Foundations of Programming Using C

by
Evan Weaver
School of Computer Studies
Seneca College of Applied Arts and Technology

July 2006

Foundations of Programming Using C

Table of Contents

Preface

At Seneca, we have been struggling for years to find an adequate text for introductory programming. There is no shortage of good books covering the C language. The problem we have had is that we want to give a good introduction to practical programming in one semester, but we simply do not have the time to teach the entire C language. Whenever we pick a text, we end up skipping large pieces of almost every chapter, knowing that the omitted material will be covered in later semesters. The student is usually forced to choose between (1) reading the entire text anyway, since many explanations in the text assume the reader has read all the preceding material, or (2) not using the text, except perhaps for the odd table of information in it, and depending on material presented in class instead.

These notes are an attempt to bridge this gap. They introduce the fundamentals of programming, using a subset of the C language. While this subset is small compared to the content of the typical C book, it is large enough to write a surprisingly wide array of robust programs. These notes do not replace the need for a more thorough treatment of C. In fact, the student is encouraged to obtain at least one "real" C text, and probably two or three, to use in conjunction with this material. Virtually any "real" C book will do. We have found that every book speaks to different people, so the student should browse the C programming section of a bookstore, to find a book (or two) with explanations that are clear to that particular student.

A question that frequently comes up is, "why teach introductory programming using the C language?" We have tried other languages, such as COBOL, BASIC and Pascal, at different times in history. The advantages to C are many. It is a widely used and highly standardized language. It is available on virtually every computing platform, giving flexibility as to the programming platform that must be provided to present the material. Most C compilers are very efficient, which is no small concern if you have several hundred students compiling programs on a multi-user computer. The C language is the basis for C++, and is also the syntactical foundation for Java. Since our students go on to learn both C++ and Java, these later subjects benefit from not having to spend much time studying the basic language syntax. And so on.

Admittedly, we have been criticized for not starting the student immediately with object oriented programming, using either C++ or Java right from the start. Our contention is, of course, that we do! You cannot write objects, which from one point of view are collections of related subroutines, until you can write subroutines, and you cannot write subroutines until you know the basic syntax of logic control and data manipulation. We start with data manipulation and logic control, and then teach subroutines before moving on to objects. It just so happens that the semester ends before we get to objects, and so our second semester programming subject, which is C++, takes up where this subject ends, and completes what is really a two semester introduction to object oriented programming. It conveniently works out that the first half of that can be accomplished with that part of C++ which is C.

The main disadvantage to C is that many people find it hard to learn, and casual programmers never quite get the hang of it. Since our students are on track to be professional programmers, however, they do not fall into the category of "casual programmers". And the very reason we start by teaching a subset of the C language, rather than the whole thing, is to make learning easier at the introductory stage. In fact, the low level nature of C programming, which gives the programmer more control at the expense of forcing the programmer to accept more responsibility, teaches the student, in a very hands-on manner, a lot more about the fundamentals of computer architecture than is possible with "safer", higher-level languages.

Chapter 1. Introduction

What Is A Computer?

The modern world is filled with electronic devices of all sorts. Almost everything seems to be "computerized". But what is a computer? Specifically, what makes a computer different from other electronic devices? A computer can be defined as an electronic device with the following five properties:

- Processing (the ability to take raw data and make coherent information out of it)

- Input (the ability to put data into the computer)

- Output (the ability to get information from the computer)

- Storage (the ability to retain data for a period of time)

- Programmability (the ability to tell the computer how the data is to be processed)

Electronic devices that are not computers lack one or more of these properties. A calculator, for example, has a set of buttons for input, an LCD or LED panel (and sometimes a paper roll) for output, a limited storage capability, and a processing capability provided by the various function buttons on the calculator. But the buttons of a calculator always do the functions they were built to do; the calculator lacks programmability.

A computer, on the other hand, doesn't do anything unless it is given a set of instructions telling it exactly what it should do. Such a set of instructions is called a program. Since a computer is useless without a program, most people consider the program(s) to be part of the computer, and use the terms hardware and software to refer to the computer equipment and the programs, respectively. (Most computers have a program, or set of programs, built into the hardware to get things started when the computer is first turned on. This special kind of program is called firmware).

[A very different, yet equivalent, view of what a computer is, is to define a computer as an electronic device capable of performing mathematical simulations of events that occur, or that you would like to occur, in the "real world". Using this definition, a program would then be defined as the description of a simulation that the computer will perform.]

What Is Programming?

The job of a computer programmer is to give instructions to the computer telling it what to do. In other words, a programmer writes software. A programmer gives instructions in a language which the computer can understand (or, more accurately, in a language which some other program already on the computer can translate into a language the computer can understand).

Such languages are usually more precise, more rigid and less flexible than the languages humans use to communicate. While this should make a computer language easier to learn than a human language, many people cannot cope with the lack of flexibility and find programming to be an exercise in frustration. But for many others, fluency in a first computer language can be developed in a year or two (compared to 10 years or so for a person's first human language), with fluency in subsequent languages often taking a matter of months (versus years in the case of human languages).

Since computers are mathematical simulation tools, computer languages are essentially forms of mathematical notation. Many elements of the C language are similar to high-school algebra, for example. Yet success in mathematics at school is not necessarily a reliable indicator of the potential to be a good programmer. While it is true that people who are very good at mathematics almost always will be good at programming, there are many people who, for one reason or another, never got "turned on" by math at school, yet will become successful at programming. These people may find math a dreary and dull subject, but find the opportunity to control a machine, by giving it a complex and precise set of instructions, a fascinating challenge.

As with any profession, there are certain personality characteristics that programmers, by necessity, share. The prospective programmer must either already have, or must develop, these characteristics. For example, a good programmer must have patience. You will be working with a machine that neither knows nor cares that you are in a hurry. A programmer is persistent. When you write a program, there may be a big difference between what you think you've told the computer to do and what it actually does. It may take a lot of investigation to find out what you have done wrong after which you may find you have to take a completely new approach. A programmer is precise. With most computer languages, simply getting the punctuation wrong can cause significant errors. The sloppier about details you are, the more time you'll be frustrated tracking down "silly" errors. A programmer plans ahead. To tell a computer exactly what to do, you have to be sure not only what things it has to do, but in what order, and under what conditions. You can not just sit down at a computer and write a program. Rather you must analyze what you want the computer to do, and plan out the entire program before you start to write it. Finally, a programmer must be confident. The computer does not have any willingness to cooperate with you. It is not going to cheer you on. You must know what the computer is capable of, and be sure that you can make it do what you want it to do.

Keep these five traits (patience, persistence, precision, ability to plan and confidence) in mind as you learn to program. If you work on developing them in yourself, you will be more successful at programming than you would otherwise be, and might even find that you enjoy it.

While it may seem from this discussion of personality traits that programming is just a cause of aggravation, well, for some people it is. But for many, programming is a fulfilling profession. There are few things as satisfying as building a complex program, and seeing your work being used by others to help them get their work done.

Computer Architecture

Before getting into the mechanics of programming in C, it helps to have a basic understanding of the components that make up a computer system.

The main circuit of a computer, called the Central Processing Unit or CPU, is the circuit that takes instructions, one at a time, and carries them out. In smaller computers, the CPU is on a single integrated circuit (IC) chip, which is then called a microprocessor. (On some very fast computers, there may be more than one CPU, a situation which allows multiple sets of instructions to run concurrently). The CPU is designed to be able to interpret certain specific combinations of electrical signal as instructions. The set of allowable combinations is called the machine language of the computer.

All computers today are binary, which means that each wire or channel within the computer carries a signal that is effectively either "on" or "off". The CPU has many wires coming into it and going out of it. Each machine language instruction is one set of "on" and "off" values for each of the input lines. Most programmers and engineers represent the state of each wire with a 0 (for "off") or 1 (for "on"), so that each machine language instruction can be represented as a series of 0s and 1s. This can mathematically be interpreted as a number in base 2 (also called the binary numbering system). Once something is represented as a number in any base, it can be converted to a number in any other base, such as base 10 (or decimal), which is what people usually use to represent numbers. The net result of all this is that the machine language for a particular CPU is commonly represented as a collection of numbers. One number will cause the CPU to add two numbers, another will cause it to subtract, a third might cause it to multiply, and so on.

Connected to the CPU is another circuit called <u>memory</u>. The memory circuit stores data while the CPU is working with it. On most computers, this memory is for short term use only. Whenever the computer is turned off, for example, the memory circuit loses everything stored in it.

Also connected to the CPU are a series of <u>interface circuits</u>, which connect various devices to the CPU and memory. Each interface circuit connects to a different device. The devices connected to these circuits fall into two general categories: storage devices and input/output (I/O) devices. Storage devices are devices that provide long-term storage facilities. Typical storage devices are magnetic disk drives, optical disk drives (such as CD-ROMs) and magnetic tape drives. I/O devices provide input and/or output facilities to the computer. Typical I/O devices are things like a keyboard (input), monitor (or screen; output), mouse (input), printer (output) and modem (used to connect two computers over a telephone line; both input and output). A typical I/O device for a multi-user computer is a <u>terminal</u>, which combines a keyboard and a monitor into a single input and output device.

