CHAPTER

# 11 Introduction to C/C++ Programming

## 11.1 Our Objective

Congratulations, and welcome to the second half of the book! We've just completed an introduction to the basic underlying structure of modern computing devices. Be it on a smartphone or in a smart car, the underlying mechanisms for processing digital data are very much the same. With these foundational concepts solidly in place, we are now well prepared to explore 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 and C++ programming languages. At every step, with every new high-level concept, we will be able to make a connection to the lower levels of the digital system. Our perspective is that nothing should be left abstract or mysterious. If we can deconstruct new concepts into operations carried out by the underlying layers, we become more proficient developers and engineers who are better at building new hardware and software systems.

Let's begin with a quick overview of the first half. In the first ten chapters, we described the LC-3, a simple computing architecture that has all the important characteristics of a more complex, real system. A basic idea behind the LC-3 (and indeed, behind all modern digital systems) is that simple elements are systematically interconnected to form elements with more complex capabilities. 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 construction principle that enables us to build digital computing systems that are, as a whole, mind-bogglingly complex.

After describing the hardware of the LC-3, we programmed 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. Systematic decomposition and subroutines are prevalent design paradigms for software. We will continue to see examples of these concepts before we are through.

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 vehicles for doing so are the C and C++ programming languages.

It is not our objective to provide complete coverage of C or C++. Both C and C++ are vast programming languages (particularly C++) that have evolved over decades to support the building of large-scale software. Billions of lines of code have been written in these languages. Some of the most widely used apps, cloud services, and devices are built using C or C++. These languages contain many features that enable stable, maintainable, scalable software development by teams of developers, but it is not necessary to cover many of these features for a first exposure to these languages.

Our objective will be to explore the **core** elements of C initially, and later C++. These core elements are the entry points of these languages, and they will enable us to write interesting and challenging programs consisting of 100 or so lines of code contained in a single file.

We must start somewhere. So in this chapter, we make the transition from programming in low-level assembly language to high-level language programming in C. The C and C++ programming languages have a fascinating history. We'll explore how and why these languages came about and why they are so widely popular even nearly 50 years after their introduction. We'll dive headfirst into C by examining a simple example program. Let us begin!

## 11.2 Bridging the Gap

It is almost always the software that enables a digital system to do its thing. The hardware provides the general fabric, and the software provides the specific capabilities. In devices we might not consider to be software-driven, such as a smart speaker or Bluetooth headphones, there is a large body of software embedded in the device to implement its features. It's the software that animates the device.

Enabling these sophisticated capabilities requires larger and more complex bodies of software. A typical smartphone app might consist of hundreds of thousands of lines of code, a web browser several million, and the code to power a modern automobile (nonautonomous) might range in the hundreds of millions of lines. Over time, from generation to generation, the software needed to power these devices has grown in complexity. As the underlying hardware becomes faster and our demand for additional capabilities grows, we expect the amount of software needed to continue to grow as well. Examine Figure 11.1. It provides a graphical view of the sizes of software systems for various applications.

