CE13

"C Programming Guide Book"

Spring 2009

Introductory C Programming
by
Steve Summit

These notes are part of the UW Experimental College course on Introductory C Programming. They are based on notes prepared (beginning in Spring, 1995) to supplement the book *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, or K&R as the book and its authors are affectionately known. (The second edition was published in 1988 by Prentice-Hall, ISBN 0-13-110362-8.) These notes are now (as of Winter, 1995-6) intended to be stand-alone, although the sections are still cross-referenced to those of K&R, for the reader who wants to pursue a more in-depth exposition.

C Programming Guide

A Short Introduction to Programming

At its most basic level, programming a computer simply means telling it what to do, and this vapid-sounding definition is not even a joke. There are no other truly fundamental aspects of computer programming; everything else we talk about will simply be the details of a particular, usually artificial, mechanism for telling a computer what to do. Sometimes these mechanisms are chosen because they have been found to be convenient for programmers (people) to use; other times they have been chosen because they're easy for the computer to understand. The first hard thing about programming is to learn, become comfortable with, and accept these artificial mechanisms, whether they make ``sense'' to you or not.

In fact, you shouldn't worry if some (or even many) of the mechanisms used for programming a computer don't make sense. It doesn't make sense that the cold water faucet has to be on the right side and the hot one has to be on the left; that's just the convention we've settled on. Similarly, many computer programming mechanisms are quite arbitrary, and were chosen not because of any theoretical motivation but simply because we needed an unambiguous way to say something to a computer.

In this introduction to programming, we'll talk about several things: skills needed in programming, a simplified programming model, elements of real programming languages, computer representation of numbers, characters and strings, and compiler terminology.

Skills Needed in Programming

I'm not going to claim that programming is easy, but I am going to say that it is not hard for the reasons people usually assume it is. Programming is not a deeply theoretical subject like Chemistry or Physics; you don't need an advanced degree to do well at it. (There are important principles of Computer Science, but it's possible to get a degree after studying them and to have only vague ideas of how to apply them to practical programming. Contrariwise, we'll experience many important Computer Science lessons by the seat of our pants, without bewildering ourselves with abstract notation; there are plenty of successful programmers who don't have Computer Science degrees.)

Comparing programming to some physical tasks, programming does not require some innate talent or skill, like gymnastics or painting or singing. You don't have to be strong or coordinated or graceful or have perfect pitch. Programming *does*, however, require care and craftsmanship, like carpentry or metalworking. If you've ever taken a shop class, you may remember that some students seemed to be able to turn out beautiful projects effortlessly, while other students were all thumbs and made the exact mistakes that the teacher told them not to make. What distinguished the successful students was not that they were better or smarter, but just that they paid more attention to what was going on and were more careful and deliberate about what they were doing. (Perhaps care and attention are innate skills too, like gymnastic ability; I don't know.)

Some things you *do* need are (1) attention to detail, (2) stupidity, (3) good memory, and (4) an ability to think abstractly, and on several levels. Let's look at these qualities in a bit more detail:

1. **Attention to detail**

   In programming, the details matter. Computers are incredibly stupid (more on this in a minute). You can't be vague; you can't describe your program 3/4 of the way and then say ``Ya know what I mean?'' and have the compiler figure out the rest. You have to dot your i's and cross your t's. If the language says you have to declare variables before using them, you have to. If the language says you have to use parentheses here and square brackets there and squiggly braces some third place, you have to.

2. **Stupidity**

   Computers are incredibly stupid. They do *exactly* what you tell them to do: no more, no less. If you gave a computer a bottle of shampoo and told it to read the directions and wash its hair, you'd better be sure it was a big bottle of shampoo, because the computer is going to wet hair, lather, rinse, repeat, wet hair, lather, rinse, repeat, wet hair, lather, rinse, repeat, wet hair, lather, rinse, repeat, ...

   I saw an ad by a microprocessor manufacturer suggesting the ``smart'' kinds of appliances we'd have in the future and comparing them to ``dumb'' appliances like toasters. I believe they had it backwards. A toaster (an old-fashioned one, anyway) has two controls, and one of them is optional: if you don't set the darkness control, it'll do the best it can. You don't have to tell it how many slices of bread you're toasting, or what kind. (``Modern'' toasters have begun to reverse this trend...) Compare this user interface to most microwave ovens: they won't even let you enter the cooking time until you've entered the power level.

   When you're programming, it helps to be able to ``think'' as stupidly as the computer does, so that you're in the right frame of mind for specifying everything in minute detail, and not assuming that the right thing will happen unless you tell it to.

   (This is not to say that you have to specify *everything*; the whole point of a high-level programming language like C is to take some of the busiwork burden off the programmer. A C compiler is willing to intuit a *few* things--for example, if you assign an integer variable to a floating-point variable, it will supply a conversion automatically. But you have to know the rules for what the compiler will assume and what things you must specify explicitly.)

3. **Good memory**

   There are a lot of things to remember while programming: the syntax of the language, the set of prewritten functions that are available for you to call and what parameters they take, what variables and functions you've defined in your program and how you're using them, techniques you've used or seen in the past which you can apply to new problems, bugs you've had in the past which you can either try to avoid or at least recognize by their symptoms. The more of these details you can keep in your head at one time (as opposed to looking them up all the time), the more successful you'll be at programming.

4. **Ability to abstract, think on several levels**

   This is probably the most important skill in programming. Computers are some of the most complex systems we've ever built, and if while programming you had to keep in mind every aspect of the functioning of the computer at all levels, it would be a Herculean task to write even a simple program.

   One of the most powerful techniques for managing the complexity of a software system (or any complex system) is to compartmentalize it into little ``black box'' processes which perform useful tasks but which hide some details so you don't have to think about them all the time.

   We compartmentalize tasks all the time, without even thinking about it. If I tell you to go to the store and pick up some milk, I don't tell you to walk to the door, open the door, go outside, open the car door, get in the car, drive to the store, get out of the car, walk into the store, etc. I especially don't tell you, and you don't even think about, lifting each leg as you walk, grasping door handles as you open them, etc. You never (unless perhaps if you're gravely ill) have to worry about breathing and pumping your blood to enable you to perform all of these tasks and subtasks.

   We can carry this little example in the other direction, as well. If I ask you to make some ice cream, you might realize that we're out of milk and go and get some without my asking you to. If I ask you to help put on a party for our friends, you might decide to make ice cream as part of that larger task. And so on.

   Compartmentalization, or *abstraction*, is a vital skill in programming, or in managing any complex system. Despite what I said in point 3 above, we can only keep a small number of things in our head at one time. A large program might have 100,000 or 1,000,000 or 10,000,000 lines of code. If it were necessary to understand all of the lines together and at once to understand the program, the program would be impossible to write or understand. Only if it is possible to think about small pieces in isolation will it ever be possible to work with a large program.

> Compartmentalization, powerful though it is, is not automatic, and not necessarily an instant cure for all of our organizational problems. We carry a lot of assumptions around about how various things work, and things work well only as long as these assumptions hold. To return to the previous example, if I ask you to go the store and get some milk, I'm assuming that you know which kind to get, where the store is, how to get there, how to drive if you need to, etc. If some of these assumptions weren't valid, or if there were several options for any of them, we might have to modify the way I gave you instructions. I might have to tell you to *drive* to the store, or to go to *Safeway*, or to get some *two percent* milk.
>
> Therefore, we can't simply compartmentalize all of our processes and subprocesses and forget about complexity problems forever. We have to remember at least some of the assumptions surrounding the compartmentalization scheme. We have to remember what we can and can't expect from the processes (people, computer programs, etc.) which we call on to do tasks for us. We have to make sure that we keep our end of the bargain and don't fall down on any of the commitments and promises we've made on the tasks we've been asked to do and which others are assuming we'll keep.
>
> Thinking about the mechanics of a design hierarchy, while also using that hierarchy to avoid having to think about every detail of it at every level all of the time, is one of the things I mean by ``thinking on several levels." It's tricky to do (obviously, it's tricky even to describe), but it's the only way to cut through large, complex problems.

What's hard about programming (besides maybe having trouble with the four traits above) is mostly picky little detail and organizational problems, and people problems. A large program is a terribly complex system; a large programming project worked on by many people has to work very hard at peripheral, picayune tasks like documentation and communication if the project is to avoid drowning in a flood of little details and bugs.

Simplified Programming Model

Imagine an ordinary pocket calculator which can add, subtract, multiply, and divide, and which has a few memory registers which you can store numbers in. At a grossly oversimplified level (so simple that we'll abandon it in just a minute), we can think of a computer as a calculator which is able to push its own buttons. A computer program is simply the list of instructions that tells the computer which buttons to push. (Actually, there are ``keystroke programmable" calculators which you program in just about this way.)

Imagine using such a calculator to perform the following task:

Given a list of numbers, compute the average of all the numbers, and also find the largest number in the list.

You can imagine giving the calculator a list of instructions for performing this task, or you can imagine giving a list of instructions to a very stupid but very patient person who is able to follow instructions blindly but accurately, as long as the instructions consist of pushing buttons on the calculator and making simple yes/no decisions. (For our purposes just now, either imaginary model will work.)

Your instructions might look something like this:

``We're going to use memory register 1 to store the running total of all the numbers, memory register 2 to store how many numbers we've seen, and register 3 to store the largest number we've seen. For each number in the input list, add it to register 1. Add 1 to register 2. If the number you just read is larger than the number in register 3, store it in register 3. When you've read all the numbers, divide register 1 by register 2 to compute the average, and also retrieve the largest number from register 3.''
There are several things to notice about the above list of instructions:

1. The first sentence, which explains what the registers are used for, is more for our benefit than the entity who will be pushing the buttons on the calculator. The entity pushing the buttons doesn't care what the numbers mean, it just manipulates them as directed. Similarly, the words ``to compute the average'' and ``largest'' in the last sentence are extraneous; they don't tell the entity pushing the button anything it needs to know (or that it can even understand).

2. The instructions use the word ``it'' several times. Even in English, where we're used to a certain amount of ambiguity which we can usually work out from the context, pronouns like ``it'' can cause problems in sentences, because sometimes it isn't obvious what they mean. (For example, in the preceding sentence, does ``they'' refer to ``pronouns,'' ``problems,'' or ``sentences?'') In programming, you can never get away with ambiguity; you have to be quite precise about which ``it'' you're referring to.

3. The instructions are pretty vague about the details of reading the next number in the input list and detecting the end of the list.

4. The ``program'' contains several bugs! It uses registers 1, 2, and 3, but we never say what to store in them in the first place. Unless they all happen to start out containing zero, the average or maximum value computed by the ``program'' will be incorrect. (Actually, if all of the numbers in the list are negative, having register 3 start out as 0 won't work, either.)

Here is a somewhat more detailed version of the ``program,'' which removes some of the extraneous information and ambiguity, makes the input list handling a bit more precise, and fixes at least some of the bugs. (To make the concept of ``the number just read from the list'' unambiguous, this ``program'' stores it in register 4, rather than referring to it by ``it.'' Also, for now, we're going to assume that the numbers in the input list are non-negative.)

``Store 0 in registers 1, 2, and 3. Read the next number from the list. If you're at the end of the list, you're done. Otherwise, store the number in register 4. Add register 4 to register 1. Add 1 to register 2. If register 4 is greater than register 3, store register 4 in register 3. When you're done, divide register 1 by register 2 and print the result, and print the contents of register 3."
When we add the initialization step (storing 0 in the registers), we realize that it's not quite obvious which steps happen once only and which steps happen once for each number in the input list (that is, each time through the processing loop). Also, we've assumed that the calculator can do arithmetic operations directly into memory registers. To make the loop boundaries explicit, and the calculations even simpler (assuming that all the calculator can do is store or recall memory registers from or to the display, and do calculations in the display), the instructions would get more elaborate still:
``Store 0 in register 1. Store 0 in register 2. Store 0 in register 3. Here is the start of the loop: read the next number from the list. If you're at the end of the list, you're done. Otherwise, store the number in register 4. Recall from register 1, recall from register 4, add them, store in register 1. Recall from register 2, add 1, store in register 2. Recall from register 3, recall from register 4, if greater store in register 3. Go back to the beginning of the loop. When you're done: recall from register 1, recall from register 2, divide them, print; recall from register 3, print."
We could continue to ``dumb down" this list of instructions even further, but hopefully you're getting the point: the instructions we use when programming computers have to be very precise, and at a level of pickiness and detail which we don't usually use with each other. (Actually, things aren't quite as bad as these examples might suggest. The ``dumbing down" we've been doing has been somewhat in the direction of assembly language, which wise programmers don't use much any more. In a higher-level language such as C, you don't have to worry so much about register assignment and individual arithmetic operators.)

Real computers can do quite a bit more than 4-function pocket calculators can; for one thing, they can manipulate strings of text and other kinds of data besides numbers. Let's leave pocket calculators behind, and start looking at what real computers (at least under the control of programming languages like C) can do.

Real Programming Model

A computer program consists of two parts: *code* and *data*. The code is the set of instructions for performing a task, and the data is the set of ``registers" or ``memory locations" which contain the intermediate results which are used as the program performs its calculations.

Note that the code is relatively *static* while the data is *dynamic*. Once you've gotten a program working, its code won't change, but every time you run it, it will typically be working with different data, so the memory locations will take on different values.

Once you've written a program, you've defined a new thing that your computer can do. The applications (text and graphic editors, spreadsheets, games, etc.) which your computer may already have are ``just'' programs, written by programmers using programming languages such as the one you're about to learn.

Elements of Real Programming Languages

There are several elements which programming languages, and programs written in them, typically contain. These elements are found in all languages, not just C. If you understand these elements and what they're for, not only will you understand C better, but you'll also find learning other programming languages, and moving between different programming languages, much easier.

1. There are *variables* or *objects*, in which you can store the pieces of data that a program is working on. Variables are the way we talk about memory locations (data), and are analogous to the ``registers'' in our pocket calculator example. Variables may be *global* (that is, accessible anywhere in a program) or *local* (that is, private to certain parts of a program).

2. There are *expressions*, which compute new values from old ones.

3. There are *assignments* which store values (of expressions, or other variables) into variables. In many languages, assignment is indicated by an equals sign; thus, we might have

```
        b = 3
```

or

```
        c = d + e + 1
```

The first sets the variable `b` to 3; the second sets the variable `c` to the sum of the variables `d` plus `e` plus 1.

The use of an equals sign can be mildly confusing at first. In mathematics, an equals sign indicates equality: two things are stated to be inherently equal, for all time. In programming, there's a time element, and a notion of cause-and-effect: after the assignment, the thing on the left-hand side of the assignment statement is equal to what the stuff on the right-hand side was before. To remind yourself of this meaning, you might want to read the equals sign in an assignment as ``gets'' or ``receives'': `a = 3` means ``a gets 3'' or ``a receives 3.''

(A few programming languages use a left arrow for assignment

```
        a <-- 3
```

to make the ``receives'' relation obvious, but this notation is not too popular, if for no other reason than that few character sets have left arrows in them, and the left arrow key on the keyboard usually moves the cursor rather than typing a left arrow.)

If assignment seems natural and unconfusing so far, consider the line

```
        i = i + 1
```
What can this mean? In algebra, we'd subtract `i` from both sides and end up with
```
        0 = 1
```
which doesn't make much sense. In programming, however, lines like
```
        i = i + 1
```
are extremely common, and as long as we remember how assignment works, they're not too hard to understand: the variable i receives (its new value is), as always, what we get when we evaluate the expression on the right-hand side. The expression says to fetch `i`'s (old) value, and add 1 to it, and this new value is what will get stored into `i`. So `i = i + 1` adds 1 to `i`; we say that it *increments* i.

(We'll eventually see that, in C, assignments are just another kind of expression.)

4. There are *conditionals* which can be used to determine whether some condition is true, such as whether one number is greater than another. (In some languages, including C, conditionals are actually expressions which compare two values and compute a ``true'' or ``false'' value.)

5. Variables and expressions may have *types*, indicating the nature of the expected values. For instance, you might declare that one variable is expected to hold a number, and that another is expected to hold a piece of text. In many languages (including C), your *declarations* of the names of the variables you plan to use and what types you expect them to hold must be explicit.

There are all sorts of data types handled by various computer languages. There are single characters, integers, and ``real'' (floating point) numbers. There are text strings (i.e. strings of several characters), and there are *arrays* of integers, reals, or other types. There are types which reference (point at) values of other types. Finally, there may be user-defined data types, such as *structures* or *records*, which allow the programmer to build a more complicated data structure, describing a more complicated object, by accreting together several simpler types (or even other user-defined types).

6. There are *statements* which contain instructions describing what a program actually does. Statements may compute expressions, perform assignments, or call functions (see below).

7. There are *control flow constructs* which determine what order statements are performed in. A certain statement might be performed only if a condition is true. A sequence of several statements might be repeated over and over, until some condition is met; this is called a *loop*.

8. An entire set of statements, declarations, and control flow constructs can be lumped together into a *function* (also called *routine, subroutine*, or *procedure*) which another piece of code can then *call* as a unit.

When you call a function, you transfer control to it and wait for it to do its job, after which it *returns* to you; it may also return a value as a result of what it has done. You may also pass values to the function on which it will operate or which otherwise direct its work.

