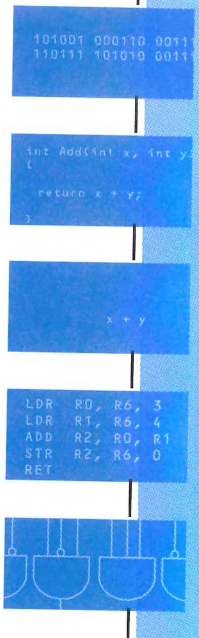
[illegible]

chapter

11


```
101001 000110 0011
110111 101010 0011
```

```
int Add(int x, int y
{
  return x + y;
}
```

```
x + y
```

```
LDR  R0, R6, 3
LDR  R1, R6, 4
ADD  R2, R0, R1
STR  R2, R6, 0
RET
```

Introduction to Programming in C

11.1 Our Objective

Congratulations, and welcome to the second half of the book! You just completed an introduction to the basic underlying structure of modern computer systems. With this foundational material solidly in place, you are now well prepared to learn the fundamentals of programming in a high-level programming language.

In the second half of this book, we will discuss high-level programming concepts in the context of the C programming language. At every step, with every new high-level concept, we will be able to make a connection to the lower levels of the computer system. From this perspective, nothing will be mysterious. We approach the computer system from the bottom up in order to reveal that there indeed is no magic going on when the computer executes the programs you write. It is our belief that with this mystery removed, you will comprehend programming concepts more quickly and deeply and in turn become better programmers.

Let's begin with a quick overview of the first half. In the first 10 chapters, we described the LC-3, a simple computer that has all the important characteristics of a more complex, real computer. A basic idea behind the design of the LC-3 (and indeed, behind all modern computers) is that simple elements are systematically interconnected to form more sophisticated devices. MOS transistors are connected to build logic gates. Logic gates are used to build memory and data path elements. Memory and data path elements are interconnected to build the LC-3. This systematic connection of simple elements to create something more sophisticated is an important concept that is pervasive throughout computing, not only in hardware design but also in software design. It is this simple design philosophy that enables us to build computing systems that are, as a whole, very complex.

After describing the hardware of the LC-3, we described how to program it in the 1s and 0s of its native machine language. Having gotten a taste of the error-prone and unnatural process of programming in machine language, we quickly moved to the more user-friendly LC-3 assembly language. We described how to decompose a programming problem systematically into pieces that could be easily coded on the LC-3. We examined how low-level TRAP subroutines perform commonly needed tasks, such as input and output, on behalf of the programmer. The concepts of systematic decomposition and subroutines are important not only for assembly-level programming but also for programming in a high-level language. You will continue to see examples of these concepts many times before the end of the book.

In this half of the book, our primary objectives are to introduce fundamental high-level programming constructs—variables, control structures, functions, arrays, pointers, recursion, simple data structures—and to teach a good problem-solving methodology for attacking programming problems. Our primary vehicle for doing so is the C programming language. It is not our objective to provide a complete coverage of C, but only the portions essential for a novice programmer to gain exposure to the fundamentals of programming and to be able to write fairly sophisticated programs. For the reader curious about aspects of C not covered in the main text, we provide a more complete description of the language in Appendix D.

In this chapter, we make the transition from programming in low-level assembly language to high-level language programming in C. We'll explain why high-level languages came about, why they are important, and how they interact with the lower levels of the computing system. We'll then dive headfirst into C by examining a simple example program. Using this example, we point out some important details that you will need to know in order to start writing your own C code.

11.2 Bridging the Gap

As computing hardware becomes faster and more powerful, software applications become more complex and sophisticated. New generations of computer systems spawn new generations of software that can do more powerful things than previous generations. As the software gets more sophisticated, the job of developing it becomes more difficult. To keep the programmer from being quickly overwhelmed, it is critical that the process of programming be kept as simple as possible. Automating any part of this process (i.e., having the computer do part of the work) is a welcome enhancement.

As we made the transition from LC-3 machine language in Chapters 5 and 6 to LC-3 assembly language in Chapter 7, you no doubt noticed and appreciated how assembly language simplified programming the LC-3. The 1s and 0s became mnemonics, and memory addresses became symbolic labels. Both instructions and memory addresses took on a form more comfortable for the human than for the machine. The assembler filled some of the *gap* between the algorithm level and the ISA level in the levels of transformation (see Figure 1.6). It would be desirable for the language level to fill more of that gap. High-level languages do just that. They help make the job of programming easier. Let's look at some ways in which they help.

