Foundations of Programming Using C

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

July 2006

©1996-2006 by Evan Weaver and Seneca College. Effective 2014, this work is made available under the Creative Commons Attribution 4.0 International License. Visit http://creativecommons.org/licenses/by/4.0/ for details on how you may use it.

Foundations of Programming Using C
Table of Contents

Copyright 1996-2006 by Evan Weaver and Seneca College

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 memory. 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 interface circuits, 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 terminal, 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 file. 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!

The Programming Process

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 text editor (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 source file, 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 compiler, which will make another file, called an executable program, 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 interpreter 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 syntax 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 structured programming. Throughout the 1980s a more sophisticated view of programming, called object oriented programming, 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 character string.

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 newline 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
```

Exercise 2.1: 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.

Input, Output and Variables

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 defining a variable. 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 whitespace 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.

Assignment Operator

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 assignment 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.

Exercise 2.2: 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))
```

Expressions, Statements and Code Blocks

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 expression 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 compound 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 infinite loop, 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.

Exercise 3.1: 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 nested ifs:

```
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.

DeMorgan's Law

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 constant:" 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

$$\text{let } g(x, y) \text{ be } (x - 2)^2 + y^2$$

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 header 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 parameter 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 return 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;
}
```

What is main, anyway?

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.

Chapter 5. Addresses and Pointers

While functions are very useful for breaking a program into small, manageable pieces, there is one limitation inherent in the syntax for functions: a function only has one return value, and so can only send one thing back to the calling program through the return value.

Suppose that we wanted to write a function, called hours_and_minutes, that would break a number of minutes (say, 147) into hours and minutes (2 hours and 27 minutes in this case). Here, we would want to pass the function an int (the total minutes), and have it send back two ints (the hours and the minutes left over). We might be tempted to write something like:

```
/* note: this function isn't correct! */
int hours_and_minutes1(int total, int minutes_left)
{
    minutes_left = total % 60; /* this just gets "lost" */
    return total / 60;
}
```

where the function sends back the hours as a return value and passes an extra variable to hold the minutes. The problem with this is that the parameters, total and minutes_left, are copies of the actual function arguments. When this function changes minutes_left, it is changing a copy of the second argument, not the argument itself. When the function returns the number of hours, the two parameter variables, which are local to the function, simply disappear.

Addresses

There is, however, a trick that can make this work. Every variable of a program is stored in the computer's memory somewhere. Computer memory is really a series of numbered storage locations, and machine language programs access these locations by specifying the number of each location. Every variable, then, has a memory location number. If we define a variable, say,

```
int x;
```

then one thing the compiler does when translating the program is to associate the name, x, with some memory location number, say, location 3204. (The actual location for each variable is dependent on such things as how many variables there are, how big the program is, how many programs are currently loaded in memory, and what other things are currently stored in memory).

There is a C operator, &, which can be used to find out what the memory location of a variable is, by placing the & just before the variable's name. We have seen this operator already, in our calls to the scanf function such as:

```
scanf("%d", &x);
```

What we are doing here is passing to scanf the memory location of the variable x. The scanf function will wait for input data and then place it in that memory location. When scanf is finished, the variable x will have a new value in it. This is the trick we need: in order to pass a variable so that it can be changed, we need to pass its memory location, not the variable itself.

In C, the memory location of a variable is usually called the address of the variable. The expression

```
&x
```

is commonly read as "the address of x" (or "the location of x").

Pointers

In order to write a function that will receive an address as an argument, we need to know how to declare a parameter that will be able to store this address. To do this, we first need to learn another operator, *, called indirection. This operator is used to access a specific memory location. For example, if the variable p stored the address of an int variable, then:

```
*p = 6;
```

would set that int variable to 6. The expression

```
*p
```

is commonly read as "the variable whose address is p" or "the variable to which p points". For this reason, variables which store addresses are usually called pointers, and the indirection operator (*) is also called pointer resolution or pointer deference. A pointer is declared

```
data type *variable name
```

where data type refers to the data type of the variable being pointed to.

For example, the variable p, above, would have been declared:

```
int *p
```

since p points to a variable of type int. Literally, this is read as "the variable, to which [the variable] p points, is an int". What is really happening here is that the variable p is being declared, and the data type of p is "int *", even though it looks like the * is part of the variable name.

Beginners often find it confusing that * is used to declare a pointer, which will store an address, when the * operator is used to dereference a pointer and it is the & operator that is used to obtain an address. If the literal translation of a pointer variable declaration (given in the preceding paragraph) doesn't make sense to you, then think of the data type in a variable declaration as a mold for the variable being declared. The symbol *, which is the inverse of &, is used for much the same reason as the fact that a mold is always the inverse of what you pour into it.

Another reason beginners get confused by * is that * is also used for multiplication. There are a few operators in C which have different meanings depending on whether they are unary (have one operand) or binary (have two operands). Binary *, where there is an expression on either side of the *, is multiplication. Unary *, with an expression on the right but not on the left, is indirection. Particularly confusing are statements such as:

```
*p = 2 * *p;
```

where the variable pointed to by p is being doubled. Here, the first * clearly has nothing on the left, and so must be unary *. The second *, just as clearly, has an expression on either side, and so must be binary *. The third * is murkier: on the right is an expression (p), but on the left is another operator (the binary *), not an expression, so it must be unary *. Note that unary * has higher precedence than both binary * and =.

Note that some people use the term pointer to be synonymous with the term address, and then speak of pointer variables, which we refer to as pointers, and pointer values, which we refer to as addresses.

Now that we know how to declare pointers, we can fix our hours_and_minutes function:

```
int hours_and_minutes2(int total, int *pminutes_left)
{
    *pminutes_left = total % 60;
    return total / 60;
}
```

(Many programmers, as a habit, always give a pointer a name that

begins with p, so they don't forget to use the * later).

A main() that calls this is:

```
main()
{
    int tmin, hr, min;

    printf("Enter total minutes for videotape: ");
    scanf("%d", &tmin);
    hr = hours_and_minutes2(tmin, &min);
    printf("That is %d hours and %d minutes\n", hr, min);
}
```

Observe carefully how the calling function (main) uses & to send the address of min to hours_and_minutes2, whereas the code for hours_and_minutes2 uses * to resolve the pointer that receives this address.

As confusing as pointers may seem, the general rule is really quite simple: if a function is going to change a variable, it needs to be passed the address of that variable. The calling function supplies an address as an argument (usually by using &) and the function itself resolves its corresponding pointer parameter by putting * before every occurrence of that parameter, including its declaration in the function header line.

Void

The hours_and_minutes2 function wants to send two things back to the calling function. We sent one through the return value, and the other through a parameter by a using a pointer. You might wonder how we decided that. Well, you can only send one thing back through the return value, and all other values to be sent back MUST be by pointers. We simply "flipped a coin" to decide which one to return. In a situation like this, where there is no compelling reason to make any one of the values being sent back into THE return value, you may choose to have NO return value at all. This is done by using the word void in place of the function return type (in the function's header line).

A function that returns nothing can only be used as a statement, not as an expression that is part of a statement. (Recall that an expression has a value associated with it, and a function returning "void" has, by definition, no value associated with it). This can be advantageous in situations, such as hours_and_minutes, where there is no single value that we wish to associate with the function's name.

In our last main, for example, both hr and min are being changed, but by completely different mechanisms. The variable

"min" is changed by the hours_and_minutes2 function, but "hr" is actually changed by main, using information sent back from hours_and_minutes2. Rewriting the function, and the program that calls it (for the last time) will make the treatment of the two variables more "symmetric":

```
#include <stdio.h>
void hours_and_minutes(int total, int *phrs, int *pmin);

main()
{
    int tmin, hr, min;

    printf("Enter total minutes for videotape: ");
    scanf("%d", &tmin);
    hours_and_minutes(tmin, &hr, &min);
    printf("That is %d hours and %d minutes\n", hr, min);
}

/* splits "total" minutes into hours and minutes */
void hours_and_minutes(int total, int *phrs, int *pmin)
{
    *pmin = total % 60;
    *phrs = total / 60;
}
```

Note how there is no return statement in hours_and_minutes. This corresponds to the fact that the return type for the function is void. If, for some reason, you want to prematurely exit a "void" function, you can issue the statement

```
return;
```

with no value. However, good programming style is for each function to have a single exit point at its end. A return statement right at the end of a "void" function is redundant (since it will return anyway), so most void functions will simply not have a return statement at all.

Just as void may be substituted for the return value of a function, void may also be substituted for the parameter list of a function in its header line, to indicate that the function does not need to be passed anything. Any code that calls a function with a void parameter list will simply leave the parentheses empty when it calls the function. We will see examples of functions with no parameters in the next chapter.

Chapter 6. Filling In The Gaps

So far, we have studied fundamental aspects of the C language, but there are a lot of little, less important features of the language which can make programming easier, or can make the programs you write more robust. In this chapter we will have a look at some of the more commonly used of these features, and explore in greater detail some things at which we have already looked.

Symbolic names for constants

Often, a program may use the same constant value over and over again. It is a good idea in such situations to give a name to the constant value, and use the name throughout the code. One way to do this is to use the directive, #define. Earlier (in Chapter 3), we wrote a program that has the user guess a number. The number the program was expecting was 42, and the constant value 42 was used throughout the code. Instead, we could put the directive

```
#define ANSWER 42
```

before the main function. Then throughout the code we would use ANSWER instead of 42. This has three main advantages: (1) if we want to change the program so that is looks for, say, 139 instead of 42, we only have one line of the program to change [the #define], (2) the likelihood is reduced that we might accidentally code the wrong constant value in one place, making the program work incorrectly, and (3) by giving the constant value a name, we help describe what the constant represents to someone else reading the code.

The syntax for #define is

```
#define name anything you like
```

The rules for the name are the same as the rules for a variable or function name. However, many programmers choose to use all capital letters for a #define name (and not all capital letters for variables and functions), so that it is clear when reading the code whether the name is a #define or a variable.

What #define does is go through the entire program and replace every occurrence of the name with whatever follows the name on the #define line. Just as the #include copies another file into the current file before the current file is compiled, the #define substitutions are done before the file is compiled. While the text that follows the name on a #define directive may be anything that fits on a line, it is common practice to use #define to give a name to a constant value. Making a name for an entire expression or statement, even though it can be done, is

generally considered to be a good way to make your program unreadable, and probably should be avoided.

In particular, note that #define is a directive, not a C statement, and does not end with a semi-colon. In fact, if you did end a #define with a semi-colon, the semi-colon would become part of what gets substituted for the name. A #define ends with the end of the line on which it is written, rather than ending with punctuation like a C statement.

While #define directives may be placed anywhere in a program, most programmers place them at the start of the source file, just after the #include directives. This makes them easy to find.

Incrementation

Adding one to a variable is something programmers often need to do. It is so common that C provides an operator, ++, that increments a variable by one. Thus,

```
x = x + 1;
```

and

```
++x;
```

both do the same thing. In fact, the ++ may either precede the variable name or the variable name may precede the ++, so

```
x++;
```

also does the same thing. There is a difference between putting ++ before or after the variable name when it is used in a more complex statement. The rule is that if the ++ precedes the variable name (sometimes called a pre-increment), then the variable will be incremented before its value is used in the statement. If the ++ is after the variable name (called a post-increment), then the variable's value will be used in the statement before the variable is incremented. For example, the program:

```
#include <stdio.h>
main()
{
    int n, m;
    m = 6;
    n = 2 * ++m;
    printf("n is %d and m is %d\n", n, m);
    n = 3 - m++;
    printf("and now n is %d and m is %d\n", n, m);
}
```

outputs:

```
n is 14 and m is 7
and now n is -4 and m is 8
```

The variable m, which starts out at 6, is incremented to 7 before its value is multiplied by 2 (making 14) to be assigned to n. In the second calculation, m's value (which is now 7) is subtracted from 3 (making -4) to calculate the new value for n, and m is incremented to 8 after this is done.

Note that you are cautioned against using a variable elsewhere in a statement in which that variable is incremented. The C language standard does not define the exact order in which the incrementations will be done in a statement such as:

```
m = (m++ + ++m) * m++; /* don't do this kind of stuff! */
```

(For example, it is not clear whether the first post-increment will be completed or not before the first pre-increment). The exact operation of such a statement is dependent on the individual compiler program that you are using, and as such should be avoided even though your program may produce correct results when tested.

Incidentally, regular assignments, using =, can also be used as part of another statement. The statement

```
n = 2 * ++m;
```

works the same as

```
n = 2 * (m = m + 1);
```

for example. The only advantages of the pre-increment operator over using = are that (1) it is more compact (hence quicker to write), (2) on some systems it is marginally more efficient and (3) ++ has very high precedence (higher than multiplication) compared to assignment (which has the lowest precedence of the operators we have look at so far), and so may make some parentheses unnecessary.

There are also two operators, pre-decrement and post-decrement, which use the symbol -- to subtract one from a variable in the same way that ++ adds one. As an example, the statement

```
printf("%d, %d\n", n--, --m);
```

displays n before subtracting one from it, and displays m after subtracting one from it.

A generalization of incrementation

Just as C provides a special syntax that allows you to add one to a variable, there is a more generalized set of operators that let you modify a variable based on its current contents. An example is

```
n += 4;
```

which is defined to be the same as

```
n = n + 4;
```

The general rule is that if you want an expression of the form

variable = variable operator value

then it can be written as

variable operator= value

where variable represents some variable name, operator is a suitable operator (the arithmetic operators we have studied are all suitable), and value is some expression. A few examples are:

```
n *= 3;       /* triples n                */
x /= 2;       /* cuts x in half           */
abc -= x * y; /* subtracts x * y from abc */
```

In the last of these, note how the precedence of these new operators is low: they have the same precedence as the normal assignment operator (=).

Initialization vs. Assignment

A variable can be given a value at the same time that it is defined, by specifying the initial value for the variable in its declaration with =. This form of setting a variable is usually called initialization to distinguish it from assignment, which involves setting a variable that has already been defined (by using =, ++, --, +=, *=, etc.). An example of variable initialization is the declaration:

```
int x = 6, y, z = 12;
```

Here, three int variables are being defined: x, y and z. The variable x is set to be 6, and z is set to be 12. The variable y is not set to any particular value; presumably the program will set it later.

The parameter variables in a function's header line may not be initialized in this manner. Rather, they are automatically

initialized when the function is called, to be copies of the arguments passed to the function.

Parameters need no names in a prototype

The header line of a function contains a list of parameters that are to be supplied to the function when it is called. We have already seen that to declare a function without actually writing it, we may supply a copy of the function's header, with a semi-colon at the end. We called this a function prototype.

The main purpose of a function prototype is to let the compiler know how many parameters, and what kinds of parameters, a function expects, so that it can make sure that the function will be used correctly. The names that the parameters might have are immaterial, however, since the parameters are only referred to by name within the code for the function itself. Consequently, when you prototype a function, you may omit the names of the parameters. For example, a function that has the header line:

```
int foo(double x, int n)
```

may be prototyped as

```
int foo(double x, int n);
```

or, more simply, as

```
int foo(double, int);
```

When defining the function, of course, the header line should have names for the parameters, so that you can refer to them in the code for the function.

Sometimes, programmers will leave the variable names in the prototype for documentation purposes. For example, the prototype

```
void set_date(int day, int month, int year);
```

makes it clear what the parameters actually are, but

```
void set_date(int, int, int);
```

does not. The presence of the names, if they are well chosen, may enable the programmer on occasion to avoid having to look up the actual definition of the function to see exactly what the parameters represent.

Casting data to different types

Suppose a function has three variables

```
int n = 7, m = 5;
double x;
```

and we execute the statement

```
x = n / m;
```

Although we might expect x to get the value 1.4, in fact x will get 1.0. This is because the division is a division between int variables, and division between ints is defined to be itself an int. It does not matter to the compiler that the destination for the calculation is a double; the compiler does not look that far ahead. If we did want x to be 1.4, we could change one of the variables (either n or m) to be of type double, to force a double division rather than an int division. But a better solution is to use a cast, which allows you to convert a value from one type to another.

You can cast an expression by preceding the expression with a data type in parentheses. For example

```
x = (double)n / (double)m;
```

would create a double from the int value in n, and another double from the int value in m, and then divide the two doubles. Actually, since only one side of the division needs to be double to force the division to be double, it would be sufficient to do

```
x = (double)n / m;
```

Note that casting has very high precedence (the same as ++, in fact), so that n is cast to a double before that double is divided by m. Note also that something like

```
x = (double)(n / m); /* Wrong! this cast is too late */
```

would perform an int division first (getting 1) and then, too late, convert it to double (giving 1.0). In this case we aren't doing anything the compiler wouldn't have done by itself.

Finally, note that casting a variable does NOT change the variable. It simply makes a temporary expression which contains a copy of the value in the variable, only converted to another data type.

