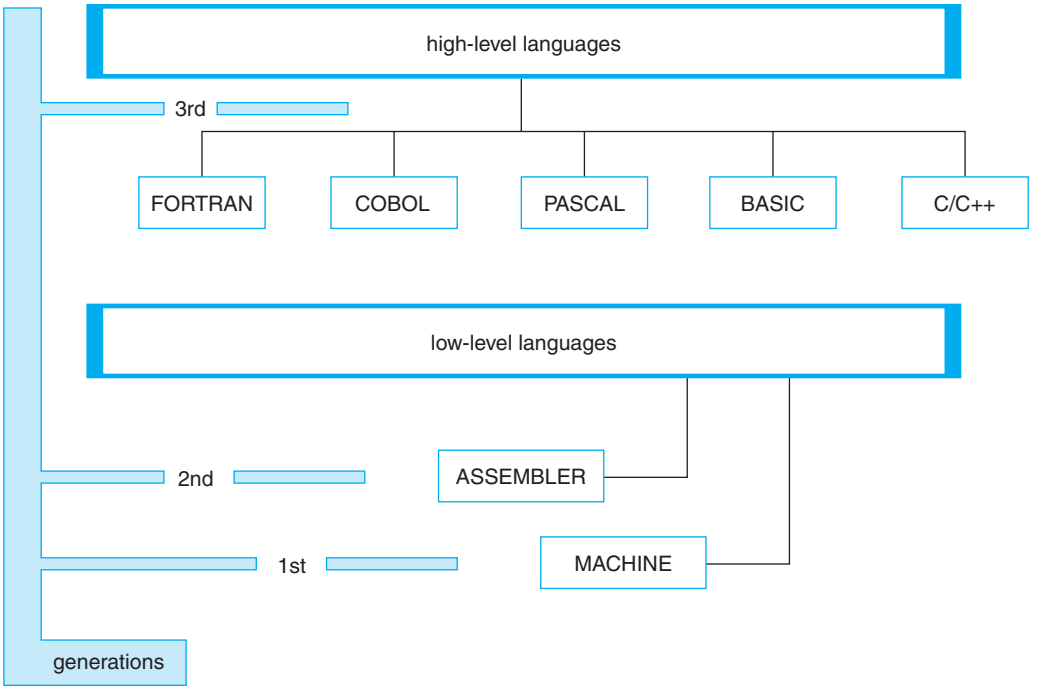
Programming is the process of writing instructions (a program) for a computer to solve a problem. These instructions must be written in a computer programming language. A program can be anything from a simple list of instructions that adds a series of numbers together, to a large and complex structure with many sections that calculates the payroll of a big corporation.

Like computer hardware, programming languages have evolved in generations. Each new generation of languages improved on the previous ones and incorporated more capabilities for programmers.

Figure 11.2 shows the hierarchy and generations of the programming languages. Let's briefly explore the different levels of this hierarchy.

See page x for an explanation of the icons used to highlight information in this chapter.

Figure 11.2
The Hierarchy of the Programming Languages



11.2.1 Low-Level Languages

Machine Language In machine language, instructions are coded as a series of zeros and ones. It is cumbersome and difficult to write programs in machine language. Machine language programs are written at the most basic level of computer operation; it is the only language computers can understand and execute. Programs written in other programming languages must be translated into the machine language of the computer on which the program is to be executed. (Programs called *compilers* do this translation and are discussed in the next section.)

Assembler Language Like machine language, assembler language is unique to a particular computer, but the instructions are represented differently. Instead of a series of zeros and ones, assembler language uses some recognized symbols, called *mnemonics* (memory aids), to represent instructions. For example, the mnemonic MUL is used to represent a multiply instruction. Because computers understand only zeros and ones, you must change a program in assembler language to machine language format for execution. This translation is done by invoking a program called an *assembler.* The assembler program translates the mnemonics in your program back to zeros and ones.

11.2.2 High-Level Languages

No matter what computer and which high-level language you use to write your program, the program must be translated into machine language before it can be executed. This conversion of a program from a high-level programming language to low-level machine language is the job done by the software programs called *compilers* and *interpreters.* High-level programming languages are simply a programmer's convenience; they

cannot be executed in their original form (source code). This section describes a few of the major programming languages.