- **High-level languages allow us to give symbolic names to values.** When programming in machine language, if we want to keep track of the iteration count of a loop, we need to set aside a memory location or a register in which to store the counter value. To access the counter, we need to remember the spot where we last stored it. The process is easier in assembly language because we can assign a meaningful label to the counter's memory location. In a higher-level language such as C, the programmer simply assigns the value a name (and, as we will see later, provides a *type*) and the programming language takes care of allocating storage for it and performing the appropriate data movement operations whenever the programmer refers to it. Since most programs contain many values, having such a convenient way to handle values is a critically useful enhancement.

- **High-level languages provide expressiveness.** Most humans are more comfortable describing the interaction of objects in the real world than describing the interaction of objects such as integers, characters, and floating-point numbers in the digital world. Because of their human-friendly orientation, high-level languages enable the programmer to be more expressive. In a high-level language, the programmer can express complex tasks with a smaller amount of code, with the code itself looking more like a human language. For example, if we wanted to calculate the area of a triangle, we could simply write:

```
area = 0.5 * base * height;
```

Another example: we often write code to test a condition and do something if the condition is true or do something else if the condition is false. In high-level languages, such common tasks can be simply stated in an English-like form. For example, if we want to `get(Umbrella)` if the condition `isItCloudy` is true, otherwise `get(Sunglasses)` if it is false, then in C we can use the following C *control structure*:

```
if (isItCloudy)
   get(Umbrella);
else
   get(Sunglasses);
```

- **High-level languages provide an abstraction of the underlying hardware.** In other words, high-level languages provide a uniform interface independent of underlying ISA or hardware. For example, often a programmer will want to do an operation that is not naturally supported by the instruction set. In the LC-3, there is no one instruction that performs an integer multiplication. Instead, an LC-3 assembly language programmer must write a small piece of code to perform multiplication. The set of operations supported by a high-level language is usually larger than the set supported by the ISA. The language will generate the necessary code to carry out the operation whenever the programmer uses it. The programmer can concentrate on the actual programming task knowing that these high-level operations will be performed correctly and without having to deal with the low-level implementation.

- **High-level languages enhance code readability.** Since common control structures are expressed using simple, English-like statements, the program itself becomes easier to read. One can look at a program in a high-level language, notice loops and decision constructs, and understand the code with less effort than with a program written in assembly language. As you will no doubt discover if you have not already, the readability of code is very important in programming. Often as programmers, we are given the task of debugging or building upon someone else's code. If the organization of the language is human-friendly to begin with, then understanding code in that language is a much simpler task.
- **Many high-level languages provide safeguards against bugs.** By making the programmer adhere to a strict set of rules, the language can make checks as the program is translated or as it is executed. If certain rules or conditions are violated, an error message will direct the programmer to the spot in the code where the bug is likely to exist. In this manner, the language helps the programmer to get his/her program working more quickly.

11.3 Translating High-Level Language Programs

Just as LC-3 assembly language programs need to be translated (or more specifically, assembled) into machine language, so must all programs written in high-level languages. After all, the underlying hardware can only execute machine code. How this translation is done depends on the particular high-level language. One translation technique is called *interpretation*. With interpretation, a translation program called an *interpreter* reads in the high-level language program and performs the operations indicated by the programmer. The high-level language program does not directly execute but rather is executed by the interpreter program. The other technique is called *compilation*, and the translator, called a *compiler*, completely translates the high-level language program into machine language. The output of the compiler is called the executable image, and it can directly execute on the hardware. Keep in mind that both interpreters and compilers are themselves programs running on the computer system.

11.3.1 Interpretation

With interpretation, a high-level language program is a set of commands for the interpreter program. The interpreter reads in the commands and carries them out as defined by the language. The high-level language program is not directly executed by the hardware but is in fact just input data for the interpreter. The interpreter is a *virtual machine* that executes the program. Many interpreters translate the high-level language program section by section, one line, command, or subroutine at a time.

For example, the interpreter might read a single line of the high-level language program and directly carry out the effects of that line on the underlying hardware. If the line said, "Take the square root of B and store it into C," the interpreter will carry out the square root by issuing the correct stream of instructions in the ISA of the computer to perform square root. Once the current line is processed, the interpreter moves on to the next line and executes it. This process continues until the entire high-level language program is done.

High-level languages that are often interpreted include LISP, BASIC, and Perl. Special-purpose languages tend to be interpreted, such as the math language called Matlab. The LC-3 simulator is also an interpreter. Other examples include the UNIX command shell.

11.3.2 Compilation

With compilation, on the other hand, a high-level language program is translated into machine code that can be directly executed on the hardware. To do this effectively, the compiler must analyze the source program as a larger unit (usually, the entire source file) before producing the translation. A program need only be compiled once and can be executed many times. Many programming languages, including C, C++, and FORTRAN, are typically compiled. The LC-3 assembler is an example of a rudimentary compiler. A compiler *processes* the file (or files) containing the high-level language program and produces an executable image. The compiler does not execute the program (though some sophisticated compilers do execute the program in order to better optimize its performance), but rather only transforms it from the high-level language into the computer's native machine language.