Note that the devices may or may not be in the same physical box as the CPU and memory. On a typical microcomputer, for example, it is common for the disk drives to be in the same box as the CPU, but the printer, monitor and keyboard are usually separate, connected to the CPU box by wires.

Programs, and the data manipulated by the programs, are stored on a storage device, usually a magnetic disk. Each program or collection of data stored on a disk is called a <u>file</u>. When a program is to be executed by the computer, it is copied from the disk into memory, where the CPU goes through it, one instruction at a time. Any data that needs to be stored for future, rather than immediate, use must be stored back on the disk. One simile people like to use is that the CPU is like a person sitting at a desk. The top of the desk itself is like memory, in that the person has immediate, back-and-forth access to it. The desk drawers are like the disk, where a file folder must be pulled out of the drawer and placed on the top of the desk to work on it. And anything that does not get put back in the drawer gets thrown out by the nightly cleaning staff!

<u>The Programming Process</u>

When a programmer wants to write a program, the following steps are usually followed:

1. The requirements for the program are analyzed, and the programmer makes sure that all the required knowledge, such as mathematical formulae, is known.

2. The program itself is planned out.

3. The program is written (or modified, if this is not the first time through - see 4 below). This may include writing documentation which explains how the program works.

4. The program is tested. If the program does not do what it was supposed to do, return to step 2 or 3, whichever is most appropriate. If, seeing the program run, it is clear the requirements weren't quite right, return to step 1.

Once the program passes the testing step, it may be used for its intended purpose. Note that in most cases, the user of the program will not be the programmer.

The mechanics of writing the program (step 3) are:

A. The program is typed in to the computer. Usually a program called a <u>text editor</u> (which allows you to enter and change text) is used to type in the program. The program is saved, on disk. The resulting file is called the <u>source file</u>, since it represents the original program as written by the programmer.

B. The source file is translated into machine language so that the computer can execute it. This translation is usually done by a program called a <u>compiler</u>, which will make another file, called an <u>executable program</u>, which is a machine language version of the program that can be run as many times as desired without having to translate the source file again. (Another kind of translation program, called an <u>interpreter</u> might also be used. An interpreter actually executes each line of the source file as it translates it, rather than storing the translated program for future execution. On almost all systems, however, C language programs are translated using a compiler).

C. If any part of the program contains badly formed instructions, the translation step will result in a <u>syntax</u> error, indicating that the translation program was unable to translate the program fully. Usually, the location of the error in syntax is shown, along with a message (though often very cryptic) indicating what the problem is suspected to be. If this is the case, return to step A. Otherwise, the program is ready for testing.

Since the programming environment might change from one semester to the next, the actual commands for editing a source file and compiling it are not described in these notes. Refer to instructions given in class instead.

The C Language

The C language is one of the most popular programming languages in use today. It was originally developed, in the 1970s, by Bell Labs for internal use within AT&T, but had such a good combination of simplicity, power and efficiency that people outside AT&T started writing C compilers. It is possibly the most widely available language, in that practically every computing environment today has a C compiler available. It is also highly standardized, compared to most languages, so that programs written on one computing environment can be moved with minimal changes to other computing environments.

C is not perfect, however. It is simple and powerful, but the power of the language gives the programmer the ability to mess up the computer unless great care is exercised. Consequently, it is usually recommended only for professional programmers who know what to watch out for. Casual programmers are encouraged to use a less powerful but safer programming language. Also, partly because of its simplicity and partly because of its power, there are usually many very different ways to achieve the same end in C. Beginning programmers often have a hard time trying to pick the "best" way to do something, because there are too many alternatives. To mitigate this, these notes will, in the early going at least, try to limit the number of alternate methods shown.

C is also based on a view of programming that developed in the late 1960s and early 1970s, called <u>structured</u> <u>programming</u>. Throughout the 1980s a more sophisticated view of programming, called <u>object</u> <u>oriented</u> <u>programming</u>, was developed to deal with the increasing complexity of programs. Fortunately, C is the basis for one of the most important object oriented languages, C++, which was also developed at Bell Labs. While there are a few minor syntactical changes required when moving from C to C++, virtually everything you'll learn about C will be required knowledge for when you later study C++.

So, without further ado, let us start to learn the C language.

Chapter 2. Basic Computations

A First C Program

This:

```
main()
{
    printf("Hello, world!\n");
}
```

is a C program that causes the line:

```
Hello, world!
```

to be displayed on the screen. (This program is the traditional "first" C program for someone who is learning C - Brian Kernighan and Dennis Ritchie presented it as their first sample program in the original book about the C language, called "The C Programming Language"). Note that fact that the words "main" and "printf" are in lower case is significant. This program would not work if you used "MAIN" or "Printf".

Every C program has a section titled "main()", immediately followed by one or more lines contained in a set of braces ("{" and "}"). The name main indicates that this is the main part of the program. Later, when we study functions, we will see how we can have other sections, with different names of our own choosing, in addition to main. We will also see the purpose of the (empty) parentheses after "main". For now, just be aware that every program has the framework

```
main()
{

}
```

with a bunch of stuff between the braces. In the case of our program, there is just one line, or statement:

```
printf("Hello, world!\n");
```

Note that, just as an English sentence normally ends with a period (.), a C statement normally ends with a semicolon (;). This particular statement is a printf statement, which in C means that something is to be displayed (or printed) on the screen. (The "f" at the end of printf is supposed to indicate that the programmer has some control over the format of what gets displayed, but some people speculate that it is just there to make the program look more intimidating). The parentheses after printf contain the data to be displayed. In this case, the data itself begins and ends with a double quote ("), which is used to tell the C compiler that what is between the quotes is not C, but is just a bunch of characters to be displayed. The C term for any text between double quotes is a <u>character string</u>.

The \n at the end of the character string shows the technique for including special characters that control the output but do not display as a single character. These special characters always begin with a backwards slash (\), followed by one (or sometimes more) characters. The special character represented by \n is called the <u>newline</u> character, and causes the display to advance to the beginning of the next line. For example, if we changed the printf statement to be:

```
printf("Hello,\nworld!\n");
```

the output would become:

```
Hello,
world!
```

Some other commonly used special characters are:

```
\a - beep (a stands for "alarm")
\b - backspace
\f - form feed (usually only affects printer output)
\t - go to the next tab stop (usually every 8 columns)
\\ - output a backslash
```

<u>Exercise 2.1</u>: Type in the "Hello, world!" program, adding a second printf below the first printf, but before the closing brace (}), so that the output of the program is:

```
Hello, world!
This program was typed in by Joan Smith
```

(where you should have your name instead of "Joan Smith"). Be sure that the second printf has a semicolon after it.

<u>Input, Output and Variables</u>

It is rare that anyone needs a program that simply displays the same thing every time you run it. All you need for that is a sign, not a computer. More commonly, people want programs that ask the user to enter something, and then perform some action based on what was entered.

In order to have the user enter something, you first need to set aside some space (in the computer's memory) where that data will be stored. In C you do this by <u>defining</u> a <u>variable</u>. A variable is a named area of memory. You define a variable by stating what kind of data you wish to store, and what name you want to use. For example,

```
int number;
```

tells the C compiler that you want to have a variable, named "number", which will be used to store integers. (An integer is a whole number which may be positive or negative). While "int" is a language keyword (we will be learning some other possible types of data shortly), "number" is simply a name of our own choosing. The rules for legal variable names are simple: only letters (a-z, A-Z), digits (0-9) and underscores (_) may be used in a name, and the first character of the name may not be a digit. The following are all legal variable names:

```
x
first_number
account_balance
DayOfWeek
r2d2
amount3
```

while the following are not:

```
day of week     (spaces are not allowed)
1st_number      (may not begin with a digit)
int             (int is a C language keyword)
```

While many compilers accept names longer than 31 characters, the limit on some compilers is 31 characters, so you probably shouldn't use names longer than that.

There are some keywords used by the C language that you should not use as variable names. These reserved words are listed in Appendix F.

The following program uses a variable to have the computer ask for a number and then display what was entered:

```
main()
{
    int number;

    printf("How many bananas do you want? ");
    scanf("%d", &number);
    printf("You want %d bananas.\n", number);
    printf("Sorry, we have no bananas.\n");
}
```

The first thing in this program is the declaration of an integer variable, number. Note that any variables you need in main must be defined after the opening brace, but before any other C statements. The program then displays the message asking the user to enter the number of bananas desired, using printf.

The next step uses another C statement, scanf, which gets input from the user (by "scanning" the keyboard). Here, scanf is being given two things, or parameters, separated by a comma (,). The first parameter is a character string that describes the format of the desired input. This string usually contains one or more format specifications, which begin with a percent sign (%) and describe the type of data to be input. The format specification, %d, is used to tell scanf that integer data is expected. (The "d" in %d comes from the fact that the integer will be be entered in base 10, or decimal, notation). Each data type has a different format specification to be used in scanf, and as we learn the various data types available, we will also learn the corresponding format specifications.

