

Chapter 1

What Is a Program?

In This Chapter

- ▶ Understanding programs
 - ▶ Writing your first “program”
 - ▶ Looking at computer languages
-

In this chapter, you get a handle on what a program is and what it means to write a program. You get to practice on a Human Computer and see some program snippets written for a real computer. Finally, you get a look at your first code snippet written in C++.

Up until now, all of the programs running on your computer were written by someone else. Very soon now, that won't be true anymore. You'll be joining the ranks of the few, the proud: the programmers.

How Does My Son Differ from a Computer?

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

In this respect, a computer is almost the exact opposite of a human: Humans respond intuitively. When I was learning a second language, I found that it wasn't enough to understand what was being said — it's just as important and considerably more difficult to understand what was left unsaid. This is information that the speaker shares with the listener through common experience or education — things that don't need to be said.

For example, I say things to my son such as, “Wash the dishes” (for all the good it does me). Such instructions seem clear enough, but the vast majority of the information contained in that sentence is implied and unspoken.

Let’s assume that my son knows what dishes are and that dirty dishes are normally in the sink. But what about knives and forks? After all, I only said dishes, I didn’t say anything about eating utensils, and don’t even get me started on glassware. And did I mean wash them manually, or is it okay to load them up into the dishwasher to be washed, rinsed, and dried automatically?

But the fact is, “Wash the dishes” is sufficient instruction for my son. He can deconstruct that sentence and combine it with information that we both share, including an extensive working knowledge of dirty dishes, to come up with a meaningful understanding of what I want him to do — whether he does it or not is a different story. I would guess that he can perform all the mental gymnastics necessary to understand that sentence in about the same amount of time that it takes me to say it — about 1 to 2 seconds.

A computer can’t make head nor tail out of a statement as vague as “wash the dishes.” You have to tell the computer exactly what to do with each different type of dish, specify that silverware is included in the task, and state how to wash a fork, versus a spoon, versus a cup. When does the program stop washing a dish (that is, how does it know when a dish is clean)? When does it stop washing (that is, how does it know when the task is finished)?

My son has gobs of memory — it isn’t clear exactly how much memory a normal human has, but it’s boatloads. Unfortunately, human memory is fuzzy. For example, witnesses to crimes are notoriously bad at recalling details, even a short time after the event. Two witnesses to the same event often disagree radically on what transpired.

Computers also have gobs of memory, and that’s very good. Once a fact is stored, a computer can retrieve that fact as often as you like without changing the fact. As expensive as memory was back in the early 1980s, the original IBM PC had only 16K (that’s 16 thousand bytes). This could be expanded to a whopping 64K. Compare this with the 2GB to 6GB of main storage available in most computers today (a gigabyte, 1GB, is *one billion bytes*).



As expensive as memory was in the early days of personal computing, the IBM PC included extra memory chips and decoding hardware to detect a memory failure. If a memory chip went bad, this circuitry was sure to find the problem and report it before the program went haywire. This so-called *parity memory* was discontinued after only a few years; as far as I know, it’s unavailable today except in specific applications where extreme reliability is required — because the memory boards almost never fail.

On the other hand, humans are very good at certain types of processing that computers do poorly, if at all. For example, humans are very good at pulling the meaning out of a sentence garbled by large amounts of background noise. By contrast, digital cellphones (which are at least as much computer as phone) have the infuriatingly bad habit of just going silent whenever the noise level gets above a built-in threshold.

The remainder of this chapter looks at instructions that come a little closer to telling the computer to “wash the dishes” (or some equally fuzzy task).

Programming a “Human Computer”

Before I dive into showing you how to write programs for computer consumption, I start by showing you a program to guide human behavior so that you can get a better look at what you’re up against. Writing a program to guide a human is much easier than writing programs for computer hardware. That’s because we human beings have a lot of experience in dealing with each other — which gives us some familiarity with (and understanding of) humans and how they work. (I’ll assume that much, anyway.) We also share a common human language; no need to translate everything into ones and zeroes. But assume that the human computer in this thought experiment takes every instruction quite literally — so the program will have to be very specific.

The problem I’ve chosen is to instruct our human computer how to change a flat tire.

Creating the algorithm

The instructions for changing a flat tire are straightforward and go something like the following:

1. Raise the car.
2. Remove the lug nuts that affix the faulty tire to the car.
3. Remove the tire.
4. Mount the new tire.
5. Install the lug nuts.
6. Lower the car.

Okay, even these everyday terms are potentially fuzzy: Technically the lug nuts hold the *wheel*, not the tire, on the car. To keep it simple, assume that the terms “wheel” and “tire” are synonymous, and that the computer understands them as the same thing.