11.3.3 Pros and Cons

There are advantages and disadvantages with either translation technique. With interpretation, developing and debugging a program is usually easier. Interpreters often permit the execution of a program one section (single line, for example) at a time. This allows the programmer to examine intermediate results and make code modifications on-the-fly. Often the debugging is easier with interpretation. Interpreted code is more easily portable across different computing systems. However, with interpretation, programs take longer to execute because there is an intermediary, the interpreter, which is actually doing the work. With the compiler's assistance, the programmer can produce code that executes more quickly and uses memory more efficiently. Since compilation produces more efficient code, most commercially produced software tends to be programmed in compiled languages.

11.4 The C Programming Language

The C programming language was developed in 1972 by Dennis Ritchie at Bell Laboratories. C was developed for use in writing compilers and operating systems, and for this reason the language has a low-level bent to it. The language allows

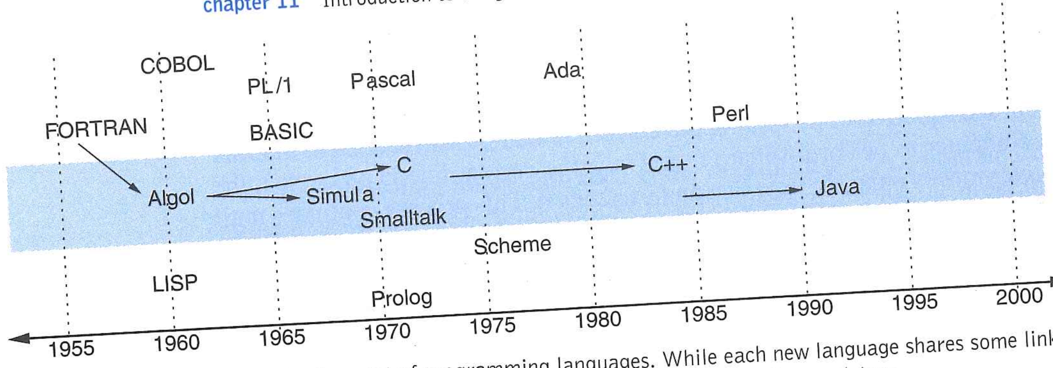


Figure 11.1 A timeline of the development of programming languages. While each new language shares some link to all previous languages, there is a strong relationship between C and both C++ and Java

the programmer to manipulate data items at a very low level yet still provides the expressiveness and convenience of a high-level language. It is for these reasons that C is very widely used today as more than just a language to develop compilers and system software.

The C programming language has a special place in the evolution of programming languages. Figure 11.1 provides a timeline of the development of some of the more significant programming languages. Starting with the introduction of the first high-level programming language FORTRAN in 1954, each subsequent language was an attempt to fix the problems with its predecessors. While it is somewhat difficult to completely track the "parents" of a language (in fact, one can only surely say that *all* previous languages have some influence on a particular language), it is fairly clear that C had a direct influence on C++ and Java, both of which are two of the more significant languages today. C++ and Java were also influenced by Simula and its predecessors. The object-oriented features of C++ and Java come from these languages. Almost all of the aspects of the C programming language that we discuss in this textbook would be the same if we were programming in C++ or Java. Once you've understood the concepts in this half of the textbook, both C++ and Java will also be easier to master because of their similarity to C.

Because of its low-level approach and because of its root influence on other current major languages, C is the language of choice for our bottom-up exploration of computing systems. C allows us to make clearer connections to the underlying levels in our discussions of basic high-level programming concepts. Learning more advanced concepts, such as object-oriented programming, is a shorter leap forward once these more fundamental, basic concepts are understood.

All of the examples and specific details of C presented in this text are based on a standard version of C called ANSI C. As with many programming languages, several variants of C have been introduced throughout the years. In 1989, the American National Standards Institute (ANSI) approved "an unambiguous and machine-independent definition of the language C" in order to standardize the popular language. This version is referred to as *ANSI C*. ANSI C is supported by most C compilers. In order to compile and try out the sample code in this textbook, having access to an ANSI-compliant C compiler will be essential.

11.4.1 The C Compiler

The C compiler is the typical mode of translation from a C source program to an *executable image*. Recall from Section 7.4.1 that an executable image is a machine language representation of a program that is ready to be loaded into memory and executed. The entire compilation process involves the preprocessor, the compiler itself, and the linker. Often, the entire mechanism is casually referred to as *the compiler*, because when we use the C compiler, the preprocessor and the linker are often automatically invoked. Figure 11.2 shows how the compilation process is handled by these components.