Placing code into functions not only avoids repetition if the same sequence of actions must be performed at several places within a program, but it also makes programs easy to understand, because you can see that some function is being called, and performing some (presumably) well-defined subtask, without always concerning yourself with the details of how that function does its job. (If you've ever done any knitting, you know that knitting instructions are often written with little sub-instructions or patterns which describe a sequence of stitches which is to be performed multiple times during the course of the main piece. These sub-instructions are very much like function calls in programming.)

9. A set of functions, global variables, and other elements makes up a *program*. An additional wrinkle is that the source code for a program may be distributed among one or more *source files*. (In the other direction, it is also common for a suite of related programs to work closely together to perform some larger task, but we'll not worry about that ``large scale integration'' for now.)

10. In the process of specifying a program in a form suitable for a compiler, there are usually a few logistical details to keep track of. These details may involve the specification of compiler parameters or interdependencies between different functions and other parts of the program. Specifying these details often involves miscellaneous syntax which doesn't fall into any of the other categories listed here, and which we might lump together as ``boilerplate.''

Many of these elements exist in a hierarchy. A program typically consists of functions and global variables; a function is made up of statements; statements usually contain expressions; expressions operate on objects. (It is also possible to extend the hierarchy in the other direction; for instance, sometimes several interrelated but distinct programs are assembled into a suite, and used in concert to perform complex tasks. The various ``office'' packages--integrated word processor, spreadsheet, etc.--are an example.)

As we mentioned, many of the concepts in programming are somewhat arbitrary. This is particularly so for the terms *expression*, *statement*, and *function*. All of these could be defined as ``an element of a program that actually does something.'' The differences are mainly in the level at which the ``something'' is done, and it's not necessary at this point to define those ``levels.'' We'll come to understand them as we begin to write programs.

An analogy may help: Just as a book is composed of chapters which are composed of sections which are composed of paragraphs which are composed of sentences which are composed of words (which are composed of letters), so is a program composed of functions which are composed of statements which are composed of expressions (which are in fact composed of smaller elements which we won't bother to define). Analogies are never perfect, though, and this one is weaker than most; it still doesn't tell us anything about what expressions, statements, and functions really *are*. If ``expression'' and ``statement'' and ``function'' seem like totally arbitrary words to you, use the analogy to understand that what they are is arbitrary words describing arbitrary levels in the hierarchical composition of a program, just as ``sentence,'' ``paragraph,'' and ``chapter'' are different levels of structure within a book.

The preceding discussion has been in very general terms, describing features common to most ``conventional'' computer languages. If you understand these elements at a relatively abstract level, then learning a new computer language becomes a relatively simple matter of finding out how that language implements each of the elements. (Of course, you can't understand these abstract elements in isolation; it helps to have concrete examples to map them to. If you've never programmed before, most of this section has probably seemed like words without meaning. Don't spend too much time trying to glean all the meaning, but do come back anbd reread this handout after you've started to learn the details of a particular programming language such as C.)

Finally, there's no need to overdo the abstraction. For the simple programs we'll be writing, in a language like C, the series of calculations and other operations that actually takes place as our program runs is a simpleminded translation (into terms the computer can understand) of the expressions, statements, functions, and other elements of the program. Expressions are evaluated and their results assigned to variables. Statements are executed one after the other, except when the control flow is modified by if/then conditionals and loops. Functions are called to perform subtasks, and return values to their callers, which have been waiting for them.

Computer Representation of Numbers

Most computers represent integers as binary numbers (see the ``math refresher'' handout) with a certain number of bits. A computer with 16-bit integers can represent integers from 0 to 65,535 (that is, from 0 to 2^{16}-1), or if it chooses to make half of them negative, from -32,767 to 32,767. (We won't get into the details of how computers handle negative numbers right now.) A 32-bit integer can represent values from 0 to 4,294,967,295, or +-2,147,483,647.

Most of today's computers represent real (i.e. fractional) numbers using exponential notation. (Again, see ``math refresher'' handout. Actually, deep down inside, computers usually use powers of 2 instead of powers of 10, but the difference isn't important to us right now.) The advantage of using exponential notation for real numbers is that it lets you trade off the range and precision of values in a useful way. Since there's an infinitely large number of real numbers (and in three directions: very large, very small, and very negative), it will never be possible to represent all of them (without using potentially infinite amounts of space).

Suppose you decide to give yourself six decimal digits' worth of storage (that is, you decide to devote an amount of memory capable of holding six digits) for each value. If you put three digits to the left and three to the right of the decimal point, you could represent numbers from 999.999 to -999.999, and as small as 0.001. (Furthermore, you'd have a resolution of 0.001 everywhere: you could represent 0.001 and 0.002, as well as 999.998 and 999.999.) This would be a workable scheme, although 0.001 isn't a very small number and 999.999 isn't a very big one.

If, on the other hand, you used exponential notation, with four digits for the base number and two digits for the exponent, you could represent numbers from 9.999 x 10^{99} to -9.999 x 10^{99}, and as small as 1 x 10^{-99} (or, if you cheat, 0.001 x 10^{-99}). You can now represent both much larger numbers and much smaller; the tradeoff is that the absolute resolution is no longer constant, and gets smaller as the absolute value of the numbers gets larger. The number 123.456 can only be represented as 123.4, and the number 123,456 can only be represented as 123,400. You can't represent 999.999 any more; you have to settle for 999.9 (9.999 x 10^2 or 1000 (1.000 x 10^3). You can't distinguish between 999.998 and 999.999 any more.

Since superscripts are difficult to type, computer programming languages usually use a slightly different notation. For example, the number 1.234 x 10^5 might be indicated by `1.234e5`, where the letter `e` replaces the ``times ten to the'' part.

You will often hear real, exponential numbers numbers referred to on computers as ``floating point numbers'' or simply ``floats,'' and you will also hear the term ``double'' which is short for ``double-precision floating point number.'' Some computers also use ``fixed point'' real numbers, (which work along the lines of our ``three to the left, three to the right'' example of a few paragraphs back), but those are comparatively rare and we won't need to discuss them.

It's important to remember that the precision of floating-point numbers is usually limited, and this can lead to surprising results. The result of a division like 1/3 cannot be represented exactly (it's an infinitely repeating fraction, 0.333333...), so the computation (1 / 3) x 3 tends to yield a result like 0.999999... instead of 1.0. Furthermore, in base 2, the fraction 1/10, or 0.1 in decimal, is *also* an infinitely repeating fraction, and cannot be represented exactly, either, so (1 / 10) x 10 may also yield 0.999999.... For these reasons and others, floating-point calculations are rarely exact. When working with computer floating point, you have to be careful not to compare two numbers for exact equality, and you have to ensure that ``round off error'' doesn't accumulate until it seriously degrades the results of your calculations.

Characters, Strings, and Numbers

The earliest computers were number crunchers only, but almost all more recent computers have the ability to manipulate alphanumeric data as well. The computer, and our programming languages, tend to maintain a strict distinction between numbers on the one hand and alphanumeric data on the other, so we have to maintain that distinction in our own minds as well.

One fundamental component of a computer's handling of alphanumeric data is its *character set*. A character set is, not surprisingly, the set of all the characters that the computer can process and display. (Each character generally has a key on the keyboard to enter it and a bitmap on the screen which displays it.) A character set consists of letters, numbers, punctuation, etc., but the point of this discussion is not so much what the characters *are* but that we have to be careful to distinguish between characters, strings, and numbers.

A *character* is, well, a single character. If we have a variable which contains a character value, it might contain the letter `A', or the digit `2', or the symbol `&'.

A *string* is a set of zero or more characters. For example, the string ``and'' consists of the characters `a', `n', and `d'. The string ``K2!'' consists of the characters `K', `2', and `!'. The string ``.'' consists of the single character `.', and the empty string ``'' consists of no characters at all. Not to belabor the point, but the string ``123'' consists of the characters `1', `2', and `3', and the string ``4'' consists of the single character `4'.

The last two examples illustrate some important and perhaps surprising or annoying distinctions. The character `4' and the string ``4'' are conceptually different, and neither of them is quite the same as the *number* 4. The string ``123'' consists of three characters, and it looks like the number 123 to us, but as far as the computer is concerned it is just a string. The number 123 is, when used for ordinary numeric purposes, *not* represented internally as a string of three characters (instead, it is typically represented as a 16- or 32-bit integer). When we have a string which contains a numeric value which we wish to manipulate as a number, we must typically ask for the string to be explicitly converted to that number somehow. Similarly, we may have reason to convert a number to a string of digits making up its decimal representation.

We may also find ourselves needing to convert back and forth between characters and the numeric codes which are assigned to each character in a character set. (For example, in the ASCII character set,the character `A' is code 65, the character `.' is code 46, and the character `4' is, perhaps surprisingly, code 52.)

Compiler Terminology

C is a *compiled* language. This means that the programs you write are translated, by a program called a compiler, into executable machine-language programs which you can actually run. Executable machine-language programs are self-contained and run very quickly. Since they are self-contained, you don't need copies of the source code (the original programming-language text you composed) or the compiler in order to run them; you can distribute copies of just the executable and that's all someone else needs to run it. Since they run relatively quickly, they are appropriate for programs which will be written once and run many times.

A compiler is a special kind of progam: it is a program that builds other programs. What happens is that you invoke the compiler (as a program), and it reads the programming language statements that you have written and turns them into a new, executable program. When the compiler has finished its work, you then invoke *your* program (the one the compiler just built) to see if it works.

The main alternative to a compiled computer language or program is an interpreted one, such as BASIC. An interpreted language is interpreted (by, not surprisingly, a program called an interpreter) and its actions performed immediately. If you gave a copy of an interpreted program to someone else, they would also need a copy of the interpreter to run it. No standalone executable machine-language binary program is produced.

In other words, for each statement that you write, a compiler translates into a sequence of machine language instructions which does the same thing, while an interpreter simply does it (where ``it'' is whatever the statement that you wrote is supposed to do).

The big advantage of an interpreted language is that your program runs right away; you don't have to perform--and wait for--the separate tasks of compiling and then running your program. (Actually, on a modern computer, neither compiling nor interpreting takes much time, so some of these distinctions become less important.)

Actually, whether a language is compiled or interpreted is not always an inherent part of the language. There are interpreters for C, and there are compilers for BASIC. However, most languages were designed with one or the other mechanism in mind, and there are usually a few difficulties when trying to compile a language which is traditionally interpreted, or vice versa.

The distinction between compilation and interpretation, while it is very significant and can make a big difference, is not one to get worked up over. Most of the time, once you get used to the details of how you get your programs to run, you don't need to worry about the distinction too much. But it is a useful distinction to have a basic understanding of, and to keep in the back of your mind, because it will help you understand why certain aspects of computer programming (and particular languages) work the way they do.

When you're working with a compiled language, there are several mechanical details which you'll want to be aware of. You create one or more *source files* which are simple text files containing your program, written in whatever language you're using. You typically use a text editor to work with source files (typically you don't want to use a full-fledged text editor, since the compiler won't understand its formatting codes). You supply each source file (you may have one, or more than one) to the compiler, which creates an *object file* containing machine-language instructions corresponding to your program. Your program is not ready to run yet, however: if you called any functions which you didn't write (such as the standard library functions provided as part of a programming language environment), you must arrange for them to be inserted into your program, too. The task of combining object files together, while also locating and inserting any library functions, is the job of the *linker*. The linker puts together the object files you give it, noticing if you call any functions which you haven't supplied and which must therefore be library functions. It then searches one or more *library files* (a library file is simply a collection of object files) looking for definitions of the still-unresolved functions, and pulls in any that it finds. When it's done, it either builds the final, executable file, or, if there were any errors (such as a function called but not defined anywhere) complains.

If you're using some kind of an integrated programming environment, many of these steps may be taken care of for you so automatically and seamlessly that you're hardly aware of them.

A Brief Refresher on Some Math Often Used in Computing

There are a few mathematical concepts which figure prominently in programming. None of these involve higher math (or even algebra, and as we'll see, knowing algebra too well can make one particular programming idiom rather confusing). You don't have to understand these with deep mathematical sophistication, but keeping them in mind will help some things make more sense.

Integers vs. real numbers

An *integer* is a number without a fractional part, a number you could use to count things (although integers may also be negative). Mathematicians may distinguish between natural numbers and cardinal numbers, and linguists may distinguish between cardinal numbers and ordinal numbers, but these distinctions do not concern us here.

A *real number* is, for our purposes, simply a number with a fractional part. Since computers do not typically implement real numbers exactly, it is not necessary or even meaningful to distinguish between rational, irrational, and transcendental numbers.

Exponential Notation

Exponential or Scientific Notation is simply a method of writing a number as a base number times some power of ten. For example, we could write the number 2,000,000 as 2×10^6, the number 0.00023 as 2.3×10^{-4}, and the number 123.456 as 1.23456×10^2.

Binary Numbers

Our familiar decimal number system is based on powers of 10. The number 123 is actually 100 + 20 + 3 or 1 x 10<sup>2</sup> + 2 x 10<sup>1</sup> + 3 x 10<sup>0</sup>.

The binary number system is based on powers of 2. The number 100101<sub>2</sub> (that is, ``100101 base two'') is 1 x 2<sup>5</sup> + 0 x 2<sup>4</sup> + 0 x 2<sup>3</sup> + 1 x 2<sup>2</sup> + 0 x 2<sup>1</sup> + 1 x 2<sup>0</sup> or 32 + 4 + 1 or 37.

We usually speak of the individual numerals in a decimal number as digits, while the ``digits'' of a binary number are usually called ``bits.''

Besides decimal and binary, we also occasionally speak of octal (base 8) and hexadecimal (base 16) numbers. These work similarly: The number 45_8 is $4 \times 8^1 + 5 \times 8^0$ or 32 + 5 or 37. The number 25_{16} is $2 \times 16^1 + 5 \times 16^0$ or 32 + 5 or 37. (So 37_{10}, 100101_2, 45_8, and 25_{16} are all the same number.)

Boolean Algebra

Boolean algebra is a system of algebra (named after the mathematician who studied it, George Boole) based on only two numbers, 0 and 1, commonly thought of as ``false'' and ``true.'' Binary numbers and Boolean algebra are natural to use with modern digital computers, which deal with switches and electrical currents which are either on or off. (In fact, binary numbers and Boolean algebra aren't just natural to use with modern digital computers, they are the fundamental basis of modern digital computers.)

There are four arithmetic operators in Boolean algebra: NOT, AND, OR, and EXCLUSIVE OR.

NOT takes one operand (that is, applies to a single value) and negates it: NOT 0 is 1, and NOT 1 is 0.

AND takes two operands, and yields a truth value if both of its operands are true: 1 AND 1 is 1, but 0 AND 1 is 0, and 0 AND 0 is 0.

OR takes two operands, and yields a truth value if either of its operands (or both) are true: 0 OR 0 is 0, but 0 OR 1 is 1, and 1 OR 1 is 1.

EXCLUSIVE OR, or XOR, takes two operands, and yields a truth value if one of its operands, but *not* both, is true: 0 XOR 0 is 0, 0 XOR 1 is 1, and 1 XOR 1 is 0.

It is also possible to take strings of 0/1 values and apply Boolean operators to all of them in parallel; these are sometimes called ``bitwise'' operations. For example, the bitwise OR of 0011 and 0101 is 0111. (If it isn't obvious, what happens here is that each bit in the answer is the result of applying the corresponding operation to the two corresponding bits in the input numbers.)

Chapter 1: Introduction

C is (as K&R admit) a relatively small language, but one which (to its admirers, anyway) wears well. C's small, unambitious feature set is a real advantage: there's less to learn; there isn't excess baggage in the way when you don't need it. It can also be a disadvantage: since it doesn't do everything for you, there's a lot you have to do yourself. (Actually, this is viewed by many as an additional advantage: anything the language doesn't do for you, it doesn't dictate to you, either, so you're free to do that something however you want.)

C is sometimes referred to as a ``high-level assembly language.'' Some people think that's an insult, but it's actually a deliberate and significant aspect of the language. If you have programmed in assembly language, you'll probably find C very natural and comfortable (although if you continue to focus too heavily on machine-level details, you'll probably end up with unnecessarily nonportable programs). If you haven't programmed in assembly language, you may be frustrated by C's lack of certain higher-level features. In either case, you should understand why C was designed this way: so that seemingly-simple constructions expressed in C would not expand to arbitrarily expensive (in time or space) machine language constructions when compiled. If you write a C program simply and succinctly, it is likely to result in a succinct, efficient machine language executable. If you find that the executable program resulting from a C program is not efficient, it's probably because of something silly you did, not because of something the compiler did behind your back which you have no control over. In any case, there's no point in complaining about C's low-level flavor: C is what it is.

A programming language is a tool, and no tool can perform every task unaided. If you're building a house, and I'm teaching you how to use a hammer, and you ask how to assemble rafters and trusses into gables, that's a legitimate question, but the answer has fallen out of the realm of ``How do I use a hammer?'' and into ``How do I build a house?''. In the same way, we'll see that C does not have built-in features to perform every function that we might ever need to do while programming.