The second parameter to scanf is the location, in the computer's memory, of the variable into which we want scanf to place the input data. The & before the name of our variable, number, means "memory location of". Every variable exists somewhere in memory, and scanf needs to be given the memory location of a variable in order to be able to fill the variable up. (If the first parameter to scanf had more than one format specification in it, there would be additional parameters for scanf, giving one memory location of a variable for each format specification.)

After the scanf is a second printf which simply displays the value stored in the "number" variable. This printf also has two parameters. The first parameter is a character string describing the format of the output. Note the format specification (%d) in the middle of this string. This indicates that rather than simply outputting some characters, printf must output some integer data in place of the %d. The second parameter is our variable (which contains integer data); the value stored in this variable which will be displayed in place of the %d.

Finally, there is a third printf, which simply causes even more output to appear. If we run this program, and enter the number 3, the output would look like this:

```
How many bananas do you want? 3
You want 3 bananas.
Sorry, we have no bananas.
```

If we run this program and enter the number 56, the output would be:

```
How many bananas do you want? 56
You want 56 bananas.
Sorry, we have no bananas.
```

Even though we don't know how to make the computer do calculations yet, at least we can see that data we enter does actually get into the program in such a way that we can make use of it.

More Numeric Data Types

It is not convenient to use integers for everything. Most programming languages allow variables to store numeric data that is not restricted to integers.

In C, the data type double (short for "double-precision floating-point") is used to create variables that will store numeric information which may have a fractional part. The way that computers are designed makes any calculations done with double variables slower and less efficient than similar calculations done with int variables. Also, double calculations are subject to rounding errors (such as you encounter when using a calculator) to which int calculations, by their more restricted nature, are not prone. So, although you could theoretically use double variables for all storage of numeric data, it is generally best to use int variables in situations where they can be used, only resorting to double variables if there are reasons why int is not good enough. The scanf/printf format specification for double data is %lf (which you can think of as short for long floating-point number).

Two other numeric data types are long and float. On some machines (so-called 16-bit computers, such as a PC running DOS or Windows 3.1), int variables can only store numbers that are between -32768 and +32767. The long data type is like int, except that the range of values that may be stored is wider, typically from -2147483648 to +2147483647, with a corresponding decrease in efficiency. While longs are slower than ints and take more space, they are smaller and faster than doubles, so it makes sense to use long (rather than double) if int does not provide the necessary range, but long does. On most 32-bit computers (such as a PC running Windows 95, or an RS/6000), int is the same as long, so you don't need to worry about choosing between them. Still, many programmers will use int and long as if they were on a 16-bit computer, so that the program will be easier to move to a 16-bit computer should that ever be necessary. The scanf/printf specification for long is %ld (for long integer in decimal format).

The float data type is an abbreviated version of double. Floats, like doubles, can store data that may have fractional parts. But whereas a double is stored accurately to 11 or 12 significant digits (in decimal), a float is only accurate to 5 or 6 significant digits. The benefit to using float is that floats occupy half as much storage space as doubles. In most cases, the loss of precision is not worth the space savings, however, so float is not used anywhere near as much as int, long, or double. The scanf/printf specification for float is %f.

See Appendix C for explanations of exactly how computers use binary patterns to internally represent the different types of numeric values.

Arithmetic Operators

One of the main purposes of a computer is to perform calculations. C uses common mathematical notation for performing arithmetic operations. The plus sign (+) adds two numeric values, the minus sign (-) subtracts, and a slash (/) divides the number on the left by the number on the right. Because a keyboard doesn't have a funny x-shaped key, C uses * for multiplication.

Different kinds of numeric data can be mixed in a calculation. If so, the result is of the more precise type. For example, if an int value is added to a double value, the result will be a double. The following small program demonstrates how easy it is to do calculations.

```
main()
{
    int quantity;
    double cost;

    printf("Enter the number of apples desired: ");
    scanf("%d", &quantity);
    printf("Now enter the cost of one apple (in $): ");
    scanf("%lf", &cost);
    printf("That will cost $%lf\n", quantity * cost);
}
```

If, when running this program we entered 5 for the number of apples and 0.56 for the cost of one apple, the output would be:

```
Enter the number of apples desired: 5
Now enter the cost of one apple (in $): 0.56
That will cost $2.800000
```

Note how the product of the int value 5 and the double value 0.56 is a double, which is why we have used %lf to display the calculation. Note also how this double is shown with 6 digits after the decimal place. A double value does not have a specific preset number of decimal places, but the printf function will always show 6 decimal places, unless we tell it otherwise. To have printf show a specific number of decimal places other than 6, put a period followed by the number of places desired right after the %, but before the lf, in the format specification. For example, we would use the format specification "%.2lf" to have a double value shown to 2 decimal places of accuracy. Thus, changing the last line of the program to:

```
    printf("That will cost $%.2lf\n", quantity * cost);
```

would change the last line of the output to:

```
    That will cost $2.80
```

which looks more like what we would want to see.

There are a few idiosyncrasies with these arithmetic operators. For one, if a calculation involves both multiplication and addition, the multiplication will be done first, regardless of the order in which the operators are written. For example, the calculation

```
    x + y * z
```

would add x and the product of y and z (rather than multiply the sum of x and y by z). This fact that multiplication has higher precedence than addition mimics the common notation used in all basic mathematics courses. In C, both + and - have the same precedence as each other, and both * and / have the same precedence as each other, but have higher precedence than + and -.

See appendix E for a chart that lists the precedence of the various C operators used in these notes.

Parentheses can be used to over-ride the precedence of operators. For example,

```
    (x + y) * z
```

is how you could multiply the sum of x and y by z. Note that only parentheses, (), and not other brackets (such as [] or {}) can be used for this purpose.

When arithmetic operators of the same precedence are mixed in one calculation without parentheses, they get computed in order from left to right, so that, for example,

```
    a - b + c - d
```

would be the same as

```
    ((a - b) + c) - d
```

Another peculiarity is that division (/) works slightly differently, depending on the type of data being divided. If both sides of the division are ints, then the result will also be an int, where any remainder is just thrown away. If either side of the division is a double, the result will be a double, with the quotient being as accurate as the machine can store. There is an additional operator, called modulus (or, more commonly, remainder) which is used like / (with ints on either side), but computes the remainder upon dividing the left side by the right side rather than the quotient. The modulus operator is represented in C with the percent sign (%), and may be used with ints or longs, but not with floats or doubles. As an example of the use of integer division and modulus, consider the following program:

```
main()
{
    int minutes;

    printf("Enter the length of a videotape, in minutes: ");
    scanf("%d", &minutes);
    printf("That tape is %d hours and %d minutes long.\n",
     minutes/60, minutes%60);
}
```

Running this program, and supplying the length of 131 minutes when asked, would produce the output:

```
Enter the length of a videotape, in minutes: 131
That tape is 2 hours and 11 minutes long.
```

This program shows a few other basic aspects of the C language that we haven't yet discussed. For one thing, the last printf has two %d format specifications in the character string used as the first parameter. For this reason, there are three parameters (the character string itself and one integer for each of the %d specifications). The value of the second parameter (the calculation minutes/60) will be displayed in place of the first %d, and the value of the third parameter (the calculation minutes%60) will be displayed in place of the second %d.

The second new point is the use of constant values in calculations. We used the number 60 in both calculations, rather than using a variable. In C, if you use a whole number (which may have a + or - sign right at the beginning), it will be considered to be an int value, and can be used in any situation where an int value could be used. Similarly, if you use a number that has a decimal point in it (such as 3.14159 or -4.0), it is considered to be a double value and can be used in any situation requiring a double value. In this program, the number of minutes in a hour (60) is never going to change, so we don't need a variable in which to store it - we just use it directly in our calculations. (This is not really the first use of constants we have seen. A character string given in double quotes, such as the first parameters we have been supplying to printf and scanf, are examples of character string constants).

A third point about this program is that the last printf is too long to neatly fit on one line of the program. Rather than keeping it on one line, and have it run off the edge of the page when we show it on paper, we have simply continued the printf on the next line. The C language doesn't really care how the program is spaced out on the screen, but rather cares about the punctuation. The very first program we looked at could have been typed in like this:

```
main(){printf("Hello, world!\n");}
```

and it would work exactly the same way. C allows spaces, tabs and newlines to be freely inserted anywhere in a program EXCEPT in the middle of a name (such as a variable name), language keyword (such as int or double) or constant (such as 60 or "Hello, world!\n"). In particular, wherever there is punctuation (such as the commas between parameters) or an operator (such as +) there is an opportunity to insert a space or even go to a new line. By the way, C programmers use the term <u>whitespace</u> to refer to any combination of one or more space, tab, vertical tab, form feed and newline characters. We have been using this ability to insert whitespace to make our programs look attractive and more readable than they would be if they were just all jumbled up on one line. While there are no hard and fast rules about using whitespace, there are some guidelines that most programmers follow:

1. Have no more than one statement per line.
2. Indent lines inside of braces ({ and }).
3. If a statement is going to be wider than the screen and/or a printed page, split it.