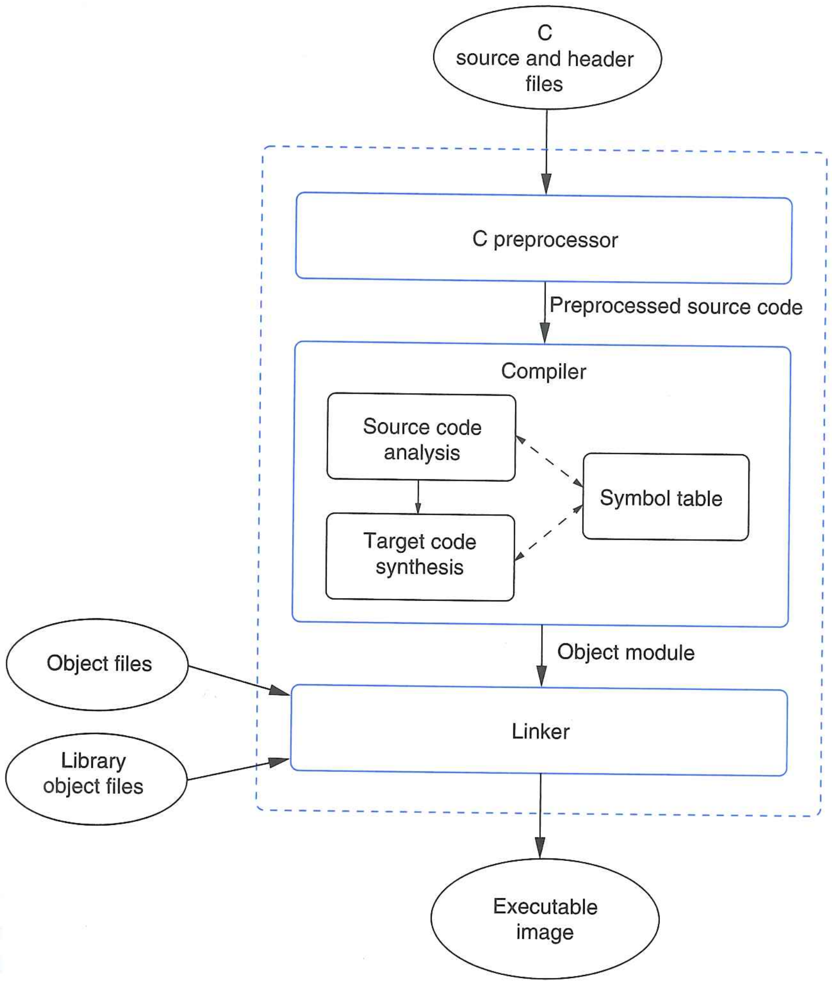


Figure 11.2 The dotted box indicates the overall compilation process—the preprocessor, the compiler, and the linker. The entire process is called compilation even though the compiler is only one part of it. The inputs are C source and header files and various object files. The output is an executable image.

The Preprocessor

As its name implies, the C preprocessor "preprocesses" the C program before handing it off to the compiler. The C preprocessor scans through the source files (the source files contain the actual C program) looking for and acting upon C preprocessor directives. These directives are similar to pseudo-ops in LC-3 assembly language. They instruct the preprocessor to transform the C source file in some controlled manner. For example, we can direct the preprocessor to substitute the character string `DAYS_THIS_MONTH` with the string `30` or direct it to insert the contents of file `stdio.h` into the source file at the current line. We'll discuss why both of these actions are useful in the subsequent chapters. All preprocessor directives begin with a pound sign, #, as the first character. All useful C programs rely on the preprocessor in some way.

The Compiler

After the preprocessor transforms the input source file, the program is ready to be handed over to the compiler. The compiler transforms the preprocessed program into an *object module*. Recall from Section 7.4.2 that an object module is the machine code for one section of the entire program. There are two major phases of compilation: analysis, in which the source program is broken down or *parsed* into its constituent parts, and synthesis, in which a machine code version of the program is generated. It is the job of the analysis phase to read in, parse, and build an internal representation of the original program. The synthesis phase generates machine code and, if directed, attempts to optimize this code to execute more quickly and efficiently on the computer on which it will be run. Each of these two phases is typically divided into subphases where specific tasks, such as parsing, register allocation, or instruction scheduling, are accomplished. Some compilers generate assembly code and use an assembler to complete the translation to machine code.

One of the most important internal bookkeeping mechanisms the compiler uses in translating a program is the *symbol table*. A symbol table is the compiler's internal bookkeeping method for keeping track of all the symbolic names the programmer has used in the program. The C compiler's symbol table is very similar to the symbol table maintained by the LC-3 assembler (see Section 7.3.3). We'll examine the C compiler's symbol table in more detail in the next chapter.