As mentioned above, C imposes relatively few built-in ways of doing things on the programmer. Some common tasks, such as manipulating strings, allocating memory, and doing input/output (I/O), are performed by calling on library functions. Other tasks which you might want to do, such as creating or listing directories, or interacting with a mouse, or displaying windows or other user-interface elements, or doing color graphics, are not defined by the C language at all. You can do these things from a C program, of course, but you will be calling on services which are peculiar to your programming environment (compiler, processor, and operating system) and which are not defined by the C standard. Since this course is about portable C programming, it will also be steering clear of facilities not provided in all C environments.

Another aspect of C that's worth mentioning here is that it is, to put it bluntly, a bit dangerous. C does not, in general, try hard to protect a programmer from mistakes. If you write a piece of code which will (through some oversight of yours) do something wildly different from what you intended it to do, up to and including deleting your data or trashing your disk, and if it is possible for the compiler to compile it, it generally will. You won't get warnings of the form ``Do you really mean to...?'' or ``Are you sure you really want to...?''. C is often compared to a sharp knife: it can do a surgically precise job on some exacting task you have in mind, but it can also do a surgically precise job of cutting off your finger. It's up to you to use it carefully.

This aspect of C is very widely criticized; it is also used (justifiably) to argue that C is not a good teaching language. C aficionados love this aspect of C because it means that C does not try to protect them from themselves: when they know what they're doing, even if it's risky or obscure, they can do it. Students of C hate this aspect of C because it often seems as if the language is some kind of a conspiracy specifically designed to lead them into booby traps and ``gotcha!''s.

This is another aspect of the language which it's fairly pointless to complain about. If you take care and pay attention, you can avoid many of the pitfalls. These notes will point out many of the obvious (and not so obvious) trouble spots.

1.1 A First Example

[This section corresponds to K&R Sec. 1.1]

The best way to learn programming is to dive right in and start writing real programs. This way, concepts which would otherwise seem abstract make sense, and the positive feedback you get from getting even a small program to work gives you a great incentive to improve it or write the next one.

Diving in with ``real'' programs right away has another advantage, if only pragmatic: if you're using a conventional compiler, you can't run a fragment of a program and see what it does; nothing will run until you have a complete (if tiny or trivial) program. You can't learn everything you'd need to write a complete program all at once, so you'll have to take some things ``on faith'' and parrot them in your first programs before you begin to understand them. (You can't learn to program just one expression or statement at a time any more than you can learn to speak a foreign language one word at a time. If all you know is a handful of words, you can't actually *say* anything: you also need to know something about the language's word order and grammar and sentence structure and declension of articles and verbs.)

Besides the occasional necessity to take things on faith, there is a more serious potential drawback of this ``dive in and program'' approach: it's a small step from learning-by-doing to learning-by-trial-and-error, and when you learn programming by trial-and-error, you can very easily learn many errors. When you're not sure whether something will work, or you're not even sure what you could use that might work, and you try something, and it does work, you do *not* have any guarantee that what you tried worked for the right reason. You might just have ``learned'' something that works only by accident or only on your compiler, and it may be very hard to un-learn it later, when it stops working.

Therefore, whenever you're not sure of something, be very careful before you go off and try it ``just to see if it will work.'' Of course, you can never be absolutely sure that something is going to work before you try it, otherwise we'd never have to try things. But you should have an expectation that something is going to work before you try it, and if you can't predict how to do something or whether something would work and find yourself having to determine it experimentally, make a note in your mind that whatever you've just learned (based on the outcome of the experiment) is suspect.

The first example program in K&R is the first example program in any language: print or display a simple string, and exit. Here is my version of K&R's ``hello, world'' program:

```
#include <stdio.h>

main()
{
printf("Hello, world!\n");
return 0;
}
```

If you have a C compiler, the first thing to do is figure out how to type this program in and compile it and run it and see where its output went. (If you don't have a C compiler yet, the first thing to do is to find one.)

The first line is practically boilerplate; it will appear in almost all programs we write. It asks that some definitions having to do with the ``Standard I/O Library'' be included in our program; these definitions are needed if we are to call the library function `printf` correctly.

The second line says that we are defining a function named `main`. Most of the time, we can name our functions anything we want, but the function name `main` is special: it is the function that will be ``called'' first when our program starts running. The empty pair of parentheses indicates that our `main` function accepts no *arguments*, that is, there isn't any information which needs to be passed in when the function is called.

The braces `{` and `}` surround a list of statements in C. Here, they surround the list of statements making up the function `main`.

The line

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

is the first statement in the program. It asks that the function `printf` be called; `printf` is a library function which prints formatted output. The parentheses surround `printf`'s argument list: the information which is handed to it which it should act on. The semicolon at the end of the line terminates the statement.

(`printf`'s name reflects the fact that C was first developed when Teletypes and other printing terminals were still in widespread use. Today, of course, video displays are far more common. `printf`'s ``prints'' to the *standard output*, that is, to the default location for program output to go. Nowadays, that's almost always a video screen or a window on that screen. If you do have a printer, you'll typically have to do something extra to get a program to print to it.)

`printf`'s first (and, in this case, only) argument is the string which it should print. The string, enclosed in double quotes `""`, consists of the words ``Hello, world!'' followed by a special sequence: `\n`. In strings, any two-character sequence beginning with the backslash `\` represents a single special character. The sequence `\n` represents the ``new line'' character, which prints a carriage return or line feed or whatever it takes to end one line of output and move down to the next. (This program only prints one line of output, but it's still important to terminate it.)

The second line in the `main` function is

```
        return 0;
```

In general, a function may return a value to its caller, and `main` is no exception. When `main` returns (that is, reaches its end and stops functioning), the program is at its end, and the return value from `main` tells the operating system (or whatever invoked the program that `main` is the main function of) whether it succeeded or not. By convention, a return value of 0 indicates success.

This program may look so absolutely trivial that it seems as if it's not even worth typing it in and trying to run it, but doing so may be a big (and is certainly a vital) first hurdle. On an unfamiliar computer, it can be arbitrarily difficult to figure out how to enter a text file containing program source, or how to compile and link it, or how to invoke it, or what happened after (if?) it ran. The most experienced C programmers immediately go back to this one, simple program whenever they're trying out a new system or a new way of entering or building programs or a new way of printing output from within programs. As Kernighan and Ritchie say, everything else is comparatively easy.

How *you* compile and run this (or any) program is a function of the compiler and operating system you're using. The first step is to type it in, exactly as shown; this may involve using a text editor to create a file containing the program text. You'll have to give the file a name, and all C compilers (that I've ever heard of) require that files containing C source end with the extension `.c`. So you might place the program text in a file called `hello.c`.

The second step is to compile the program. (Strictly speaking, compilation consists of two steps, compilation proper followed by linking, but we can overlook this distinction at first, especially because the compiler often takes care of initiating the linking step automatically.) On many Unix systems, the command to compile a C program from a source file `hello.c` is

```
        cc -o hello hello.c
```
You would type this command at the Unix shell prompt, and it requests that the `cc` (C compiler) program be run, placing its output (i.e. the new executable program it creates) in the file `hello`, and taking its input (i.e. the source code to be compiled) from the file `hello.c`.

The third step is to run (execute, invoke) the newly-built `hello` program. Again on a Unix system, this is done simply by typing the program's name:

```
        hello
```
Depending on how your system is set up (in particular, on whether the current directory is searched for executables, based on the PATH variable), you may have to type
```
        ./hello
```
to indicate that the `hello` program is in the current directory (as opposed to some ``bin'' directory full of executable programs, elsewhere).

You may also have your choice of C compilers. On many Unix machines, the `cc` command is an older compiler which does not recognize modern, ANSI Standard C syntax. An old compiler will accept the simple programs we'll be starting with, but it will not accept most of our later programs. If you find yourself getting baffling compilation errors on programs which you've typed in exactly as they're shown, it probably indicates that you're using an older compiler. On many machines, another compiler called `acc` or `gcc` is available, and you'll want to use it, instead. (Both `acc` and `gcc` are typically invoked the same as `cc`; that is, the above `cc` command would instead be typed, say, `gcc -o hello hello.c` .)

(One final caveat about Unix systems: don't name your test programs `test`, because there's already a standard command called `test`, and you and the command interpreter will get badly confused if you try to replace the system's `test` command with your own, not least because your own almost certainly does something completely different.)

Under MS-DOS, the compilation procedure is quite similar. The name of the command you type will depend on your compiler (e.g. `cl` for the Microsoft C compiler, `tc` or `bcc` for Borland's Turbo C, etc.). You may have to manually perform the second, linking step, perhaps with a command named `link` or `tlink`. The executable file which the compiler/linker creates will have a name ending in `.exe` (or perhaps `.com`), but you can still invoke it by typing the base name (e.g. `hello`). See your compiler documentation for complete details; one of the manuals should contain a demonstration of how to enter, compile, and run a small program that prints some simple output, just as we're trying to describe here.

In an integrated or ``visual'' progamming environment, such as those on the Macintosh or under various versions of Microsoft Windows, the steps you take to enter, compile, and run a program are somewhat different (and, theoretically, simpler). Typically, there is a way to open a new source window, type source code into it, give it a file name, and add it to the program (or ``project'') you're building. If necessary, there will be a way to specify what other source files (or ``modules'') make up the program. Then, there's a button or menu selection which compiles and runs the program, all from within the programming environment. (There will also be a way to create a standalone executable file which you can run from outside the environment.) In a PC-compatible environment, you may have to choose between creating DOS programs or Windows programs. (If you have troubles pertaining to the `printf` function, try specifying a target environment of MS-DOS. Supposedly, some compilers which are targeted at Windows environments won't let you call `printf`, because until you call some fancier functions to request that a window be created, there's no window for `printf` to print to.) Again, check the introductory or tutorial manual that came with the programming package; it should walk you through the steps necessary to get your first program running.

1.2 Second Example

Our second example is of little more practical use than the first, but it introduces a few more programming language elements:

```
#include <stdio.h>

/* print a few numbers, to illustrate a simple loop */

main()
{
int i;

for(i = 0; i < 10; i = i + 1)
        printf("i is %d\n", i);

return 0;
}
```

As before, the line `#include <stdio.h>` is boilerplate which is necessary since we're calling the `printf` function, and `main()` and the pair of braces `{}` indicate and delineate the function named `main` we're (again) writing.

The first new line is the line

```
        /* print a few numbers, to illustrate a simple loop */
```

which is a *comment*. Anything between the characters `/*` and `*/` is ignored by the compiler, but may be useful to a person trying to read and understand the program. You can add comments anywhere you want to in the program, to document what the program is, what it does, who wrote it, how it works, what the various functions are for and how *they* work, what the various variables are for, etc.

The second new line, down within the function `main`, is

```
        int i;
```

which *declares* that our function will use a variable named `i`. The variable's type is `int`, which is a plain integer.

Next, we set up a *loop*:

```
        for(i = 0; i < 10; i = i + 1)
```

The keyword `for` indicates that we are setting up a ``for loop.'' A `for` loop is controlled by three expressions, enclosed in parentheses and separated by semicolons. These expressions say that, in this case, the loop starts by setting `i` to 0, that it continues as long as `i` is less than 10, and that after each iteration of the loop, `i` should be incremented by 1 (that is, have 1 added to its value).

Finally, we have a call to the `printf` function, as before, but with several differences. First, the call to `printf` is within the *body* of the `for` loop. This means that control flow does not pass once through the `printf` call, but instead that the call is performed as many times as are dictated by the `for` loop. In this case, `printf` will be called several times: once when `i` is 0, once when `i` is 1, once when `i` is 2, and so on until `i` is 9, for a total of 10 times.

A second difference in the `printf` call is that the string to be printed, `"i is %d"`, contains a percent sign. Whenever `printf` sees a percent sign, it indicates that `printf` is not supposed to print the exact text of the string, but is instead supposed to read another one of its arguments to decide what to print. The letter after the percent sign tells it what type of argument to expect and how to print it. In this case, the letter `d` indicates that `printf` is to expect an `int`, and to print it in decimal. Finally, we see that `printf` is in fact being called with another argument, for a total of two, separated by commas. The second argument is the variable `i`, which is in fact an `int`, as required by `%d`. The effect of all of this is that each time it is called, `printf` will print a line containing the current value of the variable `i`:

```
        i is 0
        i is 1
        i is 2
        ...
```

After several trips through the loop, `i` will eventually equal 9. After that trip through the loop, the third control expression `i = i + 1` will increment its value to 10. The condition `i < 10` is no longer true, so no more trips through the loop are taken. Instead, control flow jumps down to the statement following the `for` loop, which is the `return` statement. The `main` function returns, and the program is finished.

1.3 Program Structure

We'll have more to say later about program structure, but for now let's observe a few basics. A program consists of one or more functions; it may also contain global variables. (Our two example programs so far have contained one function apiece, and no global variables.) At the top of a source file are typically a few boilerplate lines such as `#include <stdio.h>`, followed by the definitions (i.e. code) for the functions. (It's also possible to split up the several functions making up a larger program into several source files, as we'll see in a later chapter.)

Each function is further composed of *declarations* and *statements*, in that order. When a sequence of statements should act as one (for example, when they should all serve together as the body of a loop) they can be enclosed in braces (just as for the outer body of the entire function). The simplest kind of statement is an *expression statement*, which is an expression (presumably performing some useful operation) followed by a semicolon. Expressions are further composed of *operators*, *objects* (variables), and *constants*.

C source code consists of several *lexical elements*. Some are words, such as `for`, `return`, `main`, and `i`, which are either *keywords* of the language (`for`, `return`) or *identifiers* (names) we've chosen for our own functions and variables (`main`, `i`). There are *constants* such as `1` and `10` which introduce new values into the program. There are *operators* such as `=`, `+`, and `>`, which manipulate variables and values. There are other punctuation characters (often called *delimiters*), such as parentheses and squiggly braces `{}`, which indicate how the other elements of the program are grouped. Finally, all of the preceding elements can be separated by *whitespace*: spaces, tabs, and the ``carriage returns'' between lines.

The source code for a C program is, for the most part, ``free form.'' This means that the compiler does not care how the code is arranged: how it is broken into lines, how the lines are indented, or whether whitespace is used between things like variable names and other punctuation. (Lines like `#include <stdio.h>` are an exception; they must appear alone on their own lines, generally unbroken. Only lines beginning with `#` are affected by this rule; we'll see other examples later.) You can use whitespace, indentation, and appropriate line breaks to make your programs more readable for yourself and other people (even though the compiler doesn't care). You can place explanatory *comments* anywhere in your program--any text between the characters `/*` and `*/` is ignored by the compiler. (In fact, the compiler pretends that all it saw was whitespace.) Though comments are ignored by the compiler, well-chosen comments can make a program *much* easier to read (for its author, as well as for others).

The usage of whitespace is our first *style* issue. It's typical to leave a blank line between different parts of the program, to leave a space on either side of operators such as `+` and `=`, and to *indent* the bodies of loops and other control flow constructs. Typically, we arrange the indentation so that the subsidiary statements controlled by a loop statement (the ``loop body,'' such as the `printf` call in our second example program) are all aligned with each other and placed one tab stop (or some consistent number of spaces) to the right of the controlling statement. This indentation (like all whitespace) is not required by the compiler, but it makes programs *much* easier to read. (However, it can also be misleading, if used incorrectly or in the face of inadvertent mistakes. The compiler will decide what ``the body of the loop'' is based on its own rules, not the indentation, so if the indentation does not match the compiler's interpretation, confusion is inevitable.)

To drive home the point that the compiler doesn't care about indentation, line breaks, or other whitespace, here are a few (extreme) examples: The fragments

```
for(i = 0; i < 10; i = i + 1)
        printf("%d\n", i);
```
and
```
for(i = 0; i < 10; i = i + 1) printf("%d\n", i);
```
and
```
for(i=0;i<10;i=i+1)printf("%d\n",i);
```
and
```
        for(i = 0; i < 10; i = i + 1)
printf("%d\n", i);
```
and
```
for     (       i
=       0       ;
i       <       10
;       i       =
i       +       1
)       printf  (
"%d\n"  ,       i
)       ;
```
and
```
    for
   (i=0;
  i<10;i=
 i+1)printf
("%d\n", i);
```
are all treated exactly the same way by the compiler.

Some programmers argue forever over the best set of ``rules'' for indentation and other aspects of programming style, calling to mind the old philosopher's debates about the number of angels that could dance on the head of a pin. Style issues (such as how a program is laid out) *are* important, but they're not something to be too dogmatic about, and there are also other, deeper style issues besides mere layout and typography. Kernighan and Ritchie take a fairly moderate stance:

Although C compilers do not care about how a program looks, proper indentation and spacing are critical in making programs easy for people to read. We recommend writing only one statement per line, and using blanks around operators to clarify grouping. The position of braces is less important, although people hold passionate beliefs. We have chosen one of several popular styles. Pick a style that suits you, then use it consistently.

There is some value in having a reasonably standard style (or a few standard styles) for code layout. Please don't take the above advice to ``pick a style that suits you'' as an invitation to invent your own brand-new style. If (perhaps after you've been programming in C for a while) you have specific objections to specific facets of existing styles, you're welcome to modify them, but if you don't have any particular leanings, you're probably best off copying an existing style at first. (If you want to place your own stamp of originality on the programs that you write, there are better avenues for your creativity than inventing a bizarre layout; you might instead try to make the logic easier to follow, or the user interface easier to use, or the code freer of bugs.)

