Thinking Like a Programmer

Programming may not be exactly like any activity you've ever done. Basically, you're just giving instructions—but doing so in a logical, systematic manner.

Computers Do Only What You Tell Them

Computers do only what you tell them: This is the most important rule in this book, especially if you are new to programming. By using a computer language, such as C++, Visual Basic, Pascal, or FORTRAN, you give the computer a list of things to do; this is the *program*.

A computer needs information, of course—that's program *data*. But it also needs to know what to do with that data. The instructions that tell it what to do are called program *code*.

Determine What the Program Will Do

So, to get a computer to do anything, it has to be told exactly what to do.

So far, you've probably used a computer by running programs that other people have written for you. To this extent, you've been an *end user*—or *user*, for short.

By writing programs yourself, you'll promote yourself into the next higher echelon of computer people. Now, you'll decide what a program does. You'll make things happen.

But a computer—more so than Dustin Hoffman in *Rain Man*—is the ultimate idiot savant. It can never guess what you want. It can never make independent judgments. It is extremely literal and will carry out precisely what you say, no matter how stupid. Therefore, you have to be careful to say what you mean.

You can't even give the computer a command that might seem relatively clear to a human, such as "Convert a number from Celsius to Fahrenheit for me." Even that's too general. Instead, you have to be more specific, writing down steps such as these:

1. Print the "Enter Celsius temperature:" message.
2. Get a number from the keyboard and store it in the variable ctemp.
3. Convert to Fahrenheit by using the formula ftemp = (ctemp * 1.8) + 32.
4. Print the "The Fahrenheit temperature is:" message.
5. Print the value of the variable ftemp.

If you have to go through this much trouble just to do something simple, why even bother? The answer is that once a program is written, you can run it over and over. And though programs take time to write, they usually execute at lightning speed.

Write the Equivalent C++ Statements

After you've determined precisely what you want your program to do, step-by-step, you need to write down the equivalent C++ statements. A statement is roughly the C++ equivalent of a sentence in English.

For example, say you want your program to do the following:

1. Print the "The Fahrenheit temperature is:" message.

2. Print the value of the variable ftemp.

You'd translate these steps into the following C++ statements:

```
cout << "The Fahrenheit temperature is: ";
cout << ftemp;
```

Remember, the goal of programming is to get the computer to do a series of specific tasks. But the computer understands only its own native language—*machine code*—which consists of 1s and 0s. Back in the 1950s, programmers did write out instructions in machine code, but this was difficult and time-consuming.

To make things easier, computer engineers developed programming languages such as FORTRAN, Basic, and C, which enable humans to write programs bearing at least some resemblance to English.

To write a program, you may want to start by writing *pseudocode*—an approach I often use in this book. Pseudocode is similar to English, but it describes the action of the program in a systematic way reflecting the logical flow of the program. For example, here's a simple program written in pseudocode:

```
If a is greater than b
  Print "a is greater than b."
Else
  Print "a is not greater than b."
```

Once you've written pseudocode, you're not far from having a C++ program. All you need to do is look up the corresponding C++ statement for each action:

```
if (a > b)
     cout << "a is greater than b.";
else
     cout << "a is not greater than b.";
```

The advantage of a programming language is that it follows rules that allow no ambiguity. C++ statements are so precise that they can be translated into the 1s and 0s of machine code without

guesswork.

It should come as no surprise that programming languages have strict rules of syntax. These rules are more consistent, and usually simpler, than rules in a human language. From time to time, I summarize these rules. For example, here's **if-else** syntax:


```
if (condition)
     statement
else
     statement
```

The words in bold are *keywords*; they must be entered into the program exactly as shown. Words in italics are *placeholders*; they represent items that you supply.

The application that translates C++ statements into machine code is called a *compiler*. I'll have a lot more to say about compilers in the upcoming section "Building a C++ Program." First, however, let's review some key definitions.

Interlude: How "Smart" Are Computers, Really?

When I ran a computer lab in Tacoma, Washington, some years ago, I came across some interesting characters, some of them right off the street. One of them was a rotund little man in a straw hat, ill-matching clothes, old shoes, and a big smile, who carried around a copy of *The Daily Racing Form* like it was the Bible.

