Chapter 1. What Is C?

The ability to organize and process information is the key to success in the modern age. Computers are designed to handle and process large amounts of information quickly and efficiently, but they can't do anything until someone tells them what to do.

That's where C comes in. C is a programming language that allows a software engineer to efficiently communicate with a computer.

C is a highly flexible and adaptable language. Since its creation in 1970, it's been used for a wide variety of programs including firmware for micro-controllers, operating systems, applications, and graphics programming.

C is one of the most most widely used languages in the world and is fairly stable. An improved C language called C++ has been invented, but it is still in development, and its definition is still being worked on. C++, originally known as C with Classes, adds a number of new features to the C language, the most important of which is the class. Classes facilitate code reuse through object-oriented design (OOD).

Which is better, C or C++? The answer depends on who you talk to. C++ does great things for you behind your back, such as automatically calling constructors and destructors for variables. This processing makes some types of programming easy, but it makes static checking of programs difficult, and you need to be able to tell exactly what your program is doing if you are working on embedded control applications. So some people consider C++ the better language because it does things automatically and C doesn't. Other people consider C better for precisely the same reason.

Also, C++ is a relatively new language that's still changing. Much more C code exists than C++ code, and that C code will need to be maintained and upgraded. So C will be with us for a long time to come.

1.1 How Programming Works

Communicating with computers is not easy. They require instructions that are exact and detailed. It would be nice if we could write programs in English. Then we could tell the computer, "Add up all my checks and deposits, then tell me the total," and the machine would balance our checkbook.

But English is a lousy language when it comes to writing exact instructions. The language is full of ambiguity and imprecision. Grace Hopper, the grand old lady of computing, once commented on the instructions she found on a bottle of shampoo:

Wash
Rinse
Repeat

She tried to follow the directions, but she ran out of shampoo. (Wash-Rinse-Repeat. Wash-Rinse-Repeat. Wash-Rinse-Repeat...)

Of course, we can try to write in precise English. We'd have to be careful and make sure to spell everything out and be sure to include instructions for every contingency. But if we worked really hard, we could write precise English instructions.

It turns out that there is a group of people who spend their time trying to write precise English. They're called the government, and the documents they write are called government regulations. Unfortunately, in their effort to make the regulations precise, the government has made them almost unreadable. If you've ever read the instruction book that comes with your tax forms, you know what precise English can be like.

Still, even with all the extra verbiage that the government puts in, problems can occur. A few years ago California passed a law requiring all motorcycle riders to wear a helmet. Shortly after this law went into effect, a cop stopped a guy for not wearing one. The man suggested the policeman take a closer look at the law.

The law had two requirements: 1) that motorcycle riders have an approved crash helmet and 2) that it be firmly strapped on. The cop couldn't give the motorcyclist a ticket because he did have a helmet firmly strapped on—to his knee.

So English, with all its problems, is out. Now, how do we communicate with a computer?

The first computers cost millions of dollars, while at the same time a good programmer cost about $15,000 a year. Programmers were forced to program in a language in which all the instructions were reduced to a series of numbers, called *machine language*. This language could be directly input into the computer. A typical machine-language program looks like:

```
1010 1111
0011 0111
0111 0110
.. and so on for several hundred instructions
```

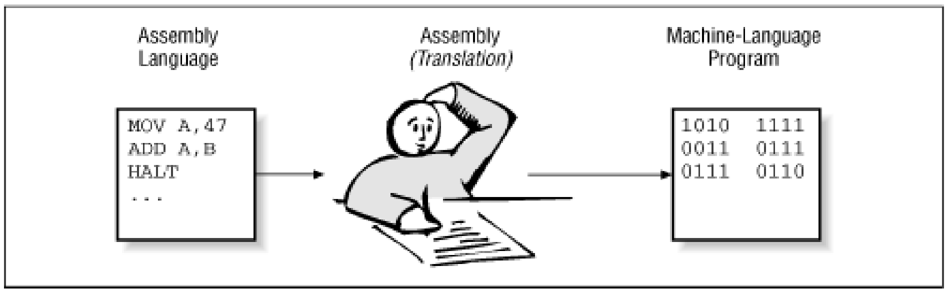
While machines "think" in numbers, people don't. To program these ancient machines, software engineers would write their programs using a simple language in which each word in the language stood for a single instruction. This language was called *assembly language* because the programmers had to hand translate, or assemble, each line into machine code.