Chapter 2: Basic Data Types & Operators

The *type* of a variable determines what kinds of values it may take on. An *operator* computes new values out of old ones. An *expression* consists of variables, constants, and operators combined to perform some useful computation. In this chapter, we'll learn about C's basic types, how to write constants and declare variables of these types, and what the basic operators are.

As Kernighan and Ritchie say, ``The type of an object determines the set of values it can have and what operations can be performed on it.'' This is a fairly formal, mathematical definition of what a type is, but it is traditional (and meaningful). There are several implications to remember:

1. The ``set of values'' is finite. C's `int` type can not represent *all* of the integers; its `float` type can not represent *all* floating-point numbers.
2. When you're using an object (that is, a variable) of some type, you may have to remember what values it can take on and what operations you can perform on it. For example, there are several operators which play with the binary (bit-level) representation of integers, but these operators are not meaningful for and may not be applied to floating-point operands.
3. When declaring a new variable and picking a type for it, you have to keep in mind the values and operations you'll be needing.

In other words, picking a type for a variable is not some abstract academic exercise; it's closely connected to the way(s) you'll be using that variable.

2.1 Types

[This section corresponds to K&R Sec. 2.2]

There are only a few basic data types in C. The first ones we'll be encountering and using are:

- char a character
- int an integer, in the range -32,767 to 32,767
- long int a larger integer (up to +-2,147,483,647)
- float a floating-point number
- double a floating-point number, with more precision and perhaps greater range than `float`

If you can look at this list of basic types and say to yourself, ``Oh, how simple, there are only a few types, I won't have to worry much about choosing among them,'' you'll have an easy time with declarations. (Some masochists wish that the type system were more complicated so that they could specify more things about each variable, but those of us who would rather not have to specify these extra things each time are glad that we don't have to.)

The ranges listed above for types `int` and `long int` are the guaranteed minimum ranges. On some systems, either of these types (or, indeed, any C type) may be able to hold larger values, but a program that depends on extended ranges will not be as portable. Some programmers become obsessed with knowing exactly what the sizes of data objects will be in various situations, and go on to write programs which depend on these exact sizes. Determining or controlling the size of an object is occasionally important, but most of the time we can sidestep size issues and let the compiler do most of the worrying.

(From the ranges listed above, we can determine that type `int` must be at least 16 bits, and that type `long int` must be at least 32 bits. But neither of these sizes is exact; many systens have 32-bit `int`s, and some systems have 64-bit `long int`s.)

You might wonder how the computer stores characters. The answer involves a *character set*, which is simply a mapping between some set of characters and some set of small numeric codes. Most machines today use the ASCII character set, in which the letter A is represented by the code 65, the ampersand & is represented by the code 38, the digit 1 is represented by the code 49, the space character is represented by the code 32, etc. (Most of the time, of course, you have *no* need to know or even worry about these particular code values; they're automatically translated into the right shapes on the screen or printer when characters are printed out, and they're automatically generated when you type characters on the keyboard. Eventually, though, we'll appreciate, and even take some control over, exactly when these translations--from characters to their numeric codes--are performed.) Character codes are usually small--the largest code value in ASCII is 126, which is the ~ (tilde or circumflex) character. Characters usually fit in a byte, which is usually 8 bits. In C, type `char` is defined as occupying one byte, so it is usually 8 bits.

Most of the simple variables in most programs are of types `int`, `long int`, or `double`. Typically, we'll use `int` and `double` for most purposes, and `long int` any time we need to hold integer values greater than 32,767. As we'll see, even when we're manipulating individual characters, we'll usually use an `int` variable, for reasons to be discussed later. Therefore, we'll rarely use individual variables of type `char`; although we'll use plenty of arrays of `char`.

2.2 Constants

[This section corresponds to K&R Sec. 2.3]

A *constant* is just an immediate, absolute value found in an expression. The simplest constants are decimal integers, e.g. `0`, `1`, `2`, `123` . Occasionally it is useful to specify constants in base 8 or base 16 (octal or hexadecimal); this is done by prefixing an extra `0` (zero) for octal, or `0x` for hexadecimal: the constants `100`, `0144`, and `0x64` all represent the same number. (If you're not using these non-decimal constants, just remember not to use any leading zeroes. If you accidentally write `0123` intending to get one hundred and twenty three, you'll get 83 instead, which is 123 base 8.)

We write constants in decimal, octal, or hexadecimal for our convenience, not the compiler's. The compiler doesn't care; it always converts everything into binary internally, anyway. (There is, however, no good way to specify constants in source code in binary.)

A constant can be forced to be of type `long int` by suffixing it with the letter `L` (in upper or lower case, although upper case is strongly recommended, because a lower case `l` looks too much like the digit `1`).

A constant that contains a decimal point or the letter `e` (or both) is a floating-point constant: `3.14`, `10.`, `.01`, `123e4`, `123.456e7` . The `e` indicates multiplication by a power of 10; `123.456e7` is 123.456 times 10 to the 7th, or 1,234,560,000. (Floating-point constants are of type `double` by default.)

We also have constants for specifying characters and strings. (Make sure you understand the difference between a character and a string: a character is exactly one character; a string is a set of zero or more characters; a string containing one character is distinct from a lone character.) A character constant is simply a single character between single quotes: `'A'`, `'.'`, `'%'`. The numeric value of a character constant is, naturally enough, that character's value in the machine's character set. (In ASCII, for example, `'A'` has the value 65.)

A *string* is represented in C as a sequence or array of characters. (We'll have more to say about arrays in general, and strings in particular, later.) A string constant is a sequence of zero or more characters enclosed in double quotes: `"apple"`, `"hello, world"`, `"this is a test"`.

Within character and string constants, the backslash character `\` is special, and is used to represent characters not easily typed on the keyboard or for various reasons not easily typed in constants. The most common of these ``character escapes'' are:

```
        \n      a ``newline'' character
        \b      a backspace
        \r      a carriage return (without a line feed)
        \'      a single quote (e.g. in a character constant)
        \"      a double quote (e.g. in a string constant)
        \\      a single backslash
```

For example, `"he said \"hi\""` is a string constant which contains two double quotes, and `'\''` is a character constant consisting of a (single) single quote. Notice once again that the character constant `'A'` is very different from the string constant `"A"`.

2.3 Declarations

[This section corresponds to K&R Sec. 2.4]

Informally, a *variable* (also called an *object*) is a place you can store a value. So that you can refer to it unambiguously, a variable needs a name. You can think of the variables in your program as a set of boxes or cubbyholes, each with a label giving its name; you might imagine that storing a value ``in'' a variable consists of writing the value on a slip of paper and placing it in the cubbyhole.

A *declaration* tells the compiler the name and type of a variable you'll be using in your program. In its simplest form, a declaration consists of the type, the name of the variable, and a terminating semicolon:

```
char c;
int i;
float f;
```

You can also declare several variables of the same type in one declaration, separating them with commas:

```
int i1, i2;
```

Later we'll see that declarations may also contain *initializers*, *qualifiers* and *storage classes*, and that we can declare *arrays, functions, pointers,* and other kinds of data structures.

The placement of declarations is significant. You can't place them just anywhere (i.e. they cannot be interspersed with the other statements in your program). They must either be placed at the beginning of a function, or at the beginning of a brace-enclosed block of statements (which we'll learn about in the next chapter), or outside of any function. Furthermore, the placement of a declaration, as well as its storage class, controls several things about its *visibility* and *lifetime,* as we'll see later.

You may wonder *why* variables must be declared before use. There are two reasons:

1. It makes things somewhat easier on the compiler; it knows right away what kind of storage to allocate and what code to emit to store and manipulate each variable; it doesn't have to try to intuit the programmer's intentions.
2. It forces a bit of useful discipline on the programmer: you cannot introduce variables willy-nilly; you must think about them enough to pick appropriate types for them. (The compiler's error messages to you, telling you that you apparently forgot to declare a variable, are as often helpful as they are a nuisance: they're helpful when they tell you that you misspelled a variable, or forgot to think about exactly how you were going to use it.)

Although there are a few places where declarations can be omitted (in which case the compiler will assume an implicit declaration), making use of these removes the advantages of reason 2 above, so I recommend always declaring everything explicitly.

Most of the time, I recommend writing one declaration per line. For the most part, the compiler doesn't care what order declarations are in. You can order the declarations alphabetically, or in the order that they're used, or to put related declarations next to each other. Collecting all variables of the same type together on one line essentially orders declarations by type, which isn't a very useful order (it's only slightly more useful than random order).

A declaration for a variable can also contain an initial value. This *initializer* consists of an equals sign and an expression, which is usually a single constant:

```
        int i = 1;
        int i1 = 10, i2 = 20;
```

2.4 Variable Names

[This section corresponds to K&R Sec. 2.1]

Within limits, you can give your variables and functions any names you want. These names (the formal term is ``identifiers'') consist of letters, numbers, and underscores. For our purposes, names must begin with a letter. Theoretically, names can be as long as you want, but extremely long ones get tedious to type after a while, and the compiler is not required to keep track of extremely long ones perfectly. (What this means is that if you were to name a variable, say, `supercalafragalisticespialidocious`, the compiler might get lazy and pretend that you'd named it `supercalafragalisticespialidocio`, such that if you later misspelled it `supercalafragalisticespialidociouz`, the compiler wouldn't catch your mistake. Nor would the compiler necessarily be able to tell the difference if for some perverse reason you *deliberately* declared a second variable named `supercalafragalisticespialidociouz`.)

The capitalization of names in C is significant: the variable names `variable`, `Variable`, and `VARIABLE` (as well as silly combinations like `variAble`) are all distinct.

A final restriction on names is that you may not use *keywords* (the words such as `int` and `for` which are part of the syntax of the language) as the names of variables or functions (or as identifiers of any kind).

2.5 Arithmetic Operators

[This section corresponds to K&R Sec. 2.5]

The basic operators for performing arithmetic are the same in many computer languages:

```
        +       addition
```

```
	-	subtraction
	*	multiplication
	/	division
	%	modulus (remainder)
```

The `-` operator can be used in two ways: to subtract two numbers (as in `a - b`), or to negate one number (as in `-a + b` or `a + -b`).

When applied to integers, the division operator `/` discards any remainder, so `1 / 2` is 0 and `7 / 4` is 1. But when either operand is a floating-point quantity (type `float` or `double`), the division operator yields a floating-point result, with a potentially nonzero fractional part. So `1 / 2.0` is 0.5, and `7.0 / 4.0` is 1.75.

The *modulus* operator `%` gives you the remainder when two integers are divided: `1 % 2` is 1; `7 % 4` is 3. (The modulus operator can only be applied to integers.)

An additional arithmetic operation you might be wondering about is exponentiation. Some languages have an exponentiation operator (typically `^ or **`), but C doesn't. (To square or cube a number, just multiply it by itself.)

Multiplication, division, and modulus all have higher *precedence* than addition and subtraction. The term ``precedence'' refers to how ``tightly'' operators bind to their operands (that is, to the things they operate on). In mathematics, multiplication has higher precedence than addition, so `1 + 2 * 3` is 7, not 9. In other words, `1 + 2 * 3` is equivalent to `1 + (2 * 3)`. C is the same way.

All of these operators ``group'' from left to right, which means that when two or more of them have the same precedence and participate next to each other in an expression, the evaluation conceptually proceeds from left to right. For example, `1 - 2 - 3` is equivalent to `(1 - 2) - 3` and gives -4, not +2. (``Grouping'' is sometimes called *associativity*, although the term is used somewhat differently in programming than it is in mathematics. Not all C operators group from left to right; a few group from right to left.)

Whenever the default precedence or associativity doesn't give you the grouping you want, you can always use explicit parentheses. For example, if you wanted to add 1 to 2 and then multiply the result by 3, you could write `(1 + 2) * 3`.

By the way, the word ``arithmetic'' as used in the title of this section is an adjective, not a noun, and it's pronounced differently than the noun: the accent is on the third syllable.

2.6 Assignment Operators

[This section corresponds to K&R Sec. 2.10]

The assignment operator `=` assigns a value to a variable. For example,

```
	x = 1
```

sets `x` to 1, and

```
        a = b
```

sets `a` to whatever `b`'s value is. The expression

```
        i = i + 1
```

is, as we've mentioned elsewhere, the standard programming idiom for increasing a variable's value by 1: this expression takes `i`'s old value, adds 1 to it, and stores it back into `i`. (C provides several ``shortcut'' operators for modifying variables in this and similar ways, which we'll meet later.)

We've called the `=` sign the ``assignment operator'' and referred to ``assignment expressions'' because, in fact, `=` *is* an operator just like `+` or `-`. C does not have ``assignment statements''; instead, an assignment like `a = b` is an expression and can be used wherever any expression can appear. Since it's an expression, the assignment `a = b` has a value, namely, the same value that's assigned to `a`. This value can then be used in a larger expression; for example, we might write

```
        c = a = b
```

which is equivalent to

```
        c = (a = b)
```

and assigns `b`'s value to both `a` and `c`. (The assignment operator, therefore, groups from right to left.) Later we'll see other circumstances in which it can be useful to use the value of an assignment expression.

It's usually a matter of style whether you initialize a variable with an initializer in its declaration or with an assignment expression near where you first use it. That is, there's no particular difference between

```
        int a = 10;
```

and

```
        int a;

        /* later... */

        a = 10;
```

2.7 Function Calls

We'll have much more to say about functions in a later chapter, but for now let's just look at how they're called. (To review: what a function *is* is a piece of code, written by you or by someone else, which performs some useful, compartmentalizable task.) You call a function by mentioning its name followed by a pair of parentheses. If the function takes any arguments, you place the arguments between the parentheses, separated by commas. These are all function calls:

```
        printf("Hello, world!\n")
        printf("%d\n", i)
        sqrt(144.)
        getchar()
```

The arguments to a function can be arbitrary expressions. Therefore, you don't have to say things like

```
        int sum = a + b + c;
        printf("sum = %d\n", sum);
```

if you don't want to; you can instead collapse it to

```
        printf("sum = %d\n", a + b + c);
```

Many functions return values, and when they do, you can embed calls to these functions within larger expressions:

```
        c = sqrt(a * a + b * b)
        x = r * cos(theta)
        i = f1(f2(j))
```

The first expression squares `a` and `b`, computes the square root of the sum of the squares, and assigns the result to `c`. (In other words, it computes `a * a + b * b`, passes that number to the `sqrt` function, and assigns `sqrt`'s return value to `c`.) The second expression passes the value of the variable `theta` to the `cos` (cosine) function, multiplies the result by `r`, and assigns the result to `x`. The third expression passes the value of the variable `j` to the function `f2`, passes the return value of `f2` immediately to the function `f1`, and finally assigns `f1`'s return value to the variable `i`.

Chapter 3: Statements and Control Flow

Statements are the ``steps'' of a program. Most statements compute and assign values or call functions, but we will eventually meet several other kinds of statements as well. By default, statements are executed in sequence, one after another. We can, however, modify that sequence by using *control flow constructs* which arrange that a statement or group of statements is executed only if some condition is true or false, or executed over and over again to form a *loop*. (A somewhat different kind of control flow happens when we call a function: execution of the caller is suspended while the called function proceeds. We'll discuss functions in chapter 5.)

My definitions of the terms *statement* and *control flow* are somewhat circular. A statement is an element within a program which you can apply control flow to; control flow is how you specify the order in which the statements in your program are executed. (A weaker definition of a statement might be ``a part of your program that does something,'' but this definition could as easily be applied to expressions or functions.)

3.1 Expression Statements

[This section corresponds to K&R Sec. 3.1]

Most of the statements in a C program are *expression statements*. An expression statement is simply an expression followed by a semicolon. The lines

```
        i = 0;
        i = i + 1;
and
        printf("Hello, world!\n");
```

are all expression statements. (In some languages, such as Pascal, the semicolon separates statements, such that the last statement is not followed by a semicolon. In C, however, the semicolon is a statement terminator; all simple statements are followed by semicolons. The semicolon is also used for a few other things in C; we've already seen that it terminates declarations, too.)

Expression statements do all of the real work in a C program. Whenever you need to compute new values for variables, you'll typically use expression statements (and they'll typically contain assignment operators). Whenever you want your program to do something visible, in the real world, you'll typically call a function (as part of an expression statement). We've already seen the most basic example: calling the function `printf` to print text to the screen. But anything else you might do--read or write a disk file, talk to a modem or printer, draw pictures on the screen--will also involve function calls. (Furthermore, the functions you call to do these things are usually different depending on which operating system you're using. The C language does not define them, so we won't be talking about or using them much.)

Expressions and expression statements can be arbitrarily complicated. They don't have to consist of exactly one simple function call, or of one simple assignment to a variable. For one thing, many functions return values, and the values they return can then be used by other parts of the expression. For example, C provides a `sqrt` (square root) function, which we might use to compute the hypotenuse of a right triangle like this:

```
        c = sqrt(a*a + b*b);
```

To be useful, an expression statement must do something; it must have some lasting effect on the state of the program. (Formally, a useful statement must have at least one *side effect*.) The first two sample expression statements in this section (above) assign new values to the variable `i`, and the third one calls `printf` to print something out, and these are good examples of statements that do something useful.