Conditional Expression

It is not uncommon to have an if/else statement where each of the sub-statements are simple statements that are almost identical. For example, consider the statement:

```
if (x < y)
    z = 3 * x + y - 2;
else
    z = 4 * x + y - 2;
```

Such a statement can be simplified to s single simple statement by using a conditional expression. A conditional expression has three parts: a condition, an expression to use if the condition is true, and an expression to use if the condition is false. The condition is given first, followed by a question mark (?), followed by the expression to use if the condition is true, followed by a colon (:), followed by the expression to use if the condition is false.

The if/else shown above could be written as

```
z = (x < y ? 3 : 4) * x + y - 2;
```

The conditional expression operator (?:), has fairly low precedence (lower than || but higher than =), so the parentheses are needed here.

A statement written using the conditional operator may be harder to read than the equivalent if/else, but it is more compact, and the fact that most of the statement isn't duplicated helps to reduce the chance of a programming error due to a typographical error. So, while many programmers avoid the conditional expression for reasons of clarity, others embrace it for brevity and robustness.

So far, this chapter has been a collection of nice little extra, but not essential, features of the C language. None of this chapter's topics has been crucial to the act of programming. The next topic, however, is quite important.

Character Data

All the data our programs have been working with have been numeric. As you are probably aware, computers can manipulate text data just as well. The way computers, which are basically devices that perform numeric computations, deal with text data is simply by treating each character of text as if it were a number. The most widely used scheme for codifying text characters is the ASCII (American Standard Code for Information Interchange) table. This is simply a list of characters that can

be stored in the computer, along with the number that will be used to represent that character.

For example, in ASCII, an upper case A is stored as the number 65, uppercase B is 66, and so on. Similarly, a lower case a is stored as 97, b is 98 and so on. Each punctuation character has a number, as do the digits. (The ASCII code for the digit 0, for example, is the number 48, and the code for the digit 1 is 49). See Appendix D for the complete list of ASCII codes.

For the most part, it doesn't matter what number is associated with what character; what is critical is that there is a scheme that everyone follows. In fact, there are other schemes in use. Most notable is IBM's EBCDIC (Extended Binary Coded Decimal Interchange Code), which is used on large IBM mainframe computers and AS/400s, and a newer Unicode, which attempts to represent characters from all human languages (such as Russian or Japanese), not just English. Unicode is actually an extension of ASCII - the English characters have the same numbers associated with them.

The idea is that with a standard code, manufacturers can build text-based equipment, such as terminals and printers, that use the code and can be made to work with computers that also understand the same code. Of course, if pieces of equipment use different codes, there will be problems; a terminal that uses EBCDIC and is designed for use with an IBM mainframe computer cannot simply be connected to an ASCII-based UNIX computer.

C provides a data type specifically for text data. The type

```
char
```

can be used to declare variables that store one character of data. Character constants are formed by placing the desired character in single quotes. For example, an uppercase A is

```
'A'
```

and a new-line character is

```
'\n'
```

Note that the rules for non-printable characters are the same as what we saw for character string constants in chapter 2. Also keep in mind that C makes a large distinction between character strings (several characters in between double quotes) and individual characters (single characters in single quotes). In this chapter, we are only going to learn how to handle individual characters. The next chapter will show us the extra complexities involved with working with strings.

The format specification to use with printf and scanf to indicate a piece of char data is %c.

The following program performs a temperature conversion from Fahrenheit to Celsius, and asks the user whether or not to repeat the process.

```
#include <stdio.h>
main()
{
    double fahr;
    char again;

    do {
        printf("Enter degrees F: ");
        scanf("%lf", &fahr);
        printf("In Celsius that is %.1lf\n",
            (fahr - 32) * 5.0 / 9.0);
        printf("Another conversion? (y/n) ");
        do {
            scanf("%c", &again);
        } while (again != 'y' && again != 'n');
    } while (again == 'y');
}
```

A sample run of this program is:

```
Enter degrees F: 32
In Celsius that is 0.0
Another conversion? (y/n) y
Enter degrees F: 100
In Celsius that is 37.8
Another conversion? (y/n) y
Enter degrees F: 72
In Celsius that is 22.2
Another conversion? (y/n) n
```

Character input introduces a set of unexpected complexities, since EVERY piece of input can be viewed as valid character data, including such things as the newline at the end of each line of input. The inside do/while loop is there because when we first try to input a character, we will in fact get the new-line that was after the "degrees F" number entered by the user. The purpose of the loop is to keep reading characters until we get either a y or an n. The reason we haven't had these problems before is that the numeric formats (%d, %ld, %f and %lf) are designed to skip over any whitespace (spaces, tabs or newlines) in the input. The %c format will NOT automatically skip these sorts of things, simply because these are the sorts of things that can be stored in a char.

This program works fine as long as the input data is correct.

But something as simple as the user typing in "yes", instead of just "y", will cause problems with our unsophisticated logic.

A detailed look at scanf

To write programs that handle input well, we will have to learn more about how scanf works. The first parameter to scanf is always a format string, which contains format specifications as well as other characters. The characters that are not format specs must be present in the user's input and will be ignored once they are checked. The remaining parameters for scanf are one address for each format spec. These are addresses of the variables to be filled by scanf, and are assigned on a left-to-right basis. For example,

```
scanf("%d,%d", &n, &m);
```

will expect the user to enter two ints, separated by a comma. The first number will be placed in the variable n, and the second number will be placed in m. The comma entered by the user is simply thrown out, although the scanf would not work right if the user forgot the comma.

There are two notable exceptions to these rules:

1. If a space is in the format string, then scanf will skip not just a space, but any collection of whitespace. For example, the inside loop of our Fahrenheit/Celsius program could have been replaced with

   ```
   scanf(" %c", &again);
   ```

   where the space before the %c would cause scanf to ignore the newline (which is an example of whitespace) after the previously entered number.

2. If there is an asterisk (*) between the % and the data type indicator in a format spec, then scanf will expect that kind of data, but instead of placing it in a variable, will discard it. This kind of format spec does NOT have a matching address. For example, another way to replace the inside loop of our program would be to

   ```
   scanf("%*c%c", &again);
   ```

   where the first %*c means to ignore one character (the newline) and place the next character in the variable again.

The scanf function processes the format string from left to right. It keeps going through, either filling up variables or throwing away matching data, until it is finished or finds data

that does not meet its requirements, whichever comes first. If it runs out of data before the end of the format string, it will simply stop and wait for another line of input, and then keeps going. If there is any input left over after scanf is finished, then that leftover input is kept around for the next scanf. (This explains why the newline after numeric input causes a problem with subsequent character input).

Checking the success of scanf

It is possible to see if scanf found the kind of data it was looking for. The scanf function does have a return value, even though we have been ignoring its presence so far. What it returns is the number of variables that it actually filled with input data. If, for example, you had two %-specs in the format string (and, correspondingly, two addresses of variables of matching data types), then scanf might return 2 (if it was able to fill up both variables), 1 (if it was only able to fill up the first variable) or 0 (if it was unable to fill up the first variable).

(Actually, scanf might also return the value -1, if it was unable to fill the first variable because the user had pressed the "end-of-data" key, signalling that there will be no more input at all. The end-of-data key is system dependent. For example, using MS-DOS, it is Control-Z, while on UNIX systems it is usually Control-D, although any user can customize the end-of-data key to be something else if desired. From scanf's point of view, it doesn't matter what the end-of-data key is; scanf will simply stop reading input and start returning -1 once that key has been encountered. To keep things relatively simple, in these notes we will not worry about correctly handling situations where the user presses the end-of-data key, even though it would not require much modification to our logic to do so. If you are interested, you can try to do such modifications as an exercise.)

It should be noted that scanf's return value does not reflect the success or failure of matching any trailing characters (or %-specs with an embedded *) after the last %-spec. Also be aware that once scanf encounters data that does not meet the requirements of the current %-spec, the offending data will be left untouched for the next scanf.

The following program shows how this knowledge can be used to write programs which do not react bizarrely to invalid data. Rather than one big main function, this program has separate functions to process different pieces of the input. Such functions can be used, unchanged, in future programs.

This program asks repeatedly for a quantity and unit price, stopping when the user finally enters a quantity of zero. It

then displays the total of all items entered, and (if the number of items was non-zero) the number of items and the average unit price. This program also will force the user to re-enter non-numeric input. Notice particularly how the main function contains the fundamental logic of the program; the details of the numeric validations have been delegated to appropriate functions. Also notice how many of the new things we have recently learned (such as +=) have been used.

```
#include <stdio.h>
void clear_input(void);
int get_an_int(void);
double get_a_double(void);

main()
{
    int qty, counter = 0;
    double price, total = 0;

    do {
        printf("Enter quantity (0 to stop): ");
        qty = get_an_int();
        if (qty != 0) {
            printf("Enter unit price: ");
            price = get_a_double();
            total += qty * price;
            counter += qty;
        }
    } while (qty != 0);
    printf("The total is %.2lf\n", total);
    if (counter != 0)
        printf(" for %d items at an average price of %.2lf\n",
         counter, total/counter);
}

/* Clears out any input data remaining unprocessed.
*  This function simply reads characters until it
*  encounters the new-line that terminates the input line.
*/
void clear_input(void)
{
    char junk;
    do {
        scanf("%c", &junk);
    } while (junk != '\n');
}

/* Gets an int value from the user and clears out the
*  rest of the input line. The value entered is returned.
*  If no int data is entered, the user is given an error
*  message and a chance to re-enter.
*/
```

```
int get_an_int(void)
{
    int n;
    while (0 == scanf("%d", &n))
    {
        clear_input(); /* throw away the bad data */
        printf(" Error! Please enter an integer: ");
    }
    clear_input(); /* throw away any extra data entered */
    return n;
}

/* Just like get_an_int, only gets a double rather than an int.
*/
double get_a_double(void)
{
    double n;
    while (0 == scanf("%lf", &n))
    {
        clear_input(); /* throw away the bad data */
        printf(" Error! Please enter a number: ");
    }
    clear_input(); /* throw away any extra data entered */
    return n;
}
```

A sample run of this program is:

```
Enter quantity (0 to stop): 2
Enter unit price: 5.99
Enter quantity (0 to stop): abc
 Error! Please enter an integer: def
 Error! Please enter an integer: 4
Enter unit price: a.50
 Error! Please enter a number: 2.50
Enter quantity (0 to stop): 0
The total is 21.98
 for 6 items at an average price of 3.66
```

Exercise 6.1: One thing this program does not do is give the user an error message if there is any garbage entered AFTER the number on an input line. Rather, it ignores such garbage. Modify the logic of the get_an_int and get_a_double functions, so that anything after the number other than a newline character will also cause the error message and re-entry of data. (Hint: don't just read a number, but read a number and a character, so that you can check that the character is a newline).

The getchar function

The header file, stdio.h, declares many functions that are related to scanf and printf. One of those is getchar, which can

be used instead of scanf to get one character of input. The getchar function is passed no parameters, and returns the next character of input. Assuming that c is a char variable, the statements

```
c = getchar();
```

and

```
scanf("%c", &c);
```

both do the same thing. However, since getchar returns the character, rather than passing it back through one of the parameters, it can be more convenient to use in some situations. For instance, the clear_input function from the previous program example could be written:

```
void clear_input(void)
{
    while (getchar() != '\n')
        ; /* loop body is intentionally empty! */
}
```

Note how the body of the while loop is an empty statement (;). In this case, the only thing that really needs to be repeated (the getchar call, to read the next character of input) is done as part of evaluating the condition for the loop. Because this may be an obscure thing to do, we have placed an explanatory comment beside the empty statement.

Calls to scanf and getchar can freely be mixed in the same program. Since the getchar call is simpler, and since it reduces the likelihood that you will use the wrong %-spec, it is generally preferred to a scanf if you are simply looking for a character. If you are looking for something other than a character, or for a number of things, only one of which is a character, then, of course, getchar is not appropriate.

(For those interested in the end-of-data character, you should realize that getchar really returns an int, not a char. If the user has entered the end-of-data character to stop input, getchar returns the int value -1. Otherwise, it returns the character code for the character - a number between 0 and 255 - which may freely be used as a char value.)

Better output control

We have seen how decimal places can be specified in a printf format specification to control how many decimal places will be displayed for a float or double value. (Incidentally, decimal places should NOT be specified in a scanf format string).

You can similarly control the width of the field in which a value will be displayed by putting a number in between the % and the type specifier. (In the case of a double value, the width goes before the decimal point, if you are also specifying the number of decimal places). Enough leading spaces will be output so that the value's display will take up the required number of positions.

If the field width is negative, then trailing spaces, rather than leading spaces, will be added, so that the value will be left-justified within its width.

If the field width begins with a zero, then leading zeroes will be used instead of leading spaces (provided that the field is numeric, not character).

For example, the test program

```
#include <stdio.h>

main()
{
    printf("12345678901234567890123456789\n");
    printf("%5d%4c%-4d%05d%10.2lf\n", 15, 'x', 23, 321, 4.56);
}
```

outputs

```
12345678901234567890123456789
   15   x23  00321      4.56
```

Notice how the field width for a double includes the decimal places and the decimal point. Be aware that if you supply a width that is too small to display the value, your field width will be ignored and the entire field will be displayed.

This control over field widths allows a program to nicely line up the output of columnar reports such as:

```
#include <stdio.h>
main()
{
    double fahr;

    printf("Fahrenheit to Celsius Conversion Chart\n\n");
    printf("    Degrees F equals Degrees C\n");
    printf("    ---------        ---------\n");
    for (fahr = -10; fahr < 106; fahr += 5)
        printf("%12.1lf%17.1lf\n", fahr, 5 * (fahr - 32) / 9);
}
```

which outputs a report starting with:

```
    Fahrenheit to Celsius Conversion Chart

        Degrees F equals Degrees C
        ---------        ---------
           -10.0            -23.3
            -5.0            -20.6
             0.0            -17.8
               ....and so on
```

Chapter 7. Arrays

We have seen techniques for repeatedly entering similar data and accumulating totals based on the values entered, but we do not yet have a technique for keeping the individual pieces of data around for later use. In fact, a new concept is required in order to be able to do this: the array.

An array, in C, is a single variable that can store several pieces of data of the same type. Each piece is accessed by specifying its position within the array, where the first element of the array has position 0, the second element has position 1, and so on.

When you define an array, you specify the desired size (which, with many compilers, must be a constant, by the way) inside brackets ([]) immediately after the variable name. For example,

```
int x[10];
```

defines an array, named x, which will store 10 int values.

After an array is defined, the individual elements of it are referenced by using the array name, followed by brackets ([]) containing the element's position. The individual elements of our array x are then x[0], x[1], x[2], and so on, up to x[9].

```
x[0] = 6;
```

sets the first element of x to 6, while

```
x[7] = 23;
```

sets the eighth element to 23, and

```
x[2] = x[0] + x[7];
```

sets the third element to be the sum of the first and eighth elements.

Of course, we could do these same sorts of things with individual, named variables. What makes an array useful is the fact that the position of an element in the array, often called its index, is an int value, and can therefore be computed to enable the coding of loops.

As an example, the following program takes in 5 numbers and then displays them in the reverse order that they were entered:

```
#include <stdio.h>
#define SIZE 5

main()
{
    int nums[SIZE], i;

    printf("Enter %d numbers: ", SIZE);
    for(i = 0; i < SIZE; i++)
        scanf("%d", &nums[i]);
    printf("Thank you.\n");
    printf("In reverse order, those numbers are:\n");
    for (i = SIZE - 1; i >= 0; i--)
        printf("%d ", nums[i]);
    printf("\n");
}
```

A sample run of this program is:

```
Enter 5 numbers: 5 10 2 -45 6
Thank you.
In reverse order, those numbers are:
6 -45 2 10 5
```

This program has used a #define to name the constant that is used as the size of the array. This allows us an easy way in the future to "resize" the program to handle a longer list of numbers, should we want to do so. The first for loop scans in nums[0], nums[1], and so on, up to nums[4], while the second for loop prints out nums[4], nums[3], and so on, down to nums[0]. Such loops are typically used to set or examine arrays.

One thing you need to be careful about, when using arrays, is not to use an index less than 0 or more than "one less than the size of the array" (4, in this case). If you do such a thing, your program will compile without complaint, but the program will not run properly, as it will be looking in memory locations that may be used by another variable or may not exist at all.

Also, be aware that you don't use the entire array, but rather the individual elements of the array. Every use of the array, nums, in the program above is followed by brackets containing an index.