The Linker

The linker takes over after the compiler has translated the source file into object code. It is the linker's job to link together all object modules to form an executable image of the program. The executable image is a version of the program that can be loaded into memory and executed by the underlying hardware. When you click on the icon for the web browser on your PC, for example, you are instructing the operating system to read the web browser's executable image from your hard drive, load it into memory, and start executing it.

Often, C programs rely upon library routines. Library routines perform common and useful tasks (such as I/O) and are prepared for general use by the developers of the system software (the operating system and compiler, for example). If a program uses a library routine, then the linker will find the object code corresponding to the routine and link it within the final executable image. This process of linking in library objects should not be new to you; we described the process in Section 9.2.5 in the context of the LC-3. Usually, library objects are stored in a particular place depending on the computer system. In UNIX, for example, many common library objects can be found in the directory `/usr/lib`.

11.5 A Simple Example

We are now ready to start discussing programming concepts in the C programming language. Many of the new C concepts we present will be coupled with LC-3 code generated by a "hypothetical" LC-3 C compiler. In some cases, we will describe what actually happens when this code is executed. Keep in mind that you are not likely to be using an LC-3–based computer but rather one based on a real ISA such as the x86. For example, if you are using a Windows-based PC, then it is likely that your compiler will generate x86 code, not LC-3 code.

Many of the examples we provide are complete programs that you can compile and execute. For the sake of clearer illustration, some of the examples we provide are not quite complete programs and need to be completed before they can be compiled. In order to keep things straight, we'll refer to these partial code examples as *code segments*.

Let's begin by diving headfirst into a simple C example. Figure 11.3 shows its *source code*. We will use this example to jump-start the process of learning C by pointing out some important aspects of a typical C program. The example is a simple one: It prompts the user to type in a number and then counts down from that number to 0.

You are encouraged to compile and execute this program. At this point, it is not important to completely understand the purpose of each line. There are however several aspects of this example that will help you with writing your own C code and with comprehending the subsequent examples in the text. We'll focus on four such aspects: the function `main`, the code's comments and programming style, preprocessor directives, and the I/O function calls.

11.5.1 The Function `main`

The function `main` begins at the line containing `int main()` (line 17) and ends at the closing brace on the last line of the code. These lines of the source code constitute a *function definition* for the function named `main`. What were called subroutines in LC-3 assembly language programming (discussed in Chapter 9) are referred to as *functions* in C. Functions are a very important part of C, and we will devote all of Chapter 14 to them. In C, the function `main` serves a special purpose: It is where execution of the program begins. Every C program, therefore, requires a function `main`. Note that in ANSI C, `main` must be declared to return

```
1   /*
2    *
3    *  Program Name : countdown, our first C program
4    *
5    *  Description  : This program prompts the user to type in
6    *  a positive number and counts down from that number to 0,
7    *  displaying each number along the way.
8    *
9    */
10
11  /* The next two lines are preprocessor directives */
12  #include <stdio.h>
13  #define STOP 0
14
15  /* Function    : main                                     */
16  /* Description : prompt for input, then display countdown */
17  int main()
18  {
19    /* Variable declarations */
20    int counter;          /* Holds intermediate count values */
21    int startPoint;       /* Starting point for count down   */
22
23    /* Prompt the user for input */
24    printf("===== Countdown Program =====\n");
25    printf("Enter a positive integer: ");
26    scanf("%d", &startPoint);
27
28    /* Count down from the input number to 0 */
29    for (counter = startPoint; counter >= STOP; counter--)
30      printf("%d\n", counter);
31  }
```

Figure 11.3 A program prompts the user for a decimal integer and counts down from that number to 0

an integer value. That is, `main` must be of type `int`, thus line 17 of the code is `int main()`.

In this example, the code for function `main` (i.e., the code in between the curly braces) can be broken down into two components. The first component contains the variable *declarations* for the function. Two variables, one called `counter` and the other `startPoint`, are created for use within the function `main`. Variables are a very useful feature provided by high-level programming languages. They give us a way to symbolically name the values within a program.

The second component contains the *statements* of the function. These statements express the actions that will be performed when the function is executed. For all C programs, execution starts in `main` and progresses, statement by statement, until the last statement in `main` is completed.

In this example, the first grouping of statements (lines 24–26) displays a message and prompts the user to input an integer number. Once the user enters a number, the program enters the last statement, which is a `for` loop (a type of iteration construct that we will discuss in Chapter 13). The loop counts downward from the number typed by the user to 0. For example, if the user entered the number 5, the program's output would look as follows:

```
===== Countdown Program =====
Enter a positive integer: 5
5
4
3
2
1
0
```

Notice in this example that many lines of the source code are terminated by semicolons, `;`. In C, semicolons are used to terminate declarations and statements; they are necessary for the compiler to break the program down unambiguously into its constituents.