(To make the distinction clear, we may note that degenerate constructions such as

```
        0;
        i;
or
        i + 1;
```

are syntactically valid statements--they consist of an expression followed by a semicolon--but in each case, they compute a value without doing anything with it, so the computed value is discarded, and the statement is useless. But if the ``degenerate'' statements in this paragraph don't make much sense to you, don't worry; it's because they, frankly, don't make much sense.)

It's also possible for a single expression to have multiple side effects, but it's easy for such an expression to be (a) confusing or (b) undefined. For now, we'll only be looking at expressions (and, therefore, statements) which do one well-defined thing at a time.

3.2 `if` Statements

[This section corresponds to K&R Sec. 3.2]

The simplest way to modify the control flow of a program is with an `if` statement, which in its simplest form looks like this:

```
        if(x > max)
                max = x;
```

Even if you didn't know any C, it would probably be pretty obvious that what happens here is that if `x` is greater than `max`, `x` gets assigned to `max`. (We'd use code like this to keep track of the maximum value of `x` we'd seen--for each new `x`, we'd compare it to the old maximum value `max`, and if the new value was greater, we'd update `max`.)

More generally, we can say that the syntax of an `if` statement is:

```
        if( expression )
                statement
```

where *expression* is any expression and *statement* is any statement.

What if you have a series of statements, all of which should be executed together or not at all depending on whether some condition is true? The answer is that you enclose them in braces:

```
        if( expression )
                {
                statement<sub>1</sub>
                statement<sub>2</sub>
                statement<sub>3</sub>
                }
```

As a general rule, anywhere the syntax of C calls for a statement, you may write a series of statements enclosed by braces. (You do not need to, and should not, put a semicolon after the closing brace, because the series of statements enclosed by braces is not itself a simple expression statement.)

An `if` statement may also optionally contain a second statement, the ``` ``else ``` clause," which is to be executed if the condition is not met. Here is an example:

```
        if(n > 0)
                average = sum / n;
        else    {
                printf("can't compute average\n");
                average = 0;
                }
```

The first statement or block of statements is executed if the condition *is* true, and the second statement or block of statements (following the keyword `else`) is executed if the condition is *not* true. In this example, we can compute a meaningful average only if `n` is greater than 0; otherwise, we print a message saying that we cannot compute the average. The general syntax of an `if` statement is therefore

```
        if( expression )
                statement<sub>1</sub>
        else
                statement<sub>2</sub>
```

(where both *statement*<sub>1</sub> and *statement*<sub>2</sub> may be lists of statements enclosed in braces).

It's also possible to nest one `if` statement inside another. (For that matter, it's in general possible to nest any kind of statement or control flow construct within another.) For example, here is a little piece of code which decides roughly which quadrant of the compass you're walking into, based on an `x` value which is positive if you're walking east, and a `y` value which is positive if you're walking north:

```
        if(x > 0)
                {
                if(y > 0)
                        printf("Northeast.\n");
                else    printf("Southeast.\n");
                }
```

```
        else    {
                if(y > 0)
                        printf("Northwest.\n");
                else    printf("Southwest.\n");
                }
```
When you have one `if` statement (or loop) nested inside another, it's a very good idea to use explicit braces `{}`, as shown, to make it clear (both to you and to the compiler) how they're nested and which `else` goes with which `if`. It's also a good idea to indent the various levels, also as shown, to make the code more readable to humans. Why do both? You use indentation to make the code visually more readable to yourself and other humans, but the compiler doesn't pay attention to the indentation (since all whitespace is essentially equivalent and is essentially ignored). Therefore, you also have to make sure that the punctuation is right.

Here is an example of another common arrangement of `if` and `else`. Suppose we have a variable `grade` containing a student's numeric grade, and we want to print out the corresponding letter grade. Here is code that would do the job:

```
        if(grade >= 90)
                printf("A");
        else if(grade >= 80)
                printf("B");
        else if(grade >= 70)
                printf("C");
        else if(grade >= 60)
                printf("D");
        else    printf("F");
```
What happens here is that exactly one of the five `printf` calls is executed, depending on which of the conditions is true. Each condition is tested in turn, and if one is true, the corresponding statement is executed, and the rest are skipped. If none of the conditions is true, we fall through to the last one, printing ``F''.

In the cascaded `if`/`else`/`if`/`else`/... chain, each `else` clause is another `if` statement. This may be more obvious at first if we reformat the example, including every set of braces and indenting each `if` statement relative to the previous one:

```
        if(grade >= 90)
                {
                printf("A");
                }
        else    {
                if(grade >= 80)
                        {
                        printf("B");
                        }
                else    {
                        if(grade >= 70)
                                {
                                printf("C");
                                }
                        else    {
                                if(grade >= 60)
```

```
                                        {
                                        printf("D");
                                        }
                                else    {
                                        printf("F");
                                        }
                                }
                        }
                }
```

By examining the code this way, it should be obvious that exactly one of the `printf` calls is executed, and that whenever one of the conditions is found true, the remaining conditions do not need to be checked and none of the later statements within the chain will be executed. But once you've convinced yourself of this and learned to recognize the idiom, it's generally preferable to arrange the statements as in the first example, without trying to indent each successive `if` statement one tabstop further out. (Obviously, you'd run into the right margin very quickly if the chain had just a few more cases!)

3.3 Boolean Expressions

An `if` statement like

```
        if(x > max)
                max = x;
```

is perhaps deceptively simple. Conceptually, we say that it checks whether the condition `x > max` is ``true'' or ``false''. The mechanics underlying C's conception of ``true'' and ``false,'' however, deserve some explanation. We need to understand how true and false values are represented, and how they are interpreted by statements like `if`.

As far as C is concerned, a true/false condition can be represented as an integer. (An integer can represent many values; here we care about only two values: ``true'' and ``false.'' The study of mathematics involving only two values is called Boolean algebra, after George Boole, a mathematician who refined this study.) In C, ``false'' is represented by a value of 0 (zero), and ``true'' is represented by any value that is nonzero. Since there are many nonzero values (at least 65,534, for values of type `int`), when we have to pick a specific value for ``true,'' we'll pick 1.

The *relational operators* such as `<`, `<=`, `>`, and `>=` are in fact operators, just like `+`, `-`, `*`, and `/`. The relational operators take two values, look at them, and ``return'' a value of 1 or 0 depending on whether the tested relation was true or false. The complete set of relational operators in C is:

```
        <       less than
        <=      less than or equal
        >       greater than
        >=      greater than or equal
        ==      equal
        !=      not equal
```

For example, `1 < 2` is 1, `3 > 4` is 0, `5 == 5` is 1, and `6 != 6` is 0.

We've now encountered perhaps the most easy-to-stumble-on ``gotcha!'' in C: the equality-testing operator is `==`, not a single `=`, which is assignment. If you accidentally write

```
        if(a = 0)
```
(and you probably will at some point; everybody makes this mistake), it will *not* test whether `a` is zero, as you probably intended. Instead, it will *assign* 0 to `a`, and then perform the ``true'' branch of the `if` statement if `a` is *non*zero. But `a` will have just been assigned the value 0, so the ``true'' branch will never be taken! (This could drive you crazy while debugging--you wanted to do something if `a` was 0, and after the test, `a` *is* 0, whether it was supposed to be or not, but the ``true'' branch is nevertheless not taken.)

The relational operators work with arbitrary numbers and generate true/false values. You can also combine true/false values by using the *Boolean operators*, which take true/false values as operands and compute new true/false values. The three Boolean operators are:

```
        &&      and
        ||      or
        !       not (takes one operand; ``unary'')
```

The `&&` (``and'') operator takes two true/false values and produces a true (1) result if both operands are true (that is, if the left-hand side is true **and** the right-hand side is true). The `||` (``or'') operator takes two true/false values and produces a true (1) result if either operand is true. The `!` (``not'') operator takes a single true/false value and negates it, turning false to true and true to false (0 to 1 and nonzero to 0).

For example, to test whether the variable `i` lies between 1 and 10, you might use

```
        if(1 < i && i < 10)
                ...
```
Here we're expressing the relation ``i is between 1 and 10'' as ``1 is less than `i` **and** `i` is less than 10.''

It's important to understand why the more obvious expression

```
        if(1 < i < 10)                  /* WRONG */
```
would not work. The expression `1 < i < 10` is parsed by the compiler analogously to `1 + i + 10`. The expression `1 + i + 10` is parsed as `(1 + i) + 10` and means ``add 1 to `i`, and then add the result to 10.'' Similarly, the expression `1 < i < 10` is parsed as `(1 < i) < 10` and means ``see if 1 is less than `i`, and then see if the result is less than 10.'' But in this case, ``the result'' is 1 or 0, depending on whether `i` is greater than 1. Since both 0

and 1 are less than 10, the expression `1 < i < 10` would *always* be true in C, regardless of the value of `i`!

Relational and Boolean expressions are usually used in contexts such as an `if` statement, where something is to be done or not done depending on some condition. In these cases what's actually checked is whether the expression representing the condition has a zero or nonzero value. As long as the expression is a relational or Boolean expression, the interpretation is just what we want. For example, when we wrote

```
        if(x > max)
```

the `>` operator produced a 1 if `x` was greater than `max`, and a 0 otherwise. The `if` statement interprets 0 as false and 1 (or any nonzero value) as true.

But what if the expression is not a relational or Boolean expression? As far as C is concerned, the controlling expression (of conditional statements like `if`) can in fact be *any* expression: it doesn't have to ``look like'' a Boolean expression; it doesn't have to contain relational or logical operators. All C looks at (when it's evaluating an `if` statement, or anywhere else where it needs a true/false value) is whether the expression evaluates to 0 or nonzero. For example, if you have a variable `x`, and you want to do something if `x` is nonzero, it's possible to write

```
        if(x)
                statement
```

and the statement will be executed if `x` is nonzero (since nonzero means ``true'').

This possibility (that the controlling expression of an `if` statement doesn't have to ``look like'' a Boolean expression) is both useful and potentially confusing. It's useful when you have a variable or a function that is ``conceptually Boolean,'' that is, one that you consider to hold a true or false (actually nonzero or zero) value. For example, if you have a variable `verbose` which contains a nonzero value when your program should run in verbose mode and zero when it should be quiet, you can write things like

```
        if(verbose)
                printf("Starting first pass\n");
```

and this code is both legal and readable, besides which it does what you want. The standard library contains a function `isupper()` which tests whether a character is an upper-case letter, so if `c` is a character, you might write

```
        if(isupper(c))
                ...
```

Both of these examples (`verbose` and `isupper()`) are useful and readable.

However, you will eventually come across code like

```
        if(n)
                average = sum / n;
```

where `n` is just a number. Here, the programmer wants to compute the average only if `n` is nonzero (otherwise, of course, the code would divide by 0), and the code works, because, in the context of the `if` statement, the trivial expression `n` is (as always) interpreted as ``true'' if it is nonzero, and ``false'' if it is zero.

``Coding shortcuts'' like these can seem cryptic, but they're also quite common, so you'll need to be able to recognize them even if you don't choose to write them in your own code. Whenever you see code like

```
        if(x)
```
or
```
        if(f())
```

where `x` or `f()` do not have obvious ``Boolean'' names, you can read them as ``if `x` is nonzero'' or ``if `f()` returns nonzero.''

3.4 `while` Loops

[This section corresponds to half of K&R Sec. 3.5]

Loops generally consist of two parts: one or more *control expressions* which (not surprisingly) control the execution of the loop, and the *body*, which is the statement or set of statements which is executed over and over.

The most basic *loop* in C is the `while` loop. A `while` loop has one control expression, and executes as long as that expression is true. This example repeatedly doubles the number 2 (2, 4, 8, 16, ...) and prints the resulting numbers as long as they are less than 1000:

```
        int x = 2;

        while(x < 1000)
                {
                printf("%d\n", x);
                x = x * 2;
                }
```
(Once again, we've used braces `{}` to enclose the group of statements which are to be executed together as the body of the loop.)

The general syntax of a `while` loop is

```
        while( expression )
                statement
```
A `while` loop starts out like an `if` statement: if the condition expressed by the *expression* is true, the *statement* is executed. However, after executing the statement, the condition is tested again, and if it's still true, the statement is executed again. (Presumably, the condition depends on some value which is changed in the body of the loop.) As long as the condition remains true, the body of the loop is executed over and over again. (If the condition is false right at the start, the body of the loop is not executed at all.)

As another example, if you wanted to print a number of blank lines, with the variable `n` holding the number of blank lines to be printed, you might use code like this:

```
	while(n > 0)
		{
		printf("\n");
		n = n - 1;
		}
```

After the loop finishes (when control ``falls out'' of it, due to the condition being false), `n` will have the value 0.

You use a `while` loop when you have a statement or group of statements which may have to be executed a number of times to complete their task. The controlling expression represents the condition ``the loop is not done'' or ``there's more work to do.'' As long as the expression is true, the body of the loop is executed; presumably, it makes at least some progress at its task. When the expression becomes false, the task is done, and the rest of the program (beyond the loop) can proceed. When we think about a loop in this way, we can seen an additional important property: if the expression evaluates to ``false'' before the very first trip through the loop, we make *zero* trips through the loop. In other words, if the task is already done (if there's no work to do) the body of the loop is not executed at all. (It's always a good idea to think about the ``boundary conditions'' in a piece of code, and to make sure that the code will work correctly when there is no work to do, or when there is a trivial task to do, such as sorting an array of one number. Experience has shown that bugs at boundary conditions are quite common.)

3.5 `for` Loops

[This section corresponds to the other half of K&R Sec. 3.5]

Our second loop, which we've seen at least one example of already, is the `for` loop. The first one we saw was:

```
	for (i = 0; i < 10; i = i + 1)
		printf("i is %d\n", i);
```

More generally, the syntax of a `for` loop is

```
	for( expr<sub>1</sub> ; expr<sub>2</sub> ; expr<sub>3</sub> )
		statement
```

(Here we see that the `for` loop has three control expressions. As always, the *statement* can be a brace-enclosed block.)

Many loops are set up to cause some variable to step through a range of values, or, more generally, to set up an initial condition and then modify some value to perform each succeeding loop as long as some condition is true. The three expressions in a `for` loop encapsulate these conditions: *expr*`<sub>`*1*`</sub>` sets up the initial condition, *expr*`<sub>`*2*`</sub>` tests whether another trip through the loop should be taken, and *expr*`<sub>`*3*`</sub>` increments or updates things after each trip through the loop and prior to the next one. In our first example, we had `i = 0` as *expr*`<sub>`*1*`</sub>`, `i < 10` as *expr<sub>2</sub>*, `i = i + 1` as *expr<sub>3</sub>*, and the call to `printf` as *statement*, the body of the loop. So the loop began by setting `i` to 0, proceeded as long as `i` was less than 10, printed out `i`'s value during each trip through the loop, and added 1 to `i` between each trip through the loop.

When the compiler sees a `for` loop, first, *expr<sub>1</sub>* is evaluated. Then, *expr<sub>2</sub>* is evaluated, and if it is true, the body of the loop (*statement*) is executed. Then, *expr<sub>3</sub>* is evaluated to go to the next step, and *expr<sub>2</sub>* is evaluated again, to see if there *is* a next step. During the execution of a `for` loop, the sequence is:

```
expr<sub>1</sub>
expr<sub>2</sub>
statement
expr<sub>3</sub>
expr<sub>2</sub>
statement
expr<sub>3</sub>
...
expr<sub>2</sub>
statement
expr<sub>3</sub>
expr<sub>2</sub>
```

The first thing executed is *expr<sub>1</sub>*. *expr<sub>3</sub>* is evaluated after *every* trip through the loop. The last thing executed is always *expr<sub>2</sub>*, because when *expr<sub>2</sub>* evaluates false, the loop exits.

All three expressions of a `for` loop are optional. If you leave out *expr<sub>1</sub>*, there simply is no initialization step, and the variable(s) used with the loop had better have been initialized already. If you leave out *expr<sub>2</sub>*, there is no test, and the default for the `for` loop is that another trip through the loop should be taken (such that unless you break out of it some other way, the loop runs forever). If you leave out *expr<sub>3</sub>*, there is no increment step.

The semicolons separate the three controlling expressions of a `for` loop. (These semicolons, by the way, have nothing to do with statement terminators.) If you leave out one or more of the expressions, the semicolons remain. Therefore, one way of writing a deliberately infinite loop in C is

```
        for(;;)
                ...
```

It's useful to compare C's `for` loop to the equivalent loops in other computer languages you might know. The C loop

```
        for(i = x; i <= y; i = i + z)
```

is roughly equivalent to:

```
        for I = X to Y step Z  (BASIC)
```

```
do 10 i=x,y,z          (FORTRAN)

for i := x to y        (Pascal)
```

In C (unlike FORTRAN), if the test condition is false before the first trip through the loop, the loop won't be traversed at all. In C (unlike Pascal), a loop control variable (in this case, `i`) is guaranteed to retain its final value after the loop completes, and it is also legal to modify the control variable within the loop, if you really want to. (When the loop terminates due to the test condition turning false, the value of the control variable after the loop will be the first value for which the condition failed, not the last value for which it succeeded.)