Passing arrays to functions

There is one instance in which the name of an array is used without brackets, and that is when passing an array to a function. When one function wants to pass an array to another function, it simply uses the name of the array as the argument. The function that is receiving the array declares the corresponding parameter as an array, except that it doesn't need

to specify a size for the array (since the size was specified when the originating array was defined). A modular version of the previous program is

```
#include <stdio.h>
#define SIZE 5
void fill(int x[]);
void reverse(int x[]);
main()
{
    int nums[SIZE];

    printf("Enter %d numbers: ", SIZE);
    fill(nums);
    printf("Thank you.\n");
    printf("In reverse order, those numbers are:\n");
    reverse(nums);
}
void fill(int x[])
{
    int i;
    for(i = 0; i < SIZE; i++)
        scanf("%d", &x[i]);
}
void reverse(int x[])
{
    int i;
    for (i = SIZE - 1; i >= 0; i--)
        printf("%d ", x[i]);
    printf("\n");
}
```

In this program, notice that even though it looks like the function, fill, is receiving a copy of the nums array, it is not. The name of an array, without any brackets after it, is really the address of the start of the array, and the parameter declaration

```
int x[]
```

in fill's header line makes x (which is a pointer to some array) point to num. Because of these mechanics concerning the passing of arrays, any time a function is passed an array, that function has the opportunity to change the originating array. While this may sound dangerous, it is done for efficiency reasons: passing an address doesn't involve much data transfer, whereas making a copy of a large array would consume memory and the time of the CPU to make the copy.

The following program is similar to the previous two programs, except that it takes in one line of characters (up to 80 characters) and reverses them, useful for a party gag, where

people can then try to pronounce sentences backwards:

```
#include <stdio.h>
#define MAX 80
void backwards(char line[], int size);
int getline(char line[]);

main()
{
    char input[MAX];
    int numchars;
    printf("Enter a line to be reversed:\n");
    numchars = getline(input);
    printf("See if you can say this:\n");
    backwards(input, numchars);
}
/* displays "size" characters from "line" in reverse order */
void backwards(char line[], int size)
{
    int i;

    for (i = size - 1; i >= 0; i--)
        printf("%c", line[i]);
    printf("\n");
}
/* gets up to MAX characters from the user and stores them
*  in "line", returning the actual number of characters entered.
*  The newline at the end of the line is discarded, as are
*  any characters in excess of MAX on the input line.
*/
int getline(char line[])
{
    int n = 0;
    char c;

    while ('\n' != (c = getchar()))
        if (n < MAX)
            line[n++] = c;
    return n;
}
```

A sample run of this program is:

```
Enter a line to be reversed:
C is a very interesting language
See if you can say this:
egaugnal gnitseretni yrev a si C
```

A very useful technique shown in this program is the way that the array is not necessarily entirely filled. Only as much of the array as is needed is used. A separate variable is used to store how many of the elements are actually occupied.

Initializing arrays

Normally, you may only set the elements of an array one by one. However, when you first define an array, you may supply initial values for all elements of the array, by placing the desired initial values in braces ({}) and separating them with commas. For example,

```
int x[5] = { 10, -2, 3, 45, 6 };
```

accomplishes the same thing as

```
int x[5];
x[0] = 10;
x[1] = -2;
x[2] = 3;
x[3] = 45;
x[4] = 6;
```

If you don't supply enough values to cover every element in the array, the remaining elements will be set to zero. This implies that an easy way to initialize an entire array to zero is:

```
int x[5] = { 0 };
```

Remember that this is a technique only for initializing an array. Once an array has been defined, the elements can only be changed one at a time.

Character strings

A short while ago, we used an array of chars to store a line of input. Recall that we declared an array big enough to store the largest line we wanted to handle, and used a separate variable to indicate how many elements were actually being used.

It is so necessary to store words and sentences that there is a collection of library functions designed to store and manipulate variable-length strings of characters, using char arrays. These functions are declared in the standard header file <string.h>, although many <stdio.h> functions (notably printf and scanf) are also designed to work with these strings.

Rather than use a char array to store the string and a separate int variable to store the length of the string, these standard functions store the length in a slightly more clever fashion.

A special char value, called the null character (or null byte, since byte is a technical term for the amount of memory used to store one character), is placed after the last character of data in the array, and all the string-handling functions are designed to recognize this character as the end of the data stored in the

string.

The null character has numeric code zero. (If you look on an ASCII table, you will see the ASCII name for the null byte is NUL). This character is non-printable, and so the special notation

```
'\0'
```

is the character constant used to represent the null character. Be aware that this character, since it marks the end of a string, cannot be stored as part of the data in the string. (If you need to store a bunch of characters that might include null bytes, then use a technique similar to the one we used before, with the length stored in a separate variable, rather than using the string functions).

Also, note that when you declare an array that will hold one of these strings, the size of the array should be one element larger than the largest string you wish to store, since the null character itself occupies one element.

The strcpy function

The strcpy function, which copies one string into another, is one of the string functions in the standard library. This function takes two character arrays as parameters, and copies the data from the second string into the first string. Thus,

```
char a[6] = { 'h', 'e', 'l', 'l', 'o', '\0' }, b[6];
strcpy(b, a);
```

defines a to hold the word "hello", and then copies that word into b. The actual working of strcpy is surprisingly simple. It copies the first character of the second string to the first position of the first string, the second character of the second string into the second position of the first string, and so on, until the null byte at the end of the second string is copied over. If we were to write it ourselves we might write a function something like:

```
/* a handwritten version of strcpy */
void mystrcpy(char to[], char from[])
{
    int i;
    for (i = 0; from[i] != '\0'; i++)
        to[i] = from[i];
    to[i] = '\0';
}
```

or, more briefly (and trickier!):

```
/* an alternate handwritten version of strcpy */
void mystrcpy(char to[], char from[])
{
    int i = 0;

    while ('\0' != (to[i] = from[i]))
        i++;
}
```

The real strcpy function in the standard library was probably not written in C, but rather directly in machine language for the ultimate in efficiency. (Also, the real strcpy has a return value - it returns the address of the first array - which we will be ignoring at this introductory level. There are a few complex issues concerning addresses as return values which we do not want to get involved with now).

One warning about strcpy: while strcpy can find the end of the second string (by looking for the null byte), it has no knowledge of the maximum capacity of the first array. Therefore, whenever you use strcpy, you should be sure to supply, as the first parameter, an array that is big enough to hold the string you are copying. If the array is too small, some other parts of memory may be overwritten, and your program will act unpredictably.

Most of the string functions are similarly quite simple. Many very good programmers know only a few of the most basic string functions from the standard library, because, in fact, it usually takes more time to look up, read about, and understand what each function does, than it does to just write similar functions from scratch as the need arises. Beginning programmers, however, are usually more than happy to use functions that have already been written for them!

String constants revisited

We have been using double quotes (") to enclose constant strings of data from the very beginning. Interestingly, the double quotes are simply an alternate notation for a series of char constants with a null byte at the end, enclosed in braces. The definition

```
char name[21] = { 'F', 'r', 'e', 'd', '\0' };
```

is the same as

```
char name[21] = "Fred";
```

both of which initialize the variable, name, to store the word, Fred. We can use character string constants as the second parameter for strcpy, so that, for example,

```
strcpy(name, "Wilma");
```

changes name to store the word, Wilma. (Recalling that arrays are really passed by passing the array's starting address, when we supply an argument like "Wilma", what is really being passed is the address of the constant string).

Note that you may not do something like:

```
name = "Wilma"; /* This will NOT work! */
```

to change the contents of the array, name, after it has been defined. You must either modify the elements of the array one by one, or else call a function like strcpy, to do that for you.

The strlen and strcmp functions

Two other commonly used string functions are strlen, which returns the current length of a string, and strcmp, which compares two strings.

The strlen function is passed a string, and returns a number indicating how many characters are being stored in it (not counting the null byte). Alternatively, you can think of strlen as returning the position in the array of the null byte (because position numbers start from zero). For example, after

```
char sentence[81] = "This is a test";
int n;
n = strlen(sentence);
```

the variable n will contain the number 14. (There are 14 characters, in positions 0 through 13, before the null byte stored in position 14).

The strcmp function is passed two strings, and returns a number. This number is less than zero if the first string is "smaller" than the second string, equal to zero if the two strings are equal, and greater than zero if the first string is "bigger" than the second string.

The mechanism that strcmp uses to determine "smaller" and "bigger" is to look through the two strings, character by character until a difference is encountered. The character codes at the first spot where the strings differ determine bigger and smaller: if the character code of the first string's character is a larger number than that of the second string's, the first string is considered to be bigger. (Note that the "order" of strings is thus dependent on the character coding system used. On an ASCII based machine, for example, the upper case letters are "smaller" than the lower case letters, while on an EBCDIC

based machine the reverse is true.) If the ends of both strings are reached without a difference, the strings are considered equal.

Some compilers return -1, 0 or 1, while others return the difference of the character codes at the first position where the two strings differ. (If this difference is negative, then the code for the second string's character must have been larger than the code for the first string's, for example. If the ends of both strings are reached, the difference returned is the difference of the two null bytes, which is 0 - 0 = 0.) Any code which checks the value returned by strcmp should only expect the value to be negative, positive or zero, and should not look for specific positive or negative values.

For example, the output of

```
char s1[] = "Betty", s2[] = "Barney";
if (strcmp(s1, s2) < 0)
    printf("Betty is smaller\n");
else if (strcmp(s1, s2) > 0)
    printf("Barney is smaller\n");
else
    printf("They are the same\n");
```

is

```
Barney is smaller
```

since the first character of difference (Betty's e compared to Barney's a) tells us that the word Barney is smaller than (i.e. alphabetically "before") the word Betty.

Keep in mind that, just as you must use strcpy (or something equivalent) rather than = to assign one string to another, you must also use strcmp (or something equivalent) rather than <, <=, ==, >=, > or != when comparing two strings.

Input and output of strings

Outputting a string is quite straightforward. The following function displays the string passed to it:

```
void printstring(char s[])
{
    int i;
    for (i = 0; s[i] != '\0'; i++)
        printf("%c", s[i]);
}
```

However, this capability has already been put into printf. You may use the specification %s in a printf format string, and

supply a char array containing a string as the corresponding parameter, to display a string.

Note that in columnar reports, where you would supply a length between the % and the s, a negative length (which will right, rather than left, justify the string within the field width) is commonly used with strings.

With the %s specification, you may also supply what looks like a number of decimal places. For a string, this is used to specify the maximum number of characters from the string to display. The specification "%-30.30s", for example, displays no more than 30 characters from a string, in a left justified 30 character field.

You may also use %s as a format spec for scanf. In this case, since arrays are automatically passed as addresses, you use the array name, with no brackets ([]) and no ampersand (&), as the corresponding value. For example,

```
char name[31];
printf("Enter your name: ");
scanf("%s", name);
```

would let you enter your name, placing the first word entered into the array, name, as a string. Like the numeric formats (but unlike %c), %s is designed to treat whitespace as a separator. A side effect of this is that %s only takes one word from input.

Another format spec that may also be used with a string variable is % followed by brackets ([]) containing a list of valid characters. For example, "%[yYnN]" would accept a string of y's and n's (either upper or lower case), stopping when some other character is encountered. Two characters have a special significance inside the brackets: - indicates a range of characters, and ^ indicates a desire to accept all characters except what is shown. For example

```
scanf("%[ a-zA-Z0-9]", name);
```

accepts only spaces, lower case letters (in the range between 'a' and 'z') upper case letters (in the range between 'A' and 'Z') and digits (in the range '0' to '9'). Once some other character (such as a new-line or punctuation) is encountered, input will stop being taken, and the string (name) will be terminated. Similarly

```
scanf("%[^\n]", name);
```

will take everything except a newline character as input for the string. This can be a convenient way to get a line of input, spaces and all, into a string. Unfortunately, if the user only

presses a new-line, scanf will fail to take input (returning 0), and not touch the string variable (rather than, say, making the string variable contain an empty string). This means that extra logic would need to be included to make sure that an empty line from the user would not cause problems due to the string variable not being set.

One good feature of scanf's use of both %s and %[] is that a field width may be specified, to indicate that no more than that many characters of input should be taken. Thus

```
scanf("%30[^\n]", name);
```

would take up to 30 characters of input from a line and place them in name. If more than 30 characters are input on a line, then the extra characters would be left for the next input operation. If fewer than 30 characters are input, then all of them (except the new-line at the end, of course) are placed in the array and scanf considers the variable to have been filled.

Just as getchar() is an alternative to scanf when entering a character, there is a function declared in <stdio.h> which is an alternative to scanf when entering a string, called gets. The gets function is passed a char array which it will fill. It takes a line of input and places into the array as a string, throwing out the newline in the process.

```
gets(name);
```

and

```
scanf("%[^\n]%*c", name);
```

are therefore very similar. The major difference between them is that if the user just presses a newline, gets will make the string empty, whereas the scanf will fail, not touching the array, and leaving the lone newline for a future input operation.

If you do use gets to input a line, be aware that gets has no way of knowing how big the array is, so make the array large enough that a careless user won't cause the array to be overfilled.

With all the idiosyncrasies of the string input functions, it really is easier to write a robust string entry function than it is to fiddle the library functions so that they do what we would like. The following variation of a function we wrote before is passed a char array, and the maximum number of characters to be placed in the array. It takes one line of input, and places as much of it as will fit into the array as a string, discarding any left over input and discarding the new-line:

```
void get_a_line(char line[], int max)
{
    int n = 0;
    char c;

    while ('\n' != (c = getchar()))
        if (n < max)
            line[n++] = c;
    line[n] = '\0';
}
```

Arrays of strings

Even though C stores a string in an array, we tend to think of a string as a single piece of data. Consequently, just as we would like to have arrays of numbers, it would be nice to have arrays of strings. This is possible, simply by using two sets of brackets when declaring a char array. For example,

```
char names[10][31];
```

defines the array, names, to be 10 arrays (named names[0], names[1] and so on up to names[9]), each 31 chars big. Each of these 10 arrays can be used like a regular char array, so that, for example,

```
gets(names[i]);
```

would take a line of input and place it in the array names[i], (where hopefully the variable i is between 0 and 9).

About the only rule change for dealing with such arrays, called 2-dimensional arrays, is that when a function is passed a 2-dimensional array, the parameter in the function header MUST have the size of the second dimension specified. For example, if the names array define above were to be passed to a function named foo, then the function call might be:

```
foo(names);
```

and the header line for foo would be:

```
void foo(char n[][31])
```

where the 31 is necessary.

(While we will not show other examples of 2-dimensional arrays, you certainly can have 2-D arrays of other types. To access specific locations of a 2-D array, you simply use two sets of brackets with a number inside each one, between 0 and one less than the size for that dimension. We have only shown 2-D arrays

of char in these introductory notes because the way we work with strings allows us to treat a 2-D array of chars as a simple one-dimensional array of strings).

Putting it all together

We have waited a while to give a good example of arrays. This is because the typical example of a program that uses arrays uses both arrays of numeric types and arrays of strings. The following program uses arrays and performs rigorous handling of unreasonable input data. It asks the user to repeatedly enter information (specifically, description, quantity purchased and unit cost) about items being purchased, and then produces an itemized invoice for these purchases.