He came up to me every other day with his horse-racing newspaper. "I got this idea," he kept saying. "We'll make a million. All we have to do is put this information into the computer and have it pick horses. I'll supply *The Daily Racing Form*. You write the program. We'll be rich!" I smiled but mumbled something about feasibility problems.

What he forgot—or probably never understood—is that no real knowledge exists inside the computer itself. There's no magical being inside that answers your questions like the ship's computer in *Star Trek*. Just entering raw data into a computer does nothing. What a computer needs first is the right *program*...a series of instructions that tells it how to move, copy, add, subtract, multiply, or otherwise evaluate data.

The real trick in getting a computer to properly pick winning horses at the racetrack, of course, would be to come up with the right *algorithm*...an algorithm being a systematic, step-by-step technique for evaluating data and getting a result, that is to say, the right formula. If you could come up with the right algorithm for picking horses, I don't deny it could make you rich. But in that case, finding a computer would be the least of your problems. You could use any computer as long as it could hold the data required.

As for finding a programmer, well, if you could get such a person to believe you really had the right algorithm, it would simply be a matter of offering to split the first 10 million dollars in winnings. I know any number of people who would go for that.

Or better yet, you could use the techniques in this book to write the program yourself.

Some Nerdy Definitions—A Review

I like to avoid jargon. I really do. But when you start learning programming, you enter into a world that requires new terminology. The following are some definitions you need to survive in this world.

application

Essentially the same thing as a program but seen from a user's point of view. An application is a program that a user runs to accomplish some task. A word processor is an application; so is an Internet browser or a database manager. Even a *compiler* (defined in a moment) is an application, but of a very special kind, because it's used by programmers. To put the matter simply, when your program is written, built, and tested, it's an application.

code

Another synonym for "the program," but from the programmer's point of view. The use of the term *code* stems from the days in which programmers wrote in machine code. Each machine instruction is encoded into a unique combination of 1s and 0s and is therefore the computer's code for doing some action.

Programmers have continued to talk about "code" even when using languages such as C++, Java, FORTRAN, or Visual Basic. (See the definition of *source code*.) Also, the term *code* is used to differentiate between the passive information in a program (its data) and the part of the program that performs actions (its code).

compiler

The language translator that takes C++ statements (that is, C++ source code) as input and outputs the program in machine-code form. This is necessary, because the computer itself—its central processing unit (CPU)—only understands machine code.

data

Information operated on by a program. At its most basic level, this information consists of words and/or numbers.

machine code

The CPU's native language, in which each computer instruction consists of a unique combination (or *code*) of 1s and 0s. It is still possible to program in machine code, but this requires looking up each instruction, as well as having elaborate knowledge of the CPU's architecture.

Languages such as C++ provide a way to write programs that is closer to English but still logically precise enough to be translated into machine code.

program

A series of instructions to be carried out by a computer along with initial data. As I mentioned earlier, a program can take time to write, but once it's completed, it usually executes at lightning-fast speed and can be run over and over.

source code

A program written out in a high-level language such as C++. Source code needs to be translated into machine code before it can actually be run on a computer. Machine code is typically represented in hexadecimal (base 16), so it looks something like this:

```
08 A7 C3 9E 58 6C 77 90
```

It's not obvious what this does, is it? Unless you look up all the instruction codes, such a program is incomprehensible—which is why almost no one writes in machine code anymore. In contrast, source code bears at least some resemblance to English. The following C++ statement says, "If salary is less than 0, print error message":

```
if (salary < 0)
     print_error_message();
```

statement

A simple statement usually corresponds to one line of a C++ program, and it is terminated by a semicolon (just as an English statement is terminated by a period). But C++ supports compound statements, just as English does, and these can take up multiple lines. Most C++ statements perform an action, although some perform many actions.

user

The person who runs a program—that is, the person who utilizes the computer to do something useful, such as edit a text file, read email, explore the Internet, or balance a checkbook. The more official name for user is *end user*.

When I was at Microsoft, the user was the person who caused most of the trouble in the world, but he or she was also the person who paid the bills. Even if this hypothetical person was a nontechnical "bozo," there would be no Microsoft without users buying its products. So when you start designing serious programs, it's important that you carefully consider the needs of the user.