It's also worth noting that a `for` loop can be used in more general ways than the simple, iterative examples we've seen so far. The ``control variable'' of a `for` loop does not have to be an integer, and it does not have to be incremented by an additive increment. It could be ``incremented'' by a multiplicative factor (1, 2, 4, 8, ...) if that was what you needed, or it could be a floating-point variable, or it could be another type of variable which we haven't met yet which would step, not over numeric values, but over the elements of an array or other data structure. Strictly speaking, a `for` loop doesn't have to have a ``control variable'' at all; the three expressions can be anything, although the loop will make the most sense if they are related and together form the expected initialize, test, increment sequence.

The powers-of-two example of the previous section does fit this pattern, so we could rewrite it like this:

```
int x;

for(x = 2; x < 1000; x = x * 2)
        printf("%d\n", x);
```

There is no earth-shaking or fundamental difference between the `while` and `for` loops. In fact, given the general `for` loop

```
for(expr<sub>1</sub>; expr<sub>2</sub>; expr<sub>3</sub>)
        statement
```

you could usually rewrite it as a `while` loop, moving the initialize and increment expressions to statements before and within the loop:

```
expr<sub>1</sub> ;
while(expr<sub>2</sub>)
        {
        statement
        expr<sub>3</sub> ;
        }
```

Similarly, given the general `while` loop

```
while(expr)
        statement
```

you could rewrite it as a `for` loop:

```
for(; expr; )
        statement
```

Another contrast between the `for` and `while` loops is that although the test expression (*expr*<sub>*2*</sub>) is optional in a `for` loop, it is required in a `while` loop. If you leave out the controlling expression of a `while` loop, the compiler will complain about a syntax error. (To write a deliberately infinite `while` loop, you have to supply an expression which is always nonzero. The most obvious one would simply be `while(1)` .)

If it's possible to rewrite a `for` loop as a `while` loop and vice versa, why do they both exist? Which one should you choose? In general, when you choose a `for` loop, its three expressions should all manipulate the same variable or data structure, using the initialize, test, increment pattern. If they don't manipulate the same variable or don't follow that pattern, wedging them into a `for` loop buys nothing and a `while` loop would probably be clearer. (The reason that one loop or the other can be clearer is simply that, when you see a `for` loop, you *expect* to see an idiomatic initialize/test/increment of a single variable, and if the `for` loop you're looking at doesn't end up matching that pattern, you've been momentarily misled.)

3.6 `break` and `continue`

[This section corresponds to K&R Sec. 3.7]

Sometimes, due to an exceptional condition, you need to jump out of a loop early, that is, before the main controlling expression of the loop causes it to terminate normally. Other times, in an elaborate loop, you may want to jump back to the top of the loop (to test the controlling expression again, and perhaps begin a new trip through the loop) without playing out all the steps of the current loop. The `break` and `continue` statements allow you to do these two things. (They are, in fact, essentially restricted forms of `goto`.)

To put everything we've seen in this chapter together, as well as demonstrate the use of the `break` statement, here is a program for printing prime numbers between 1 and 100:

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

main()
{
int i, j;

printf("%d\n", 2);

for(i = 3; i <= 100; i = i + 1)
        {
        for(j = 2; j < i; j = j + 1)
                {
                if(i % j == 0)
                        break;
                if(j > sqrt(i))
                        {
                        printf("%d\n", i);
                        break;
                        }
                }
        }

return 0;
}
```

The outer loop steps the variable `i` through the numbers from 3 to 100; the code tests to see if each number has any divisors other than 1 and itself. The trial divisor `j` loops from 2 up to `i`. `j` is a divisor of `i` if the remainder of `i` divided by `j` is 0, so the code uses C's ``remainder'' or ``modulus'' operator `%` to make this test. (Remember that `i % j` gives the remainder when `i` is divided by `j`.)

If the program finds a divisor, it uses `break` to break out of the inner loop, without printing anything. But if it notices that `j` has risen higher than the square root of `i`, without its having found any divisors, then `i` must not have any divisors, so `i` is prime, and its value is printed. (Once we've determined that `i` is prime by noticing that `j > sqrt(i)`, there's no need to try the other trial divisors, so we use a second `break` statement to break out of the loop in that case, too.)

The simple algorithm and implementation we used here (like many simple prime number algorithms) does not work for 2, the only even prime number, so the program ``cheats'' and prints out 2 no matter what, before going on to test the numbers from 3 to 100.

Many improvements to this simple program are of course possible; you might experiment with it. (Did you notice that the ``test'' expression of the inner loop `for(j = 2; j < i; j = j + 1)` is in a sense unnecessary, because the loop always terminates early due to one of the two `break` statements?)

Chapter 4: More about Declarations

4.1 Arrays

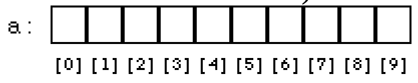
So far, we've been declaring simple variables: the declaration

```
        int i;
```

declares a single variable, named `i`, of type `int`. It is also possible to declare an *array* of several elements. The declaration

```
        int a[10];
```

declares an array, named `a`, consisting of ten elements, each of type `int`. Simply speaking, an array is a variable that can hold more than one value. You specify which of the several values you're referring to at any given time by using a numeric *subscript*. (Arrays in programming are similar to vectors or matrices in mathematics.) We can represent the array `a` above with a picture like this:



In C, arrays are *zero-based*: the ten elements of a 10-element array are numbered from 0 to 9. The subscript which specifies a single element of an array is simply an integer expression in square brackets. The first element of the array is `a[0]`, the second element is `a[1]`, etc. You can use these ``array subscript expressions" anywhere you can use the name of a simple variable, for example:

```
        a[0] = 10;
        a[1] = 20;
        a[2] = a[0] + a[1];
```

Notice that the subscripted array references (i.e. expressions such as `a[0]` and `a[1]`) can appear on either side of the assignment operator.

The subscript does not have to be a constant like `0` or `1`; it can be any integral expression. For example, it's common to loop over all elements of an array:

```
        int i;

        for(i = 0; i < 10; i = i + 1)
                a[i] = 0;
```

This loop sets all ten elements of the array `a` to 0.

Arrays are a real convenience for many problems, but there is not a lot that C will do with them for you automatically. In particular, you can neither set all elements of an array at once nor assign one array to another; both of the assignments

```
        a = 0;                  /* WRONG */
```

and

```
        int b[10];
        b = a;                  /* WRONG */
```

are illegal.

To set all of the elements of an array to some value, you must do so one by one, as in the loop example above. To copy the contents of one array to another, you must again do so one by one:

```
        int b[10];

        for(i = 0; i < 10; i = i + 1)
                b[i] = a[i];
```

Remember that for an array declared

```
        int a[10];
```

there is no element `a[10]`; the topmost element is `a[9]`. This is one reason that zero-based loops are also common in C. Note that the `for` loop

```
        for(i = 0; i < 10; i = i + 1)
                ...
```

does just what you want in this case: it starts at 0, the number 10 suggests (correctly) that it goes through 10 iterations, but the less-than comparison means that the last trip through the loop has `i` set to 9. (The comparison `i <= 9` would also work, but it would be less clear and therefore poorer style.)

In the little examples so far, we've always looped over all 10 elements of the sample array `a`. It's common, however, to use an array that's bigger than necessarily needed, and to use a second variable to keep track of how many elements of the array are currently in use. For example, we might have an integer variable

```
        int na;         /* number of elements of a[] in use */
```

Then, when we wanted to do something with `a` (such as print it out), the loop would run from 0 to `na`, not 10 (or whatever `a`'s size was):

```
        for(i = 0; i < na; i = i + 1)
                printf("%d\n", a[i]);
```

Naturally, we would have to ensure ensure that `na`'s value was always less than or equal to the number of elements actually declared in `a`.

Arrays are not limited to type `int`; you can have arrays of `char` or `double` or any other type.

Here is a slightly larger example of the use of arrays. Suppose we want to investigate the behavior of rolling a pair of dice. The total roll can be anywhere from 2 to 12, and we want to count how often each roll comes up. We will use an array to keep track of the counts: `a[2]` will count how many times we've rolled 2, etc.

We'll simulate the roll of a die by calling C's random number generation function, `rand()`. Each time you call `rand()`, it returns a different, pseudo-random integer. The values that `rand()` returns typically span a large range, so we'll use C's *modulus* (or ``remainder'') operator `%` to produce random numbers in the range we want. The expression `rand() % 6` produces random numbers in the range 0 to 5, and `rand() % 6 + 1` produces random numbers in the range 1 to 6.

Here is the program:

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

main()
{
	int i;
	int d1, d2;
	int a[13];	/* uses [2..12] */

	for(i = 2; i <= 12; i = i + 1)
		a[i] = 0;

	for(i = 0; i < 100; i = i + 1)
		{
		d1 = rand() % 6 + 1;
		d2 = rand() % 6 + 1;
		a[d1 + d2] = a[d1 + d2] + 1;
		}

	for(i = 2; i <= 12; i = i + 1)
		printf("%d: %d\n", i, a[i]);

	return 0;
}
```

We include the header `<stdlib.h>` because it contains the necessary declarations for the `rand()` function. We declare the array of size 13 so that its highest element will be `a[12]`. (We're wasting `a[0]` and `a[1]`; this is no great loss.) The variables `d1` and `d2` contain the rolls of the two individual dice; we add them together to decide which cell of the array to increment, in the line

```
	a[d1 + d2] = a[d1 + d2] + 1;
```

After 100 rolls, we print the array out. Typically (as craps players well know), we'll see mostly 7's, and relatively few 2's and 12's.

(By the way, it turns out that using the `%` operator to reduce the range of the `rand` function is *not* always a good idea. We'll say more about this problem in an exercise.)

4.1.1 Array Initialization

Although it is not possible to assign to all elements of an array at once using an assignment expression, it is possible to initialize some or all elements of an array when the array is defined. The syntax looks like this:

```
	int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

The list of values, enclosed in braces `{}`, separated by commas, provides the initial values for successive elements of the array.

(Under older, pre-ANSI C compilers, you could not always supply initializers for ``local'' arrays inside functions; you could only initialize ``global'' arrays, those outside of any function. Those compilers are now rare, so you shouldn't have to worry about this distinction any more. We'll talk more about local and global variables later in this chapter.)

If there are fewer initializers than elements in the array, the remaining elements are automatically initialized to 0. For example,

```
        int a[10] = {0, 1, 2, 3, 4, 5, 6};
```

would initialize `a[7]`, `a[8]`, and `a[9]` to 0. When an array definition includes an initializer, the array dimension may be omitted, and the compiler will infer the dimension from the number of initializers. For example,

```
        int b[] = {10, 11, 12, 13, 14};
```

would declare, define, and initialize an array `b` of 5 elements (i.e. just as if you'd typed `int b[5]`). Only the dimension is omitted; the brackets `[]` remain to indicate that `b` is in fact an array.

In the case of arrays of `char`, the initializer may be a string constant:

```
        char s1[7] = "Hello,";
        char s2[10] = "there,";
        char s3[] = "world!";
```

As before, if the dimension is omitted, it is inferred from the size of the string initializer. (We haven't covered strings in detail yet--we'll do so in chapter 8--but it turns out that all strings in C are terminated by a special character with the value 0. Therefore, the array `s3` will be of size 7, and the explicitly-sized `s1` does need to be of size at least 7. For `s2`, the last 4 characters in the array will all end up being this zero-value character.)

4.1.2 Arrays of Arrays (``Multidimensional'' Arrays)

[This section is optional and may be skipped.]

When we said that ``Arrays are not limited to type `int`; you can have arrays of... any other type,'' we meant that more literally than you might have guessed. If you have an ``array of `int`,'' it means that you have an array each of whose elements is of type `int`. But you can have an array each of whose elements is of type *x*, where *x* is any type you choose. In particular, you can have an array each of whose elements is another array! We can use these arrays of arrays for the same sorts of tasks as we'd use multidimensional arrays in other computer languages (or matrices in mathematics). Naturally, we are not limited to arrays of arrays, either; we could have an array of arrays of arrays, which would act like a 3-dimensional array, etc.

The declaration of an array of arrays looks like this:

```
        int a2[5][7];
```

You have to read complicated declarations like these ``inside out.'' What this one says is that `a2` is an array of 5 somethings, and that each of the somethings is an array of 7 `int`s. More briefly, ``a2 is an array of 5 arrays of 7 `int`s,'' or, ``a2 is an array of array of `int`.'' In the declaration of `a2`, the brackets closest to the identifier `a2` tell you what `a2` first and foremost is. That's how you know it's an array of 5 arrays of size 7, not the other way around. You can think of `a2` as having 5 ``rows'' and 7 ``columns,'' although this interpretation is not mandatory. (You could also treat the ``first'' or inner subscript as ``x'' and the second as ``y.'' Unless you're doing something fancy, all you have to worry about is that the subscripts when you access the array match those that you used when you declared it, as in the examples below.)

To illustrate the use of multidimensional arrays, we might fill in the elements of the above array `a2` using this piece of code:

```
int i, j;
for(i = 0; i < 5; i = i + 1)
        {
        for(j = 0; j < 7; j = j + 1)
                a2[i][j] = 10 * i + j;
        }
```

This pair of nested loops sets `a[1][2]` to 12, `a[4][1]` to 41, etc. Since the first dimension of `a2` is 5, the first subscripting index variable, `i`, runs from 0 to 4. Similarly, the second subscript varies from 0 to 6.

We could print `a2` out (in a two-dimensional way, suggesting its structure) with a similar pair of nested loops:

```
for(i = 0; i < 5; i = i + 1)
        {
        for(j = 0; j < 7; j = j + 1)
                printf("%d\t", a2[i][j]);
        printf("\n");
        }
```

(The character `\t` in the `printf` string is the tab character.)

Just to see more clearly what's going on, we could make the ``row'' and ``column'' subscripts explicit by printing them, too:

```
for(j = 0; j < 7; j = j + 1)
        printf("\t%d:", j);
printf("\n");

for(i = 0; i < 5; i = i + 1)
        {
        printf("%d:", i);
        for(j = 0; j < 7; j = j + 1)
                printf("\t%d", a2[i][j]);
        printf("\n");
        }
```

This last fragment would print

```
        0:      1:      2:      3:      4:      5:      6:
0:      0       1       2       3       4       5       6
1:      10      11      12      13      14      15      16
2:      20      21      22      23      24      25      26
3:      30      31      32      33      34      35      36
4:      40      41      42      43      44      45      46
```

Finally, there's no reason we have to loop over the ``rows'' first and the ``columns'' second; depending on what we wanted to do, we could interchange the two loops, like this:

```
	for(j = 0; j < 7; j = j + 1)
		{
		for(i = 0; i < 5; i = i + 1)
			printf("%d\t", a2[i][j]);
		printf("\n");
		}
```

Notice that `i` is still the first subscript and it still runs from 0 to 4, and `j` is still the second subscript and it still runs from 0 to 6.

4.2 Visibility and Lifetime (Global Variables, etc.)

We haven't said so explicitly, but variables are channels of communication within a program. You set a variable to a value at one point in a program, and at another point (or points) you read the value out again. The two points may be in adjoining statements, or they may be in widely separated parts of the program.

How long does a variable last? How widely separated can the setting and fetching parts of the program be, and how long after a variable is set does it persist? Depending on the variable and how you're using it, you might want different answers to these questions.

The *visibility* of a variable determines how much of the rest of the program can access that variable. You can arrange that a variable is visible only within one part of one function, or in one function, or in one source file, or anywhere in the program. (We haven't really talked about source files yet; we'll be exploring them soon.)

Why would you want to limit the visibility of a variable? For maximum flexibility, wouldn't it be handy if all variables were potentially visible everywhere? As it happens, that arrangement would be *too* flexible: everywhere in the program, you would have to keep track of the names of all the variables declared anywhere else in the program, so that you didn't accidentally re-use one. Whenever a variable had the wrong value by mistake, you'd have to search the entire program for the bug, because any statement in the entire program could potentially have modified that variable. You would constantly be stepping all over yourself by using a common variable name like `i` in two parts of your program, and having one snippet of code accidentally overwrite the values being used by another part of the code. The communication would be sort of like an old party line--you'd always be accidentally interrupting other conversations, or having your conversations interrupted.

To avoid this confusion, we generally give variables the narrowest or smallest visibility they need. A variable declared within the braces `{}` of a function is visible only within that function; variables declared within functions are called *local variables*. If another function somewhere else declares a local variable with the same name, it's a different variable entirely, and the two don't clash with each other.

On the other hand, a variable declared outside of any function is a *global variable*, and it is potentially visible anywhere within the program. You use global variables when you *do* want the communications path to be able to travel to any part of the program. When you declare a global variable, you will usually give it a longer, more descriptive name (not something generic like `i`) so that whenever you use it you will remember that it's the same variable everywhere.

Another word for the visibility of variables is *scope*.

How long do variables last? By default, local variables (those declared within a function) have *automatic duration*: they spring into existence when the function is called, and they (and their values) disappear when the function returns. Global variables, on the other hand, have *static duration*: they last, and the values stored in them persist, for as long as the program does. (Of course, the values can in general still be overwritten, so they don't necessarily persist forever.)

Finally, it is possible to split a function up into several source files, for easier maintenance. When several source files are combined into one program (we'll be seeing how in the next chapter) the compiler must have a way of correlating the global variables which might be used to communicate between the several source files. Furthermore, if a global variable is going to be useful for communication, there must be exactly one of it: you wouldn't want one function in one source file to store a value in one global variable named `globalvar`, and then have another function in another source file read from a *different* global variable named `globalvar`. Therefore, a global variable should have exactly one *defining instance*, in one place in one source file. If the same variable is to be used anywhere else (i.e. in some other source file or files), the variable is declared in those other file(s) with an *external declaration*, which is not a defining instance. The external declaration says, ``hey, compiler, here's the name and type of a global variable I'm going to use, but don't define it here, don't allocate space for it; it's one that's defined somewhere else, and I'm just referring to it here.'' If you accidentally have two distinct defining instances for a variable of the same name, the compiler (or the linker) will complain that it is ``multiply defined.''

It is also possible to have a variable which is global in the sense that it is declared outside of any function, but private to the one source file it's defined in. Such a variable is visible to the functions in that source file but not to any functions in any other source files, even if they try to issue a matching declaration.

You get any extra control you might need over visibility and lifetime, and you distinguish between defining instances and external declarations, by using *storage classes*. A storage class is an extra keyword at the beginning of a declaration which modifies the declaration in some way. Generally, the storage class (if any) is the first word in the declaration, preceding the type name. (Strictly speaking, this ordering has not traditionally been necessary, and you may see some code with the storage class, type name, and other parts of a declaration in an unusual order.)

We said that, by default, local variables had automatic duration. To give them static duration (so that, instead of coming and going as the function is called, they persist for as long as the function does), you precede their declaration with the `static` keyword:

```
static int i;
```

By default, a declaration of a global variable (especially if it specifies an initial value) is the defining instance. To make it an external declaration, of a variable which is defined somewhere else, you precede it with the keyword `extern`:

```
extern int j;
```

Finally, to arrange that a global variable is visible only within its containing source file, you precede it with the `static` keyword:

```
static int k;
```

Notice that the `static` keyword can do two different things: it adjusts the duration of a local variable from automatic to static, or it adjusts the visibility of a global variable from truly global to private-to-the-file.

To summarize, we've talked about two different attributes of a variable: visibility and duration. These are orthogonal, as shown in this table:

	duration	
visibility	automatic	static
local	normal local variables	static local variables
global	N/A	normal global variables

We can also distinguish between file-scope global variables and truly global variables, based on the presence or absence of the `static` keyword.

We can also distinguish between external declarations and defining instances of global variables, based on the presence or absence of the `extern` keyword.

4.3 Default Initialization

The duration of a variable (whether static or automatic) also affects its default initialization.

If you do not explicitly initialize them, automatic-duration variables (that is, local, non-`static` ones) are not guaranteed to have any particular initial value; they will typically contain garbage. It is therefore a fairly serious error to attempt to use the value of an automatic variable which has never been initialized or assigned to: the program will either work incorrectly, or the garbage value may just happen to be ``correct'' such that the program appears to work correctly! However, the particular value that the garbage takes on can vary depending literally on *anything*: other parts of the program, which compiler was used, which hardware or operating system the program is running on, the time of day, the phase of the moon. (Okay, maybe the phase of the moon is a bit of an exaggeration.) So you hardly want to say that a program which uses an uninitialized variable ``works''; it may seem to work, but it works for the wrong reason, and it may stop working tomorrow.

Static-duration variables (global and `static` local), on the other hand, are guaranteed to be initialized to 0 if you do not use an explicit initializer in the definition.

(Once upon a time, there was another distinction between the initialization of automatic vs. static variables: you could initialize *aggregate* objects, such as arrays, only if they had static duration. If your compiler complains when you try to initialize a local array, it's probably an old, pre-ANSI compiler. Modern, ANSI-compatible compilers remove this limitation, so it's no longer much of a concern.)

4.4 Examples

Here is an example demonstrating almost everything we've seen so far:

```
int globalvar = 1;
extern int anotherglobalvar;
static int privatevar;

f()
{
        int localvar;
        int localvar2 = 2;
        static int persistentvar;
}
```

Here we have six variables, three declared outside and three declared inside of the function `f()`.

`globalvar` is a global variable. The declaration we see is its defining instance (it happens also to include an initial value). `globalvar` can be used anywhere in this source file, and it could be used in other source files, too (as long as corresponding external declarations are issued in those other source files).

`anotherglobalvar` is a second global variable. It is *not* defined here; the defining instance for it (and its initialization) is somewhere else.

`privatevar` is a ``private'' global variable. It can be used anywhere within this source file, but functions in other source files cannot access it, even if they try to issue external declarations for it. (If other source files try to declare a global variable called ``` ``privatevar'' ```, they'll get their own; they won't be sharing this one.) Since it has static duration and receives no explicit initialization, `privatevar` will be initialized to 0.

`localvar` is a local variable within the function `f()`. It can be accessed only within the function `f()`. (If any other part of the program declares a variable named ``` ``localvar'' ```, that variable will be distinct from the one we're looking at here.) `localvar` is conceptually ``created'' each time `f()` is called, and disappears when `f()` returns. Any value which was stored in `localvar` last time `f()` was running will be lost and will not be available next time `f()` is called. Furthermore, since it has no explicit initializer, the value of `localvar` will in general be garbage each time `f()` is called.

`localvar2` is also local, and everything that we said about `localvar` applies to it, except that since its declaration includes an explicit initializer, it will be initialized to 2 each time `f()` is called.

Finally, `persistentvar` is again local to `f()`, but it *does* maintain its value between calls to `f()`. It has static duration but no explicit initializer, so its initial value will be 0.

The defining instances and external declarations we've been looking at so far have all been of simple variables. There are also defining instances and external declarations of functions, which we'll be looking at in the next chapter.

(Also, don't worry about static variables for now if they don't make sense to you; they're a relatively sophisticated concept, which you won't need to use at first.)

The term *declaration* is a general one which encompasses defining instances and external declarations; defining instances and external declarations are two different kinds of declarations. Furthermore, either kind of declaration suffices to inform the compiler of the name and type of a particular variable (or function). If you have the defining instance of a global variable in a source file, the rest of that source file can use that variable without having to issue any external declarations. It's only in source files where the defining instance hasn't been seen that you need external declarations.

You will sometimes hear a defining instance referred to simply as a ``definition,'' and you will sometimes hear an external declaration referred to simply as a ``declaration.'' These usages are mildly ambiguous, in that you can't tell out of context whether a ``declaration'' is a generic declaration (that might be a defining instance or an external declaration) or whether it's an external declaration that specifically is not a defining instance. (Similarly, there are other constructions that can be called ``definitions'' in C, namely the definitions of preprocessor macros, structures, and typedefs, none of which we've met.) In these notes, we'll try to make things clear by using the unambiguous terms *defining instance* and *external declaration*. Elsewhere, you may have to look at the context to determine how the terms ``definition'' and ``declaration'' are being used.

Chapter 5: Functions & Program Structure

[This chapter corresponds to K&R chapter 4.]

A function is a ``black box" that we've locked part of our program into. The idea behind a function is that it *compartmentalizes* part of the program, and in particular, that the code within the function has some useful properties:

1. It performs some well-defined task, which will be useful to other parts of the program.
2. It might be useful to other programs as well; that is, we might be able to reuse it (and without having to rewrite it).
3. The rest of the program doesn't have to know the details of how the function is implemented. This can make the rest of the program easier to think about.
4. The function performs its task *well*. It may be written to do a little more than is required by the first program that calls it, with the anticipation that the calling program (or some other program) may later need the extra functionality or improved performance. (It's important that a finished function do its job well, otherwise there might be a reluctance to call it, and it therefore might not achieve the goal of reusability.)
5. By placing the code to perform the useful task into a function, and simply calling the function in the other parts of the program where the task must be performed, the rest of the program becomes clearer: rather than having some large, complicated, difficult-to-understand piece of code repeated wherever the task is being performed, we have a single simple function call, and the name of the function reminds us which task is being performed.
6. Since the rest of the program doesn't have to know the details of how the function is implemented, the rest of the program doesn't care if the function is reimplemented later, in some different way (as long as it continues to perform its same task, of course!). This means that one part of the program can be rewritten, to improve performance or add a new feature (or simply to fix a bug), without having to rewrite the rest of the program.

Functions are probably the most important weapon in our battle against software complexity. You'll want to learn when it's appropriate to break processing out into functions (and also when it's not), and *how* to set up function interfaces to best achieve the qualities mentioned above: reuseability, information hiding, clarity, and maintainability.

5.1 Function Basics

So what defines a function? It has a *name* that you call it by, and a list of zero or more *arguments* or *parameters* that you hand to it for it to act on or to direct its work; it has a *body* containing the actual instructions (statements) for carrying out the task the function is supposed to perform; and it may give you back a *return value*, of a particular type.

Here is a very simple function, which accepts one argument, multiplies it by 2, and hands that value back:

```
int multbytwo(int x)
{
        int retval;
        retval = x * 2;
        return retval;
}
```

On the first line we see the return type of the function (`int`), the name of the function (`multbytwo`), and a list of the function's arguments, enclosed in parentheses. Each argument has both a name and a type; `multbytwo` accepts one argument, of type `int`, named `x`. The name `x` is arbitrary, and is used only within the definition of `multbytwo`. The caller of this function only needs to know that a single argument of type `int` is expected; the caller does not need to know what name the function will use internally to refer to that argument. (In particular, the caller does not have to pass the value of a variable named `x`.)

Next we see, surrounded by the familiar braces, the body of the function itself. This function consists of one declaration (of a local variable `retval`) and two statements. The first statement is a conventional expression statement, which computes and assigns a value to `retval`, and the second statement is a `return` statement, which causes the function to return to its caller, and also specifies the value which the function returns to its caller.

The `return` statement can return the value of any expression, so we don't really need the local `retval` variable; the function could be collapsed to

```
int multbytwo(int x)
{
        return x * 2;
}
```

How do we call a function? We've been doing so informally since day one, but now we have a chance to call one that we've written, in full detail. Here is a tiny skeletal program to call `multby2`:

```
#include <stdio.h>

extern int multbytwo(int);

int main()
{
        int i, j;
        i = 3;
        j = multbytwo(i);
        printf("%d\n", j);
        return 0;
}
```

This looks much like our other test programs, with the exception of the new line

```
        extern int multbytwo(int);
```
This is an *external function prototype declaration*. It is an external declaration, in that it declares something which is defined somewhere else. (We've already seen the defining instance of the function `multbytwo`, but maybe the compiler hasn't seen it yet.) The function prototype declaration contains the three pieces of information about the function that a caller needs to know: the function's name, return type, and argument type(s). Since we don't care what name the `multbytwo` function will use to refer to its first argument, we don't need to mention it. (On the other hand, if a function takes several arguments, giving them names in the prototype may make it easier to remember which is which, so names may optionally be used in function prototype declarations.) Finally, to remind us that this is an external declaration and not a defining instance, the prototype is preceded by the keyword `extern`.

The presence of the function prototype declaration lets the compiler know that we intend to call this function, `multbytwo`. The information in the prototype lets the compiler generate the correct code for calling the function, and also enables the compiler to check up on our code (by making sure, for example, that we pass the correct number of arguments to each function we call).

Down in the body of main, the action of the function call should be obvious: the line

```
        j = multbytwo(i);
```
calls `multbytwo`, passing it the value of `i` as its argument. When `multbytwo` returns, the return value is assigned to the variable `j`. (Notice that the value of `main`'s local variable `i` will become the value of `multbytwo`'s parameter `x`; this is absolutely not a problem, and is a normal sort of affair.)

This example is written out in ``longhand,'' to make each step equivalent. The variable `i` isn't really needed, since we could just as well call

```
        j = multbytwo(3);
```
And the variable `j` isn't really needed, either, since we could just as well call
```
        printf("%d\n", multbytwo(3));
```
Here, the call to `multbytwo` is a subexpression which serves as the second argument to `printf`. The value returned by `multbytwo` is passed immediately to `printf`. (Here, as in general, we see the flexibility and generality of expressions in C. An argument passed to a function may be an arbitrarily complex subexpression, and a function call is itself an expression which may be embedded as a subexpression within arbitrarily complicated surrounding expressions.)

We should say a little more about the mechanism by which an argument is passed down from a caller into a function. Formally, C is *call by value*, which means that a function receives *copies* of the values of its arguments. We can illustrate this with an example. Suppose, in our implementation of `multbytwo`, we had gotten rid of the unnecessary `retval` variable like this:

```
        int multbytwo(int x)
```

```
	{
		x = x * 2;
		return x;
	}
```

We might wonder, if we wrote it this way, what would happen to the value of the variable `i` when we called

```
	j = multbytwo(i);
```

When our implementation of `multbytwo` changes the value of `x`, does that change the value of `i` up in the caller? The answer is no. `x` receives a copy of `i`'s value, so when we change `x` we don't change `i`.

However, there is an exception to this rule. When the argument you pass to a function is not a single variable, but is rather an array, the function does *not* receive a copy of the array, and it therefore *can* modify the array in the caller. The reason is that it might be too expensive to copy the entire array, and furthermore, it can be useful for the function to write into the caller's array, as a way of handing back more data than would fit in the function's single return value. We'll see an example of an array argument (which the function deliberately writes into) in the next chapter.

5.2 Function Prototypes

In modern C programming, it is considered good practice to use prototype declarations for all functions that you call. As we mentioned, these prototypes help to ensure that the compiler can generate correct code for calling the functions, as well as allowing the compiler to catch certain mistakes you might make.

Strictly speaking, however, prototypes are optional. If you call a function for which the compiler has not seen a prototype, the compiler will do the best it can, assuming that you're calling the function correctly.

If prototypes are a good idea, and if we're going to get in the habit of writing function prototype declarations for functions we call that we've written (such as `multbytwo`), what happens for library functions such as `printf`? Where are their prototypes? The answer is in that boilerplate line

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

we've been including at the top of all of our programs. `stdio.h` is conceptually a file full of external declarations and other information pertaining to the ``Standard I/O'' library functions, including `printf`. The `#include` directive (which we'll meet formally in a later chapter) arranges that all of the declarations within `stdio.h` are considered by the compiler, rather as if we'd typed them all in ourselves. Somewhere within these declarations is an external function prototype declaration for `printf`, which satisfies the rule that there should be a prototype for each function we call. (For other standard library functions we call, there will be iother ``header files'' to include.) Finally, one more thing about external function prototype declarations. We've said that the distinction between external declarations and defining instances of normal variables hinges on the presence or absence of the keyword `extern`. The situation is a little bit different for functions. The ``defining instance'' of a function is the function, including its body (that is, the brace-enclosed list of declarations and statements implementing the function). An external declaration of a function, even without the keyword `extern`, looks nothing like a function declaration. Therefore, the keyword `extern` is optional in function prototype declarations. If you wish, you can write

```
        int multbytwo(int);
```

and this is just as good an external function prototype declaration as