<u>Assignment Operator</u>

Often, it is desirable to store a calculation in a variable, rather than simply display it. If you need one calculation over and over again, it makes sense to calculate it once, storing the result, and use that stored value over and over. To assign a calculation to a variable, you put the variable name first, then an equals sign (=, called the <u>assignment</u> operator in C) followed by the desired calculation, with a semi-colon (;) at the end. Consider the following program, which is a variation on an earlier one.

```
main()
{   int quantity;
    double cost, total, tax;

    printf("Enter the number of apples desired: ");
    scanf("%d", &quantity);
    printf("Now enter the cost of one apple (in $): ");
    scanf("%lf", &cost);

    total = quantity * cost;
    tax = 0.15 * total;

    printf("That will cost ($%.2lf plus $%.2lf tax): $%.2lf\n",
     total, tax, total + tax);
}
```

If, when running this program we entered 4 for the number of apples and 0.50 for the cost of one apple, the output would be:

```
Enter the number of apples desired: 4
Now enter the cost of one apple (in $): .50
That will cost ($2.00 plus $0.30 tax): $2.30
```

This program assumes that there is a 15% tax to be added to the purchase of apples, and shows the total before tax, the tax and the total including tax. To compute the tax, we need the total before tax, and to compute the total after tax, we need both the total before tax and the tax, so this program (1) asks for the quantity and cost, (2) computes the total before tax and stores it in "total", (3) computes the tax, based on the value in "total" and stores that in "tax", (4) displays "total", "tax" and their sum.

It is important to realize that a mathematical formula, like

```
y = m*x + b
```

is a little bit different from a C assignment statement, like

```
tax = quantity * cost;
```

A mathematical formula generally is used to establish a relationship which will hold throughout the remainder of the discussion. Mathematical statements tend to be things that are always true, and specific assumptions (such as "assume that x is 3") may usually be done later. If you know the value of m and x and b, then you can determine y; if you know y and m and x, you can determine b; and so on.

The statements in a program, on the other hand, are things to be done, one at a time, in the order they appear in the program. The C assignment statement does a computation once, and stores its value in the variable on the left side. In the case above, quantity and cost must already have values, and the variable tax will be set to be the product of those values. If you later change the variable quantity (or cost), the variable tax will not be affected (unless you then repeat the assignment statement).

One final comment about the "apples" program concerns the definition of variables. We had one line:

```
    double cost, total, tax;
```

to define three variables, named "cost", "total" and "tax", all of the same type (double). We could have used three separate definitions:

```
    double cost;
    double total;
    double tax;
```

The C language allows you to define several variables of the same type at one time, by separating the different names with a comma, ending with a semi-colon. There is no advantage to doing this other than the fact that the program is a bit shorter to type in, but that is enough to make it common practice.

<u>Exercise 2.2</u>: Suppose that a taxi fare is always $2.00 plus $0.20 for each kilometer travelled and $1.50 for every large suitcase. Write a program that asks the user to specify how many kilometers were travelled, and how many large suitcases there were, and displays (1) the total for the trip without the suitcase charge, (2) the charge for suitcases and (3) the entire cost of the trip.

Chapter 3. Basic Logic

The ability to perform the same computation on different pieces of data, while useful, is not the essential element of computer programming. Most calculators, other than the most basic models, can store one formula and allow you to enter different data to be plugged into it. What distinguishes programming from simply typing in formulae is the ability to tell the computer to do different things under different conditions.

The if Statement

In C, the if statement is one way to instruct the computer to decide what should be done. The syntax of the if statement is:

```
if (some condition)
    some statement
```

where "some statement" is to be replaced with some C statement, and "some condition" is to be replaced with the condition that determines whether or not the statement should be executed. As an example, the following program is an enhancement of our earlier "apple" program, where we have decided to give a 10 percent discount on any amount (before taxes) over the first $100:

```
main()
{
    int quantity;
    double cost, total, tax;

    printf("Enter the number of apples desired: ");
    scanf("%d", &quantity);
    printf("Now enter the cost of one apple (in $): ");
    scanf("%lf", &cost);

    total = quantity * cost;
    if (total > 100)
        total = 100 + (total - 100) * 0.9;
    tax = 0.15 * total;

    printf("That will cost ($%.2lf plus $%.2lf tax): $%.2lf\n",
     total, tax, total + tax);
}
```

Note how after calculating total, but before figuring out the taxes, we decide whether or not to recalculate total to reflect the discount. In this case, we want to use the discount only if the total is over $100. Make sure that you understand the calculation that is being done under this condition. If the total were, say, $120, then there would be a 10% discount on $20 (the amount in excess of $100), so the pretax total would be

$118 ($100 plus 90% of $20, which is another way of looking at $120 minus 10% of $20). On the other hand, when the total is $100 or less, it is not changed at all.

Notice also how symbols like $ and % are not used to represent dollars and percents. In C, any numeric amount is just a number. Whether a number is a special kind of number or not can be reflected in the output of the program (by placing a $ in a printf statement, for example), but has no bearing on the simple arithmetic calculations that you ask the computer to perform. As an example, we have multiplied by 0.9 to get 90% of an amount; we cannot multiply by 90%. (We have already seen that the % sign means something else - modulus - in C computations).

Relational Operators

The simplest kinds of conditions that you can use in if statements involve the comparison of numeric amounts. A normal computer keyboard does not have all the common mathematical symbols for comparisons, so the following symbols are used in C:

Symbol	Meaning
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
!=	not equal to
==	equal to

The above meanings refer to what is on the left of the operator, compared to what is on the right. For example, if we had two variables, x and y, then the statement:

```
if (x >= y)
    printf("Hello\n");
```

would print out "Hello" (and move to the next line) if, and only if, the number stored in x is greater than or equal to the number stored in y. Be especially careful when asking if two things are equal. The following statements look essentially the same:

```
right:
    if (x == y)
        printf("They are the same\n");

wrong:
    if (x = y)
        printf("They are the same\n");
```

In the second one, the value stored in y is actually being copied into x using the assignment operator! We will see later how this statement would use the assignment as a condition, but for now realize that the second (and wrong) example is actually legal C code, and most compilers will just happily compile it. (A few compilers will display a warning message telling you that you just might be doing something dubious, but will still compile it. No doubt, such compilers were written by programmers who frequently use = when they mean ==).

These relational operators have lower precedence than the arithmetic operators, so that a condition such as

```
x + y < z * (200 + y)
```

would be the same as

```
(x + y) < (z * (200 + y))
```

<u>Expressions, Statements and Code Blocks</u>

We have already said that a statement is to a C program what a sentence is to something written in English. Another way to describe what a statement is, is to define it as one complete step in a program. An <u>expression</u> in C is some part of a C statement that has a value associated with it. In the statement

```
x = y * (3 + z) + w;
```

there are many expressions, specifically: y, 3, z, w, 3 + z, y * (3 + z), and y * (3 + z) + w. The value of the last of these expressions is what is ultimately stored in x by this statement. Note how an expression can be composed of other expressions. Even conditions (using the relational operators) are considered to be expressions; in a condition, the possible values are "true" and "false", rather than a number.

Just as expressions are often composed of other expressions, statements can be composed of other statements. For example, in the statement:

```
if (x >= y)
    printf("Hello\n");
```

we see the statement

```
printf("Hello\n");
```

For this reason, the if statement is sometimes called a <u>compound</u> statement, since an if statement always includes at least one other statement as part of itself.

Another way to form a compound statement is to take a bunch of statements and enclose them in braces ({}). This makes the collection of statements into one big statement, called a code block. (This term comes from the fact that programmers often refer to the programs they write as "code"). Although we didn't know the term then, we have already seen that every program must have at least one code block - the "main" section of the program. The statement that forms the latter part of an if statement can be a code block. (In fact, any statement may be replaced by a code block). This allows several statements to be executed (in order) if a particular condition is true. Let us modify the apple program even more to display the amount of discount, but only when a discount is applied:

```
main()
{
    int quantity;
    double cost, total, tax, discount;

    printf("Enter the number of apples desired: ");
    scanf("%d", &quantity);
    printf("Now enter the cost of one apple (in $): ");
    scanf("%lf", &cost);

    total = quantity * cost;
    if (total > 100) {
        discount = (total - 100) * 0.1;
        total = total - discount;
        printf("Receiving a discount of $%.2lf\n", discount);
    }
    tax = 0.15 * total;

    printf("That will cost ($%.2lf plus $%.2lf tax): $%.2lf\n",
     total, tax, total + tax);
}
```

Here, we have changed the way we calculate the discount, and have even added a variable named "discount", to make it easier to display the amount of the discount, but the final total will be the same as it was before. Make sure that you can follow the change in calculations, and are satisfied that the end result will be the same. Programmers often have to change the details of a calculation when asked to display some of the intermediate values.