A sample run of this program looks like this:

```
Enter item description (or quit to stop): HAL Computer
Enter quantity: 1
Enter unit price: 1999.99
Enter item description (or quit to stop): Box of diskettes
Enter quantity: 2
Enter unit price: 9.99
Enter item description (or quit to stop): quit

                    Customer Invoice

Description                    Quan      Cost     Total
-----------                    ----      ----     -----
HAL Computer                      1   1999.99   1999.99
Box of diskettes                  2      9.99     19.98
                                                --------
Total                                            2019.97
Tax 15.00 percent                                 303.00
                                                --------
Total with tax                                   2322.97
                                                ========
```

As you look through this program, notice how we have attempted to re-use functions we have written previously. Also, notice how we have grouped together functions which are specific to this particular application (main, get_data, and invoice, all of which work with items that will appear on the invoice), and general purpose functions (clear_input, get_a_double, get_a_line and get_an_int) which we are likely to use again, completely unmodified, in other programs. Observe the mechanics of the passing of data back and forth between the various functions, each of which performs a useful, easy to describe task and no one of which is overwhelmingly complex.

```
/* This program produces a customer invoice for a store.
*
*  Author: Evan Weaver        Last Modified: 18-Nov-1996
*/
#include <stdio.h>
#include <string.h>

#define DESCSIZE 41     /* size of item description arrays */
#define MAX 20          /* Maximum number of items on invoice */
#define TAXRATE 15.0    /* Tax Rate in percent */

/* Application specific functions */
int get_data(char desc[][DESCSIZE], int qty[], double prc[],
 int max);
void invoice(char desc[][DESCSIZE], int qty[], double prc[],
 int max);

/* General purpose functions */
void clear_input(void);
double get_a_double(void);
void get_a_line(char line[], int max);
int get_an_int(void);

main()
{
    char descriptions[MAX][DESCSIZE];
    int quantities[MAX], itemcount;
    double prices[MAX];

    itemcount = get_data(descriptions, quantities, prices, MAX);
    if (itemcount > 0)
        invoice(descriptions, quantities, prices, itemcount);
}

/* Asks the user for description, quantity and price for
*  up to "max" items. The user may stop before "max" by
*  entering "quit" for the description.
*  The data entered is sent back through the array parameters,
*  and the function returns the number of items actually entered.
*/
int get_data(char desc[][DESCSIZE], int qty[], double prc[],
 int max)
{
    int i = 0;
    printf("Enter item description (or quit to stop): ");
    get_a_line(desc[i], DESCSIZE - 1);
    while (i < max && strcmp(desc[i], "quit") != 0) {
        printf("Enter quantity: ");
        qty[i] = get_an_int();
        printf("Enter unit price: ");
        prc[i] = get_a_double();
        i++;
```

```
            if (i < max) {
                printf("Enter item description (or quit to stop): ");
                get_a_line(desc[i], DESCSIZE - 1);
            }
        }
        return i;
    }

    /* Produces an invoice showing description, quantity, price
    *  and extended price for "max" items. Tax is added at the
    *  prevailing tax rate (TAXRATE).
    */
    void invoice(char desc[][DESCSIZE], int qty[], double prc[],
     int max)
    {
        int i;
        double subtotal, total = 0, tax;

        /* display column headings */
        printf("\n\n                    Customer Invoice\n\n");
        printf("%-30s%8s%10s%10s\n", "Description", "Quan", "Cost",
         "Total");
        printf("%-30s%8s%10s%10s\n", "-----------", "----", "----",
         "-----");

        /* show details for each item, accumulating total */
        for (i = 0; i < max; i++) {
            subtotal = qty[i] * prc[i];
            total += subtotal;
            printf("%-30.30s%8d%10.2lf%10.2lf\n", desc[i], qty[i],
             prc[i], subtotal);
        }

        /* display totals and taxes */
        printf("%58s\n", "--------");
        printf("%-30s%28.2lf\n", "Total", total);
        tax = total * TAXRATE / 100.0;
        printf("Tax %5.2lf percent%41.2lf\n", TAXRATE, tax);
        printf("%58s\n", "--------");
        printf("%-30s%28.2lf\n", "Total with tax", total + tax);
        printf("%58s\n", "========");
    }

    /* Clears out any input data remaining unprocessed.
    *  This function simply reads characters until it
    *  encounters the new-line that terminates the input line.
    */
    void clear_input(void)
    {
        while (getchar() != '\n')
            ; /* intentional empty statement! */
    }
```

```
/* Gets an int value from the user and clears out the
*  rest of the input line. The value entered is returned.
*  If no int data is entered, the user is given an error
*  message and a chance to re-enter.
*/
int get_an_int(void)
{
    int n;
    while (0 == scanf("%d", &n))
    {
        clear_input(); /* throw away the bad data */
        printf(" Error! Please enter an integer: ");
    }
    clear_input(); /* throw away any extra data entered */
    return n;
}

/* Just like get_an_int, only gets a double rather than an int.
*/
double get_a_double(void)
{
    double n;
    while (0 == scanf("%lf", &n))
    {
        clear_input(); /* throw away the bad data */
        printf(" Error! Please enter a number: ");
    }
    clear_input(); /* throw away any extra data entered */
    return n;
}

/* Gets a line of input from the user. Up to "max" characters
*  are placed into "line" as a string, and any remainder,
*  including the trailing new-line, is discarded.
*/
void get_a_line(char line[], int max)
{
    int n = 0;
    char c;

    while ('\n' != (c = getchar()))
        if (n < max)
            line[n++] = c;
    line[n] = '\0';
}
```

Chapter 8. Files

In the last chapter, we learned how to use arrays to store in memory repeated copies of the same sort of data. But since arrays are stored in memory, once the program ends, the values that were stored are lost. In this chapter we will learn the techniques of accessing data stored on disk, so that it stays around even when the program isn't running.

At the very beginning of these notes, the term file was introduced to help explain the edit-compile-run process with which you are, by now, intimately familiar. A file is simply a named area of disk storage. The C programs you write are each stored in a file, as are the compiled, machine language versions of your programs.

But you can also have files which contain data that is accessed by one or more programs. Accessing a file from a C program involves three steps:

1. The program opens the file. This establishes a connection between the program and the file. The library function, fopen, is used to open a file.

2. The program accesses data in the file. This may mean that the program stores data on the file (sometimes called "writing to the file"), or it may mean that the program retrieves data from the file (sometimes called "reading from the file"). Typically, the file is accessed numerous times (in a loop, for example) while the file is open. The library function, fprintf, is used to write to a file, while the function fscanf is used to read from a file. These functions make file access look very much like output to and input from the terminal.

3. The program closes the file. This breaks the connection between the program and the file. The library function fclose is used to close the file.

The file functions mentioned above are all declared in the stdio.h header file. Also, stdio.h sets up the name FILE (by using a #define directive) to correspond to the kind of data that the computer will use to maintain the connection with the file. To work with a file, you will first need a variable that points to a FILE. For example,

```
FILE *fp;
```

declares such a variable. This pointer will be used throughout the program to refer to the file. (It is important not to forget the *).

The fopen function

To open the file, issue a statement like:

```
fp = fopen("data.dat", "r");
```

The first parameter to the fopen function is a character string containing the name of the file you wish to use. The second parameter is also a character string, and indicates the manner in which you wish to access the data in the file. We will explore three possibilities for this:

"r" - tells fopen you will be reading data from the file. If no such file exists, fopen will fail.

"w" - tells fopen that you will be writing to a new file. If a file with this name already exists, all pre-existing data will be wiped out.

"a" - tells fopen that you will be writing new data to the end of the file (i.e. appending data to the file). If the file already exists, all data written to the file will appear after the pre-existing data. If the file doesn't exist, a new (empty) file will be created.

(The access string may also contain a b, indicating binary - rather than text - access, or a +, indicating that the file can be both read and written, but we will ignore these possibilities for now).

The fopen function will attempt to open the file. If that is successful, fopen returns the address of the area in memory where it is keeping track of the file. Future file operations will need to be given this address, so we store it in a pointer for later use. If the file cannot be opened for some reason, then fopen returns the address zero. In stdio.h, the address 0 is given the name NULL (using a #define). It is common to check the pointer in which you store fopen's return value, and make sure that it is not NULL before beginning to process the file.

Incidentally, the files we will be working with in these notes are called "sequential text files". Sequential means that we will be either reading data from the beginning of the file to the end of the file, or writing data immediately after previously written data, without jumping around from one position in the file to another. Text means that the data will always be stored in files that can be viewed or even modified using a regular text editor such as the one you use to write your C programs.

The fprintf and fscanf functions

To write data to a sequential text file, you may use the function, fprintf. The fprintf function works exactly like printf, except that there is an extra argument, the address of the FILE that was obtained from fopen, which appears before the format string (i.e. at the beginning of the argument list). The data that printf would have displayed on the screen gets instead written to the file. For example,

```
fprintf(fpout, "%s:%.2lf,%s\n", name, salary, job);
```

would write the character string name, followed by a colon, followed by the double variable salary (formatted to 2 decimal places), followed by a comma, followed by the character string job, followed by a newline, to a file that was previously opened for "w" or "a" access. The FILE address for this file (from fopen) was stored in the pointer fpout in this case.

Similarly, the fscanf function may be used to read data from a sequential text file. Like fprintf, fscanf works the same way as scanf, except that it has an extra argument, the FILE address resulting from opening a file for "r" access, at the beginning of the argument list. The statement

```
fscanf(fpin, "%[^:]:%lf,%[^\n]\n", name, &salary, job);
```

would read the same sort of data from a file (into variables name, salary and job) that the previously shown fprintf would have written. Here, fin is a pointer to a FILE opened for "r" access.

The fclose function

When you are done with a file, you must remember to close it. If the FILE address from fopen was stored in a pointer named fp, you would close it with the statement

```
fclose(fp);
```

Closing the file ensures that all data the program may have written to the file actually gets onto the disk (usually, the file functions may "cache" some of the data destined for the disk in memory until there is enough accumulated data to justify a disk write, which is a comparatively slow operation), and also tells the operating system that you no longer need to use the file, which may allow another program to use the file.

Failure to close the file may prevent later programs from accessing the file, or may cause some data loss if you are writing to the file, although the severity of these problems varies a great deal from one system to another. To be on the

safe side, always remember to call fclose for every file that was successfully opened, even though you may determine through experimentation that forgetting to close does not seem to cause problems on your system.

Some other file functions

Occasionally, you may find a need, when reading through a file, to move right back to the beginning of the file. (For example, you might want to read all the data in the file to perform some statistics, and then pass through the file a second time to look for specific records based on the results of the statistics.) One way to do this would be to close the file and then immediately reopen it. A more efficient way is to use the function, rewind, which simply moves back to the start of the file. The rewind function has a single parameter, the FILE pointer for the open file you are working with, for example:

```
rewind(fp);
```

Also, just as getchar and gets perform duties that overlap what scanf does, there are similar file functions related to fscanf. The fgetc function reads one character from a file, so that

```
c = fgetc(fp);
```

does essentially the same thing as

```
fscanf(fp, "%c", &c);
```

The one way in which fgetc differs from the equivalent fscanf call is what happens if there is no more data in the file. The fscanf call will fail, returning -1 and not setting the character variable. The fgetc function returns the integer value -1. The value -1 is given the name EOF (again, using a #define) in stdio.h, by the way.

Consequently, if you plan to use fgetc, it is recommended that you use an int variable, rather than a char variable, to store fgetc's return value. After checking to make sure it isn't EOF, you can then copy it to a char variable.

In much the same way, fgets reads one line from a file into a string.

```
fgets(s, 31, fp);
```

works much the same as:

```
fscanf(fp, "%30[^\n]\n", s);
```

except that the fgets call places the newline in the string s,

where the fscanf shown will simply discard it. (Note that the fgets file function works a bit differently from the gets input function, in that you supply fgets with the size of the char array you are passing, and in that it stores, rather than discards, the newline.)

There are also similar counterparts for fprintf. The fputc function writes a character to a file, so that

```
fputc(ch, fp);
```

works the same as

```
fprintf(fp, "%c", ch);
```

Similarly, fputs writes a character string to a file, so that

```
fputs(str, fp);
```

works the same as

```
fprintf(fp, "%s", str);
```

Incidentally, there are output functions corresponding to fputc and fputs which we haven't mentioned before: putchar and puts.

```
putchar(ch);
```

works the same as

```
printf("%c", ch);
```

while

```
puts(str);
```

works the same as

```
printf("%s\n", str);
```

Note that puts displays a newline after the string, whereas the fputs functions does not write a newline character to the file after writing the string.

Records and fields

You may write any data you like to a file, in any order you want. But files are most commonly used in two different situations: (1) storing data that is destined to be viewed later or printed out (called a report file), and (2) storing data to be used again later by the same or another program (called a data file).

Creating a report file is completely analogous to writing a program that displays a report on the screen. The only real differences are that you must open the file before you start, close the file when you are done, and use fprintf instead of printf.

Creating or reading a data file is conceptually a bit different, however, from the sorts of things we have been doing before. Most data stored in files tends to be the same sorts of information repeated over and over again in the same pattern. For example, an employee file might store an employee name, job title and salary for thousands of employees.

In such a file, the data for one employee is called a record, while the individual pieces of data (the name, the job title and the salary) are called fields. The term record layout is often used to refer to a description of how each record is broken into fields. While there is no requirement for one record to be stored as one line of a file, it is common practice to do so when using text files (as we are in these notes). If each line of the file stores one record, the file can easily be viewed directly by printing it out or using a text editor, without undue confusion for the person looking at it.

The following program lets the user add records to an employee file, where the record layout is:

- the employee's name of up to 35 characters
- the employee's job title of up to 40 characters
- the employee's annual salary

where each record forms one line of the file and the fields are separated by semi-colons (;). A sample of the file might be:

```
Joseph Blow;Manager, Clerical Staff;45000.00
Betty Boop;Vice President, Administration;175400.50
Peter Piper;Taste Tester; 37500.00
```

Note that for brevity, this program does not do any input validation; you may add in functions similar to those we have written before to make the entry less error prone. Also, note that if we had wanted to create a new file from scratch, rather than adding to the end of any data that may already be in the file, we would have used "w" instead of "a" in the fopen statement.

```
/* This program lets the user add employee records to
*  the end of the file "employee.dat". Entry stops when
*  an empty employee name is entered.
*
*  Author: Evan Weaver        Last Modified: 25-Nov-1996
*/
#include <stdio.h>
#include <string.h>
int enter_employee(char name[], char job[], double *psal);

main()
{
    FILE *fp;
    char name[36], job[41];
    double salary;

    fp = fopen("employee.dat", "a");
    if (fp == NULL)
        printf("Cannot open the employee.dat file\n");
    else {
        while (enter_employee(name, job, &salary))
            fprintf(fp, "%s;%s;%.2lf\n", name, job, salary);
        fclose(fp);
    }
}

/* Lets the user enter data for an employee. If no name
*  is entered, no more data is asked for and a false value
*  is returned. Otherwise, a true value is returned after
*  placing the user's input into the variables pointed
*  to by the parameters.
*/
int enter_employee(char name[], char title[], double *psal)
{
    int ok = 0; /* false */

    printf("Enter employee name (or nothing to stop): ");
    gets(name);
    if (strcmp(name, "")) {
        printf("Enter %s's job title: ", name);
        gets(title);
        printf("Enter %s's salary: ", name);
        scanf("%lf", psal); /* no & since psal is a pointer! */
        while (getchar() != '\n')
            ; /* discard junk entered after the number */
        ok = 1; /* true */
    }
    return ok;
}
```

The nature of truth, according to C

This program shows a useful technique: the enter_employee function returns a "true" value if an employee is entered, and a "false" value if one is not. It turns out that C represents false with the numeric value 0, and true with any value that is non-zero. Whenever you want a function to return "true" or "false", simply return an int, using 0 for false and anything else (say, 1) for true. Then you may use the function call itself as a condition for an if statement or a loop.

The relational operators (such as < and ==), as well as the logical operators (such as && and ||) all return ints, using 0 for false, and 1 for true. But be aware that the if statement and the looping statements all accept any nonzero value as true. This means that any time you do a numeric comparison like

```
x != 0
```

you could simply use x as the condition, since if x is not zero, it will automatically be true. We have used this fact in the if statement by using the condition

```
strcmp(name, "")
```

rather than

```
strcmp(name, "") != 0
```

to see if the name is empty. (We could have gone one step further and simply used name[0] as the condition - this asks if the first character of the array is non-zero - which will only be false if name[0] is the numeric value 0, i.e. the null character).

The variable, ok, in the enter_employee function is used to store "true" if we enter an "OK" record, and "false" if we don't. Such a variable is often called a flag (after naval practices of holding different flags up and down to communicate out of sound range but within visual range).

The not operator

Once you start to store true/false flags, you often want to be able to reverse the flag, in order to, for example, apply DeMorgan's Law. The operator ! can be placed in front a condition to reverse it. For example,

```
!(x < y)
```

is the same as

```
x >= y
```

and

```
!(ok && x < y)
```

is (using DeMorgan's Law) the same as

```
!ok || x >= y
```

(The ! operator has very high precedence, making all the parentheses above necessary). If you think about it, you will realize that the ! operator is the same as asking if the condition is == 0.

The following program reads the employee.dat file to produce a report of proposed raises, designed to make low and middle income employees feel less badly treated. The proposal is to give each employee earning under $100,000 an increase of 10% of the first $40,000 in annual salary. The report shows what each employee's raise would be as well as the total cost for everyone's raise.

```
/* This program displays proposed raises for low and
*  middle income employees, using employee data from
*  the file "employee.dat".
*
*  Author: Evan Weaver        Last Modified: 25-Nov-1996
*/

#include <stdio.h>
double show_raise(char name[], double sal);

main()
{
    FILE *fp;
    char name[36], job[41];
    double salary, total_cost = 0;

    fp = fopen("employee.dat", "r");
    if (fp) {
        printf("            Proposed Employee Raises\n");
        printf("%-35s%10s\n", "Name", "Raise");
        while (3 == fscanf(fp, "%35[^;];%40[^;];%lf\n", name,
         job, &salary))
            total_cost += show_raise(name, salary);
        printf("\nTotal cost for raises: $%.2lf\n", total_cost);
        fclose(fp);
    }
    else
        printf("Cannot read employee.dat file\n");
}
```

```
/* Computes raise for employee: 10% of the first $40,000
*  of annual salary for every employee earning less than
*  $100,000. The employee's name and raise is shown, and
*  the raise is also returned.
*/
double show_raise(char name[], double sal)
{
    double raise;
    if (sal < 40000)
        raise = sal * 0.1;
    else if (sal < 100000)
        raise = 4000;
    else
        raise = 0;
    printf("%-35s%10.2lf\n", name, raise);
    return raise;
}
```

The output of this program would be something like

```
            Proposed Employee Raises
Name                                    Raise
Joseph Blow                           4000.00
Betty Boop                               0.00
Peter Piper                           3750.00

Total cost for raises: $7750.00
```

In this program, note how we read each record of the file with the following fscanf call:

```
fscanf(fp, "%35[^;];%40[^;];%lf\n", name, job, &salary)
```

We have carefully designed the layout of the format string to match the record layout, discarding the semi-colons and the newline that we don't need. If there is a record that can be read, this fscanf should return 3, indicating that 3 variables were successfully filled. Our program stops as soon as one such fscanf fails.