COBOL The COBOL (COmmon Business Oriented Language) programming language was introduced in 1959. Developed in response to business community requirements, it was used mostly on mainframe computers providing data-processing services to large companies. More efficient and faster general-purpose computer programming languages exist in the market, but COBOL is going to remain with us because more than half of all business application programs are written in COBOL.

FORTRAN The FORTRAN (FORmula TRANslator) programming language was developed in 1955. It is most suitable for scientific or engineering programming, and remains the most popular scientific language. Of course, FORTRAN has been modified to meet the demands of new computers. The current version of FORTRAN is FORTRAN77, which is a general-purpose language capable of dealing with numeric and symbolic problems. As with COBOL, a large body of existing code was written in FORTRAN, and it is still used in engineering and science.

Pascal The Pascal programming language was developed in 1968 and is named after the seventeenth-century French mathematician Blaise Pascal. The idea of good style and habits in programming was around and the buzzwords "structured programming" were taking shape when Pascal was created, which incorporated *structured programming* ideas. Pascal was designed as a language to help students learn structured programming and develop good habits in programming. However, the use of Pascal is not limited to educational institutions; it is used in industry to create readable and maintainable codes.

BASIC The BASIC (Beginners All-purpose Symbolic Instruction Code) programming language was developed in 1964 to help students with no computer background learn about computers and programming. It is accepted as the most effective programming language for instructional purposes. Since then, BASIC has evolved into a general-purpose language, and its use is not limited to the education field. BASIC is widely implemented and is used in business applications.

C The C programming language was developed in 1972, and it is based on principles practiced in Pascal programming. It is mainly for targeted system programming, creating operating systems, compilers, and so on. Most of the UNIX operating system is written in C. C is a fast, efficient, general-purpose language. It is also a portable language; it is relatively machine independent. A C program written for one type of computer can be run on another with little or no modification. C is by far the language of choice for many programmers developing programs for business, scientific, and other applications.

C++ In the 1980s, C++ was developed to add to the C language the tools needed to make it an object-oriented language. *Object-oriented programming* (OOP) is one of the many programming paradigms. The OOP techniques make programmers' lives easier and reduce the program development time, especially when programs reach a certain size. C++ provides the language mechanism for implementing the OOP concept.

Java The Java programming language was developed by Sun Microsystems, Inc., in 1990 and was released to the public in 1995. Originally, it was designed for use in control systems of consumer electronics devices such as television sets, microwave oven, and so forth. Later, it was embedded into Internet browsers and programs for Web pages. Java is syntactically similar to C++, but many of the confusing features of C++ are discarded.

11.3 PROGRAMMING MECHANICS

To write a program, you have to choose a computer programming language. The choice of programming language depends on the nature of the application. Many general-purpose and specialized programming languages exist that fit any requirement. The exact steps you must follow to produce a program depend on the computer environment. The following discussion traces the steps in a typical program's development.

11.3.1 Steps to Creating an Executable Program

Regardless of the computer operating system and the programming language you use, the following steps are necessary to create an executable program:

1. Create the source file (source code).
2. Create the object file (object code/object module).
3. Create the executable file (executable code/load module).

Source Code You usually use an editor (such as the vi editor) to write a program and save what you write in a file. This file is the *source code.* The source code is written in the programming language of your choice. By itself, your computer does not understand it. The goal is to convert the source code file to an executable file.

A source file is a text file, which is also called an ASCII file. You can display it on the screen, modify it using one of the available editors, or send it to the printer to obtain a hard copy of your program's source code.

Object Code Source code is incomprehensible to the computer. Remember, computers understand only machine language (the pattern of zeros and ones). Thus, your source code must be translated to machine-understandable language. This is the job of a compiler or an interpreter. They produce the object code. The *object code* is the machine language translation of your source code. However, it is not an executable file yet; it lacks some necessary parts. These necessary parts are programs that provide the interface between the program and the operating system. They are usually grouped together in files called *library files.*

You cannot send an object file to the printer. It is a file of zeros and ones. If you display it, you may lock your keyboard or hear beeps as some of the zeros and ones are translated into the ASCII codes that represent codes for locking the keyboard, the beep at your terminal, stop scrolling, and so on.

Executable Code Your object code might refer to other programs that are not part of your *object module.* Before your program can be executed, these references to other programs must be resolved. This is the job of the *linker* or *link editor.* It creates the *executable code,* the *load module.* The *load module* is a complete, ready-to-be-executed program with all its parts put together.