In this program we are now doing three statements (the calculation of discount, recalculating total by taking away discount, and displaying discount) if the total is over $100. Notice how we have placed the opening brace ({) just after the condition, and have the closing brace (}) lined up under the "i" of "if", with the three statements indented a bit. This is a common coding style used by many programmers. A similar style that many beginners prefer is to line both the opening and closing braces up under the "if", indenting everything in between:

```
if (total > 100)
{
    discount = (total - 100) * 0.1;
    total = total - discount;
    printf("Receiving a discount of $%.2lf\n", discount);
}
```

This does occupy one more line of the page, but makes it harder to "lose" the opening braces when you are quickly scanning the program. Remember that the spacing of the program code is not an issue to the compiler or to the correctness of the program, but it can be tremendously helpful to someone (perhaps even yourself!) reading your program at a later time. The generally accepted rule is to indent lines of a program whenever the execution of those lines may be dependent on something. That way, it really stands out that the lines may or may not be executed. There are many variations of this theme, and as you continue to practice programming, you will settle on a style that you like the best. It is more important that you develop a style that you use consistently, than it is that you follow the style used by any one book.

At any rate, here we have one statement, an if statement, which contains three other statements. Earlier, we stated that statements "usually" end with a semi-colon (;), but when the "statement" following the condition in an if statement is really a code block, the if statement actually ends with the closing brace rather than a semi-colon. This is the only exception to the semi-colon: in C, compound statements might end with the closing brace of a code block - otherwise, a semi-colon marks the end of a statement.

Note that the statement which is part of an if statement, or any statement in a code block, for that matter, may be another compound statement, such as an if statement. Let us modify our program yet again, this time doubling the discount if the pre-discount total exceeds $1000:

```
main()
{
    int quantity;
    double cost, total, tax, discount;

    printf("Enter the number of apples desired: ");
    scanf("%d", &quantity);
    printf("Now enter the cost of one apple (in $): ");
    scanf("%lf", &cost);

    total = quantity * cost;
```

```
        if (total > 100) {
            discount = (total - 100) * 0.1;
            if (total > 1000)
                discount = 2 * discount;
            total = total - discount;
            printf("Receiving a discount of $%.2lf\n", discount);
        }
        tax = 0.15 * total;

        printf("That will cost ($%.2lf plus $%.2lf tax): $%.2lf\n",
         total, tax, total + tax);
    }
```

Notice how, using our coding style, the second if, which is part of the first if, causes a secondary indentation.

The while Statement

The while statement, which has the same syntax as the if statement, works almost the same way except that it repeats the statement (or code block) over and over again. More precisely, the while statement checks the condition. If it is true, it does the statement (or code block) and then checks the condition again. If the condition is still true, it does the statement (or code block) and checks the condition yet again. This continues until finally the condition, when checked, is false.

The following program plays a simple little game with the user. It asks for a number, expecting a particular value. If the user's response isn't the one it expected, it gives a hint ("too high" or "too low") and asks again. Eventually (hopefully) the user will enter the correct number and the game ends.

```
    main()
    {
        int guess;

        printf("Guess a number: ");
        scanf("%d", &guess);
        while (guess != 42) {
            if (guess > 42)
                printf("Too high! ");
            if (guess < 42)
                printf("Too low! ");
            printf("Try again: ");
            scanf("%d", &guess);
        }
        printf("You guessed the magic number! YOU WIN!!\n");
    }
```

A sample run of this program is:

```
Guess a number: 34
Too low! Try again: 50
Too high! Try again: 40
Too low! Try again: 42
You guessed the magic number! YOU WIN!!
```

(Of course, since you have read the code for this game, you know what the answer is and would simply enter 42 right at the beginning. But someone who hasn't read the program wouldn't be able to get it right away without a great deal of luck).

Note how the last step in the while statement is to have the user enter a number into the variable, guess, which also is checked in the while's condition. It is critical that some value involved in the condition has the opportunity to change during the execution of the while statement. Otherwise, the statements will simply be executed over and over again, forever. This kind of programming error is called an <u>infinite loop</u>, because the program keeps repeating, or "looping", the same code over and over. If you do write a program with an infinite loop, and it just keeps repeating over and over, there is a way to stop it, but each system uses a slightly different method. On some systems there is a key labelled "break" or "attention", and on others there is a special key sequence (often, it is Control/C - pressing C while holding down the key labelled "Control"), which will terminate a runaway program.

Let us now add a little bit more to the game, so that the computer can tell us how many tries it took:

```
main()
{
    int guess, counter;

    counter = 1;
    printf("Guess a number: ");
    scanf("%d", &guess);
    while (guess != 42) {
        if (guess > 42)
            printf("Too high! ");
        if (guess < 42)
            printf("Too low! ");
        printf("Try again: ");
        scanf("%d", &guess);
        counter = counter + 1;
    }
    if (counter == 1)
        printf("WOW! Either you are very lucky or you CHEAT!\n");
    if (counter != 1)
        printf("You win in %d tries.\n", counter);
}
```

Here, we have set up a second variable, which we have named counter, and we start it out at 1. Each time the user has to make another guess (i.e. each time the loop executes) we add one to the value stored in counter. Thus, this variable will always store the number of times the user has taken a guess.

Compound Conditions

Sometimes a simple condition, such as a single comparison, is not sufficient to control an if statement or a while statement. In these situations, it is usually a combination of circumstances that determine what code should be executed.

You can combine two separate conditions into a larger condition using one of two logical operators, and (&&) and or (||).

When two conditions are joined with && (and), the resulting compound condition will be true if, and only if, both of the sub-conditions are true. When two conditions are joined with || (or), the resulting condition will be true if either (or both) of the sub-conditions are true. This matches what we usually use "and" and "or" to mean in English, when we use them to combine conditions.

These two operators have lower precedence than the relational operators, so that the expression

```
x < y && x < z
```

is the same as

```
(x < y) && (x < z)
```

However, && has a higher precedence than || (in much the same way as * has a higher precedence than +), so that

```
x < y || x < z && y < z
```

is the same as

```
(x < y) || ((x < z) && (y < z))
```

The following samples demonstrate various cases. They assume that x is 5, y is -10 and z is 3. Use them to make sure you understand the way the && and || work:

```
Condition          Value
x < z && y < z     false (since x < z is false)
y < z && z < x     true  (since both y < z and z < x are true)
x < z || y < z     true  (since y < z is true)
y > x || y > z     false (since neither y > x nor y > z is true)
```

Just as a common programming error is to use = in a condition where you mean ==, it is another common programming error to use the symbols & and | instead of && and ||. The operators & (bitwise and) and | (bitwise or) are valid C operators, and will not generate syntax errors from a compiler. However & and |, the use of which are an advanced topic beyond the scope of these notes, do not quite work the same way as && and ||, and should not be used to join a series of true/false conditions, even though they may often seem to work.

Now that we can join conditions, we will refine our game a bit more. We will make it stop when they get it right, or have made 100 tries, whichever comes first. If they don't get it in 100 tries, they lose. If they do win, we'll give them a different message depending on how few or how many turns they took.

```
main()
{
    int guess, counter;

    counter = 1;
    printf("Guess a number: ");
    scanf("%d", &guess);
    while (guess != 42 && counter < 100) {
        if (guess > 42)
            printf("Too high! ");
        if (guess < 42)
            printf("Too low! ");
        printf("Try again: ");
        scanf("%d", &guess);
        counter = counter + 1;
    }
    if (counter == 1)
        printf("WOW! Either you are very lucky or you CHEAT!\n");
    if (counter >= 2 && counter < 5)
        printf("Very good. You got it in %d turns\n", counter);
    if (counter >= 5 && counter < 15)
        printf("You win in %d turns, an average performance\n",
         counter);
    if (counter >= 15 && counter < 100)
        printf("Duh, it took you %d tries to get it!\n", counter);
    if (counter == 100 && guess == 42)
        printf("You got it in the nick of time\n");
    if (guess != 42)
        printf("You lose\n");
}
```

In this program we have made the condition controlling the while loop into a compound condition using &&. There are two ways out of this loop: the user could enter 42 (making the first sub-condition false) or the user could take 100 guesses (making the second condition false). Note that if the user does take 100 guesses, that 100th guess might be right (a narrow victory) or wrong (a loss).

After leaving the loop, we take great pains to split the possibilities into six distinct cases: right on the first try, right in fewer than 5 tries, right in fewer than 15 tries, right in fewer than 100 tries, right on the 100th try, and wrong. The conditions in the six if statements are carefully arranged so that one and only one will work out to be true, thereby causing one and only one of the the printf statements to execute.

As you look at this last version of the game you should be starting to notice how quickly things go from being simple to being tricky. In fact, the little bit of C syntax that we have learned so far is enough to write programs as complex as they come. We know how to do input (scanf), output (printf) and mathematical computations, and we can selectively execute parts of our program (if) or make parts of our program repeat (while). These are the fundamental elements of programming. From now on we will just be learning variations of these basic elements, and techniques of using them that will make it easier to develop and write programs.