A typical program might look like:

```
Program                  Translation
MOV A,47     1           010 1111
ADD A,B                  0011 0111
HALT               0111 0110
.. and so on for several hundred instructions
```

This process is illustrated by Figure 1-1.

Figure 1-1. Assembling a program



Translation was a difficult, tedious, and exacting task. One software engineer decided that this was a perfect job for a computer, so he wrote a program called an assembler that would do the job automatically.

He showed his new creation tohis boss and was immediately chewed out: "How dare you even think of using such an expensive machine for a mere `clerical' task." Given the cost of an hour of computer time versus the cost of an hour of programmer time, this attitude was not unreasonable.

Fortunately, as time passed the cost of programmers went up and the cost of computers went down. So letting the programmers write programs in assembly language and then using a program called an assembler to translate them into machine language became very cost effective.

Assembly language organized programs in a way that was easy for the programmers to understand. However, the program was more difficult for the machine to use. The program had to be translated before the machine could execute it. This method was the start of a trend. Programming languages became more and more convenient for the programmer to use, and started requiring more and more computer time for translation into something useful.

Over the years, a series of *higher-level* languages have been devised. These languages attempt to let the programmer write in a medium that is easy for him to understand, and that is also precise and simple enough for the computer to understand.

Early high-level languages were designed to handle specific types of applications. FORTRAN was designed for number crunching, COBOL was for writing business reports, and PASCAL was for student use. (Many of these languages have far outgrown their initial uses. Nicklaus Wirth has been rumored to have said, "If I had known that PASCAL was going to be so successful, I would have been more careful in its design.")

1.2 Brief History of C

In 1970 a programmer, Dennis Ritchie, created a new language called C. (The name came about because it superceded the old programming language he was using: B.) C was designed with one goal in mind: writing operating systems. The language was extremely simple and flexible, and soon was used for many different types of programs. It quickly became one of the most popular programming languages in the world.

C's popularity was due to two major factors. The first was that the language didn't get in the way of the programmer. He could do just about anything by using the proper C construct. (As we will see, this flexibility is also a drawback, as it allows the program to do things that the programmer never intended.)

The second reason that C is popular is that a portable C compiler was widely available. Consequently, people could attach a C compiler for their machine easily and with little expense.

In 1980, Bjarne Stroustrup started working on a new language, called "C with Classes." This language improved on C by adding a number of new features. This new language was improved and augmented, and finally became C++.

One of the newest languages, Java, is based on C++. Java was designed to be "C++ with the bugs fixed." At the time of this writing, Java has limited use despite being heavily marketed by Sun Microsystems and others.

1.3 How C Works

C is designed as a bridge between the programmer and the raw computer. The idea is to let the programmer organize a program in a way that he can easily understand. The compiler then translates the language into something that the machine can use.

Computer programs consist of two main parts: data and instructions. The computer imposes little or no organization on these two parts. After all, computers are designed to be as general as possible. The programmer should impose his organization on the computer, not the other way around.

The data in a computer is stored as a series of bytes. C organizes those bytes into useful data. Data declarations are used by the programmer to describe the information he is working with. For example:

```
int total;      /* Total number accounts */
```

tells C that we want to use a section of the computer's memory to store an integer named `total`. We let the compiler decide what particular bytes of memory to use; that decision is a minor bookkeeping detail that we don't want to worry about.

Our variable `total` is a simple variable. It can hold only one integer and describe only one total. A series of integers can be organized into an array as follows:

```
int balance[100];  /* Balance (in cents) for all 100 accounts
*/
```

Again, C will handle the details of imposing that organization on the computer's memory. Finally, there are more complex data types. For example, a rectangle might have a width, a height, a color, and a fill pattern. C lets us organize these four items into one group called a structure.

```
struct rectangle {
    int width;       /* Width of rectangle in pixels */
    int height;      /* Height of rectangle in pixels */
    color_type color; /* Color of the rectangle */
    fill_type fill;   /* Fill pattern */
};
```

The point is that structures allow the programmer to arrange the data to suit his needs no matter how simple or complex that data is. Translation of this data description into something the computer can use is the job of the compiler, not the programmer.