The load module, like the object file, is not a file to be sent to the printer or displayed on your terminal.

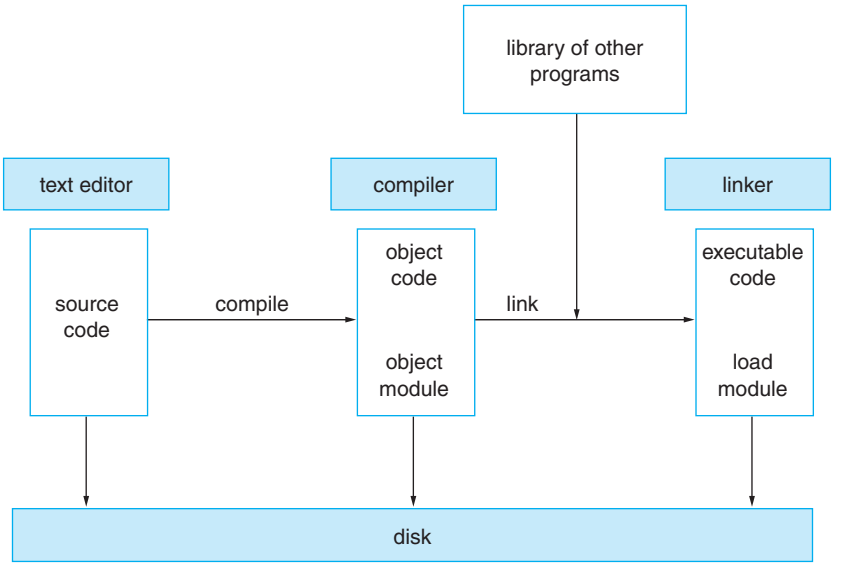
In some systems, you invoke the compiler, and after the compilation process is completed, you invoke the linker to create the executable file. Other systems have the

compiler start the linker automatically. On those systems, you just give the compile command, and, if your program does not have any compilation error, the linker is invoked to link your program.

Exactly how this works varies among systems, depending on the system environment/ configuration setup as done by the system administrator.

This process, starting from a source file and ending with creating an executable file, is depicted in Figure 11.3.

Figure 11.3
Program Development Steps



11.3.2 Compilers/Interpreters

The main function of a compiler or an interpreter is to translate your source code (program instructions) to machine code so the computer can understand your instructions. The compiled languages and interpreted languages represent two different categories of computer languages. Each has its advantages and disadvantages.

Compiler The compiler is a system software program that translates high-level program instructions, such as a Pascal program, into machine language that the computer can interpret and execute. It compiles the entire program at one time and does not give you any feedback until it compiles the entire program.

A separate compiler is required for each programming language you intend to use on your computer system. To execute C++ and Pascal programs, you must have a C++ compiler and a Pascal compiler. Compilers produce a better and more efficient object code than interpreters, so a compiled program runs faster and needs less space.

Interpreter Like the compiler, the interpreter translates a high-level language program into machine language. However, instead of translating the entire source program, it

translates a single line at a time. An interpreter gives you immediate feedback. If the code contains an error, the interpreter picks it up as soon as you press **[Return]** to finish a line of code. Therefore, you can correct errors during program development. The interpreter does not produce a separate object code file, and it must perform the translation process each time a program is executed. Interpreters usually are used in an educational environment, and the executable code they produce is less efficient than the code produced by a compiler.

11.4 A SIMPLE C++ PROGRAM

Let's write a simple C++ program and go through the steps to see how this process works. The goal is to understand and practice the compilation process, not to learn the C++ language. However, you will have to understand some very basic characteristics of the C++ language to be able to write a simple C++ program and compile it successfully.

Using the vi editor (or any other editor), create a source file called `first.cpp`. Figure 11.4 shows the contents of the file.