Note that it could fail because we have already read all the data and there is none left, or because the data that is there is not of the correct type. We could distinguish between these two if we wanted (the fscanf would return -1 if there were no more data, and 0-2 if it were bad data) but we have chosen simply to stop reading at the first sign of trouble, regardless of the reason.

Exercise 8.1: Make the output of this program go to a file, named raises.txt, rather than the screen. (You will need a second FILE pointer for this). Furthermore, find out how many lines are printed on a page with your printer, and then make the

logic print a fresh heading (with page number) at the top of each page. Hint: make one variable to count the lines output, and another to count the pages output. Every time you output a line, add one to the line count. If the line count reaches the limit for one page, increase the page count, output a page heading and reset the line count.

A Final Word

This chapter finishes these notes. We have covered a great deal of the syntax of the C language. For many of you this will have been a lot of material in a fairly short period of time, while others may be worried that we haven't covered enough C to be a real programmer. The subset of the C language that we have studied is enough to do some serious programming. But no one becomes a programmer in 3 months.

You have started to learn the "vocabulary" of programming. The concepts of logic control structures, subroutines, data types, variables, arrays and files are fundamental concepts in the world of programming, regardless of the language. But you have a great deal more to learn about programming and effective program design. The best way to prepare yourself for your future development as a programmer is to write as many programs as you can. At the same time, look at other programs written by good programmers, to learn as many little tricks and techniques as you can.

Once you have an understanding of all the basics of programming, which will take a year or more, it will be time to specialize. There are so many different areas of programming, such as writing programs that display intense computer graphics, writing programs that manage large databases, writing programs for other programmers to use, writing operating system programs, and so on, that no one can know it all.

It is better, after getting a firm grounding in the fundamentals, to learn a lot about something particular, than to learn a little about a lot of things. Once you learn something really thoroughly, you develop confidence that you will be able to learn other things just as thoroughly, should the need ever arise. And in the field of programming, about the only thing you can be sure of is that the need to learn new things in depth arises often.

APPENDIX A. Sample Walkthrough Exercises

At Seneca, we have found walkthrough exercises to be a valuable tool when learning to program. Given a program, the objective is to determine exactly what the program does. Typically, this means determining the precise output which the program displays on the screen. In order to do this accurately, you must "become the computer", and step (or "walk") through the program, line by line. You must also keep track of each and every variable as it changes through time.

Most people develop their own technique for keeping track of things as the program's logic is traced. Your instructor will no doubt show you a workable technique. As your programming skills develop, you will probably find that you need to write down less and less in order to perform a walkthrough. But even the best programmers, who may simply be able to write down the output of a program upon looking at the code, have to keep track of everything. They just may be able to keep track of the details mentally, rather than having to write it all down. Beginning programmers, on the other hand, should be careful to write down everything that happens as the program executes, in order to avoid the situation of getting three quarters of the way through a problem and then forgetting where you are!

The following walkthrough exercises have appeared on actual tests and exams over the years. In most of them meaningless variable and function names are used, in order to force you to accurately trace through the logic, and not simply to rely on those names to guess what should happen. Keep in mind that in your own programs, you should not imitate this characteristic: always use meaningful names.

Even though the variable names may be meaningless, you should still pay attention to how the variables change as you proceed through the logic. Walkthroughs are a good opportunity to see programming techniques which may not otherwise occur to you. Just keeping accurate track will get you the right answer, but if you can "step back" and observe the flow of the logic over the data, your skills as a programmer will develop more.

While these exercises should give you some feel for what you can expect to have to be able to do on tests, realize that every walkthrough you encounter will be a unique combination of the syntactical elements you have been studying. Knowing how these particular programs work should not be your goal. Rather, you should be learning the concepts of programming, and the syntax of the C language, well enough to be able to figure out programs like these. If you can figure these out, it is a good sign. But if you need too much help to get through them, that means you need to do more work.

1. Basic numeric data and logic.

```
main()
{
    int a;
    double b, c;

    a = 6;
    b = 0.7;
    while (a < 10 && b < 3.0) {
        if (a < 8) {
            a = a + 1;
            b = b * 2;
            c = a - b;
        }
        else {
            a = a - 2;
            b = b + 0.8;
        }
        c = a - b;
        printf("%.2lf-%d-%.2lf\n", c, a, b);
    }
}
```

---

2. Basic numeric data and logic.

```
main()
{
    int num1;
    double num2, num3;

    num1 = 0;
    num2 = 1.5;
    num3 = num2 * 3.0;
    if (num3 > num2) {
        printf("I ");
        num1 = num1 + 1;
        while (num3 > num2) {
            num2 = num2 + num1;
            printf("*");
        }
    }
    else
        printf("You ");
    printf(" C so well now\n");
    printf("One: %d, Two: %.1lf, Three: %.1lf\n",
     num1, num2, num3);
}
```

3. Basic numeric data and logic.

```
main()
{
    double x;
    int n, m;

    n = 0;
    m = 10;
    for (x = 2.5; n < m; x = x + 0.3) {
        if (n < 3) {
            n = n + 1;
            m = m - 2;
        }
        else
            n = n + 1;
        printf("%d %d %.1lf\n", n, m, x);
    }
    printf("%.1lf\n", x);
}
```

---

4. Basic logic and char data.

```
#include <stdio.h>
main()
{
    int n = 8;
    char c = 'f';

    do {
        switch (n) {
          case 8:
          case 11:
            n++;
            break;

          case 9:
          case 10:
          case 12:
            n += 2;
            c = 'h';
            break;

          default:
            c = 'q';
        }
        printf("n stores %d while c stores %c\n", n, c);
    } while (c != 'q');
}
```

5. Basic modularity.

```
#include <stdio.h>

int do_something(int a, int b)
{
    int c;

    printf("a is %d, ", a);
    c = a + b - 1;
    printf("c is %d\n", c);
    return c;
}

main()
{
    int n, m;
    n = 1;
    m = 7;
    while (m < 10) {
        n = do_something(n, m);
        m = m + 1;
        printf("n is %d and m is %d\n", n, m);
    }
    printf("Phew!\n");
}
```

6. Basic modularity.

```
#include <stdio.h>

double foo(double x, int n)
{
    double y;

    if (x < n)
        y = x + n;
    else
        y = x - n;
    printf("In foo (%.1lf)...", y);
    return y;
}

main()
{
    double a, b;

    a = 6;
    b = 12.5;
    while (a < b) {
        a = foo(b, 5);
        printf("In main (%.1lf)\n", b);
        b = b - 4.5;
    }
}
```

7. Pointers.

```
#include <stdio.h>

double foo(int a, char *p);

main()
{
    double x;
    int i;
    char c = 'a';

    for (i = 0; i < 4; i++)
        x = foo(i, &c);
    printf("one:%c two:%.1lf three:%d\n", c, x, i);
}

double foo(int a, char *p)
{
    double d;

    printf("%c ", *p);
    switch (*p) {
      case 'a':
        *p = 'c';
        d = a * 0.5;
        break;
      case 'b':
        *p = 'a';
        d = a * 1.5;
        break;
      case 'c':
        *p = 'b';
        d = a * 0.1;
        break;
      default:
        d = 1.7;
    }
    printf("%.1lf\n", d);
    return d;
}
```

8. Arrays.

```
#include <stdio.h>
#define SZ 10
main()
{
    int x[SZ], i, j = 5;

    for (i = 0; i < SZ; i++) {
        x[i] = (i + 1) * j;
        j--;
    }
    while (i > 0)
        printf("%4d", x[--i]);
}
```

---

9. Strings.

```
#include <stdio.h>
#include <string.h>
int foo(char s1[], char s2[]);
main()
{
    char name[21],
         title[21];
    strcpy(name, "Fred");
    strcpy(title, "Crane Operator");
    while (1 == foo(name, title))
        printf("Looping...\n");
    printf("All done\n");
}
int foo(char s1[], char s2[])
{
    int num = 1;
    printf("%s is a %s\n", s1, s2);
    if (strcmp(s1, "Betty") == 0) {
        strcpy(s1, "Barney");
        strcpy(s2, "Programmer");
    }
    else if (strcmp(s1, "Fred") == 0) {
        strcpy(s1, "Betty");
        strcpy(s2, "Manager");
    }
    else {
        strcpy(s1, "Joe");
        strcpy(s2, "Leaf Blower");
        num = 0;
    }
    return num;
}
```

10. Arrays.

```
#include <stdio.h>
#define MAX 5

void fill(double x[], int sz);
int find(double num, double x[], int sz);
void show(double x[], int sz);

main()
{
    double nums[MAX];
    int n;

    fill(nums, MAX);
    n = find(4.5, nums, MAX);
    show(nums, n);
}

void fill(double x[], int sz)
{
    int i;
    for (i = 0; i < sz; i++)
        x[i] = 1.5 * (i + 1);
}

int find(double num, double x[], int sz)
{
    int i = 0;
    double diff;

    do {
        diff = x[i] - num;
        printf("%.2lf\n", diff);
        i++;
    } while (i < sz && (diff < -0.5 || diff > 0.5));
    return i;
}

void show(double x[], int sz)
{
    int i;
    for (i = sz - 1; i >= 0; i--)
        printf("%.2lf ", x[i]);
    printf("\n");
}
```

11. Arrays.

```
#include <stdio.h>

int cp(double nfrom[], int nto[], int s);
void print(int n[], int s, int x);

main()
{
    double a[4], x = 0.5, y = 1.1, z;
    int b[4], i;

    for (i = 0; i < 4; i++) {
        z = x;
        x = x + y;
        y = z;
        printf("%5.1lf\n", x);
        a[i] = x;
    }
    i = cp(a, b, 4);
    print(b, 4, i);
}

void print(int n[], int s, int x)
{
    int i;
    printf("x is %d and the array is:\n", x);
    for (i = 0; i < s; i++)
        printf("%d\n", n[i]);
}

int cp(double nfrom[], int nto[], int s)
{
    int i, t = 0;

    for (i = 0; i < s; i++) {
        nto[i] = nfrom[s - (i + 1)];
        t = t + nto[i];
    }
    return t;
}
```

12. Arrays and pointers.

```
#include <stdio.h>
#define SIZE 4
int calc(int nums[], int n, int *pe, int *po);
void set(int nums[], int n);
main()
{
    int x[SIZE], even, odd, val;

    set(x, SIZE);
    val = calc(x, SIZE, &even, &odd);
    printf("The amounts are %d, %d and %d\n", val, even, odd);
}
int calc(int nums[], int n, int *pe, int *po)
{
    int i;
    *pe = 0;
    *po = 0;
    for (i = 0; i < n; i+=2) {
        *pe += nums[i];
        if (i + 1 < n)
            *po += nums[i + 1];
    }
    return *po + *pe;
}
void set(int nums[], int n)
{
    int i, x;
    x = 6;
    for (i = 0; i < n; i++) {
        printf("%d..", x);
        nums[i] = x;
        x = 2 * x;
        if (x > 20)
            x = 4;
    }
}
```

13. Strings.

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

void fold(char s[], int k)
{
    int i, j, len;

    len = strlen(s);
    for (i = 1; i + k < len; i++) {
        for (j = i; s[j + k]; j++)
            s[j] = s[j + k];
        s[j] = '\0';
        len -= k;
        k++;
    }
}

main()
{
    char s1[31], s2[31];

    strcpy(s1, "Show me less!? We always");
    strcpy(s2, "worry:kiss and neck, or split");
    fold(s1, 1);
    fold(s2, 0);
    printf("%s %s all, however\n", s1, s2);
}
```

14. Strings.

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

void disp1(char str[])
{
    int i = 1, oldi = 1, newi;

    while (i < 10) {
        putchar(str[i-1]);
        newi = i + oldi;
        oldi = i;
        i = newi;
    }
    putchar('\n');
}

void disp2(char str[])
{
    int i, j;

    for(i = 0; i < 6; i++) {
        for (j = 0; j <= i; j++)
            putchar(' ');
        printf("%c\n", str[1+(4*i)]);
    }
}

main()
{
    char s[30];

    strcpy(s, "dirutawy5prefeghmr8jks");
    puts("babies!");
    disp1(s);
    disp2(s);
}
```

15. Arrays.

```
#include <stdio.h>

int did(int x[])
{
    int i=1, j=2, t=0;

    while ((i + j) < 10)
        t += x[i++] + x[j++];
    return t;
}

int didnot(int x[])
{
    int i, j=1, t=0;

    for (i=2; i<10; j=i) {
        t += x[i];
        i+=j;
    }
    return t;
}

main()
{
    int n[10], i;

    for (i = 0; i < 10; i++)
        n[i] = 101 * (i + 1);
    printf("Did O.J. do it? %d say yes, %d say no!\n",
      did(n), didnot(n));
}
```

16. Files. Tricky - the program reads its own source file.

```
/* this is the first line of the source file: q1.c */
#include <stdio.h>

void q1get(FILE *file, char *s, int count)
{
    int i;
    for (i = 0; i < count; i++)
        s[i] = fgetc(file);
    s[i] = '\0';
}
main()
{
    FILE *fp;
    char s1[10], s2[10], ch;
    int keepatit = 1;

    fp = fopen("q1.c", "r");
    while (keepatit) {
        if ('*' == (ch = fgetc(fp))) {
            q1get(fp, s1, 5);
            q1get(fp, s2, 3);
            fclose(fp);
            keepatit = 0;
        }
    }
    printf("Well,%s%s it?!?\n", s2, s1);
}
```

APPENDIX B. Sample Word Problems

In this appendix, you will find a collection of word problems typical of what has been given on past in tests and exams.

While the walkthroughs from the previous appendix are only training exercises, these word problems are typical of the sorts of things programmers do on the job, day in and day out. Admittedly, these problems have a much smaller scale than the problems a professional programmer must solve. But the process of taking a verbal description of a problem and proceeding through to working code is what programming is all about, and it needs to be practiced as often as possible.

Note that some of the questions ask for a simple program to be written, while others present an already analyzed situation, and simply require one or more functions to be written.

1. Write a program that asks the user to enter repeatedly a number of gallons, stopping once the user enters a value of 0. As each value is entered, the corresponding number of litres is displayed. Assume that 1 gallon is 3.76 litres. The following shows a sample run of the program:

```
How many gallons (0 to stop): 2
That is 7.52 litres
How many gallons (0 to stop): 1.1
That is 4.14 litres
How many gallons (0 to stop): 0
```

2. Write a program that helps a cook "resize" a recipe. The program first asks for a factor (for example, 0.5 to cut a recipe in half or 3 to triple a recipe), and then repeatedly asks for an amount. After each amount is entered, the program displays that amount multiplied by the factor. The program stops when an amount of 0 is entered. The following shows a "sample" run of the program:

```
***Recipe Resizer***
Enter the factor: 2
Enter an amount (or 0 to stop): 1.5
You should use 3.00 instead.
Enter an amount (or 0 to stop): 4
You should use 8.00 instead.
Enter an amount (or 0 to stop): 13
You should use 26.00 instead.
Enter an amount (or 0 to stop): 0
```

(Notes: (1) these value are just examples; any factor and any amount should work! (2) make your output match the format of this sample as closely as possible.)

3. Write a C program that asks the user, a wholesale sales person, to enter the total amount of a sale to a client. The program rejects (with an appropriate error message) any amount less than zero, forcing the user to re-enter the amount. Once an allowable amount has been entered, the program displays the client's volume discount, the total after the discount, and the sales person's commission, based on the following information:

    Discount - 0% on the total, if the total is less than $1000.
    - 5% of the total if the total is between $1000 and $20,000.
    - 10% of the total if the total is over $20,000.

    Commission - based on the total less the discount
    - 5% of the first $10,000
    - 10% on everything over $10,000

    As an example, a sale of $15,000 would receive a discount of $750, making the net sale $14,250. The commission would be $925 (which is 500 + 425).

4. Write a C program that asks the user to enter a monetary amount, and then gives the user the following choice:
    1. Tax Exempt
    2. PST only (8%)
    3. GST only (7%)
    4. Both PST and GST

    If the user enters any number other than 1-4, he or she is given an error message and is forced to choose again, until a proper choice is made.

    The program displays the original amount, the tax on that amount (which depends on the choice they made), and the total including tax.

5. Write a function

    void histogram(int array[], int count)

    which is passed an array of numbers and the number of elements in the array, and then displays a histogram of that array using asterisks. For example, if an array named x contained the six values { 5, 1, 2, 4, 3, 4 }, then histogram(x, 6) would output:

```
*****
*
**
****
***
****
```

    (If a value is 0 or less, output an empty line for that value)

6. Write a function (not a main), with the following header line:

```
void add_a_minute(int *h, int *m, char *ap)
```

This function is passed two addresses of integer variables (containing an hour [h:1-12] and a minute [m:0-59]) and the address of a character variable [ap: either 'a' for AM or 'p' for PM]. The data pointed to by the parameters represents the time on a 12-hour clock.