But data is only one part of a program. We also need instructions. As far as the computer is concerned, it knows nothing about the layout of the instructions. It knows what it's doing for the current instruction and where to get the next one, but nothing more.

C is a high-level language. It lets us write a high-level statement like:

```
area = (base * height) / 2.0;  /* Compute area of triangle */
```

The compiler will translate this statment into a series of cryptic low-level machine instructions. This sort of statement is called an *assignment statement*. It is used to compute and store the value of an arithmetic expression.

We can also use *control statements* to control the order of processing. Statements like the **if** and **switch** statements enable the computer to make simple decisions. Statements can be repeated over and over again by using looping statements such as **while** and **for.**

Groups of statements can be wrapped to form functions. Thus, we only have to write a general-purpose function to draw a rectangle once, and then we can reuse it whenever we want to draw a new rectangle.

C provides the program with a rich set of *standard functions* that perform common functions such as searching, sorting, input, and output.

A set of related functions can be grouped together in a single source file. Many source files can be compiled and linked together to form a program.

One of the major goals of the C language is to organize instructions into reusable components. After all, you can write programs much faster if you can "borrow" most of your code from somewhere else. Groups of reusable functions can be combined into a library. In this manner, when you need, for example, a sort routine, you can grab the standard function qsort from the library and link it into your program.

The data declarations, structures and control statements, and other C language elements, are not for the computer's benefit. The computer can't tell the difference between a million random bytes and a real program. All the C language elements are designed to allow the programmer to express and organize his ideas clearly in a manner tailored to him, not to the computer.

Organization is the key to writing good programs. For example, in this book you know that the Table of Contents is in the front and the Index is in the back. We use this structure because books are organized that way. Organization makes this book easier to use.

The C language lets you organize your programs using a simple yet powerful syntax. This book goes beyond the C syntax and teaches you style rules that enable you to make highly readable and reliable programs. By combining a powerful syntax with good programming style, you can create powerful programs that perform complex and wonderful operations, yet are also organized in a way that makes them easy for you to understand when change time comes around.

1.4 How to Learn C

There is only one way to learn how to program and that is to write programs. You'll learn a lot more by writing and *debugging* programs than you ever will by reading this book. This book contains many programming exercises. You should try to do as many of them as possib le. When you do the exercises, keep good programming style in mind. Always comment your programs, even if you're only doing the exercises for yourself. Commenting helps you organize your thoughts and keeps you in practice when you go into the real world.

Don't let yourself be seduced by the idea that "I'm only writing these programs for myself, so I don't need to comment them." First of all, code that looks obvious to a programmer as he writes it is often confusing and cryptic when he revisits it a week later. Writing comments also helps you to get organized before you write the actual code. (If you can write out an idea in English, you're halfway to writing it in C.)

Finally, programs tend to be around far longer than expected. I once wrote a program that was designed to work only on the computer at Caltech. The program was highly system-dependent. Because I was the only one who would ever use it, the program would print the following message if you got the command line wrong:

```
?LSTUIT User is a twit
```

A few years later, I was a student at Syracuse University, and the Secretary at the School of Computer Science needed a program that was similar to my Caltech listing program. So I adapted my program for her use. Unfortunately, I forgot about the error message.

Imagine how horrified I was when I came into the Computer Science office and was accosted by the Chief Secretary. This lady had so much power that she could make the Dean cringe. She looked at me and said, "User is a twit, huh!" Luckily she had a sense of humor, or I wouldn't be here today.

Sprinkled throughout this book are many broken programs. Spend the time to figure out why they don't work. Often, the problem is very subtle, such as a misplaced semicolon or the use of = instead of ==. These programs let you learn how to spot mistakes in a small program. Then, when you make similar mistakes in a big program, and you *will* make mistakes, you will be trained to spot them.