11.5.2 Formatting, Comments, and Style

C is a free-format language. That is, the amount of spacing between words and between lines within a program does not change the meaning of the program. The programmer is free to structure the program in whatever manner he/she sees fit while obeying the syntactic rules of C. Programmers use this freedom to format the code in a manner that makes it easier to read. In the example program, notice that the `for` loop is indented in such a manner that the statement being iterated is easier to identify. Also in the example, notice the use of blank lines to separate different regions of code in the function `main`. These blank lines are not necessary but are used to provide visual separation of the code. Often, statements that together accomplish a larger task are grouped together into a visually identifiable unit. The C code examples throughout this book use a conventional indentation style typical for C. Styles vary. Programmers sometimes use style as a means of expression. Feel free to define your own style, keeping in mind that the objective is to help convey the meaning of the program through its formatting.

Comments in C are different than in LC-3 assembly language. Comments in C begin with `/*` and end with `*/`. They can span multiple lines. Notice that this example program contains several lines of comments, some on a single line, some spanning multiple lines. Comments are expressed differently from one programming language to another. For example, comments in C++ can also begin with the sequence `//` and extend to the end of the line. Regardless of how comments are expressed, the purpose is always the same: They provide a way for programmers to describe in human terms what their code does.

Proper commenting of code is an important part of the programming process. Good comments enhance code readability, allowing someone not familiar with the code to understand it more quickly. Since programming tasks often involve working in teams, code very often gets shared or borrowed between programmers. In order to work effectively on a programming team, or to write code that is worth sharing, you must adopt a good commenting style early on.

One aspect of good commenting style is to provide information at the beginning of each source file that describes the code contained within it, the date it was last modified, and by whom. Furthermore, each function (see function `main` in the example) should have a brief description of what the function accomplishes, along with a description of its inputs and outputs. Also, comments are usually interspersed within the code to explain the intent of the various sections of the code. But overcommenting can be detrimental as it can clutter up your code, making it harder to read. In particular, watch out for comments that provide no additional information beyond what is obvious from the code.

11.5.3 The C Preprocessor

We briefly mentioned the C preprocessor in Section 11.4.1. Recall that it transforms the original C program before it is handed off to the compiler. Our simple example contains two commonly used preprocessor directives: `#define` and `#include`. The C examples in this book rely only on these two directives.

The `#define` directive is a simple yet powerful directive that instructs the C preprocessor to replace occurrences of any text that matches X with text Y. That is, the *macro* X gets *substituted* with Y. In the example, the `#define` causes the text `STOP` to be substituted with the text `0`. So the following source line

```
for (counter = startPoint; counter >= STOP; counter--)
```

is transformed (internally, only between the preprocessor and compiler) into

```
for (counter = startPoint; counter >= 0; counter--)
```

Why is this helpful? Often, the `#define` directive is used to create fixed values within a program. Following are several examples.

```
#define NUMBER_OF_STUDENTS  25
#define MAX_LENGTH          80
#define LENGTH_OF_GAME      300
#define PRICE_OF_FUEL       1.49
#define COLOR_OF_EYES       brown
```

So for example, we can symbolically refer to the price of fuel as `PRICE_OF_FUEL`. If the price of fuel were to change, we would simply modify the definition of the macro `PRICE_OF_FUEL` and the preprocessor would handle the actual substitution for us. This can be very convenient—if the cost of fuel was used heavily within a program, we would only need to modify one line in the source code to change the price throughout the code. Notice that the last example is slightly different from the others. In this example, one string of characters `COLOR_OF_EYES` is being substituted for another, `brown`. The common programming style is to use uppercase for the macro name.

The `#include` directive instructs the preprocessor literally to insert another file into the source file. Essentially, the `#include` directive itself is replaced by the contents of another file. At this point, the usefulness of this command may not be completely apparent to you, but as we progress deeper into the C language, you will understand how *C header files* can be used to hold `#defines` and declarations that are useful among multiple source files.

For instance, all programs that use the C I/O functions must include the I/O library's header file `stdio.h`. This file defines some relevant information about the I/O functions in the C library. The preprocessor directive, `#include <stdio.h>` is used to insert the header file before compilation begins.

There are two variations of the `#include` directive:

```
#include <stdio.h>
#include "program.h"
```

The first variation uses angle brackets (`< >`) around the filename. This tells the preprocessor that the header file can be found in a predefined directory. This is usually determined by the configuration of the system and contains many system-related and library-related header files, such as `stdio.h`. Often we want to include headers files we have created ourselves for the particular program we are writing. The second variation, using double quotes (`" "`) around the filename, instructs the preprocessor that the header file can be found in the same directory as the C source file.

Notice that none of the preprocessor macros ends with a semicolon. Since `#define` and `#include` are preprocessor directives and not C statements, they are not required to be terminated by semicolons.