The function advances the time by one minute.

If this should cause the minute to be 60, then the hour should be advanced by one and the minute should be reset to 0.

If, in turn, this should cause the hour to be 13, then the hour should be reset to 1.

If, in the course of changing the hour, the hour changes from 11 to 12, then the AM/PM indicator should be changed from 'a' to 'p', or vice versa, depending on its current setting.

For example, if we had the following variables:

```
int hour = 11, minute = 59;
char am_or_pm = 'p';
```

then the statement:

```
add_a_minute(&hour, &minute, &am_or_pm);
```

would change hour to 12, minute to 0, and am_or_pm to 'a'.

Similarly, if we reset the variables:

```
hour = 12;
minute = 59;
am_or_pm = 'a';
```

then the statement:

```
add_a_minute(&hour, &minute, &am_or_pm);
```

would change hour to 1, change minute to 0 and leave am_or_pm at 'a'.

A second call:

```
add_a_minute(&hour, &minute, &am_or_pm);
```

would then leave hour at 1, change minute to 1 and leave am_or_pm at 'a'.

7. Write a C program that asks the user whether to convert Litres or Gallons. If the user answers L (or l), the program asks the user to enter the number of Litres, and then displays that in Gallons. If the user answers G (or g) the program asks the user to enter the number of Gallons, and then displays that in Litres. Any other response by the user causes the program to produce an error message and then stop. Assume that 1 Gallon is 3.76 Litres.

8. Write a function

```
int deblank(char str1[])
```

that removes all spaces from the null-terminated string in "str1" and returns the number of spaces removed.

Hint: Move through "str1" looking for spaces. Every time you find one, copy everything on the right of it over one character to the left.

9. Write a function

```
int trim(char str1[])
```

that removes all leading and trailing spaces from the null-terminated string in "str1" and returns the new length of the string.

Hint: Move through "str1" looking for the first non-blank. Once you find it, move it to position 0, move the next character to position 1, and so on until the null byte has been moved back. Then, keep moving the null byte back until the character before it is non-blank. The final position of the null byte is the new length.

10. Write a function

```
int vanna(char str[], int consonant, int vowel)
```

that figures out the "cost" of the word(s) stored in the null-terminated string, str, assuming that "consonant" contains the dollar value of each consonant, that "vowel" contains the dollar value of each vowel, and that non-letter characters have a value of zero dollars. For example,

```
vanna("Wheel Of Fortune", 25, 50)
```

would return 500, because it has 8 consonants at $25 each, and 6 vowels at $50 each. (The vowels are: a, e, i, o and u).

11. Write a C program that asks the user to enter three names, up to 30 characters each, and then displays the names in alphabetical order. (Note that the order should be case sensitive, with upper case letters first as in the ASCII table, so that Fred is after FRED, but before fred, and aaa is after ZZZ).

12. Write a function that is passed (1) an array of doubles and (2) an int containing the number of elements in the array, and has no return value. The function should increase (by 15%) every element of the array that is 4.0 or larger, leaving untouched all the elements that are smaller than 4.0.

13. Write a function

```
void firstword(char word[], char sentence[])
```

    that copies the first word of the null-terminated string "sentence" into the null-terminated string "word". If there are no words in "sentence" (because it is empty or all blanks) then "word" should be set to be empty. Assume that a word is simply a series of non-blank characters, separated from other words by blanks. Thus, for example,

```
firstword(x, "This is a sentence");
```

    would copy "This" into the array x, and

```
firstword(x, " and$%^# so is this");
```

    would copy "and$%^#" into x.

    (Hint: find the position of the first non-blank, then copy things over until you hit a blank or the end of the string).

14. Suppose a file, named inventory.dat, stores inventory data. Each line of the file has an item name, quantity on hand and unit price, separated by semi-colons. A sample file might be:

```
Hammer;10;9.99
Phillips Head Screwdriver;5;7.99
Saw;2;9.95
```

    Write a program that displays the total value of all the inventory (which is the sum of quantity times price for all records of the file). For example, with the data shown above, the program outputs:

```
Total inventory value is $159.75
```

    You may assume the maximum description will be no longer than 35 characters. If the file cannot be read, your program should output a meaningful error message.

15. Suppose a file, named "comedy.dat" looks something like:

```
Seinfeld 50 75
Carson 40 20
Leno 45 50
Letterman 20 95
```

    where each line of the file has a comedian's last name, his or her F.I. (funny index, higher means funnier) and his or her N.I. (nastiness index, higher means nastier). Spaces separate each field in a record of this file.

    Write a program that reads through this file and then displays the minimum, maximum and average values for each of the F.I. and the N.I.

16. Assume a file, named "times.dat", contains records that look like:

```
12:56a 15
1:20p 20
3:01a 5
11:12a 55
```

    where each line contains a time (hour, followed by a colon, followed by a minute, followed by 'a' for AM or 'p' for PM), and a number of minutes, separated by a space. The number of minutes will never be negative.

    Write a program that, for each line in the file, displays the new time that you get by adding the supplied number of minutes to the supplied time. For example, using the sample data shown above, the output of the program would be:

```
1:11a 1:40p 3:06a 12:07p
```

    Hints:

    Read question 6 first.

    Write a loop to read through the file, one line at a time. For each line from the file, write a loop which calls, over and over again, the function described in question 6, to add the required number of minutes.

    Regardless of whether or not your answer to question 6 is correct, your answer may assume that the function described in 6 has been written by someone and works according to its specifications.

17. Write a program that displays a hexadecimal dump of a file named "testdata.dat", 16 bytes of the file per line of output. If the file cannot be read, an error message is displayed. Otherwise a dump looking similar to this is displayed:

```
54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 6f 66 20 74
68 65 20 65 6d 65 72 67 65 6e 63 79 20 62 72 6f 61 64 63
61 73 74 20 73 79 73 74 65 6d 2e 0a
```

    (Hint: printf("%02x", c) will display a char or an int in hexadecimal, with a leading 0 if needed. Read through the file a byte at a time, displaying each byte in hex. Display a newline after every 16th one, and when you reach the end of the file.)

18. Write a program that asks the user to enter a file name and then tells the user (1) how many characters are in the file, and (2) how many of those characters are spaces. If the user gives a bad file name, then an appropriate error message should be displayed instead.

APPENDIX C. Data Representation

Today, most operating systems, programming languages and hardware devices make the underlying assumption that sophisticated users are familiar with how information is stored in a computer. Without this knowledge, anything other than very basic use of a computer is impossible, as is understanding of many of the fundamental design concepts of digital computers.

Numbering Systems

We use the decimal system, also called base 10, because we have ten fingers. In decimal, a numeric quantity is represented by a series of digits from 0 to 9. The rightmost digit represents a number of units. The digit to the left of that represents the number of tens of units, the next number to the left is the number of tens of tens of units (=hundreds of units), the next is the number of tens of tens of tens of units (=thousands of units), and so on. The actual quantity expressed by the number is the sum of all these quantities.

Example: Decimal number 3450

```
        3  4  5  0
       /   |  |   \
      /    |  |    Zero single items
     /     |  Five groups of ten items
    /      Four groups of ten groups of ten items
   Three groups of ten groups of ten groups of ten items
```

Any number other than 10 can be used as the base in which numbers are represented (i.e. there is nothing special about humans usually having ten fingers).

For example, if we had been born with one finger on each hand (two fingers in total), we might count in binary (or base 2). Here each digit would range from 0 to 1, and each successive digit from right to left would represent the number of pairs of what the previous digit represented.

Example: binary 1101 is the same as decimal 13

```
       Binary number
        1  1  0  1
       /   |  |   \
      /    |  |    One single item (= one item)
     /     |  Zero groups of two items (= none)
    /      One group of two groups of two items (= four)
   One group of two groups of two groups of two items (= eight)
```

Eight plus four plus one is, in decimal, 13.

A simple way to convert from binary to decimal:

Write successive powers of two over each digit from right to left, and add up those numbers under which a 1 appears.

Example: Convert binary 110100 to decimal

```
   32   16    8    4    2    1
    1    1    0    1    0    0

   32 + 16 + 4 = 52 in decimal
```

A simple way to convert from decimal to binary:

Divide the number by two. The remainder (which will be either 0 or 1) is the rightmost binary digit. Divide the quotient by two. This remainder will be the next binary digit to the left. Continue dividing the successive quotients by two and using the remainder as the next binary digit to the left, and stop when the quotient is finally zero.

Example: Convert decimal 52 to binary

```
   52 / 2 = 26 remainder 0 -------------|
   26 / 2 = 13 remainder 0 ------------||
   13 / 2 =  6 remainder 1 -----------|||
    6 / 2 =  3 remainder 0 ----------||||
    3 / 2 =  1 remainder 1 ---------|||||
    1 / 2 =  0 remainder 1 --------||||||
                                   ||||||
   The binary equivalent is:       110100
```

Exercises:

Note how the number base is written as a subscript to the number.

```
   100101   =       ?
          2    ------------ 10

    00011   =       ?
          2    ------------ 10

     287    =       ?
         10    ------------- 2

     101    =       ?
         10    ------------- 2
```

Other bases are just as easy. For example, hexadecimal (base 16) uses digits from 0 to 15. Since we can't represent the quantities ten, eleven, ... , fifteen as a single digit with our normal digits, we have to make up some symbols to represent them. Typically, computer people use A for 10, B for 11, ... ,

and F for 15.

Example: Convert hexadecimal A5E to decimal

```
       A   5   E
      /    |    \
     /     |     Fourteen single items (= 14)
    /      Five groups of 16 items (= 5x16 = 80)
   Ten groups of 16 groups of 16 items (= 10x16x16 = 2560)

   2560 + 80 + 14 = 2654 in decimal
```

Example: Convert decimal 3107 to hexadecimal

```
   3107 / 16 = 194 remainder  3-----------|
    194 / 16 =  12 remainder  2----------||
     12 / 16 =   0 remainder 12---------|||
                                        |||
   The hexadecimal equivalent is:       C23
```

When using non-decimal numbering systems, the numeric base should always be mentioned if it is not entirely obvious from the context. Thus

```
   101  (= 5  ), 101   and  101  (= 257  )
      2     10      10         16      10
```

will not get confused with each other. In the case of hexadecimal (often called simply hex for short), many other styles of identifying this base are used. 101x, 101h, x101 and $101 are all common notations used to reference the hexadecimal number 101.

Any number base can be used. Because it is easier and more reliable to design electronic circuits that recognize only two states, "on" and "off", than to design circuits that recognize three or more different states, computers today are based on the binary system. Since any numeric quantity can be represented as a sequence of 0s and 1s, any numeric quantity can be represented as a sequence of "offs" and "ons".

Even though it is easy (relatively speaking!) to design binary computers, it is difficult for humans to deal with large volumes of binary numbers. Imagine a page full of 0s and 1s - picking out patterns, even if you knew what different sequences of 0s and 1s were supposed to mean, would be extremely laborious. For this reason, computer people usually convert binary data to some other number base before working with it.

It turns out that it is very simple to convert binary numbers to hexadecimal, and vice versa. Mathematically, because 16 (the number base for hex) is 2 (the number base for binary) raised to the 4th power, there is a direct correspondence between four binary digits and one hex digit. Notice, for example, that the largest 4-digit binary number is 1111, while the largest single hexadecimal digit is F. Both of these, of course, represent the same thing: the decimal number 15.

To convert a hex number to binary, you could convert the hex number to decimal, then convert the decimal number to binary. The easy way, however, is to take each hex digit, and replace it with the equivalent 4-digit binary number. To go from binary straight to hexadecimal, group the binary number into groups of 4 digits starting from the RIGHTmost digit. (A few leading zeros may have to be added to the binary number to make this work out properly). Each group of 4 can then be replaced with the corresponding hex digit. The following table summarizes the equivalences between a hex digit and 4 binary digits.

Equivalence_between_binary_and_hexadecimal

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Example: Convert hexadecimal A40D to binary

```
  A    4    0    D
1010 0100 0000 1101
```

The binary equivalent is 1010010000001101

Example: Convert binary 11100000100001 to hex

```
0011 1000 0010 0001
  3    8    2    1
```

The hex equivalent is 3821

Note in the last example how the binary digits were grouped from the right to the left, making it necessary to pad with two leading zeros.

Hexadecimal, used as a shorthand for binary, is particularly useful because on most machine a byte is eight bits (each "binary digit" is called a bit) long. Thus, one byte of data can be represented by exactly 2 hex digits, which is much more compact than 8 binary digits. (Four bits, which can be represented by exactly one hex digit, is often called a nibble or nybble).

Incidentally, bits towards the left are referred to as high-order bits, and bits towards the right are referred to low-order bits, since they correspond to higher and lower powers of two, respectively. Similarly, people often describe the bytes or nibbles comprising a larger value as being higher- or lower-order, depending on their relative positions within the value.

Some older devices (for example, calculators from a few years ago) do not have the letters A, B, ... F, but only have the digits 0 through 9. For this reason, octal (base eight) has occasionally been used as a shorthand for binary. Eight is two to the third power, and so three binary digits are equivalent to one octal digit in the same way that four bits are equivalent to one hex digit. The advantage of octal is that it only uses the digits from 0 to 7, and so can be implemented on devices that don't support the alphabet. The main disadvantage to octal is that a byte (on most machines) works out to be exactly two and two thirds octal digits long (3 bits plus 3 bits plus 2 bits), which can be awkward at times. On the old DEC-20 (one of DEC's pre-VAX minicomputers, which had 9-bit bytes), however, octal was more convenient than hex, since a byte was exactly 3 octal digits long (and an awkward two and a quarter hex digits).

Example: Convert decimal 98 to octal

```
98 / 8 = 12 remainder  2----------|
12 / 8 =  1 remainder  4----------||
 1 / 8 =  0 remainder  1---------|||
                                 |||
The octal equivalent is:        142
```

Example: Convert octal 260 to decimal

```
       2   6   0
      /    |    \
     /     |     Zero single items (= 0)
    /      Six groups of 8 items (= 6 x 8 = 48)
   Two groups of 8 groups of 8 items (= 2 x 8 x 8 = 128)

   128 + 48 + 0 = 176 in decimal
```

Example: Convert binary 1101110010 to octal

```
   001 101 110 010
    1   5   6   2

   The octal equivalent is 1562
```

Example: Convert octal 570 to binary

```
    5   7   0
   101 111 000

   The binary equivalent is 101111000
```

Example: Convert hex A0E to octal

```
   Simplest method: Convert to binary, then to octal

     A    0    E
   1010 0000 1110
   (regrouping from the right...)
   101 000 001 110
    5   0   1   6

   The octal equivalent is 5016
```

Example: Convert octal 177 to hex

```
    1   7   7
   001 111 111
   (regrouping...)
   0000 0111 1111
     0    7    F

   The hex equivalent is 7F
```

Note that conversions between octal and hex COULD be done by converting to base 10 as the middle step. Since octal-binary and hex-binary conversions are so simple, however, it is much easier to use binary as the middle step.

Exercise: Complete the following table

Number Base			
2	8	10	16
1101			
	6		
	71		
		99	
			ABC

Exercise:

You just got a marvelous new printer. It has two different fonts (Pica and Roman), two print qualities (draft and "Near Letter Quality"), two different horizontal pitches (10 characters per inch and 12 characters per inch), two different vertical pitches (6 lines per inch and 8 lines per inch), optional italics, optional bold, and optional underline.

The manual says that you can change any of these settings by sending two bytes to the printer. The first byte should be what the manual calls the escape character, and it says that the binary pattern for this is 00011011. The second byte should be 8 bits where each bit controls a different feature, as follows:

1st (leftmost) bit: Always zero
2nd bit: (font selection) 0 for Pica, 1 for Roman
3rd bit: (print quality) 0 for draft mode, 1 for NLQ
4th bit: (horizontal pitch) 0 for 10 cpi, 1 for 12 cpi
5th bit: (vertical pitch) 0 for 6 lpi, 1 for 8 lpi
6th bit: (italics select) 0 for normal, 1 for italics
7th bit: (bold select) 0 for normal, 1 for bold print
8th bit: (underline select) 0 for normal, 1 for underline

The manual also says that you can send a byte to the printer in BASIC by issuing the statement LPRINT CHR$(x); where x is the byte you want to send. After some experimentation, you discover that BASIC expects x to be a decimal number! Your problem is to find the two decimal numbers you need to send the two bytes that will set the printer for NLQ Roman italics, as small as possible both vertically and horizontally.

