CHAPTER 1

Introduction to Programming

"Begin at the beginning, and go on till you come to the end: then stop." This method of telling a story is as good today as it was when the King of Hearts prescribed it to the White Rabbit. In this book, we must begin with you, the reader, since my job is to explain a technical subject to you. It might appear that I'm at a severe disadvantage; after all, I've never met you.

Nevertheless, I can make some pretty good guesses about you. You almost certainly own a computer and know how to use its most common application, word processing. If you use the computer in business, you probably also have an acquaintance with spreadsheets and perhaps some database experience as well. Now you have decided to learn how to program the computer yourself rather than relying completely on programs written by others. On the other hand, you might be a student using this book as a text in an introductory course on programming. In that case, you'll be happy to know that this book isn't written in the dry, overly academic style employed by many textbook writers. I hope that you will enjoy reading it, as my previous readers have.

Whether you are using this book on your own or in school, there are many good reasons to learn how to program. You may have a problem that hasn't been solved by commercial software; you may want a better understanding of how commercial programs function so you can figure out how to get around their shortcomings and peculiarities; or perhaps you're just curious about how computers perform their seemingly magical feats. Whatever the initial reason, I hope you come to appreciate the great creative possibilities opened up by this most ubiquitous of modern inventions.[1]

Before we begin, however, we should agree on definitions for some fundamental words in the computing field. Susan had some incisive observations about the power of words. Here is our exchange on that issue:

> **Susan:** I will read something usually at face value, but often there is much more to it; that is why I don't get it. Then, when I go back and really think about what those words mean, it will make more sense. This book almost needs to be written in ALL CAPS to get the novice to pay closer attention to each and every word.
>
> **Steve:** IMAGINE WRITING A BOOK IN ALL CAPS! THAT WOULD BE VERY DIFFICULT TO READ, DON'T YOU THINK?

Many of the technical words used in this book are in the glossary at the end of the book; it is also very helpful to have a good technical dictionary of computer terms, as well as a good English dictionary.

Of course, you may not be able to remember all of these technical definitions the first time through. If you can't recall the exact meaning of one of these terms, just look up the word or phrase in the index, and it will direct you to the page where the definition is stated.

1. Of course, it's also possible that you already know how to program in another language and are using this book to learn how to do so in C++. If so, you'll have a head start; I hope that you'll learn enough to repay the effort of wading through some material you already know.

Before we continue, let's check in again with Susan. The following is from her first letter to me about the contents of this book:

> **Susan:** I like the one-on-one feel of your text, like you are talking just to me. Now, you did make a few references to how simple some things were which I didn't catch on to, so it kinda made me feel I was not too bright for not seeing how apparently simple those things were...
>
> I think maybe it would have been helpful if you could have stated from the onset of this book just what direction you were taking, at least chapter by chapter. I would have liked to have seen a goal stated or a least a summary of objectives from the beginning. I often would have the feeling I was just suddenly thrown into something as I was reading along. Also (maybe you should call this *C++ for Dummies*, or is that taken already?)[2], you might even *define* what programming is! What a concept! Because it did occur to me that since I have never seen it done, I really don't know what programming *is*! I just know it's something that nerds do.

Susan's wish is my command, so I have provided a list of objectives at the beginning of each chapter after this one. I've also fulfilled her request for a definition of some programming terms.

Definitions

An **algorithm** is a set of precisely defined steps to calculate an answer to a problem or set of problems, and which is guaranteed to arrive at such an answer eventually. As this implies, a set of steps that might never end is *not* an algorithm.

2. As it happens, that title is indeed taken. However, I'm not sure it's been applied appropriately, since the book with that title assumes previous knowledge of C! What that says about C programmers is better left to the imagination.

Programming is the art and science of solving problems by the following procedure:[3]

1. Find or invent a general solution to a class of problems.
2. Express this solution as an algorithm or set of algorithms.
3. Translate the algorithm(s) into terms so simple that a stupid machine like a computer can follow them to calculate the specific answer for any specific problem in the class.

At this point, let's see what Susan had to say about the above definition and my response.

> **Susan:** Very descriptive. How about this definition: Programming is the process of being creative using the tools of science such as incremental problem solving to make a stupid computer do what you want it to. That I understand!
>
> Your definition is just fine. A definition has to be concise and descriptive, and that you have done — and covered all the bases. But you know what is lacking? An example of what it looks like. Maybe just a little statement that really looks bizarre to me, and then say that by the end of the chapter you, the reader, will actually know what this stuff really means! Sort of like a coming attraction type of thing.
>
> **Steve:** I understand the idea of trying to draw the reader into the "game". However, I think that presenting a bunch of apparent gibberish with no warning could frighten readers as easily as it

3. This definition is possibly somewhat misleading since it implies that the development of a program is straightforward and linear, with no revisions required. This is known as the "waterfall model" of programming, since water going over a waterfall follows a preordained course in one direction. However, real-life programming doesn't usually work this way; rather, most programs are written in an incremental process as assumptions are changed and errors are found and corrected.

might intrigue them. I think it's better to delay showing examples until they have some background.

Now let's return to our list of definitions:

Hardware refers to the physical components of a computer, the ones you can touch. Examples include the keyboard, the monitor, the printer.