<u>Exercise 3.1</u>: Modify the "apple" program we have been developing, so that it can be used at the check-out stand of an apple farm. When the program is started (presumably at the start of the day), it should ask once for the price of an apple. Then it should repeatedly ask for the number of apples purchased. After each number is entered, the total cost (including taxes and discounts) for that number of apples is shown. (Each number entered represents a separate customer coming to the check-out). At the end of the day, 0 is entered for the number of apples, and the program stops with the message, "Have a nice day!". A sample run of this program would look like this:

```
Enter today's price of one apple: 0.25
Enter the number of apples purchased (or 0): 100
That will cost ($25.00 plus $3.75 tax): $28.75
Enter the number of apples purchased (or 0): 1000
Receiving a discount of $15.00
That will cost ($235.00 plus $35.25 tax): $270.25
Enter the number of apples purchased (or 0): 10000
Receiving a discount of $480.00
That will cost ($2020.00 plus $303.00 tax): $2323.00
Enter the number of apples purchased (or 0): 4
That will cost ($1.00 plus $0.15 tax): $1.15
Enter the number of apples purchased (or 0): 0
Have a nice day!
```

While any logic can be programmed using the simple if and while statements, the C language provides a few alternative logic control statements which can make a program more efficient or easier to read.

The if/else Statement

The most common variation is the if/else statement, which has the syntax:

```
if (some condition)
    some statement
else
    some other statement
```

(Just as with the simple if and while, some statement can be a single statement or a code block, as can some other statement).

If the condition is true, then some statement is executed and some other statement is skipped. If the condition is false, then some statement is skipped and some other statement is executed. In a situation such as:

```
if (x < 5)
    printf("That number is too small\n");
if (x >= 5)
    printf("That is a good number\n");
```

an if/else could be used instead:

```
if (x < 5)
    printf("That number is too small\n");
else
    printf("That is a good number\n");
```

Here, the use of if/else would make the program more efficient, because it would only have to compare x to 5 once, rather than twice. The use of else is also more readable, in the sense that it is clear that one thing or the other is going to be done. In the first example, you must carefully examine both conditions to see that they are opposites in order to tell that only one of the two dependent statements is going to be executed.

A common situation is

```
if (x < 5)
    printf("That number is too small\n");
else
    if (x > 10)
        printf("That number is too big\n");
    else
        printf("That number is nice\n");
```

where one of the dependent statements is itself an if/else. Notice how our indenting style causes the whole dependent if/else to be indented. In recognition of the fact that, even though this is syntactically two two-way decisions, it is really a three-way decision, many programmers prefer the following indentation style for these so-called <u>nested</u> <u>if</u>s:

```
if (x < 5)
    printf("That number is too small\n");
else if (x > 10)
    printf("That number is too big\n");
else
    printf("That number is nice\n");
```

Using the if/else, we can make our game logic a little bit cleaner:

```
main()
{
    int guess, counter;

    counter = 1;
    printf("Guess a number: ");
    scanf("%d", &guess);
    while (guess != 42 && counter < 100) {
        if (guess > 42)
            printf("Too high! ");
        else
            printf("Too low! ");
        printf("Try again: ");
        scanf("%d", &guess);
        counter = counter + 1;
    }
    if (counter == 1)
        printf("WOW! Either you are very lucky or you CHEAT!\n");
    else if (counter < 5)
        printf("Very good. You got it in %d turns\n", counter);
    else if (counter < 15)
        printf("You win in %d turns, an average performance\n",
         counter);
    else if (counter < 100)
        printf("Duh, it took you %d tries to get it!\n", counter);
    else if (guess == 42)
        printf("You got it in the nick of time\n");
    else
        printf("You lose\n");
}
```

Compare this version of the game closely to the previous one. In particular, notice how the conditions, in the nested ifs at the end, are much simpler. The condition in the second of the ifs, for example, went from

```
counter >= 2 && counter < 5
```

to

```
counter < 5
```

because we have already ruled out the case where counter is 1 by making the second if statement dependent on the failure of the first if's condition (counter == 1).

The do/while Statement

The while statement always checks its condition before deciding whether or not to execute its contents. Under certain conditions (when the condition is false right away) the contents of a while loop may not even be executed once. There are situations, however, where you might have logic that will be repeated at least once. A while loop can still be used for this (you simply need to force the variables used in the condition to be in a state where the condition is true when the loop is first started), but another statement, the do/while, is more convenient. The syntax for the do/while is:

```
do
    some statement
while (some condition);
```

(As always, some statement could be a code block). The do/while performs some statement first, and then checks some condition to decide whether (true) or not (false) to repeat it. For example, the situation:

```
main()
{
    int n;

    printf("Please enter a number between 1 and 5:");
    scanf("%d", &n);
    while (n < 1 || n > 5) {
        printf("Please enter a number between 1 and 5:");
        scanf("%d", &n);
    }
    printf("Thank you. You selected %d\n", n);
}
```

could be simplified, still using the basic while statement, as:

```
main()
{
    int n;

    n = 0;
    while (n < 1 || n > 5) {
        printf("Please enter a number between 1 and 5:");
        scanf("%d", &n);
    }
    printf("Thank you. You selected %d\n", n);
}
```

where we force n to zero, in order to make the condition true so that the loop will execute at least once. Using do/while, we can avoid this arbitrary setting of n:

```
main()
{
    int n;

    do {
        printf("Please enter a number between 1 and 5:");
        scanf("%d", &n);
    } while (n < 1 || n > 5);
    printf("Thank you. You selected %d\n", n);
}
```

Since we don't check n until after executing the contents of the loop, we don't need to set n before starting the loop. Note how we have placed the closing brace (}) of the loop's code block just before the while keyword. This is done by many programmers to highlight the fact that the while is the end of a do/while, rather than the start of a simple while statement.

<u>DeMorgan's Law</u>

An element of the last program that often causes confusion is the condition for the loop, where we want to keep looping as long as n is outside the range of 1 to 5. We used an or (||) to join n < 1 and n > 5, because we want to keep looping if one is true or the other is true. Using and (&&) would not be appropriate, because these two sub-conditions can never both be true at the same time. Still, many people are tempted to use &&, because they know the condition for a valid number (which causes us to leave the loop) is

```
x >= 1 && x <= 5
```

There is a mathematical fact, called DeMorgan's Law, which says that to get the opposite of a compound condition, you must reverse all the sub-conditions and, at the same time, change all &&s to ||s, and all ||s to &&s. By applying DeMorgan's Law to the above condition (which would cause us to leave the loop), we obtain

```
x < 1 || x > 5
```

as the condition to stay in the loop, which is what we used. When using DeMorgan's Law to reverse a condition, be aware that as you switch ||s to &&s, you may occasionally need to add parentheses to keep the order of operations the same.

The for Statement

Yet another looping variation is the for statement. Like the do/while, it does not give you any capability that you don't already have with while, but it can be more convenient in certain situations. The syntax for the for statement is:

```
for (initial; condition; trailing)
    statement
```

which does the same thing as

```
initial;
while (condition) {
    statement
    trailing;
}
```

Inside the parentheses of the for, there are three things, separated by semi-colons (;). The first is something to be executed before starting the loop. The second is the condition to be checked to determine whether or not to repeat the loop. Like the while statement, the condition is checked before executing the loop for the first time. The third is something to be executed after each time through the loop, before checking the condition for the next time through.

The following program, which sums 10 numbers input by the user, shows a good use of the for statement:

```
main()
{
    int n, sum, counter;

    sum = 0;
    for (counter = 1; counter <= 10; counter = counter + 1) {
        printf("Please enter number %d: ", counter);
        scanf("%d", &n);
        sum = sum + n;
    }
    printf("The sum of those numbers is %d\n", sum);
}
```

The advantage of the for statement, in a situation like this, is that it allows you to place all the pieces that control the loop in the first line of the loop, rather than having one before the loop and one at the bottom of the loop. This makes it easier to see what is making the loop continue. Using for rather than while is simply an issue of style; it carries no efficiency benefit. Although for is not restricted to loops where a variable is counting the number of times through the loop (such as the example above), this is the typical situation where for is considered to be more readable than while.

The switch Statement

A final logic control statement in C is the switch statement. The switch is an alternative to a nested if statement in certain circumstances. The syntax for switch is:

```
switch (expression) {
    case constant1:
        zero or more statements
    case constant2:
        zero or more statements
    ...and so on (as many cases as you like)...
    default:
        zero or more statements
}
```

The computer first figures out the value of expression, and then compares that to constant1. If there is a match, it begins to execute statements immediately after the colon (:) at the end of the first "case" line. If it doesn't match, it compares the same value to constant2. If that matches, it starts executing statements immediately after the colon at the end of the second "case". This continues until it finds a match or reaches "default:", whichever comes first. If there is no match, the statements after "default:" are executed.

Another new statement,

```
break;
```

is almost always used with switch. The break statement tells the computer to leave the current switch statement (and to resume processing after the closing brace of the switch), and is usually the last statement at the end of a "case" section. Without a break statement, the logic from one case section simply flows into the next.