Non-Decimal Arithmetic

Often a programmer has, say, binary or hex data and needs to do simple arithmetic with it, such as addition or subtraction. One technique is to convert the data to decimal, do the arithmetic, then convert back to the base in question. This will work. But addition, subtraction and multiplication are quite easy to do even in non-decimal bases, so it is usually simpler to do the arithmetic directly. Essentially all you have to remember is

that when you carry (or borrow) a 1, you are carrying (or borrowing) 2, 8 or 16 (depending on what base you're working with) and not 10.

Example: Hexadecimal addition; A0F + F5

```
                    1                11
   A0F             A0F              A0F
    F5      --->    F5       --->    F5
   ---             ---              ---
     4 carry 1       04 carry 1      B04
```

Note that F+5 is 15+5 = 20 (decimal) which is 14 hex. Similarly, F+1 is 15+1 = 16 (decimal) which is 10 hex. Finally A+1 is 10+1 = 11 (decimal) which is B in hex.

Looking closely, you should notice that the arithmetic is really being done in decimal, since the decimal addition tables are the only ones that people normally have memorized. It's just that the conversion to decimal and back again is being done one digit at a time, so it is easier to perform than converting larger numbers.

Example: Hexadecimal subtraction; 207 - B6

```
                      1            1
   2 0 7   borrow 1   1 0 7        1 0 7
  -  B 6        ---> -  B 6  ---> -  B 6
   -----             -----        -----
       1               5 1        1 5 1
```

Here 7-6 = 1 (same in decimal and hex). 0-B is not possible, so 1 is borrowed from the next column, making 10-B which is 16-11 = 5 (decimal). 1 was borrowed, leaving 1 in the leftmost column.

Example: Hexadecimal multiplication; D21 x 32

```
     D21
    x 32
     ---
    1A42
   2763
   -----
   29072 (remember, this is a hex number)
```

Example: Binary subtraction; 100011-110

```
                            1
   1 0 0 0 1 1  borrow 1    0 0 0 0 1 1  borrow 1
  -      1 1 0       ---> -       1 1 0       --->
   -----------             -----------
          0 1                       0 1
```

```
    1                          1
  0 1 0 0 1 1  borrow 1    0 1 1 0 1 1
- 1 1 0        --->    -   1 1 0
 -----------              -----------
        0 1                 1 1 1 0 1
```

If all this borrowing confuses you, try to do a decimal subtraction such as 20005-99, just to remind yourself how borrowing actually works.

Exercise: Non-decimal arithmetic

```
Hexadecimal
  A001        2B03        E10
 + FFF       -1C00       x 4A
  ----        ----        ---


Octal
  724         3301         41
 +555        - 277        x23
  ----        ----         --


Binary
  1100010      1010001      100101
 +   1111     - 111000     x  1100
  -------      -------      ------
```

Exercise:

You have been told that a program is located starting at byte number A3F00h in the computer's memory. You are also told that there is a programming error in the program at byte number A4010h in memory. You must calculate how many bytes the error is from the start of the program.

Two's Complement

It should be clear by this point that binary addition and multiplication are somewhat simpler processes than binary subtraction. (If you think about it, you might even realize that binary multiplication is really just a series of additions). It happens that, through a mathematical construct called the two's complement, subtraction can be made into one simple operation (the two's complement) followed by an addition. Such simplifications are sought after by computer circuit designers who prefer to design as simple and as few different circuits as possible to increase the reliability and decrease the size of

their designs.

First, let us define what a two's complement is. Let us suppose that a fixed size for a number has been determined in the design of the computer. This size generally determines how many wires run throughout the computer. For example, an addition circuit might be built to handle 16 bit numbers, so it would have two sets of 16 wires going in, and one set of 16 wires coming out. For our current purposes, let us assume that this size (often called the word size of the computer) is 16 bits, though many computers have larger word sizes, such as 32 bits or 64 bits.

To form the two's complement of a number, first write it in binary with leading zeros to pad it out to the full size of a word. Then, "flip all the bits" (i.e. make each 0 into a 1, and each 1 into a 0). Finally, add one to the "flipped" result, and you have the two's complement of the original number.

Example: Two's Complement of decimal 48

```
   Step 1: Write all 16 bits: 0000000000110000
   Step 2: Flip all the bits: 1111111111001111
   Step 3: Add 1:                           + 1
                              ----------------
   Two's Complement of 48 is: 1111111111010000
```

One neat property of the two's complement is that if you take the two's complement of a two's complement, you get the original number back again.

Example: Two's Complement of 1111111111010000

```
   1111111111010000 --(flip bits)--> 0000000000101111
                                                   + 1
                                     ----------------
                                     0000000000110000
                              (which is 48 in decimal)
```

So what is the use of this? Well, it is possible to prove mathematically that adding the two's complement is the same as subtracting the original number. While we will not attempt a mathematical proof here, we can demonstrate this with an example.

Example: 26 - 48 = -22

```
26 in binary:              0000000000011010
2's complement of 48:     +1111111111010000
                           ----------------
                           1111111111101010
```

```
2's complement of 1111111111101010 is:
    0000000000010101
                 + 1
    ----------------
    0000000000010110 which is binary for 22!
```

The net result is that if the two's complement form is used to represent a negative number, then a subtraction circuit in the computer is unnecessary, since addition of the two's complement can be used in its place.

If this scheme is used (and it is on almost all modern computers), there are a couple of restrictions. First, the largest positive number that can be used is a 0 followed by all 1s. (In our 16-bit example, the largest positive number is 0111111111111111 or 32767 in decimal). This is so that we can distinguish a two's complement from a positive (a two's complement will always begin with a one). (Note also that the "largest" negative number is 1000000000000000 or -32768 in decimal). The second restriction is that some additions will cause an overflow condition, creating a number that exceeds the limits. For example, adding 0100000000000000 (decimal 16384) to itself should yield a large positive number (32768), but in fact yields 1000000000000000 (decimal -32768). (It is not difficult to design the addition circuit to trap such overflow errors).

Exercises: Two's Complement

In the following, assume a 16-bit word.

a. Find the two's complement of:

   11001

   111111111111111

   1111111111111111

b. The following hex value represents the two's complement of a number. Find the decimal value of the number.

   FE02

   (Hint: first convert the hex to binary)

Data Format: Unsigned Binary

Numeric information which will always be a non-negative integer (i.e. a whole number greater than or equal to zero) is often stored in a computer in binary. Usually, a fixed size (such as one byte for numbers up to 255, two bytes for numbers up to 65535 or four bytes for numbers up to 4294967295) is used to hold such a number. (The size is usually related to the word size of the computer). This format for data is called unsigned binary because the data is stored as a binary number, with no way to express a negative quantity. The size, unless obvious from the context, should be specified as well, as in the phrase "a 32-bit unsigned binary number". In C, as you'll learn later, an example of unsigned binary is the data type "unsigned int".

Data Format: Signed Binary

Numeric information, which will always be an integer, but may be positive or negative, is often stored as a binary number with a leading zero for a positive number, and the two's complement of a binary number for a negative number. This format is called signed binary. In this case, as with unsigned binary, a fixed size is usually used. A one byte signed binary number would therefore range from -128 to +127, while a two byte signed binary ranges from -32768 to +32767, and a four byte signed binary, from -2147483648 to +2147483647. Remember that the first bit of a negative number will always be a 1 while that of a positive will always be a 0. In C, "int" is an example of a signed binary type.

Data Format: Floating Point

The numeric data types so far assume that only integers (whole numbers) are to be stored. If fractional amounts are required, one scheme (called fixed point) is to decide how many decimal places (or "binary places") of accuracy is desired. All numbers are then calculated to that degree of accuracy, and stored, without the decimal point (or binary point), as a whole number. Only when data is output on reports or screens is the decimal point inserted. As an example, suppose monetary values are being stored. Then two decimal places of accuracy are what is required (for dollars and cents). A four byte signed binary value could be used, storing the number of cents, and would be able to store values from -21,474,836.48 to +21,474,836.47. Note that special output routines would be required to display these values properly.

In some areas, such as engineering, statistics or science, an approach like fixed point is not acceptable, because often values are so large (100 digits, say) or so small (0.00000000000001 for example) that it is impractical to attempt to store numbers to the accuracy required. In these areas,

scientific notation is generally used to represent numbers in everyday use. In scientific notation, 16000000 is written as 1.6 x 10^7 (here ^ is used to represent exponentiation, or "to the power"), and .0000000034 is written as 3.4 x 10^-9. In this way, both very large and very small numbers can be written in a fairly compact manner.

In scientific notation, a number like 1.6 x 10^7 has three parts. The 1.6 is called the mantissa, the 10 is the base and the 7 is called the exponent. Note that both the mantissa and the exponent can be signed. A negative mantissa means that the numeric quantity is negative, whereas a negative exponent means the number is less than 1 in magnitude.

On computers, a variation on this scientific notation is called floating point format. In floating point, a numeric quantity is broken down into a mantissa, a base (usually either 2 or 16, not 10) and an exponent. Since all floating point numbers on the same computer would use the same base, only the sign, the mantissa and the exponent need to be stored.

There are almost as many different floating point formats as there are types of computer, each being a minor variation on the same theme. Commonly, either 32, 64 or 80 bits are used to store a number. One of these bits is for the sign (0 means positive, 1 means negative). Some of these bits are devoted to the mantissa and the rest to the exponent. A typical 32-bit floating point format might have 25 bits devoted to the mantissa and 7 for the exponent. If the base used were 2, this would allow numbers from 2 to the -128th power to 2 to the 127th power (approximately 10 to the -38th to 10 to the 38th).

Since floating point formats vary considerably in exactly which bits are used for what part of the number, it is important only to remember the general structure of floating point. The bit-by-bit details (whether the exponent or the mantissa comes first, for example) can be looked up in the appropriate manual, should you need to investigate floating point format further.

Data Format: Character

Since computers are only designed to interpret binary data, textual information must somehow be converted to binary. The method used is simplistic: a number is associated with each possible character. Two codes are in wide use: EBCDIC (pronounced EB-sih-dik) and ASCII (pronounced ASS-key). A newer code, Unicode, is beginning to gain promenance.

EBCDIC (Extended Binary Coded Decimal Interchange Code) is used on IBM mini and mainframe computers, as well as mainframes from some other manufacturers. It associates an 8-bit (one byte) number with each character, and is descended from the hole

patterns used to represent characters on early punched card systems (IBM has been around a long time!). Generally speaking, C is not used on EBCDIC-based machines.

ASCII (American Standard Code for Information Interchange) is used on most other computers, and was developed by a committee with representatives from a variety of computer manufacturers. By definition, ASCII associates a 7-bit number with each character (allowing only 128 different characters instead of EBCDIC's 256), although most computer manufacturers that use ASCII provide an 8-bit version of the code which uses the additional 128 characters to provide a richer set of characters than 7-bit ASCII allows. For example, on the IBM PC - an ASCII computer - characters 128 through 255 are used to represent mostly "graphics" characters. These characters, supported by the video hardware of the PC, allow the drawing of boxes and other simple shapes on the screen in text mode. ASCII is closely associated with C programming (the char data type), as almost every environment supporting the C language is ASCII-based.

Unicode attempts to address the English-language bias of the more traditional ASCII and EBCDIC codes. In Unicode, 16 bits are used to represent each character. Unicode is based on ASCII, in that the values from 0 to 127 represent the same characters in both codes. But the additional 65408 character codes available in Unicode allow room for all commonly used alphabets, as well as shapes which can be combined to generate characters for non-alphabet based languages such as Chinese. Two obvious downsides of Unicode are that English-based text requires twice the storage space than would be required using ASCII, and that most programming languages have built in support for either EBCDIC or ASCII, rather than Unicode, thus requiring special routines to be used to support Unicode.

All of these codes include "printable" characters (such as the alphabet, digits and punctuation marks typically found on a typewriter) and "non-printable" characters (such as the FF character which causes a printer to go to the top of a new page and the BEL character which causes a terminal to beep, as well as other characters used for control of devices other than display devices).

A table showing the ASCII code is included in the next Appendix.

Other Data Formats

There are many different possible data formats, and there are several others in wide use, although they will be most commonly encountered on EBCDIC-based machines using traditional business programming languages such as COBOL and RPG.

Zoned decimal is a data format often used in business

applications. A number is stored in a decimal (base 10) format rather than a binary format. Essentially, the decimal representation of the number is stored as character data. In a system using the EBCDIC code, for example, the number 385 stored in a 6-byte zoned decimal field would be stored as the character string "000385". The actual data as stored shown in hex would be F0F0F0F3F8F5 (the EBCDIC for the digit "0" is F0, and so on). Note that the same number stored as a 4-byte binary would have the hex representation 00000181.

With signed binary numbers, the first bit is used to indicate the sign, and two's complement form is used for negative numbers. Zoned decimal format uses an even stranger method to specify a signed quantity. The reason for the strangeness is due to the way the Hollerith code (the punched card code from which EBCDIC is descended) works. In the Hollerith code, if a number punched on the card is signed, the last digit has an extra hole punched on the card to indicate either a + or a - sign. When the Hollerith code was made into the EBCDIC code, it turned out that the area in which the sign hole was punched was made to be the first half of the byte, called the zone portion. The area of the card in which the number itself was punched was made into the second half of the byte, called the digit portion. For example, the digit "5" in EBCDIC is represented by the hex number F5. The F (the first half of the byte) is the zone portion of that byte, and the 5 is the digit portion. While no punch in the zone portion of the card is represented in EBCDIC with the hex number F, the punch representing a + sign is represented with a hex C and the punch for the - sign is represented with a hex D. Thus, if a zoned decimal number is positive, the first half of the last byte is a hex C, and if it is negative, the first half is a D.

Example: Zoned Decimal Numbers (EBCDIC)
(Assuming a 4-byte zoned decimal number)

	--------viewed as--------	
Quantity	Hex	Character
385	F0F3F8F5	0385
+385	F0F3F8C5	038E
-385	F0F3F8D5	038N

The character representation is shown in the table above because most programmers that enter data directly into zoned decimal format (for creating sample data test usually) would use a text editor to create the test data rather than a hexadecimal editor. The data would be entered as if it were character data, even though it isn't quite because of the strange way of specifying the sign. The following table shows the character representing all possible last bytes of a zoned decimal number.

Example: Last Digit of Zoned Decimal Numbers
(Assuming the EBCDIC code is used)

Last	-----Character (Hex Code)-----		
Digit	Unsigned	Positive	Negative
0	"0" (F0)	"{" (C0)	"}" (D0)
1	"1" (F1)	"A" (C1)	"J" (D1)
2	"2" (F2)	"B" (C2)	"K" (D2)
3	"3" (F3)	"C" (C3)	"L" (D3)
4	"4" (F4)	"D" (C4)	"M" (D4)
5	"5" (F5)	"E" (C5)	"N" (D5)
6	"6" (F6)	"F" (C6)	"O" (D6)
7	"7" (F7)	"G" (C7)	"P" (D7)
8	"8" (F8)	"H" (C8)	"Q" (D8)
9	"9" (F9)	"I" (C9)	"R" (D9)

Realize that zoned decimal format is less compact than binary format. A four byte signed binary number can store values from -2147483648 to +2147483647, whereas a four byte zoned decimal number can only hold values from -9999 to +9999. Zoned decimal format tends to be used in languages such as COBOL, which have a long history dating back to the days when programmers had to enter the data in machine readable form (e.g. using punched cards).

Packed decimal format is a compromise between the compactness of binary format and the decimal orientation of zoned decimal. In packed decimal, rather than using an entire byte to store each decimal digit, only half a byte is used. The very last half-byte of the number is used to store the sign of the number, using the same hex codes that EBCDIC zoned decimal does: F for an unsigned number, C for a positive number and D for a negative.

Example: Packed Decimal Numbers

(Assuming a 4-byte packed decimal number)

Quantity	Stored As (Hex)
385	0000385F
+385	0000385C
-385	0000385D

Packed decimal format is used because some machines (such as the IBM mainframe computers) have special hardware instructions for doing arithmetic directly on packed decimal numbers, rather than internally converting the decimal quantities to binary, performing calculations, and converting the numbers back to decimal. These special instructions, while less efficient than normal binary arithmetic instructions, can usually be used to calculate with larger numbers than the wordsize of the computer might allow for binary calculations. For example, the largest

binary number on the IBM/370 is 8 bytes big, which allows for numbers approximately 18 to 19 decimal digits long. The largest packed decimal number on this machine is 16 bytes big, allowing numbers 31 decimal digits long.

Exercise:

Explain why a 16 byte packed decimal number may be up to 31 decimal digits long.

Using Different Data Formats

With all the different data formats available (and the possibilities are endless, we have only discussed the more popular formats), how does the computer know which format to use when? It doesn't. To the computer, data all looks like 0s and 1s. It is the programs that the computer executes which interpret the data one way or another. That means that if a program is written to access data in one format, but the data is actually stored in another format, the data used will effectively be garbage.

Example - Incompatible data formats:

In the computer labs, you are continually asked not to print object files or executable files. If you attempt this, the printer seems to go wild, and the technician on duty usually has to shut it off to reset it. The program which prints a file on the printer assumes that the file will contain only printable characters, and views the file simply as a stream of bytes to be sent to the printer.