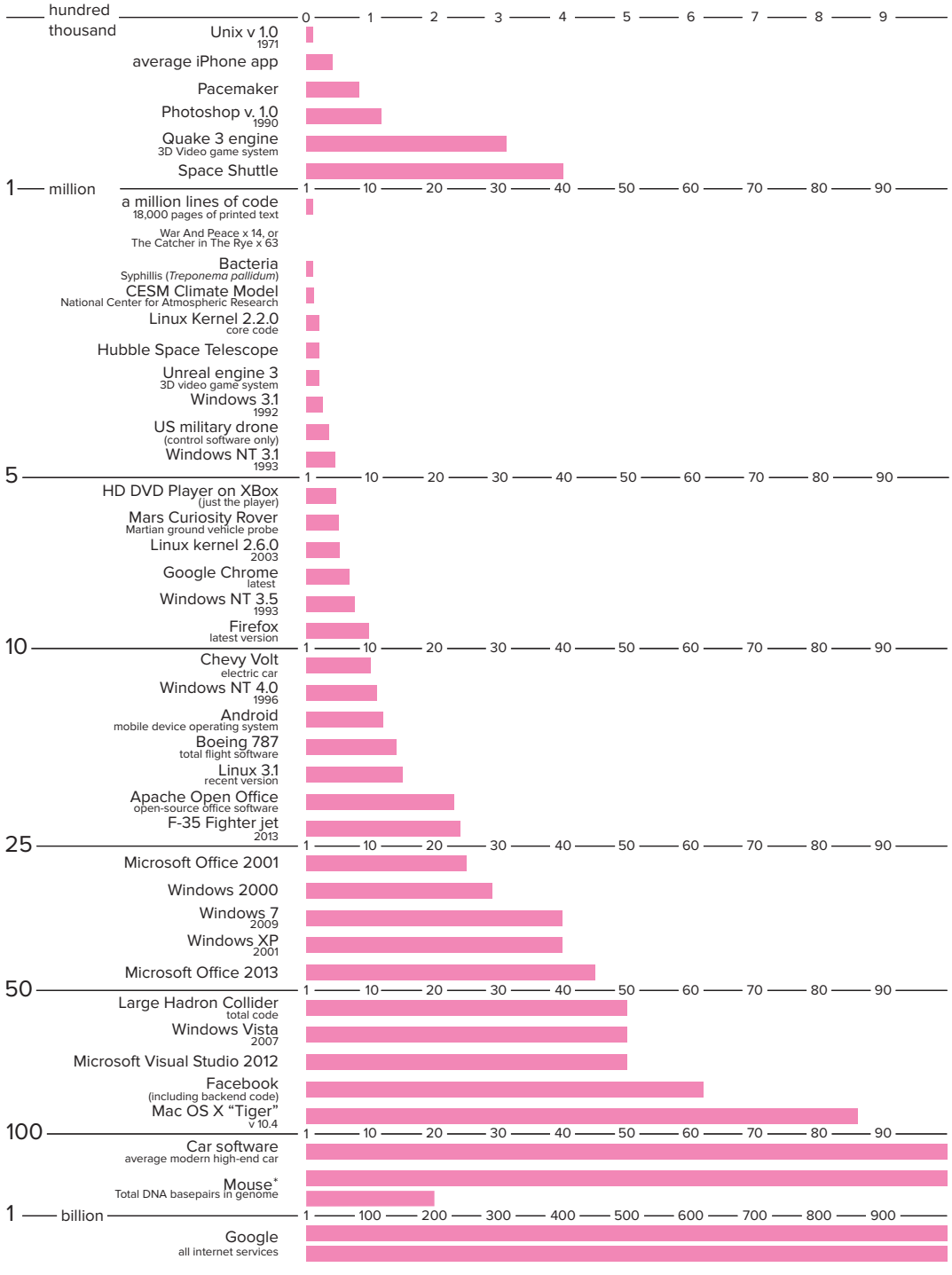


**Figure 11.1** The size in estimated lines of code of various computing applications. Certain applications require large bodies of code. And, looking at specific items, such as the Windows operating system, we can see that their complexity grows with each new version. Codebases: Millions of Line of Code, concept and design by David McCandless, research by Pearl Doughty-White and Miriam Quick. Informationisbeautiful.net, September 24, 2015. David McCandless@informationisbeautiful.net. Used with permission.

Programming languages such as C and C++ were created for one primary purpose: to increase programmer productivity. The most commonly used languages enable teams of programmers to more rapidly and maintainably develop correctly working code. Without effective programming languages and corresponding development environments, our ability to create these incredible devices would be severely diminished. Just as building a 100-story skyscraper or a Boeing 787 aircraft is an engineering feat requiring their own sets of technologies and tools, so are developing and maintaining a web service requiring ten million lines of code.

Clearly, LC-3 assembly language isn't going to be the language of choice for developing most real-world software (nor are the assembly languages for x86 or ARM, which are the two most widespread instruction set architectures today, powering nearly all of our PCs, servers, and mobile devices). But it is the case that whatever the development language, C or C++ or Java or Python, the code is ultimately, and automatically, translated into the instruction set of the underlying hardware (be it ARM or x86 or LC-3). So the conceptual process here is that the programming language enables us humans to express code in a much more human-friendly way and in a manner that can be automatically, and unambiguously, translated into a machine-executable form. Ideally we might wish to express things in a human language, but we know how ambiguous and error-prone that would be. So we choose more precise, machine-oriented languages like C and C++.