Consider the following "game show" program using a nested if:

```
main()
{
    int door;
    double retail;

    printf("Welcome to \"Why Not Make a Deal?\"\n");
    printf("Would you like to look behind\n");
    printf("door number 1, door number 2 or door number 3? ");
    scanf("%d", &door);
    printf("You have selected ");
    if (door == 1) {
        printf("a new kitchen by Matilda Kitchens!\n");
        retail = 2500;
    }
    else if (door == 2) {
        printf("your very own.....\n");
        printf("...DONKEY! And a shovel, too!!\n");
        printf("Too bad. Better luck next time.\n");
        retail = 7.50;
    }
    else if (door == 3) {
        printf("a BRAND NEW CAR!!\n");
        printf("The fabulous 4-door\n");
        printf("Liability by Generic Motors!\n");
        printf("Congratulations!\n");
        retail = 15700;
    }
    else {
        printf("a door that doesn't exist\n");
        retail = 0;
    }
    printf("The retail value of your prize is $%.2lf\n", retail);
}
```

A sample run of this program would be:

```
Welcome to "Why Not Make a Deal?"
Would you like to look behind
door number 1, door number 2 or door number 3? 1
You have selected a new kitchen by Matilda Kitchens!
The retail value of your prize is $2500.00
```

assuming that the user entered 1.

This program is a candidate for using a switch, since the nested if logic involves comparing the same value (the variable, door) to a number of different constants. Using switch, the program would look like this:

```
main()
{
    int door;
    double retail;

    printf("Welcome to \"Why Not Make a Deal?\"\n");
    printf("Would you like to look behind\n");
    printf("door number 1, door number 2 or door number 3? ");
    scanf("%d", &door);
    printf("You have selected ");
    switch (door) {
        case 1:
          printf("a new kitchen by Matilda Kitchens!\n");
          retail = 2500;
          break;

        case 2:
          printf("your very own.....\n");
          printf("...DONKEY! And a shovel, too!!\n");
          printf("Too bad. Better luck next time.\n");
          retail = 7.50;
          break;

        case 3:
          printf("a BRAND NEW CAR!!\n");
          printf("The fabulous 4-door\n");
          printf("Liability by Generic Motors!\n");
          printf("Congratulations!\n");
          retail = 15700;
          break;

        default:
          printf("a door that doesn't exist\n");
          retail = 0;
    }
    printf("The retail value of your prize is $%.2lf\n", retail);
}
```

Note that the only values allowed after the word "case" are constants, not variables or calculations. This means that switch is only useful when there are certain specific values to which you want to compare the original expression. You cannot specify ranges, although you can have several "case <u>constant</u>:" specifications, one after the other with no statements in between them. If you do this, any of the listed constant values will trigger the execution of the same code: the statements that start after the last of the back-to-back cases. (In fact, it is to allow this that the break statement is required).

Note also that you may omit the "default" section altogether. If there is no "default" section, then nothing happens if there is no match.

Another thing to note is that although braces surround the whole collection of statements in a switch, braces are not required to group together the individual sections.

And keep in mind that many beginners using switch for the first time will forget the break statements, which will cause one case to run into the next, probably producing strange results. So if you write a program using switch, and it compiles fine, but doesn't seem to work right, first look for missing break statements.

The switch statement is not as flexible as a nested if statement. But in situations where switch can work, it is actually more efficient than an equivalent nested if. Some people find switch easier to read, as well.

Choosing Appropriate Control Structures

The formal term for statements executing one after the other is sequence. To be complete, a programming language needs two ways to modify the normal sequence of statements: selection, the ability to select which statements will execute based on current conditions, and iteration, the ability to repeat a sequence of statements as much as necessary.

We have seen that C has three ways of performing selection (if, if/else and switch) and three ways of performing iteration (while, do/while and for). If you find the choices daunting, keep in mind that the basic if and while statements are all you really need. We will attempt to use the most suitable control statements in our examples from now on. However, it is not wrong to use different ones from the one we will use. If in doubt, always use a simple if for selection and a simple while for looping. As you see more examples and do more programming, you will start to use the others more and more. Eventually you will become comfortable at choosing an appropriate control structure for your programs.

Don't Believe Everything You Read

A final word concerning control structures has to do with what you may read in other books. You may read that there are two statements, goto and continue, that also alter the normal sequence of execution of a program, and that the break statement can be used to exit early from loops. It is generally accepted that using goto and continue, as well as using break anywhere except in a switch statement, leads to code that is hard to read and harder to fix than if you avoid these practices. So, please, ignore those parts of the books you read, at least until you are an advanced programmer!

Chapter 4. Modularity

We have seen how to control the computer using the techniques of selection and iteration. We have also seen how quickly the coding of even a simple program can get complex. The major technique for dealing with this complexity is to subdivide a program into a series of smaller programs, sometimes called modules. The name for this technique is modularity.

While some programming languages have a variety of module styles to choose from, C has just one type of module you can code: a function.

In algebra, we might say something like:

let $f(x)$ be $1.5x + 5$

Here, f is a function of x, and we can establish equations, such as

$y = f(x)$

which you might recognize as an equation for a straight line with slope 1.5, and which crosses the y-axis at 5.

Similarly, we can mathematically define a function of 2 variables, such as

let $g(x, y)$ be $(x - 2)^2 + y^2$

In this case, the equation

$9 = g(x, y)$

is the formula for a circle centered at (2, 0), with radius 3.

Now, C statements are different from mathematical equations. A C statement is executed at a certain point in time, usually setting some variable(s) based on values currently stored in other variables. A mathematical equation, on the other hand, is usually a formula to be used later, when some of the variables (which represent "unknowns") are given values. But just as C assignment statements are modelled after mathematical equations, so are C functions modelled after functions from algebra.

In C, we could write:

```
double f(double x)
{
    return 1.5 * x + 5;
}
```

to define a function just like the mathematical f(x) above. The first line is called the <u>header</u> line, and tells the compiler about the function we are going to write. The first thing on the header line is the data type of the value which the function is going to calculate. In this case, our function is going to calculate a double. The next thing (after whitespace) is the name we want to give to the function, in this case, f. The rules for choosing a function name are actually the same as the rules for choosing a variable name (may only contain letters, digits and underscores, and may not begin with a digit, and may not be a C language keyword). Following the name of the function is a set of parentheses, containing a <u>parameter</u> list: a series of variable definitions, separated by commas, which will be given to the function by the program that calls it. In this case, our function, f, will be given one double value, which we are going to name x. After the parameter list is a code block, contained in braces ({}), which is the logic we want to be used whenever the function is invoked.

One of the statements in a function will be the <u>return</u> statement, where after the word return is some expression. The value of the expression following the word return will be the value sent back to the program that used the function. The data type of this expression, by the way, is what was specified at the beginning of the function's header line. If the function contains logic (as opposed to this very simple function which only returns the result of a single calculation), there can be several return statements buried within ifs and whiles, for example. The first return statement encountered will cause the function to immediately stop and send the specified value back to the calling program. Good programming style, however, is to have no more than one return statement. This one return statement will necessarily then be the last statement in the function.

A C implementation of the g(x, y) function, from above, would be:

```
double g(double x, double y)
{
    return (x - 2)*(x - 2) + y*y;
}
```

Here, there are two parameters, which we have called x and y and which are both doubles. Note how a comma (rather than a semi-colon) separates the parameter definitions. Note also how there is no C arithmetic operator to raise a value to a certain power; we have simply used multiplication to compute the squares of (x - 2) and y.

As a more practical example, let us suppose that we want to compute compound interest on an investment, where the interest is compounded annually. The mathematical formula for the future value of an investment using compound interest is

$$p(1+r)^t$$

where p is the original investment amount (the principal), r is the interest rate for one period, expressed as a factor (e.g. 7% would be 0.07), and t is the number of time periods for which the principal is to be invested.

We want a program that will repeatedly ask the user for a principal amount, an annual interest rate and the number of years to invest, and shows what the value of the investment will be at the end. We will use a function, called invest, which is given the principal, the rate and the number of years:

```
double invest(double principal, double rate, int time)
{
    int i;

    for (i = 1; i <= time; i = i + 1)
        principal = principal * (1 + rate/100);
    return principal;
}

main()
{
    double start, end, rate;
    int years;

    printf("Enter an investment amount (or 0 to stop): ");
    scanf("%lf", &start);
    while (start > 0) {
        printf("Enter rate (e.g. 10.5 for 10.5%% per annum): ");
        scanf("%lf", &rate);
        printf("Enter years to invest: ");
        scanf("%d", &years);
        end = invest(start, rate, years);
        printf("%.2lf invested at %.1lf%% for %d years: %.2lf\n",
         start, rate, years, end);
        printf("Enter another investment amount (0 to stop): ");
        scanf("%lf", &start);
    }
}
```

A sample run of this program is:

```
Enter an investment amount (or 0 to stop): 10000
Enter rate (e.g. 10.5 for 10.5% per annum): 5
Enter years to invest: 3
10000.00 invested at 5.0% for 3 years: 11576.25
Enter another investment amount (0 to stop): 2000
Enter rate (e.g. 10.5 for 10.5% per annum): 10
Enter years to invest: 2
2000.00 invested at 10.0% for 2 years: 2420.00
Enter another investment amount (0 to stop): 0
```

There are several points to note about this example. First, notice how the program is now split into two parts: invest and main. In fact, main is a function, and the rule is that a program may have as many functions as you want, as long as there is one named main. The execution of the program always starts at the beginning of main, regardless of the order in which the functions are written.