At the same time, the printer is built to print any printable characters sent to it. Some non-printable characters are used to do things like reset the top-of-form, change character sets, go to the top of a new page, and so on.

Both object files and executable files contain machine language instructions, which are just binary codes. Some of these codes, by coincidence, may be printable characters, while others will be unprintable. When you run the program to print the file, all the bytes in the file will be sent to the printer as if they were character data, which they are not. Consequently, the printer does the best it can, printing the printable characters (regardless of how random they are), acting on those unprintables that have a control function for the printer, and ignoring the rest.

If you really want to see the data that makes up an executable file, you should use a dump utility, such as the UNIX od (octal dump) command or the MS-DOS DEBUG command.

APPENDIX D. ASCII Table

Code Dec	Code Hex	Character	Code Dec	Code Hex	Character	Code Dec	Code Hex	Character
0	00	NUL	43	2B	+	86	56	V
1	01	SOH	44	2C	,	87	57	W
2	02	STX	45	2D	-	88	58	X
3	03	ETX	46	2E	.	89	59	Y
4	04	EOT	47	2F	/	90	5A	Z
5	05	ENQ	48	30	0	91	5B	[
6	06	ACK	49	31	1	92	5C	\
7	07	BEL (bell)	50	32	2	93	5D	]
8	08	BS (backspace)	51	33	3	94	5E	^
9	09	HT (tab)	52	34	4	95	5F	_
10	0A	LF (linefeed)	53	35	5	96	60	`
11	0B	VT	54	36	6	97	61	a
12	0C	FF (formfeed)	55	37	7	98	62	b
13	0D	CR (carriage return)	56	38	8	99	63	c
14	0E	SO	57	39	9	100	64	d
15	0F	SI	58	3A	:	101	65	e
16	10	DLE	59	3B	;	102	66	f
17	11	DC1	60	3C	<	103	67	g
18	12	DC2	61	3D	=	104	68	h
19	13	DC3	62	3E	>	105	69	i
20	14	DC4	63	3F	?	106	6A	j
21	15	NAK	64	40	@	107	6B	k
22	16	SYN	65	41	A	108	6C	l
23	17	ETB	66	42	B	109	6D	m
24	18	CAN	67	43	C	110	6E	n
25	19	EM	68	44	D	111	6F	o
26	1A	SUB	69	45	E	112	70	p
27	1B	ESC (escape)	70	46	F	113	71	q
28	1C	FS	71	47	G	114	72	r
29	1D	GS	72	48	H	115	73	s
30	1E	RS	73	49	I	116	74	t
31	1F	US	74	4A	J	117	75	u
32	20	SP (space)	75	4B	K	118	76	v
33	21	!	76	4C	L	119	77	w
34	22	"	77	4D	M	120	78	x
35	23	#	78	4E	N	121	79	y
36	24	$	79	4F	O	122	7A	z
37	25	%	80	50	P	123	7B	{
38	26	&	81	51	Q	124	7C	|
39	27	'	82	52	R	125	7D	}
40	28	(	83	53	S	126	7E	~
41	29	)	84	54	T	127	7F	DEL
42	2A	*	85	55	U			

APPENDIX E. Operator Precedence

Precedence	Operator	Grouping
1 highest	[] array index	left-to-right
1	++ post-increment	left-to-right
1	-- post-decrement	left-to-right
2	++ pre-increment	right-to-left
2	-- pre-decrement	right-to-left
2	+ (unary) identity	right-to-left
2	- (unary) negation	right-to-left
2	& (unary) address	right-to-left
2	* (unary) pointer resolution	right-to-left
2	! logical not	right-to-left
3	(type) cast	right-to-left
4	* (binary) multiplication	left-to-right
4	/ division	left-to-right
4	% remainder	left-to-right
5	+ (binary) addition	left-to-right
5	- (binary) subtraction	left-to-right
6	< less than	left-to-right
6	<= less than or equal to	left-to-right
6	> greater than	left-to-right
6	>= greater than or equal to	left-to-right
7	== equality	left-to-right
7	!= inequality	left-to-right
8	&& logical and	left-to-right
9	|| logical or	left-to-right
10	?: conditional	right-to-left
11 lowest	= assignment	right-to-left
11	+=, -=, *=, /=, %=	right-to-left

Operators with the same precedence level number have the same precedence regardless of their order in this table.

"Grouping" refers to the precedence of multiple consecutive operators of the same level. For example, -*ptr is the same as -(*ptr) since unary - and * have the same precedence (2) and group right to left, while x - y + z is the same as (x - y) + z since binary - and + have the same precedence (5) and group left to right.

APPENDIX F. Reserved Words

The following C keywords should be avoided when creating identifier names:

```
auto
break
case
char
const
continue
default
restrict
do
double
else
enum
extern
float
for
goto
if
inline
int
long
register
return
short
signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile
while
_Bool
_Complex
_Imaginary
```

APPENDIX G. Decision Tables

Decision tables are a technique used to help formulate logic in the case where you have a complex set of conditions and appropriate actions.

The basic format of a decision table is to have a list of conditions followed by a list of actions to be taken, depending on the conditions. To the right of the list are columns representing the different possible situations, called results, where T/F is used to indicate whether a particular condition is true or false, and X or nothing is used to indicate whether a particular action should be taken.

For example, consider a chequing account which allows overdrafts. If a cheque is written, then it is deducted from the account balance, as well as a $.50 charge. However, if the cheque (or the $0.50 charge) would cause the balance to become negative, then and overdraft charge of $10.00 should be charged instead of $0.50, as long as the negative balance does not exceed $1000. If the cheque would cause the balance to become negative in excess of $1000, then the cheque will bounce, and $15 is deducted from the account as an NSF charge, instead of any of the other charges.

This confusing mess can be programmed a number of ways, but lets make a decision table. In this case there are two different conditions and four distinct actions:

Conditions	Results		
Balance - (Cheque + 0.50) >= 0	T	F	F
Balance - (Cheque + 10) >= -1000		T	F
Actions			
Deduct Cheque from Balance	X	X	
Deduct 0.50 from Balance	X		
Deduct 10.00 from Balance		X	
Deduct 15.00 from Balance			X

Notice how in the Results section, a blank condition entry means that for this particular result, it doesn't matter whether or not the condition is true or false. (Alternatively, we could have had four results columns, TT, TF, FT, FF, with the same set of actions for the first two columns).

Each column in the Results section determines one part of a nested if/else. In this example, following the decision table, we would get

```
if (Balance - (Cheque + 0.50) >= 0) {
    Balance -= Cheque;
    Balance -= 0.50;
}
else if (!(Balance - (Cheque + 0.50) >= 0) &&
         Balance - (Cheque + 10) >= -1000) {
    Balance -= Cheque;
    Balance -= 10;
}
else if (!(Balance - (Cheque + 0.50) >= 0) &&
         !(Balance - (Cheque + 10) >= -1000)) {
    Balance -= 15;
}
```

Note how in each section of the if/else, the relevant conditions (using NOT (!) in the case of a condition entry of F) are joined with AND (&&).

Note also that this process doesn't necessarily generate the simplest or most efficient code. It will generate correct code, which you can then simplify, if you are able. A decision table can help you to analyze a seemingly confusing mess and generate working code. It can also be used as documentation, as a way of graphically describing what a section of logic does.

APPENDIX H. Flowcharts

Flowcharting is another technique frequently used to assist the process of developing logic.

A flowchart is a pictorial representation of program logic, using arrows to show progression through the various steps of a program.

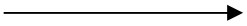
The following pages introduce the standard flowcharting symbols and show an example of a flowchart, along with the corresponding C code.

Certain people seem to be able to develop logic more easily by drawing flowcharts than by writing code, while others prefer to write code (or an approximation of code, omitting petty details, called pseudocode) directly. If you find you have trouble developing your program logic, you should try first creating your logic by drawing a flowchart, to see if it makes life easier for you.

Even if you decide that flowcharting is not a particularly useful technique for you, you should still be familiar with the standard flowcharting symbols, since you may be required to read flowcharts that others have developed, or to create flowcharts which show the logic of a program you have written, as part of the documentation.

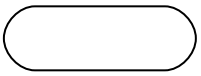
Basic Logic Flowchart Symbols

Sequence



Shows flow of logic by connecting two other symbols. If practical, make lines point down or to the left

Terminal



Used to show beginning or end of program. Will either have one line going out (beginning) or one line coming in (end)

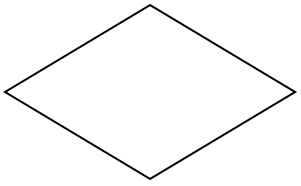
Normal Process

Signifies one step of the program, other than input, output or a decision. Always has one line coming in and one line going out

Input/Output Process

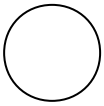
Signifies an input or an output operation (such as terminal I/O or file I/O). Always has one line coming in and one line going out

Decision



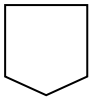
Used to show a decision. Always has one line coming in and two lines going out, where one represents a true condition and the other false

On-Page Connector



Used to join logic flows (with two or more lines in and one out), or to jump to another spot on the page (with one line going into one labeled connector, and another connector with the same label elsewhere with one line coming out

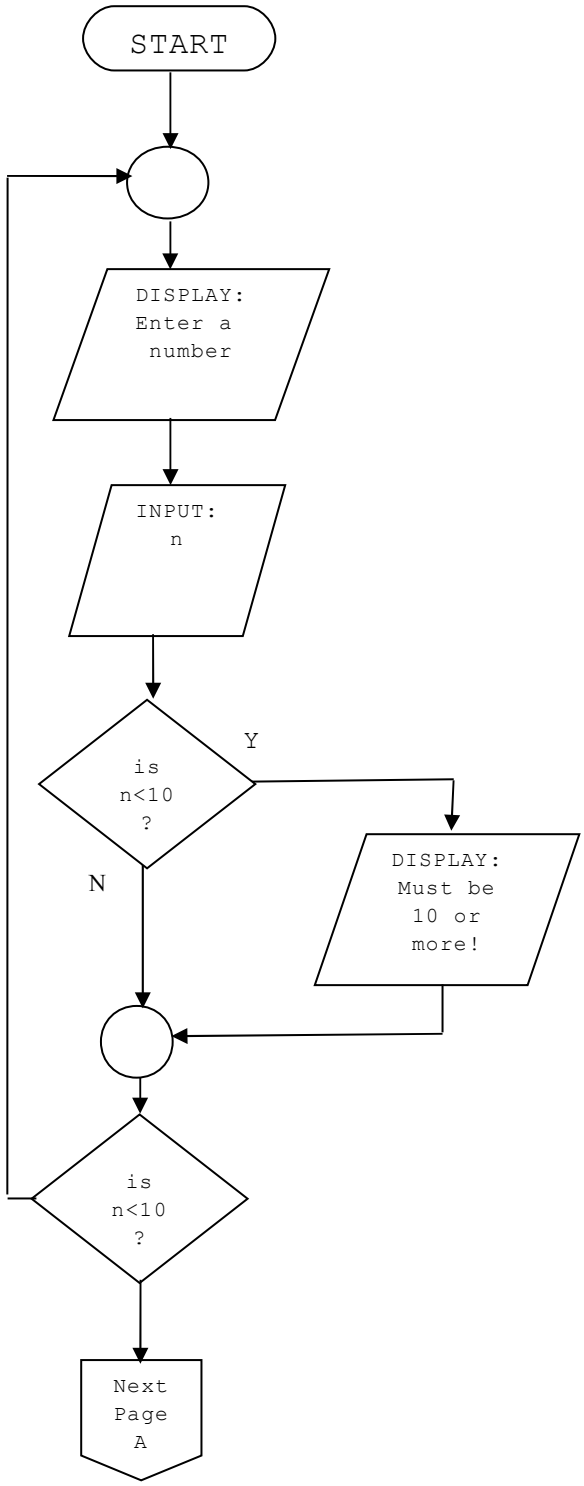
Off-page Connector

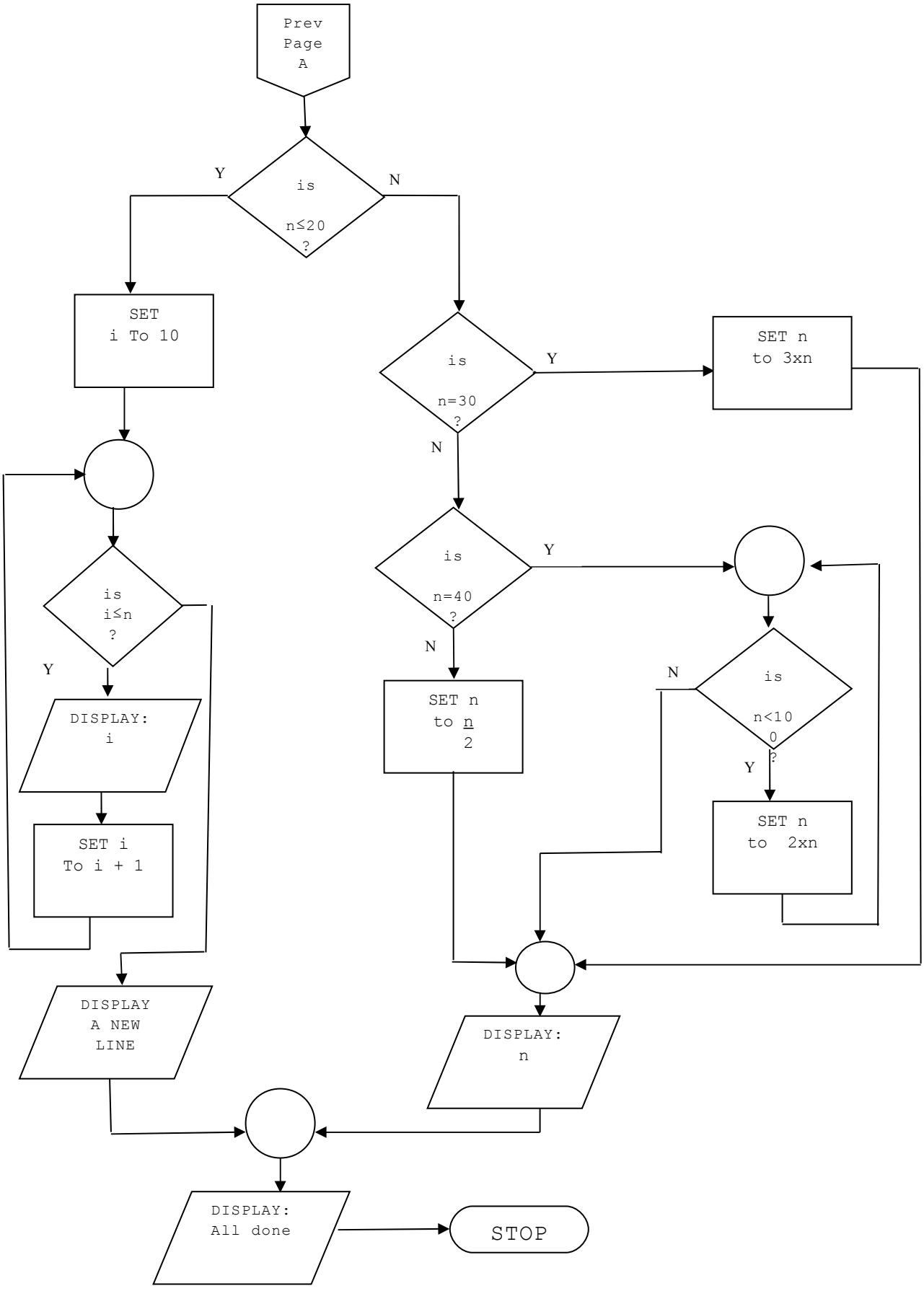


Used to jump to another spot on a different page. Always has one line going in, or one line coming out. Matching connectors should have identical labels, hopefully with the page containing the matching connector also shown

```
main ()
{
 int n, i;

 do {
  printf("Enter a number:");
  scanf("%d", &n);
  if (n < 10)
   printf("Must be 10 or more!\n");
 } while (n < 10);
 if (n <= 20) {
  for (i = 10; i <= n; i++)
   printf ("%d ", i);
  printf("\n"):
 }
 else {
  switch (n) {
   case 30:
    n = n * 3;
    break;
   case 40:
    while (n < 100)
     n = n * 2;
    break;
   default:
    n = n / 2;
  }
  printf("%d\n", n);
 }
 printf("All done\n");
}
```




Prev Page A
is n≤20 ?
Y
N
SET i To 10
is n=30 ?
Y
SET n to 3xn
N
is n=40 ?
Y
N
SET n to n 2
is n<10 0 ?
N
Y
SET n to 2xn
is i≤n ?
Y
DISPLAY: i
SET i To i + 1
DISPLAY A NEW LINE
DISPLAY: n
DISPLAY: All done
STOP