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 machinery. The assembler filled some of the *gap* between the algorithm level and the ISA level in the levels of transformation (see Figure 1.6). We would like 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 help manage the values upon which we are computing.** 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 update the counter, we need to remember the spot where we last stored it, load it, operate, and store it back in that location. 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 gives the value a meaningful symbolic name and the programming language takes care of allocating storage for it and performing the appropriate data movement operations whenever we refer to it in our code. Since most programs contain lots of values, having such a convenient way to handle these values is critical to enhancing programmer productivity.

- **High-level languages provide a human-friendly way to express computation.** 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 in C, we would write:

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

Here's 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 would use the following *control structure*:

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

This expressiveness enables us to achieve more with fewer lines than with assembly code. And this dramatically enhances our productivity as programmers.

- **High-level languages provide an abstraction of the underlying hardware.** High-level languages provide a uniform programmer interface independent of underlying hardware. And this provides two distinct advantages. First, our code becomes *portable*. C or C++ or Java code can be easily and efficiently targeted for a variety of different devices, independent of their hardware. Code written directly in assembly language takes advantage of the specifics of a particular hardware system and isn't as easy to run on different hardware. It's easy to understand why portability is important: An app developer who has written an app for Android will want to quickly port the app to iOS to take advantage of the large base of Apple devices.

  The second advantage is that we use operations that aren't natively supported by the hardware. For example, in the LC-3, there is no single 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 maintainability.** Since common control structures are expressed using simple, English-like statements, the program itself becomes easier to read and therefore easier for others to modify and fix. 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. No doubt you've had to get reacquainted with your own LC-3 assembly code after spending even a couple of hours away from it. It's no fun! 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 stricter 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 programmers to get their programs 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 design of 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 is a *compiler*. The compilation process completely translates the high-level language program into machine language, or a form very close to machine language. An executable image is an output of the compilation process. It can directly execute on the hardware. Keep in mind that both interpreters and compilers are themselves pieces of software running on some device.

## 11.3.1 Interpretation

If you've ever run Python code, you've run an interpreter. 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 in an isolated sandbox.

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.

### 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 translated version. A program need only be compiled once, and it can be executed many times. Many programming languages, including C, C++, and their variants are typically compiled. 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 (although some sophisticated compilers do execute the program in order to better optimize its performance), but instead 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 are usually easier. Interpreters often permit the execution of a program one section (a 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 compilation, the programmer can produce code that executes more quickly and uses memory more efficiently. Since compilation produces more efficient code, most production software tends to be programmed in compiled languages.

## 11.4 The C/C++ Programming Languages

### 11.4.1 The Origins of C and C++

Let's cover a bit of history. The C programming language was developed in the early 1970s by Dennis Ritchie at Bell Laboratories. Ritchie was part of a group at Bell Labs developing the Unix operating system. C was created to meet the need for a programming language that was powerful, yet compact enough for a simple compiler to generate efficient code. The computer systems being used by Ritchie and his team had very little memory, so the code generated by the compiler had to be small, and all with a compiler that was small, too! It was this pragmatic bent that gave rise to the popularity of C and propelled it to become one of the most popular production programming languages of its time.

It's remarkable that anything in the computing industry can still be useful 50 years after its introduction. (The authors of this textbook are an exception to this, of course.) The C programming language is going strong, and it is still among the top languages for software development.

In 1979, shortly after C was introduced, a computer scientist named Bjarne Stroustrup, also at Bell Labs, working alongside Ritchie and his colleagues, introduced a set of improvements to the C language that helped programmers to better organize their C code, particularly for large and complex programs. Stroustrup introduced the notion of *classes* to C, and he thereby created the C++ programming language. C++ uses the basic types, syntax, and constructs of C, so it is as efficient as C, but it also has additional features, such as classes and templates, that are essential for creating the building blocks for large-scale coding projects. C has enabled us to create computing applications consisting of hundreds of thousands of lines of code, and C++ has enabled us to create applications with hundreds of millions of lines.