Second, note how the main function calls the invest function mid-way through the while loop. Three variables from main (start, rate and years) are passed to invest, which does some calculations with them and sends back a value, which gets put into main's variable, end. The values that main passes to invest are called the arguments for the function call.

Look at the invest function now, and we see three variables declared in its parameter list (principal, rate, and time). These correspond to the variables passed by main. In fact, main's variable, start, will be copied into invest's variable, principal. Similarly, main's rate is copied into invest's rate, and main's years is copied into invest's time. Notice that the names of the arguments (where the function is called) are independent of the names of the parameters (where the function is written). The arguments are copied into the parameters based on position in the list, not by name. In fact, the arguments need not be variables, but may be constants or expressions; when the function is called the values of the arguments are copied into the parameters before the logic of the function is executed.

This concept of parameter passing allows one program (corresponding to one main function) to be split into many functions, where each function is a little self-contained program with a very limited scope, and where the parameters of the function, along with its return value, indicate the only data that are shared with the calling function. The fact that the functions are self-contained makes it possible to test them separately from the main, which can be a big advantage when writing a large and very complex program. It also allows the functions to be used in other programs that have similar needs. For example, the following main() function also uses the same invest function:

```
main()
{
    double percent;

    printf("Comparison of 5-year returns at different rates\n");
    printf("    Rate     Profit (per Thousand $)\n");
    printf("    ----     -----------------------\n");
    for (percent = 3.5; percent < 9.6; percent = percent + 0.5)
        printf("    %.2lf           %.2lf\n", percent,
         invest(1000.0, percent, 5) - 1000);
}
```

yet produces the following very different output:

```
Comparison of 5-year returns at different rates
    Rate     Profit (per Thousand $)
    ----     -----------------------
    3.50          187.69
    4.00          216.65
    4.50          246.18
    5.00          276.28
    5.50          306.96
    6.00          338.23
    6.50          370.09
    7.00          402.55
    7.50          435.63
    8.00          469.33
    8.50          503.66
    9.00          538.62
    9.50          574.24
```

In this second program, the value returned by invest is used as part of a calculation which is then printed. Note that two of the arguments to invest are constants. (Of course, the source file for this second program would have to have both invest and main in it).

Function Prototypes

While these last two programs have two functions each (invest and main), a typical program will have many functions. The hope is that as a program grows more complex, the number of functions increases rather than the complexity of the functions. Since programs always start at the beginning of the function named main, and since there will be many functions in a program, it is common practice to place the main function first, so that it will be easy to find.

A problem with putting main first is that the compiler will not be able to tell if you are using any functions correctly in main until those functions are defined. At that point, which is after the compiler is finished with main, the compiler may give you a message telling you that the function is wrong (not that main is wrong), or may simply create an incorrect program without any error messages. To avoid these difficulties, you may declare the functions (which means to tell the computer the correct usage of the functions), rather than fully defining them (which means to write the code for the functions) at the top of the program, then code the main function, then code the remaining functions in any order you like (possibly in alphabetical order by function name to make them easy to locate). To declare a function without defining it, simply write its header line, and put a semicolon at the end of it rather than putting a code block with the function's logic. This declaration of the function is called the function's prototype.

The #include Directive

From the very beginning our programs have been calling functions, without our knowing it. Both printf and scanf are functions that were written not by us, but by the programmers that created the compiler. These functions are part of the standard C library, which is a collection of functions that are defined by the ANSI (American National Standards Institute) committee governing the standards for the C language. Every C compiler comes with the standard C library, and as long as you use only standard library functions and functions you write yourself, you will be able to move your programs from one machine (using one C compiler) to another (with a different C compiler) without difficulty.

So all our programs thus far have been using functions without properly declaring them. This is not a good thing. It has been a carefully constructed "coincidence" that our examples haven't had problems using printf and scanf. Every C compiler comes with a set of files, called header files, which contain, among other things, function prototypes for the functions in the standard C library. The printf and scanf functions, for example, are prototyped in a file named "stdio.h". (As we learn other standard library functions, we will also learn the names of their header files). In order to allow the compiler to ensure that we are using standard library functions correctly, we can copy the appropriate header file into our program, using a line like

```
#include <stdio.h>
```

The #include directive (which is not, by the way, a C statement) is an instruction to the compiler telling it to copy another file into the current file before translating it into machine language. (Later, we will see another directive, which also begins with # and which also causes the compiler to do something before the translation step). The angle brackets (< and >)

around the header file name indicates that the file is part of the standard library and should be located where all the library header files are. (You may use double quotes, ", instead of the angle brackets to #include a file in the same location as the file being compiled).

Program Comments

As programs become larger, and are consequently split into more and more pieces, it becomes handy to be able to explain parts of the code, in English. The symbols /* and */ are used to denote the beginning and end, respectively, of a comment. A comment is simply ignored by the compiler and can be used to give verbal descriptions of what is going on in the code. A comment may be inserted anywhere whitespace may be, although most programmers only place comments on lines by themselves or at the end of a line.

A good practice to follow, is to put a comment at the beginning of a program, briefly describing the program as a whole. Also place a comment before each function (except main) describing the specifics of that function. Anywhere else within the code where there is complex or confusing code should also have a clarifying comment.

Putting it all together

A typical program organization is to have the following pieces, written in the following order:

1. A comment describing what the program does, and who wrote it.

2. #include directives for all functions from the standard library (or from other function libraries that may have been purchased or written).

3. function prototypes for all functions, other than main, that appear in the source file.

4. the definition of the main function.

5. the definition of the remaining functions, each preceded by a comment.

To apply all these guidelines to the last program, we would get something like this:

```
/*********************************************
* Display a table of investment profits over a  *
* 5-year period, using different interest rates *
* Written by: Evan Weaver      October 9, 1996 *
*********************************************/

#include <stdio.h>
double invest(double principal, double rate, int time);

main()
{
    double percent;

    printf("Comparison of 5-year returns at different rates\n");
    printf("    Rate     Profit (per Thousand $)\n");
    printf("    ----     ----------------------\n");
    for (percent = 3.5; percent < 9.6; percent = percent + 0.5)
        printf("    %.2lf           %.2lf\n", percent,
         invest(1000.0, percent, 5) - 1000);
}

/* returns the future value of "principal", if it is
*  invested for "time" compounding periods, at an
*  interest rate of "rate" percent per period.
*/
double invest(double principal, double rate, int time)
{
    int i;

    for (i = 1; i <= time; i = i + 1)
        principal = principal * (1 + rate/100);
    return principal;
}
```

<u>What is main, anyway?</u>

You may be wondering why there is neither a return data type, nor any parameters, in the header line for the main function. The main function is the only function in your program that your own code does not call. It is the operating system software of the computer that calls your main function. Since the purpose of the return value and the parameters is to communicate with the calling program, the return value and parameters of main would be used to communicate with the operating system. At this level of programming, we won't be learning how to communicate with the operating system through the main function, and so all our programs will simply ignore the return value and parameters for main.

In fact, there is a return type (int) and a parameter list (int argc, char *argv[]) that we could use in our main functions. The return value from main is given back to the operating system (which is the program that called the main, based on some user action), and most operating systems have a way of checking this return value. It is common practice for main to return the integer value 0 unless you have some reason to return a non-zero value, but as long as you don't check the return value (by, say, calling the program from an operating system script that performs such a check), the value you actually return from main is irrelevant. By ignoring the return value, our programs have been returning a garbage value to the operating system. It would be "cleaner" if we returned 0 at the end our main functions, but unless we are going to check the value at the operating system level, it really isn't necessary. The parameters for main, the syntactical details of which you are not yet ready to understand, represent the things typed at the operating system level (along with the program's name) that caused the program to be run. By ignoring the parameters, as our main functions have all done, we are simply not using this information.

Global Variables - Don't Try This At Home

Most books describe the use of global variables. A variable can be defined before any of the functions, including before main itself. This variable can then be used by any of the functions that follow. Such variables are described as global because they are known to all the functions. This is in contrast to variables that are defined within a function, or in the header line of a function, which are local to the function, and aren't known to the other functions.

At first glance, global variables might seem like a useful thing, and could cut down dramatically on the parameter passing that might be required. However, history has shown that the use of global variables to avoid passing parameters leads to code that is less flexible, hence harder to maintain, than programs that restrict themselves to local variables. Specifically, the design work, that goes in to deciding what data needs to be passed to a function in order for the function to do its work, actually results in programs that have a better overall structure. Furthermore, the functions themselves can often be re-used in other programs without modification, whereas a function that uses global variables generally needs to be changed to work in another program with different global variables. For these reasons, we will avoid the use of global variables in these notes, much as we will avoid the use of the goto and continue statements as noted in the previous chapter.