11.5.4 Input and Output

We close this chapter by pointing out how to perform input and output from within a C program. We describe these functions at a high level now and save the details for Chapter 18, when we have introduced enough background material to understand C I/O down to a low level. Since all useful programs perform some form of I/O, learning the I/O capabilities of C is an important first step. In C, I/O is performed by library functions, similar to the IN and OUT trap routines provided by the LC-3 system software.

Three lines of the example program perform output using the C library function `printf` or *print formatted* (refer to lines 24, 25, and 30). The function `printf` performs output to the standard output device, which is typically the monitor. It requires a *format string* in which we provide two things: (1) text to print out and (2) specifications on how to print out values within that text. For example, the statement

```
printf("43 is a prime number.");
```

prints out the following text to the output device.

```
43 is a prime number.
```

In addition to text, it is often useful to print out values generated within a program. Specifications within the format string indicate how we want these values to be printed out. Let's examine a few examples.

```
printf("%d is a prime number.", 43);
```

This first example contains the format specification `%d` in its format string. It causes the value listed after the format string to be embedded in the output as a decimal number in place of the `%d`. The resulting output would be

```
43 is a prime number.
```

The following examples show other variants of `printf`.

```
printf("43 plus 59 in decimal is %d.", 43 + 59);
printf("43 plus 59 in hexadecimal is %x.", 43 + 59);
printf("43 plus 59 as a character is %c.", 43 + 59);
```

In the first `printf`, the format specification causes the value 102 to be embedded in the text because the result of "43 + 59" is printed as a decimal number. In the next example, the format specification `%x` causes 66 (because 102 equals x66) to be embedded in the text. Similarly, in the third example, the format specification of `%c` displays the value interpreted as an ASCII character which, in this case, would be lowercase `f`. The output of this statement would be

```
43 plus 59 as a character is f.
```

What is important to notice is that the binary pattern being supplied to `printf` after the format string is the same for all three statements. Here, `printf` interprets the binary pattern 0110 0110 (decimal 102) first as a decimal number, then as a hexadecimal number, and finally as an ASCII character. The C output function `printf` converts the bit pattern into the proper sequence of ASCII characters based on the format sepecifications we provide it. Table D.6 contains a list of all the format specifications that can be used with `printf`. All format specifications begin with the percent sign, `%`.

The final example demonstrates a very common and powerful use of `printf`.

```
printf("The wind speed is %d km/hr.", windSpeed);
```

Here, a value generated during the execution of the program, in this case the variable `windSpeed`, is output as a decimal number. The value displayed depends on the value of `windSpeed` when this line of code is executed. So if `windSpeed` equals 2 when the statement containing `printf` is executed, the following output would result:

```
The wind speed is 2 km/hr.
```

If you were to execute a program containing the five preceding `printf` statements in these examples, you would notice that they would all be displayed on one single line without any line breaks. If we want line breaks to appear, we must put them explicitly within the format string in the places we want them to occur. New lines, tabs, and other special characters require the use of a special backslash (`\`) sequence. For example, to print a new line character (and thus cause a line break), we use the special sequence `\n`. We can rewrite the preceding `printf` statements as such:

```
printf("%d is a prime number.\n", 43);
printf("43 plus 59 in decimal is %d.\n", 43 + 59);
printf("43 plus 59 in hexadecimal is %x.\n", 43 + 59);
printf("43 plus 59 as a character is %c.\n", 43 + 59);
printf("The wind speed is %d km/hr.\n", windSpeed);
```

Notice that each format string ends by printing the new line character `\n`, so therefore each subsequent `printf` will begin on a new line. Table D.1 contains a list of other special characters that are useful when generating output. The output generated by these five statements would look as follows:

```
43 is a prime number.
43 plus 59 in decimal is 102.
43 plus 59 in hexadecimal is 66.
43 plus 59 as a character is f.
The wind speed is 2 km/hr.
```

In our sample program in Figure 11.3, `printf` appears three times in the source. The first two versions display only text and no values (thus, they have no format specifications). The third version prints out the value of variable `counter`. Generally speaking, we can display as many values as we like within a single `printf`. The number of format specifications (for example, `%d`) must equal the number of values that follow the format string.

Question: What happens if we replace the third `printf` in the example program with the following? The expression "`startPoint - counter`" calculates the value of `startPoint` minus the value of `counter`.

```
printf("%d %d\n", counter, startPoint - counter);
```

Having dealt with output, we now turn to the corresponding input function `scanf`. The function `scanf` performs input from the standard input device, which is typically the keyboard. It requires a format string (similar to the one required by `printf`) and a list of variables into which the values retrieved from the keyboard should be stored. The function `scanf` reads input from the keyboard and, according to the conversion characters in the format string, converts the input and assigns the converted values to the variables listed. Let's look at an example.