### 11.4.2 How We Will Approach C and C++

Because of their popularity and close-to-the-metal, low-level approach, C and C++ are the ideal languages for our bottom-up exploration. We'll spend the bulk of our time with the C language in Chapters 11 through 19, and we will introduce some core C++ concepts in Chapter 20. Because C++ is based on C, the C-specific material we cover will help in our understanding of C++.

Both C and C++ are highly developed, heavily evolved languages that support large-scale programming. We will not be covering all aspects of C or C++ in this textbook; instead, we will cover the common core subset that will enable you to write code on the order of hundreds of lines by yourself in a single source code file. Our objective is to give you a grounding in digital systems, hardware, and software. This grounding will be useful in your subsequent courses in data structures and software engineering for more complex software projects.

All of the examples and specific details of C presented in this text are based on a standard version of C called ANSI C, or C18. As with many programming languages, several variants of C have been introduced throughout the years. The American National Standards Institute (ANSI) approves "an unambiguous and machine-independent definition of the language C" in order to standardize the language, promoting portability of C code across different systems and compilers. The most recently adopted version of ANSI C is from 2018 and is thus referred to as C18. Likewise, we'll use ISO C++, often called Standard C++, in our examples involving C++ code.

Many of the new C and 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 or ARM. 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*.

### 11.4.3 The Compilation Process

Both C and C++ are compiled languages. The C or C++ compiler follows the typical mode of translation from a source program to an *executable image*. An executable image (refer to Section 7.4.1 for a refresher on this concept) is a machine language representation of a program that is ready to be loaded into memory and executed. The compilation mechanism involves several distinct components, notably the preprocessor, the compiler itself, and the linker. Figure 11.2 shows the overall compilation process for C. Let's briefly take a look at each of the major components.

#### 11.4.3.1 The Preprocessor

As its name implies, the preprocessor "preprocesses" the source program before handing it off to the compiler. The preprocessor scans through the source files (the source files contain the actual C program) looking for and acting upon preprocessor directives. These directives are similar to pseudo-ops in LC-3 assembly language. They instruct the preprocessor to transform the 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 later chapters. All preprocessor directives begin with a pound sign, #, as the first character. All useful C and C++ programs rely on the preprocessor in some way.

#### 11.4.3.2 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, tries 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