As detailed as these instructions might seem to be, they still don’t make up a program. Such a set of instructions is called an *algorithm* — a description, usually at a high level of abstraction, of the steps to be performed. An algorithm is detailed but general. I could use this algorithm to repair any flat tire I’ve experienced or ever will experience. But an algorithm does not contain sufficient detail to allow even our intentionally obtuse human computer to perform the task.

Setting the tire-changing language

Before we can write a program, we need a language that we can all agree on. For the remainder of this book, that language is C++, but for this example, I use an imaginary language: TCL (Tire-Changing Language). I have specifically adapted TCL to the problem of changing tires.

TCL includes a few nouns common in the tire-changing world:

- ✓ car
- ✓ tire
- ✓ nut
- ✓ jack
- ✓ toolbox
- ✓ spare tire
- ✓ wrench

TCL also includes the following verbs:

- ✓ grab
- ✓ move
- ✓ release
- ✓ turn

Finally, the TCL-executing processor will need the ability to count and to make simple decisions. Finally, the TCL processor knows directions like up vs. down, left vs. right and clockwise vs. counter-clockwise.

These words in TCL are all that the tire-changing robot (the imaginary human computer) understands. Any other command that's not part of Tire-Changing Language generates a blank stare of incomprehension from the human tire-changing processor.

Constructing the program

Now it's time to convert the algorithm, written in everyday English, into a program written in Tire-Changing Language. It's not as easy as it might seem. Take, for example, the phrase, "Remove the lug nuts." Actually, quite a bit is left unstated in that sentence. The word *remove* is not in the processor's vocabulary. In addition, a *wrench* (which is a word the computer knows) is never mentioned in that phrase, though we all know that a wrench must be involved. We can't assume that the computer knows what we know.

(If you haven't changed a flat tire — and didn't know that a wrench is required to remove lug nuts, or what a lug nut is — then just play along for now. You'll figure it out.)

The following steps *implement* the phrase "Remove a lug nut" using only the verbs and nouns contained in Tire-Changing Language:

1. Grab wrench.
2. Move wrench to lug nut.
3. Turn wrench counterclockwise five times.
4. Move wrench to toolbox.
5. Release wrench.

At this point, consider these aspects of the *syntax* (required word order) implied in this example of Tire-Changing Language:

- ✓ Every command starts with a single verb.
- ✓ The verb `grab` requires a single noun as its object.
- ✓ The verb `turn` requires a noun, a direction, and a count of the number of turns to make.



Even so, the program snippet should be easy enough to read (after all, this isn't a book about Tire-Changing Language).

You can skate by on this quick look at Tire-Changing Language, but you'll have to learn the grammar of each C++ command. Otherwise it won't work.

The program begins at Step 1 and continues through each step in turn until reaching Step 5. In programming terminology, we say that the program *flows* from Step 1 through Step 5. Of course, the program's not going anywhere — the processor is doing all the work — but *program flow* is a common term for this smooth execution of steps.

Even a cursory examination of this program reveals a problem: What if there is no lug nut? I suppose it's harmless to spin the wrench around a bolt with no nut on it, but doing so wastes time and isn't my idea of a good solution. The Tire-Changing Language needs a branching capability that allows the program to take one path or another in response to external conditions. We need an IF statement such as the following:

1. Grab wrench.
2. If lug nut is present
3. {
4. Move wrench to lug nut.
5. Turn wrench counterclockwise five times.
6. }
7. Move wrench to toolbox.
8. Release wrench.

The program starts with Step 1 just as before, and grabs a wrench. In the second step, however, before the program waves the wrench uselessly around an empty bolt, it checks to see if a lug nut is present. If so, flow continues with Steps 3, 4, and 5 as before. If not, however, program flow skips these unnecessary steps and goes straight on to Step 7 to return the wrench to the toolbox.

In computerese, you say that the program executes the logical expression “is lug nut present?” This expression returns either a `true` (yes, the lug nut is present) or a `false` (no, there is no lug nut there).



What I call a step, a programming language would normally call a *statement*. An *expression* is a type of statement that returns a value, such as `1 + 2`, is an expression. A *logical expression* is an expression that returns a `true` or `false` value; for example, the value of “Is the author of this book handsome?” is `true`.



The braces `{ }` in Tire-Changing Language are necessary to tell the program which steps are to be skipped if the condition is not `true`. Steps 4 and 5 are executed only if the condition is `true`.

I realize that there’s no need to grab a wrench if there’s no lug nut to remove, but work with me here.

This improved program still has a problem: How do you know that five turns of the wrench will be sufficient to remove the lug nut? It most certainly will not be enough for most of the tires with which I am familiar. You could increase the number of turns to something that seems likely to be more than enough, say, 25 turns. If the lug nut comes loose after the 20th turn, for example, the wrench will turn an extra five times. This is a harmless but wasteful solution.