In the example program in Figure 11.3, we use `scanf` to read in a single decimal number using the format specification `%d`. Recall from our discussion on LC-3 keyboard input, the value received via the keyboard is in ASCII. The format specification `%d` informs `scanf` to expect a sequence of *numeric* ASCII keystrokes (i.e., the digits 0 to 9). This sequence is interpreted as a decimal number and converted into an integer. The resulting binary pattern will be stored in the variable called startPoint. The function scanf automatically performs type conversions (in this case, from ASCII to integer) for us! The format specification %d is one of several that can be used with scanf. Table D.5 lists them all. There are specifications to read in a single character, a floating point value, an integer expressed as a hexadecimal value, and so forth.

A very important thing to remember about scanf is that variables that are being modified by the scanf function (for example, startPoint) must be preceded by an & character. This may seem a bit mysterious, but we will discuss the reason for this strange notation in Chapter 16.

Following are several more examples of scanf.

```
/* Reads in a character and stores it in nextChar */
scanf("%c", &nextChar);

/* Reads in a floating point number into radius */
scanf("%f", &radius);

/* Reads two decimal numbers into length and width */
scanf("%d %d", &length, &width);
```

11.6 Summary

In this chapter, we have introduced some key characteristics of high-level programming languages and provided an initial exposure to the C programming language. We conclude this chapter with a listing of the major topics we've covered.

- **High-Level Programming Languages.** High-level languages aim to make the programming process easier by connecting real-world objects with the low-level concepts, such as bits and operations on bits, that a computer natively deals with. Because computers can only execute machine code, programs in high-level languages must be translated using the process of compilation or interpretation into machine code.
- **The C Programming Language.** The C programming language is an ideal language for a bottom-up exposure to computing because of its low-level nature and because of its root influence on current popular programming languages. The C compilation process involves a preprocessor, a compiler, and a linker.
- **Our First C Program.** We provided a very simple program to illustrate several basic features of C programs. Comments, indentation, and style can help convey the meaning of a program to someone trying to understand the code. Many C programs use the preprocessor macros #define and #include. The execution of a C program begins at the function main, which itself consists of variable declarations and statements. Finally, I/O in C can be accomplished using the library functions printf and scanf.

Exercises

11.1 Describe some problems or inconveniences you found when programming in lower-level languages.

11.2 How do higher-level languages help reduce the tedium of programming in lower-level languages?

11.3 What are some disadvantages to programming in a higher-level language?

11.4 Compare and contrast the execution process of an interpreter versus the execution process of a compiled binary. What implication does interpretation have on performance?

11.5 A language is portable if its code can run on different computer systems, say with different ISAs. What makes interpreted languages more portable than compiled languages?

11.6 The UNIX command line shell is an interpreter. Why can't it be a compiler?

11.7 Is the LC-3 simulator a compiler or an interpreter?

11.8 Another advantage of compilation over interpretation is that a compiler can optimize code more thoroughly. Since a compiler can examine the entire program when generating machine code, it can reduce the amount of computation by analyzing what the program is trying to do.

The following algorithm performs some very straightforward arithmetic based on values typed at the keyboard. It outputs a single result.

```
1. Get W from the keyboard
2. $X \leftarrow W + W$
3. $Y \leftarrow X + X$
4. $Z \leftarrow Y + Y$
5. Print Z to the screen
```

a. An interpreter would execute the program statement by statement. In total, five statements would execute. At least how many arithmetic operations would the interpreter perform on behalf of this program? State what the operations would be.

b. A compiler would analyze the entire program before generating machine code, and possibly optimize the code. If the underlying ISA were capable of all arithmetic operations (i.e., addition, subtraction, multiplication, division), at least how many operations would be needed to carry out this program? State what the operations would be.

11.9 For this question refer to Figure 11.2.

a. Describe the input to the C preprocessor.
b. Describe the input to the C compiler.
c. Describe the input to the linker.

11.10 What happens if we changed the second-to-last line of the program in Figure 11.3 from `printf("%d\n", counter);` to:

a. `printf("%c\n", counter + 'A');`
b. `printf("%d\n%d\n", counter, startPoint + counter);`
c. `printf("%x\n", counter);`

11.11 The function `scanf` reads in a character from the keyboard and the function `printf` prints it out. What do the following two statements accomplish?

```
scanf("%c", &nextChar);
printf("%d\n", nextChar);
```

11.12 The following lines of C code appear in a program. What will be the output of each `printf` statement?

```
#define LETTER '1'
#define ZERO    0
#define NUMBER 123

printf("%c", 'a');

printf("x%x", 12288);

printf("$%d.%c%d\n", NUMBER, LETTER, ZERO);
```

11.13 Describe a program (at this point we do not expect you to be able to write working C code) that reads a decimal number from the keyboard and prints out its hexadecimal equivalent.