Figure 11.4
A Simple C++ Program

```
$ cat first.cpp
// My first C++ program
#include <iostream>
int main()
{
 std::cout << "Hi there!" << std::endl";
 std::cout << "This is my first C++ program" << std::endl";
 return(0);
}
$_
```

1. **Type the source code in lowercase letters. Like UNIX, the C++ language is a lowercase language, and all the keywords must be typed in lowercase.**
2. **In most systems, the extension `.cpp` at the end of the source code filename is mandatory.**

The next step is to compile your source code. The command is as follows:

```
$ g++ first.cpp [Return]
```

The **g++** command compiles your source code and, if you do not have any errors, automatically invokes the linker. The end result is that you have an executable file called `a.exe`. By default, `a.exe` is the name of the executable file. Now if you want to execute the file to see the program output, you type

```
$ a.exe [Return]  . . . . Execute your program.
Hi there!
This is my first C++ program.
$_ . . . . . . . . . . . . And the prompt appears.
```

The output of `a.exe` *is displayed on the standard output device, your terminal.*

What if you do not want to call your executable file **`a.exe`**? You can use the **g++** command with the **-o** option to specify the name of the output file.

Compile `first.cpp` and indicate the name of the output file. Call it `first`.

```
$ g++ -c -o first first.cpp [Return] . . .  Use g++ with the -o
                                             option.
$_ . . . . . . . . . . . . . . . . . . . .  No errors, and the
                                             prompt is back.
```

Make sure the name of the specified output file and the name of the source code file are different; otherwise, your source code file is overwritten and becomes your executable file—which means your source code file is destroyed.

Now your executable file is called `first`, and any time you type **first** at the prompt, you see program output on the screen:

```
$ first [Return]  . . . . . . . . . . Execute first.
Hi there!
This is my first C++ program.
$_ . . . . . . . . . . . . . . . . . . Ready for the next command.
```

What if you do not want the output on the screen? Maybe you want to store the output of your program in a file. Using the output redirection capability of the shell, you can redirect the program output into another file.

Run **first** and save its output in a file:

```
$ first > first.out [Return] . . . .  Output is redirected to a file
                                       called first.out, so it is not
                                       displayed on the screen.
$ cat first.out [Return] . . . . . .  Check the contents of
                                       first.out.
Hi there!
This is my first C++ program.
$_ . . . . . . . . . . . . . . . . .  And the prompt appears.
```

As you expected, the contents of the `first.out` *file are the output of your C++ program. The* `first.out` *file was created by redirecting the output of your program into it.*

Execute **first**, show its output on the screen, and save it in a file:

```
$ first | tee -a first.out [Return]  .  The output is displayed on
                                        the screen and also saved in
                                        first.out.
Hi there!
This is my first C++ program.
$_ . . . . . . . . . . . . . . . . . .  The shell prompt appears.
```

Using the **tee** (split output) command with the **pipe** operator, the output of the program **first** is displayed on the screen and also saved in `first.out`. The **tee** command **-a** option appends the output to the end of the `first.out` file.

11.4.1 Correcting Mistakes

Assuming you did not make any mistake in writing your first C++ program, things are fairly simple. You compile and execute your program. But this is not always the case when you write large programs. The chances are good that you will make syntax and logical mistakes that you will have to correct before being able to run a program successfully. The C++ compiler recognizes syntax errors and displays them on the screen with a line number reference.

Suppose you modify `first.cpp` to look like the file in Figure 11.5, so that it has no semicolon (**;**) at the end of the first **cout**() statement. This time when you compile `first.cpp`, the compiler is going to complain.

Figure 11.5
A Simple C++ Program with a Syntax Error

```
$ cat first.cpp
// My first C++ program
#include <iostream>
int main()
{
 std::cout << "Hi there!" << std::endl" // Semi colon omitted intentionally
 std::cout << "This is my first C++ program" << std::endl;
 return(0);
}
$_
```

Now compile first.cpp.

```
$ g++ -c -o first first.cpp [Return]  . .   Compile and call the
                                            output file first.
first.cpp: In function 'int main()':
first.cpp:6: error: expected ';' before "std"
$_. . . . . . . . . . . . . . . . . . . . . Prompt.
```

The error messages indicate that you have some sort of syntax error on line 6. The compiler is not clever enough to recognize the exact error and its location; it just directs you to the vicinity of the error in the source code.

At this point, the compiler has not produced an object code file. To correct the errors, you must go back to the source code, `first.cpp`, and add the semicolon. Then, to get the correct executable file, you have to recompile the file.

It is easy to look at one or two error lines on the screen, but when you have a large amount of source code, the chances are good that you will have more than a few lines with errors. Remembering the errors and their line numbers when you want to edit your source file to correct the errors is not a trivial task. Therefore, it is desirable to save the compiler error messages in a file for easy reference. The redirection capability of the shell becomes handy again!

The default error device is usually the same as the output device, your terminal, so the errors are displayed on your terminal. Suppose you want to redirect the errors to a specified file.