Software refers to the nonphysical components of a computer, the ones you cannot touch. If you can install it on your hard disk, it's software. Examples include a spreadsheet, a word processor, a database program.

Source code is a program in a form suitable for reading and writing by a human being.

An **executable program** (or an *executable*, for short) is a program in a form suitable for running on a computer.

Object code is a portion of a program in a form suitable for incorporation into an executable program.

Compilation is the process of translating source code into object code. Almost all of the software on your computer was created by this process.

A **compiler** is a program that performs compilation as defined above.

How to Write a Program

Now you have a definition of programming. Unfortunately, however, this doesn't tell you how to write a program. The process of solving a problem by programming in C++ follows these steps:

1. Problem: After discussions between the user and the programmer, the programmer defines the problem precisely.
2. Algorithms: The programmer finds or creates algorithms that will solve the problem.
3. C++: The programmer implements these algorithms as source code in C++.
4. Executable: The programmer runs the C++ compiler, which must already be present on the programmer's machine, to translate the source code into an executable program.
5. Hardware: The user runs the resulting executable program on a computer.

These steps advance from the most abstract to the most concrete, which is perfectly appropriate for an experienced C++ programmer. However, if you're using this book to learn how to program in C++, obviously you're not an experienced C++ programmer, so before you can follow this path to solving a problem you're going to need a fairly thorough grounding in all of these steps.

This description is actually a bit oversimplified, as we'll see in the discussion of *linking* in Chapter 5, "Functional Literacy". For now, let's see what Susan thinks about this issue.

> **Susan:** With all the new concepts and all the new language and terms, it is so hard to know what one thing has to do with the other and where things are supposed to fit into the big picture. Anyway, you have to understand; for someone like me, this is an enormous amount of new material to be introduced to all at once. When you are bombarded with so many new terms and so many abstract concepts, it is a little hard to sort out what is what. Will you have guidelines for each of the steps? Since I know a little about this already, the more I look at the steps, I just know that what is coming is going to be a big deal. For example, take step 1; you have to give the ingredients for properly defining a problem. If something is left out, then everything that follows won't work.

> **Steve:** I hope you won't find it that frustrating, because I explain all of the steps carefully as I do them. Of course, it's possible that I haven't been careful enough, but in that case you can let me know and I'll explain it further.
>
> Unfortunately, it's not possible for me to provide a thorough guide to all of those steps, as that would be a series of books in itself. However, there's a wonderful small book called *How to Solve It* by G. Polya, that you should be able to get at your local library. It was written to help students solve geometry problems, but the techniques are applicable in areas other than geometry. I'm going to recommend that readers of my book read it if they have any trouble with general problem solving.

The steps for solving a problem via programming might sound reasonable in the abstract, but that doesn't mean that you can follow them easily without practice. Assuming that you already have a pretty good idea of what the problem is that you're trying to solve, the algorithms step is likely to be the biggest stumbling block. Therefore, it might be very helpful to go into that step in a bit more detail.

Baby Steps

If we already understand the problem we're going to solve, the next step is to figure out a plan of attack, which we will then break down into small enough steps to be expressed in C++. This is called **stepwise refinement**, since we start out with a "coarse" solution and refine it until the steps are within the capability of the C++ language. For a complex problem, this may take several intermediate steps, but let's start out with a simple example. Say that we want to know how much older one person is than another. We might start with the following general outline:

1. Get two ages from user.

2. Calculate difference of ages.
3. Print the result.

This in turn can be broken down further, as follows:

1. Get two ages from user.
   a. Ask user for first age.
   b. Ask user for second age.
2. Subtract second age from first age.
3. Print result.

This looks okay, except that if the first person is younger than the second one, then the result will be negative. That may be acceptable. If so, we're just about done, since these steps are simple enough for us to translate them into C++ fairly directly. Otherwise, we'll have to modify our program to do something different, depending on which age is higher. For example,

1. Get two ages from user.
   a. Ask user for first age.
   b. Ask user for second age.
2. Compute difference of ages.
   a. If first age is greater than second, subtract second age from first age.
   b. Otherwise, subtract first age from second age.
3. Print result.

You've probably noticed that this is a much more detailed description than would be needed to tell a human being what you want to do. That's because the computer is extremely stupid and literal: it does only what you tell it to do, not what you meant to tell it to do.

Unfortunately, it's very easy to get one of the steps wrong, especially in a complex program. In that case, the computer will do something ridiculous, and you'll have to figure out what you did wrong. This debugging, as it's called, is one of the hardest parts of programming. Actually, it shouldn't be too difficult to understand why that is the case. After all, you're looking for a mistake you've made yourself. If you knew exactly what you were doing, you wouldn't have made the mistake in the first place.

I hope that this brief discussion has made the process of programming a little less mysterious. In the final analysis, it's basically just logical thinking.[4]

On with the Show

Now that you have some idea how programming works, it's time to see exactly how the computer actually performs the steps in a program, which is the topic of Chapter 2, "Hardware Fundamentals".

4. Of course, the word *just* in this sentence is a bit misleading; taking logical thinking for granted is a sure recipe for trouble.