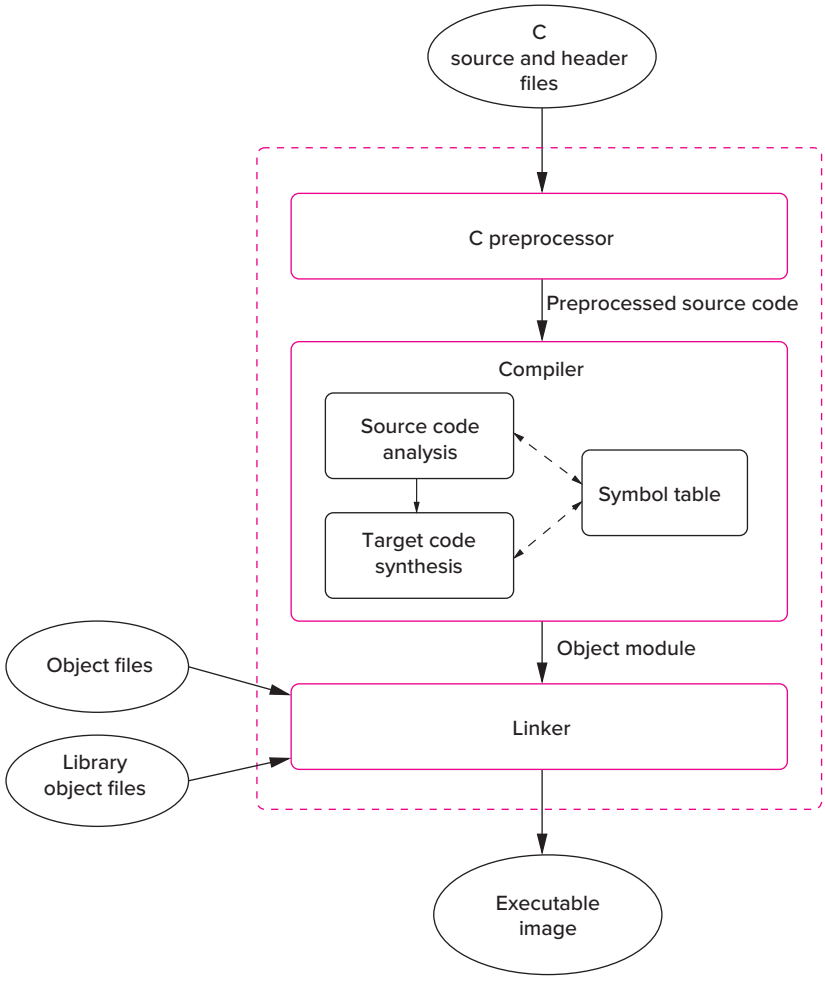


**Figure 11.2** The dotted box indicates the overall compilation process—the preprocessor, the compiler, and the linker. The entire process is called compilation even though the compiler is only one part of it. The inputs are C source and header files and various object files. The output is an executable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

programmer has used in the program. The 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 compiler's symbol table in more detail in Chapter 8.

#### 11.4.3.3 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 or launch an app on your phone, for example, you are instructing the operating system to load the app's executable image 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 8.4.1 in the context of the LC-3. Usually, library files are stored in a particular place, depending on the computer system.

### 11.4.4 Software Development Environments

A simple, effective development flow for our purposes is to write our C or C++ source code using a text editor such as Notepad, TextEdit, or vim. The source code, once complete, is compiled using a C or C++ compiler using a command line interface to invoke the full compilation process. The resulting executable then can be executed also directly from the command line interface. For the short programs we develop here, this development flow will take us quite far.

For most real-world software development, an integrated development environment (or IDE) is often a better choice. An IDE combines several often-used software development tools into a single framework. An IDE will include a text editor with which we can create and edit our source code, along with a compiler, and also a debugger that we can use to more easily debug our code (we'll take a look at debuggers in Chapter 15). With an IDE, since the tools are more tightly coupled, a programmer can more rapidly edit, compile, and debug throughout the development process.

## 11.5 A Simple Example in C

Let's begin by diving headfirst into a simple C example. Figure 11.3 provides the source code. The program prompts the user to type in a number and then counts down from that number to 0.

You can download this example from the textbook website and compile it using your choice of development environment (be sure to use ANSI C to avoid compilation issues). No need to worry about what each line of code does for now. Rather, we'll walk through the high-level organization of this source code and touch upon some important details, with the objective of helping you get started with writing your own code. We'll focus on four aspects: the function `main`, the programming style, the preprocessor directives, and the use of input/output.