Compile `first.cpp` and save compilation errors, if any, in another file:

```
$ g++ -c -o first first.cpp 2> error [Return] .  Recompile.
$_ . . . . . . . . . . . . . . . . . . . . . . . The shell
                                                 prompt appears.
```

The digit 2 before the > sign is necessary and indicates the redirection of the standard error device.

Where does the digit 2 come from? The command just shown needs some explanation. Let's go back to the UNIX redirection concept and explore the origins of the digit *2*.

11.4.2 Redirecting the Standard Error

The shell interprets the **>** as standard output redirection. The notation **1>** is the same as **>** and tells the shell to redirect the standard output. The number *1* in **1>** is the file descriptor number; by default, file descriptor 1 is assigned to the *standard output device*. For example, the following two commands do the same job: They redirect the output of the **ls** command to a file.

```
$ ls -C > list [Return]
```

or

```
$ ls -C 1> list [Return]
```

File descriptor 2 is assigned to the *standard error device*. The shell interprets the notation **2>** as the redirection of the error output. For example, suppose you do not have a file called **Y** in your current directory, and you issue the following command:

```
$ cat Y > Z [Return]. . . . . . . . . . Copy Y to Z.
cat: cannot open Y. . . . . . . . . . . Error message.
```

The error message appears on the screen. Now redirect the error output to a file.

```
$ cat Y > Z 2> error [Return]. . . . . Copy Y to Z. Keep errors in a
                                       file called error in the
                                       current directory.
$_ . . . . . . . . . . . . . . . . . . No error messages. The
                                       prompt is back.
$ cat error [Return]. . . . . . . . . . Show the contents of the
                                       error file.
cat: cannot open Y. . . . . . . . . . . As you expected.
$_ . . . . . . . . . . . . . . . . . . And the prompt appears.
```

Now back to your C++ program and its compilation errors; you can redirect the compilation errors to another file by typing:

```
$ g++ -c -o first first.cpp 2> error [Return]  . . . . Error message redirected
                                                       to the error file.
$_ . . . . . . . . . . . . . . . . . . . . . . . . . . No error is displayed.
$ cat error [Return] . . . . . . . . . . . . . . . . . Check the compilation
                                                       errors.
"first.cpp",: 6: error: expected`;` before "std"
$_ . . . . . . . . . . . . . . . . . . . . . . . . . . Prompt.
```

11.5 UNIX PROGRAMMING TRACKING UTILITIES

Other computer language compilers are available that work in a UNIX environment. You can obtain compilers for nearly every language to run under UNIX.

The goal of this chapter is to introduce you to programming development under UNIX—not to present you with a comprehensive list of languages, compilers, and UNIX programming. However, it is important to know that UNIX provides you with utilities to help you organize your program development process. These utilities become especially useful and important when you are developing large-scale software. Following is a very brief explanation of these utilities and their functions.

11.5.1 The make Utility

The **make** utility is useful when your program consists of more than one file. **make** automatically keeps track of the source files that are changed and that need recompilation, and relinks your programs if required. The **make** program gets its information from a control file. The *control file* contains rules that specify source files' dependencies and other information.

11.5.2 The SCCS Utility

The **SCCS** (Source Code Control System) is a collection of programs that helps you to maintain and manage the development of your programs. If your program is under **SCCS** control, then you can create different versions of your program easily. The **SCCS** keeps track of all the changes among different versions.

REVIEW EXERCISES

1. Explain the steps necessary to write a program and produce an executable file.
2. What is a source code?
3. What is the function of a compiler?
4. What is the difference between a compiler and an interpreter?
5. What is the command to compile a C++ program, and what is the default name of the executable file?
6. Why can't you send your executable file to a printer?

Terminal Session

In this terminal session, you are to write a sample C++ program. It is not expected that you know C++ programming. Copy the simple C++ program example in this chapter or one from any C++ programming book. The purpose is to familiarize yourself with the process of program development.

1. Write a simple C++ program.
2. Compile it.
3. Run the program.
4. Compile it again and specify the executable file.
5. Run it again.
6. Save the output of your program in another file.
7. Modify your source code and make an intentional syntax error.
8. Compile again.
9. Observe the error messages. See if you can decipher them.
10. Recompile your program and save the error messages in a file.
11. Look at the file that contains the compilation errors.