A better approach is to add some type of “loop and test” statement to the Tire-Changing Language:

1. Grab wrench.
2. If lug nut is present
3. {
4. Move wrench to lug nut.
5. While (lug nut attached to car)
6. {
7. Turn wrench counterclockwise one turn.
8. }
9. }
10. Move wrench to toolbox.
11. Release wrench.

Here the program flows from Step 1 through Step 4 just as before. In Step 5, however, the processor must make a decision: Is the lug nut attached? On the first pass, assume that the answer is yes so the processor will execute Step 7 and turn the wrench counter-clockwise one turn. At this point, the program returns

to Step 5 and repeats the test. If the lug nut is still attached, the processor repeats Step 7 before returning to Step 5 again. Eventually, the lug nut will come loose and the condition in Step 5 will return a `false`. At this point, control within the program will pass to Step 9, and the program will continue as before.

This solution is superior to its predecessor: It makes no assumptions about the number of turns required to remove a lug nut. It doesn't waste effort by requiring the processor to turn a lug nut that is no longer attached, nor does it fail by leaving a lug nut only half-removed.

As nice as this solution is, however, it still has a problem: It removes only a single lug nut. Most medium-size cars have five nuts on each wheel. We could repeat Steps 2 through 9 five times, once for each lug nut. However, this doesn't work very well either. Most compact cars have only four lug nuts, and large pickups have up to eight.

The following program expands our grammar to include the ability to loop across lug nuts. This program works irrespective of the number of lug nuts on the wheel:

```
1. Grab wrench.
2. For each lug bolt on wheel
3. {
4.     If lug nut is present
5.     {
6.         Move wrench to lug nut.
7.         While (lug nut attached to car)
8.         {
9.             Turn wrench counterclockwise one turn.
10.        }
11.    }
12. }
13. Move wrench to toolbox.
14. Release wrench.
```

This program begins just as before with the grabbing of a wrench from the toolbox. Beginning with Step 2, however, the program loops through Step 12 for each lug-nut bolt on the wheel.

Notice how Steps 7 through 10 are still repeated for each bolt. This is known as a *nested loop*. Steps 7 through 10 are called the *inner loop*; Steps 2 through 12 are the *outer loop*.

The complete program consists of the addition of similar implementations of each step in the algorithm.

Computer processors

Removing the wheel from a car seems like such a simple task, and yet it takes 11 instructions in a language designed specifically for tire changing just to get the lug nuts off. Once completed, this program is likely to include over 60 or 70 steps with numerous loops. Even more steps are needed if you add logic to check for error conditions like stripped or missing lug nuts.

Think of how many instructions have to be executed just to do something as mundane as moving a window about on the display screen (remember that a typical screen may have 1280 x 1024 or a little over a million pixels or more displayed). Fortunately, though stupid, a computer processor is very fast. For example, the processor that's in your PC can likely execute several billion instructions per second. The instructions in your generic processor don't do very much — it takes several instructions just to move one pixel — but when you can rip through a billion or so at a time, scrolling a mere million pixels becomes child's play.

The computer will not do anything that it hasn't already been programmed for. The creation of a Tire-Changing Language was not enough to replace my flat tire — someone had to write the program instructions to map out, step by step, what the computer will do. And writing a real-world program designed to handle all the special conditions that can arise is not an easy task. Writing an industrial-strength program is probably the most challenging enterprise you can undertake.

So the question becomes: "Why bother?" Because once the computer is properly programmed, it can perform the required function repeatedly, tirelessly, and usually at a greater speed than is possible under human control.

Computer Languages

The Tire-Changing Language isn't a real computer language, of course. Real computers don't have machine instructions like *grab* or *turn*. Worse yet, computers "think" by using a series of ones and zeros. Each internal

command is nothing more than a sequence of binary numbers. Real computers have instructions like 01011101, which might add 1 to a number contained in a special-purpose register. As difficult as programming in TCL might be, programming by writing long strings of numbers is even harder.



The native language of the computer is known as *machine language* and is usually represented as a sequence of numbers written either in binary (base 2) or hexadecimal (base 16). The following represents the first 64 bytes from the Conversion program in Chapter 3.

```
<main+0>: 01010101 10001001 11100101 10000011 11100100 11110000 10000011 11101100
<main+8>: 00100000 11101000 00011010 01000000 00000000 00000000 11000111 01000100
<main+16>: 00100100 00000100 00100100 01110000 01000111 00000000 11000111 00000100
<main+24>: 00100100 10000000 01011111 01000111 00000000 11101000 10100110 10001100
<main+32>: 00000110 00000000 10001101 01000100 00100100 00010100 10001001 01000100
```

Fortunately, no one writes programs in machine language anymore. Very early on, someone figured out that it's much easier for a human to understand ADD 1, REG1 as “add 1 to the value contained in register 1,” rather than 01011101. In the “post-machine-language era,” the programmer wrote her programs in this so-called *assembly language* and then submitted it to a program called an *assembler* that converted each of these instructions into its machine-language equivalent.