### 11.5.1 The Function `main`

The function `main` begins at the line containing `int main(void)` (line 17) and ends at the closing brace on line 31. These lines of the source code constitute a *function definition* for the function named `main`. What were called subroutines in

```
1  //
2  //
3  // 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 countdown
17 int main(void)
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** Our first C program. It prompts the user to enter a positive number and then counts down to 0.

LC-3 assembly language programming are referred to as *functions* in C. Functions are a vital 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 complete C program requires a function `main`. Note that in ANSI C, `main` must be declared to return an integer value, and therefore the keyword `int` precedes the name of the function `main`. As for the `(void)`, more on that later.

In this example, the code for function `main` (i.e., the code in between the curly braces at line 18 and line 31) can be broken down into two components. The first component contains the variable *declarations* for the function, lines 20 and 21. Two variables, one called `counter` and the other `startPoint`, are created for use within the function `main`. Variables are a useful and elementary feature provided by all high-level programming languages. They give us a way to symbolically name the values within a program rather than referring to them by the memory location or register in which they are stored.

The second component contains the *statements* of the function, lines 24 through 31. These statements express the actions that will be performed when the function is executed. For all C programs, execution starts at the first statement of `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 parse the program down unambiguously into its constituents.

### 11.5.2 Formatting, Comments, and Style

C is a free-format language. That is, white space (spaces and tabs and line breaks) 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 or 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 (Figure 11.3), 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 proceed through the end of the line. Notice that this example program contains several lines of comments and also source lines with comments at the end of them. Comments in C++ can also begin with the sequence `//` and extend to the end of the line, and this is one of the many ways in which the two languages are similar.

Every programming language provides a way for the programmer to express comments. These comments enable programmers to describe in human terms what their code does. Proper commenting of code is an important part of the software development 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 over-commenting can be detrimental because 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 preprocessor in Section 11.4.3. Recall that it transforms the original source program before it is handed off to the compiler. Our simple example contains two commonly used preprocessor directives: `#define` and `#include`. The C and C++ examples in this book rely only on these two directives. The `#define` directive is a simple yet powerful directive that instructs the preprocessor to replace occurrences of any text that matches X with text Y. That is, the 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 `PRICE_OF_FUEL` and the preprocessor would handle the actual substitution throughout the code 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 C and C++ programming style is to use uppercase for the name being substituted.

The other directive we'll encounter is the `#include` directive. It instructs the preprocessor literally to insert another source file into the code at the point where the `#include` appears. 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 and C++ languages, you will understand how *header files* can be used to hold `#defines` and declarations that are useful among multiple source files.

We'll often encounter the following example: `#include <stdio.h>` in our C programs. All C programs that perform typical input and output must include C's input/output library's header file `stdio.h`. This file defines some relevant information about the I/O functions in the 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 system directory. This is usually determined by the configuration of the system, and it contains many common system-related header files, such as `stdio.h`. Often we want to include header files we have created ourselves for the particular program we are writing. The second variation, using double quotes (`" "`) around the filename, tells the preprocessor that the header file can be found in the same directory as the C source file or in some other directory known to the compiler.

Notice that none of the preprocessor macros end 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 describing how to perform input and output from within a C program. In C, I/O is accomplished through a set of I/O functions. 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, getting familiar with the I/O capabilities of C is an important first step.

The C I/O functions are similar to the IN and OUT trap routines provided by the LC-3 system software. Three lines of the example program in Figure 11.3 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 display device. It requires a *format string* in which we provide two things: (1) text to print out and (2) some specifications on how to print out program 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 the code. 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 the same `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 example above, 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 that, 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 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) as a decimal number in the first example, as a hexadecimal number in the second example, and as an ASCII character in the third. The C output function `printf` converts the bit pattern into the proper sequence of ASCII characters based on the format specifications we provide it. 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` were to equal 2 when the statement containing `printf` is executed, the following output would result:

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

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`. Consider the following statements:

```
printf("%d is a prime number.\n", 43);
printf("43 plus 59 in decimal is %d.\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 each subsequent `printf` will begin on a new line. The output generated by these five statements would look as follows:

```
43 is a prime number.
43 plus 59 in decimal is 102.
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 (e.g., `%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);
```

Let's turn our attention to the 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 input 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.

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 that 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`. There are specifications to read in a single character, a floating point value, an integer expressed as a hexadecimal value, and so forth.

One important point to note about `scanf` : Variables that are being modified by the `scanf` function (e.g., `startPoint`) must be preceded by the `&` (ampersand) character. We will discuss why this is required when we discuss pointers 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, C and C++ in particular, and provided an initial exposure to the C programming language. We covered the following major topics:

- **High-Level Programming Languages.** High-level languages aim to make the programming process easier by providing human-friendly abstractions for the bits and memory that a digital system natively operates upon. Because computers can only execute machine code, programs in high-level languages must be translated using the process of compilation or interpretation into a form native to the hardware.

- **The C and C++ Programming Languages.** C and C++ are ideal languages for a bottom-up exposure to computing because of their low-level, close-to-the-metal nature. They are also two of the most popular programming languages in use today. The C/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 encountered 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 with the execution process of a compiled binary. What effect does interpretation have on performance?

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

**11.6** A command line interface 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·W + W
3. Y·X + X
4. Z·Y + Y
5. Print Z to the screen

*a.* An interpreter would execute the program statement by statement. In total, five statements would execute. 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.

*b.* A compiler would analyze the entire program before generating machine code, and it would 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 exercise, 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 change 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.