The programs that people write are known as *source code* because they are the source of all evil — just kidding — actually it's because they are the source of the program. The ones and zeros that the computer actually executes are called *object code* because they are the object of so much frustration.



The following represents the first few assembler instructions from the Conversion program when compiled to run on an Intel processor executing Windows. This is the same information previously shown in binary form.

```
<main>:      push    ebp
<main+1>:    mov     ebp, esp
<main+3>:    and     esp, 0xffffffff
<main+6>:    sub     esp, 0x20
<main+9>:    call    0x40530c <__main>
<main+14>:   movl    [esp+0x04], 0x477024
<main+22>:   movl    [esp], 0x475f80
<main+29>:   call    0x469fac <operator<<>
<main+34>:   lea     eax, [esp+0x14]
<main+38>:   mov     [esp+0x04], eax
```

This is still not very intelligible, but it's clearly a lot better than just a bunch of ones and zeros. Don't worry — you won't have to write any assembly-language code in this book either.



The computer doesn't actually ever execute the assembly-language instructions. It executes the machine instructions that result from converting the assembly instructions.

High-level languages

Assembly language might be easier to remember than machine language, but there's still a lot of distance between an algorithm like the tire-changing algorithm and a sequence of `MOVE` and `ADD` instructions. In the 1950s, people started devising progressively more expressive languages that could be automatically converted into machine language by a program called a *compiler*. These were called *high-level languages* because they were written at a higher level of abstraction than assembly language.

One of the first of these languages was COBOL (Common Business-Oriented Language). The idea behind COBOL was to allow the programmer to write commands that were as much like English sentences as possible. Suddenly programmers were writing sentences like the following to convert temperature from Celsius to Fahrenheit (believe it or not, this is exactly what the machine and assembly-language snippets shown earlier do):

```
INPUT CELSIUS_TEMP  
SET FAHRENHEIT_TEMP TO CELSIUS_TEMP * 9/5 + 32  
WRITE FAHRENHEIT_TEMP
```

The first line of this program reads a number from the keyboard or a file and stores it into the variable `CELSIUS_TEMP`. The next line multiplies this number by $9/5$ and adds 32 to the result to calculate the equivalent temperature in degrees Fahrenheit. The program stores this result in a variable called `FAHRENHEIT_TEMP`. The last line of the program writes this converted value to the display.

People continued to create different programming languages, each with its own strengths and weaknesses. Some languages, like COBOL, were very wordy but easy to read. Other languages such as database languages or the languages used to create interactive web pages, were designed for very specific jobs. These languages include powerful constructs designed to handle specific problem areas.

The C++ language

C++ (pronounced “C plus plus,” by the way) is a symbolically oriented high-level language. C++ started out life as simply C in the 1970s at Bell Labs. A couple of guys were working on a new idea for an operating system known as Unix (the predecessor to Linux and Mac OS and still used across industry and academia today). The original C language created at Bell Labs in the 1970s was modified slightly and adopted as a worldwide ISO standard in early 1989. C++ was created as an extension to the basic C language mostly by adding the features that I discuss in Parts V and VI of this book.

When I say that C++ is *symbolic*, I mean that it isn’t very wordy; it uses symbols instead of the long words in languages like COBOL. However, C++ is easy to read once you’re accustomed to what the symbols mean. The same Celsius-to-Fahrenheit conversion code shown in COBOL earlier appears as follows in C++:

```
cin >> celsiusTemp;  
fahrenheitTemp = celsiusTemp * 9 / 5 + 32;  
cout << fahrenheitTemp;
```

The first line reads a value into the variable `celsiusTemp`. The subsequent calculation converts this Celsius temperature to Fahrenheit, just as before; the third line outputs the result.

C++ has several other advantages compared with other high-level languages. For one, C++ is universal. There is a C++ compiler for almost every computer in existence.

In addition, C++ is efficient. The more tasks a high-level language tries to do automatically (to make your programming job easier), the less efficient the machine code generated tends to be. That doesn’t make much of a difference for a small program like most of those in this book, but it can make a big difference when manipulating large amounts of data, as when you’re moving pixels around on the screen, or when you want blazing real-time performance. It’s no accident that Unix and Windows are written in C++ and the Macintosh O/S is written in a language very similar to C++.

The goal of the remaining chapters of this book is get you programming in C++. You won’t have to cram every detail of the C++ language into your head, but you’ll end up with enough of it under your belt to write some pretty awesome programs.