```
        extern int multbytwo(int);
```

(In the first form, without the `extern`, as soon as the compiler sees the semicolon, it knows it's not going to see a function body, so the declaration can't be a definition.) You may want to stay in the habit of using `extern` in all external declarations, including function declarations, since ``` ``extern = external declaration'' ``` is an easier rule to remember.

5.3 Function Philosophy

What makes a good function? The most important aspect of a good ``building block'' is that have a single, well-defined task to perform. When you find that a program is hard to manage, it's often because it has not been designed and broken up into functions cleanly. Two obvious reasons for moving code down into a function are because:

1. It appeared in the main program several times, such that by making it a function, it can be written just once, and the several places where it used to appear can be replaced with calls to the new function.

2. The main program was getting too big, so it could be made (presumably) smaller and more manageable by lopping part of it off and making it a function.

These two reasons are important, and they represent significant benefits of well-chosen functions, but they are not sufficient to automatically identify a good function. As we've been suggesting, a good function has at least these two additional attributes:

3. It does just one well-defined task, and does it well.

4. Its interface to the rest of the program is clean and narrow.

Attribute 3 is just a restatement of two things we said above. Attribute 4 says that you shouldn't have to keep track of too many things when calling a function. If you know what a function is supposed to do, and if its task is simple and well-defined, there should be just a few pieces of information you have to give it to act upon, and one or just a few pieces of information which it returns to you when it's done. If you find yourself having to pass lots and lots of information to a function, or remember details of its internal implementation to make sure that it will work properly this time, it's often a sign that the function is not sufficiently well-defined. (A poorly-defined function may be an arbitrary chunk of code that was ripped out of a main program that was getting too big, such that it essentially has to have access to all of that main function's local variables.)

The whole point of breaking a program up into functions is so that you don't have to think about the entire program at once; ideally, you can think about just one function at a time. We say that a good function is a ``black box,'' which is supposed to suggest that the ``container'' it's in is opaque--callers can't see inside it (and the function inside can't see out). When you call a function, you only have to know what it does, not how it does it. When you're *writing* a function, you only have to know what it's supposed to do, and you don't have to know why or under what circumstances its caller will be calling it. (When designing a function, we should perhaps think about the callers just enough to ensure that the function we're designing will be easy to call, and that we aren't accidentally setting things up so that callers will have to think about any internal details.)

Some functions may be hard to write (if they have a hard job to do, or if it's hard to make them do it truly well), but that difficulty should be compartmentalized along with the function itself. Once you've written a ``hard'' function, you should be able to sit back and relax and watch it do that hard work on call from the rest of your program. It should be pleasant to notice (in the ideal case) how much easier the rest of the program is to write, now that the hard work can be deferred to this workhorse function.

(In fact, if a difficult-to-write function's interface is well-defined, you may be able to get away with writing a quick-and-dirty version of the function first, so that you can begin testing the rest of the program, and then go back later and rewrite the function to do the hard parts. As long as the function's original interface anticipated the hard parts, you won't have to rewrite the rest of the program when you fix the function.)

What I've been trying to say in the preceding few paragraphs is that functions are important for far more important reasons than just saving typing. Sometimes, we'll write a function which we only call once, just because breaking it out into a function makes things clearer and easier.

If you find that difficulties pervade a program, that the hard parts can't be buried inside black-box functions and then forgotten about; if you find that there are hard parts which involve complicated interactions among multiple functions, then the program probably needs redesigning.

For the purposes of explanation, we've been seeming to talk so far only about ``main programs'' and the functions they call and the rationale behind moving some piece of code down out of a ``main program'' into a function. But in reality, there's obviously no need to restrict ourselves to a two-tier scheme. Any function we find ourself writing will often be appropriately written in terms of sub-functions, sub-sub-functions, etc. (Furthermore, the ``main program,'' `main()`, is itself just a function.)

5.4 Separate Compilation--Logistics

When a program consists of many functions, it can be convenient to split them up into several source files. Among other things, this means that when a change is made, only the source file containing the change has to be recompiled, not the whole program.

The job of putting the pieces of a program together and producing the final executable falls to a tool called the *linker*. (We may or may not need to invoke the linker explicitly; a compiler often invokes it automatically, as needed.) The linker looks through all of the pieces making up the program, sorting out the external declarations and defining instances. The compiler has noted the definitions made by each source file, as well as the declarations of things used by each source file but (presumably) defined elsewhere. For each thing (global variable or function) used but not defined by one piece of the program, the linker looks for another piece which does define that thing.

The logistics of writing a program in several source files, and then compiling and linking all of the source files together, depend on the programming environment you're using. We'll cover two possibilities, depending on whether you're using a traditional command-line compiler or a newer integrated development environment (IDE) or other graphical user interface (GUI) compiler.

When using a command-line compiler, there are usually two main steps involved in building an executable program from one or more source files. First, each source file is compiled, resulting in an *object file* containing the machine instructions (generated by the compiler) corresponding to just the code in that source file. Second, the various object files are *linked* together, with each other and with *libraries* containing code for functions which you did not write (such as `printf`), to produce a final, executable program.

Under Unix, the `cc` command can perform one or both steps. So far, we've been using extremely simple invocations of `cc` such as

```
        cc -o hello hello.c
```

This invocation compiles a single source file, `hello.c`, links it, and places the executable in a file named `hello`.

Suppose we have a program which we're trying to build from three separate source files, `x.c`, `y.c`, and `z.c`. We could compile all three of them, and link them together, all at once, with the command

```
        cc -o myprog x.c y.c z.c
```

Alternatively, we could compile them separately: the `-c` option to `cc` tells it to compile only, but not to link. Instead of building an executable, it merely creates an object file, with a name ending in `.o`, for each source file compiled. So the three commands

```
        cc -c x.c
        cc -c y.c
        cc -c y.c
```

would compile `x.c`, `y.c`, and `z.c` and create object files `x.o`, `y.o`, and `z.o`. Then, the three object files could be linked together using

```
        cc -o myprog x.o y.o z.o
```

When the `cc` command is given an `.o` file, it knows that it does not have to compile it (it's an object file, already compiled); it just sends it through to the link process.

Above we mentioned that the second, linking step also involves pulling in library functions. Normally, the functions from the Standard C library are linked in automatically. Occasionally, you must request a library manually; one common situation under Unix is that the math functions tend to be in a separate math library, which is requested by using `-lm` on the command line. Since the libraries must typically be searched *after* your program's own object files are linked (so that the linker knows which library functions your program uses), any `-l` option must appear *after* the names of your files on the command line. For example, to link the object file `mymath.o` (previously compiled with `cc -c mymath.c`) together with the math library, you might use

```
        cc -o mymathprog mymath.o -lm
```

(The `l` in the `-l` option is the lower case ell, for **l**ibrary; it is *not* the digit `1`.)

Everything we've said about `cc` also applies to most other Unix C compilers. (Many of you will be using `gcc`, the FSF's GNU C Compiler.)

There are command-line compilers for MS-DOS systems which work similarly. For example, the Microsoft C compiler comes with a `CL` (``compile and link'') command, which works almost the same as Unix `cc`. You can compile and link in one step:

```
        cl hello.c
```

or you can compile only:

```
        cl /c hello.c
```

creating an object file named `hello.obj` which you can link later.

The preceding has all been about command-line compilers. If you're using some kind of integrated development environment, such as Borland's Turbo C or the Microsoft Programmer's Workbench or Visual C or Think C or Codewarrior, most of the mechanical details are taken care of for you. (There's also less I can say here about these environments, because they're all different.) Typically you define a ``project,'' and there's a way to specify the list of files (modules) which make up your project. The modules might be source files which you typed in or obtained elsewhere, or they might be source files which you created within the environment (perhaps by requesting a ``New source file,'' and typing it in). Typically, the programming environment has a single ``build'' button which does whatever's required to build (and perhaps even execute) your program. There may also be configuration windows in which you can specify compiler options (such as whether you'd like it to accept C or C++). ``See your manual for details.''

Chapter 6: Basic I/O

So far, we've been using `printf` to do output, and we haven't had a way of doing any input. In this chapter, we'll learn a bit more about `printf`, and we'll begin learning about character-based input and output.

6.1 `printf`

`printf`'s name comes from **print f**ormatted. It generates output under the control of a *format string* (its first argument) which consists of literal characters to be printed and also special character sequences--*format specifiers*--which request that other arguments be fetched, formatted, and inserted into the string. Our very first program was nothing more than a call to `printf`, printing a constant string:

```
        printf("Hello, world!\n");
```
Our second program also featured a call to `printf`:
```
        printf("i is %d\n", i);
```
In that case, whenever `printf` ``printed" the string `"i is %d"`, it did not print it verbatim; it replaced the two characters `%d` with the value of the variable `i`.

There are quite a number of format specifiers for `printf`. Here are the basic ones :

```
        %d      print an int argument in decimal
        %ld     print a long int argument in decimal
        %c      print a character
        %s      print a string
        %f      print a float or double argument
        %e      same as %f, but use exponential notation
        %g      use %e or %f, whichever is better
        %o      print an int argument in octal (base 8)
        %x      print an int argument in hexadecimal (base 16)
        %%      print a single %
```

It is also possible to specify the width and precision of numbers and strings as they are inserted (somewhat like FORTRAN `format` statements); we'll present those details in a later chapter. (Very briefly, for those who are curious: a notation like `%3d` means to print an `int` in a field at least 3 spaces wide; a notation like `%5.2f` means to print a `float` or `double` in a field at least 5 spaces wide, with two places to the right of the decimal.)

To illustrate with a few more examples: the call

```
        printf("%c %d %f %e %s %d%%\n", '1', 2, 3.14, 56000000.,
"eight", 9);
```
would print
```
        1 2 3.140000 5.600000e+07 eight 9%
```
The call
```
        printf("%d %o %x\n", 100, 100, 100);
```

would print

```
        100 144 64
```

Successive calls to `printf` just build up the output a piece at a time, so the calls

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

would also print `Hello, world!` (on one line of output).

Earlier we learned that C represents characters internally as small integers corresponding to the characters' values in the machine's character set (typically ASCII). This means that there isn't really much difference between a character and an integer in C; most of the difference is in whether we choose to interpret an integer as an integer or a character. `printf` is one place where we get to make that choice: `%d` prints an integer value as a string of digits representing its decimal value, while `%c` prints the character corresponding to a character set value. So the lines

```
        char c = 'A';
        int i = 97;
        printf("c = %c, i = %d\n", c, i);
```

would print `c` as the character A and `i` as the number 97. But if, on the other hand, we called

```
        printf("c = %d, i = %c\n", c, i);
```

we'd see the decimal value (printed by `%d`) of the character `'A'`, followed by the character (whatever it is) which happens to have the decimal value 97.

You have to be careful when calling `printf`. It has no way of knowing how many arguments you've passed it or what their types are other than by looking for the format specifiers in the format string. If there are more format specifiers (that is, more `%` signs) than there are arguments, or if the arguments have the wrong types for the format specifiers, `printf` can misbehave badly, often printing nonsense numbers or (even worse) numbers which mislead you into thinking that some other part of your program is broken.

Because of some automatic conversion rules which we haven't covered yet, you have a small amount of latitude in the types of the expressions you pass as arguments to `printf`. The argument for `%c` may be of type `char` or `int`, and the argument for `%d` may be of type `char` or `int`. The string argument for `%s` may be a string constant, an array of characters, or a pointer to some characters (though we haven't really covered strings or pointers yet). Finally, the arguments corresponding to `%e`, `%f`, and `%g` may be of types `float` or `double`. But other combinations do *not* work reliably: `%d` will not print a `long int` or a `float` or a `double`; `%ld` will not print an `int`; `%e`, `%f`, and `%g` will not print an `int`.

6.2 Character Input and Output

[This section corresponds to K&R Sec. 1.5]

Unless a program can read some input, it's hard to keep it from doing exactly the same thing every time it's run, and thus being rather boring after a while.

The most basic way of reading input is by calling the function `getchar`. `getchar` reads one character from the ``standard input,'' which is usually the user's keyboard, but which can sometimes be redirected by the operating system. `getchar` returns (rather obviously) the character it reads, or, if there are no more characters available, the special value `EOF` (``end of file'').

A companion function is `putchar`, which writes one character to the ``standard output.'' (The standard output is, again not surprisingly, usually the user's screen, although it, too, can be redirected. `printf`, like `putchar`, prints to the standard output; in fact, you can imagine that `printf` calls `putchar` to actually print each of the characters it formats.)

Using these two functions, we can write a very basic program to copy the input, a character at a time, to the output:

```
#include <stdio.h>

/* copy input to output */

main()
{
	int c;

	c = getchar();

	while(c != EOF)
		{
		putchar(c);
		c = getchar();
		}

	return 0;
}
```

This code is straightforward, and I encourage you to type it in and try it out. It reads one character, and if it is not the `EOF` code, enters a `while` loop, printing one character and reading another, as long as the character read is not `EOF`. This is a straightforward loop, although there's one mystery surrounding the declaration of the variable `c`: if it holds characters, why is it an `int`?

We said that a `char` variable could hold integers corresponding to character set values, and that an `int` could hold integers of more arbitrary values (up to +-32767). Since most character sets contain a few hundred characters (nowhere near 32767), an `int` variable can in general comfortably hold all `char` values, and then some. Therefore, there's nothing wrong with declaring `c` as an `int`. But in fact, it's important to do so, because `getchar` can return every character value, *plus* that special, non-character value `EOF`, indicating that there are no more characters. Type `char` is only guaranteed to be able to hold all the character values; it is *not* guaranteed to be able to hold this ``no more characters'' value without possibly mixing it up with some actual character value. (It's like trying to cram five pounds of books into a four-pound box, or 13 eggs into a carton that holds a dozen.) Therefore, you should always remember to use an `int` for anything you assign `getchar`'s return value to.

When you run the character copying program, and it begins copying its input (your typing) to its output (your screen), you may find yourself wondering how to stop it. It stops when it receives end-of-file (EOF), but how do you send EOF? The answer depends on what kind of computer you're using. On Unix and Unix-related systems, it's almost always control-D. On MS-DOS machines, it's control-Z followed by the RETURN key. Under Think C on the Macintosh, it's control-D, just like Unix. On other systems, you may have to do some research to learn how to send EOF.

(Note, too, that the character you type to generate an end-of-file condition from the keyboard is *not* the same as the special `EOF` value returned by `getchar`. The `EOF` value returned by `getchar` is a code indicating that the input system has detected an end-of-file condition, whether it's reading the keyboard or a file or a magnetic tape or a network connection or anything else. In a disk file, at least, there is not likely to be any character *in* the file corresponding to `EOF`; as far as your program is concerned, `EOF` indicates the absence of any more characters to read.)

Another excellent thing to know when doing any kind of programming is how to terminate a runaway program. If a program is running forever waiting for input, you can usually stop it by sending it an end-of-file, as above, but if it's running forever *not* waiting for something, you'll have to take more drastic measures. Under Unix, control-C (or, occasionally, the DELETE key) will terminate the current program, almost no matter what. Under MS-DOS, control-C or control-BREAK will sometimes terminate the current program, but by default MS-DOS only checks for control-C when it's looking for input, so an infinite loop can be unkillable. There's a DOS command,

```
        break on
```

which tells DOS to look for control-C more often, and I recommend using this command if you're doing any programming. (If a program is in a really tight infinite loop under MS-DOS, there can be no way of killing it short of rebooting.) On the Mac, try command-period or command-option-ESCAPE.

Finally, don't be disappointed (as I was) the first time you run the character copying program. You'll type a character, and see it on the screen right away, and assume it's your program working, but it's only your computer echoing every key you type, as it always does. When you hit RETURN, a full line of characters is made available to your program. It then zips several times through its loop, reading and printing all the characters in the line in quick succession. In other words, when you run this program, it will probably seem to copy the input a line at a time, rather than a character at a time. You may wonder how a program could instead read a character right away, without waiting for the user to hit RETURN. That's an excellent question, but unfortunately the answer is rather complicated, and beyond the scope of our discussion here. (Among other things, how to read a character right away is one of the things that's not defined by the C language, and it's not defined by any of the standard library functions, either. How to do it depends on which operating system you're using.)

Stylistically, the character-copying program above can be said to have one minor flaw: it contains two calls to `getchar`, one which reads the first character and one which reads (by virtue of the fact that it's in the body of the loop) all the other characters. This seems inelegant and perhaps unnecessary, and it can also be risky: if there were more things going on within the loop, and if we ever changed the way we read characters, it would be easy to change one of the `getchar` calls but forget to change the other one. Is there a way to rewrite the loop so that there is only one call to `getchar`, responsible for reading all the characters? Is there a way to read a character, test it for `EOF`, and assign it to the variable `c`, all at the same time?

There is. It relies on the fact that the assignment operator, `=`, is just another operator in C. An assignment is not (necessarily) a standalone statement; it is an expression, and it has a value (the value that's assigned to the variable on the left-hand side), and it can therefore participate in a larger, surrounding expression. Therefore, most C programmers would write the character-copying loop like this:

```
        while((c = getchar()) != EOF)
                putchar(c);
```

What does this mean? The function `getchar` is called, as before, and its return value is assigned to the variable `c`. Then the value is immediately compared against the value `EOF`. Finally, the true/false value of the comparison controls the `while` loop: as long as the value is not `EOF`, the loop continues executing, but as soon as an `EOF` is received, no more trips through the loop are taken, and it exits. The net result is that the call to `getchar` happens inside the test at the top of the `while` loop, and doesn't have to be repeated before the loop and within the loop (more on this in a bit).

Stated another way, the syntax of a `while` loop is always

```
        while( expression ) ...
```

A comparison (using the `!=` operator) is of course an expression; the syntax is

```
        expression != expression
```

And an assignment is an expression; the syntax is

```
        expression = expression
```

What we're seeing is just another example of the fact that expressions can be combined with essentially limitless generality and therefore infinite variety. The left-hand side of the `!=` operator (its first *expression*) is the (sub)expression `c = getchar()`, and the combined expression is the *expression* needed by the `while` loop.

The extra parentheses around

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

are important, and are there because because the *precedence* of the `!=` operator is higher than that of the `=` operator. If we (incorrectly) wrote

```
        while(c = getchar() != EOF)            /* WRONG */
```

the compiler would interpret it as
```
        while(c = (getchar() != EOF))
```
That is, it would assign the result of the `!=` operator to the variable `c`, which is *not* what we want.

(``Precedence'' refers to the rules for which operators are applied to their operands in which order, that is, to the rules controlling the default grouping of expressions and subexpressions. For example, the multiplication operator `*` has higher precedence than the addition operator `+`, which means that the expression `a + b * c` is parsed as `a + (b * c)`. We'll have more to say about precedence later.)

The line

```
        while((c = getchar()) != EOF)
```
epitomizes the cryptic brevity which C is notorious for. You may find this terseness infuriating (and you're not alone!), and it can certainly be carried too far, but bear with me for a moment while I defend it.

The simple example we've been discussing illustrates the tradeoffs well. We have four things to do:

1. call `getchar`,
2. assign its return value to a variable,
3. test the return value against `EOF`, and
4. process the character (in this case, print it out again).

We can't eliminate any of these steps. We have to assign `getchar`'s value to a variable (we can't just use it directly) because we have to do two different things with it (test, and print). Therefore, compressing the assignment and test into the same line is the only good way of avoiding two distinct calls to `getchar`. You may not agree that the compressed idiom is better for being more compact or easier to read, but the fact that there is now only one call to `getchar` *is* a real virtue.

Don't think that you'll have to write compressed lines like

```
        while((c = getchar()) != EOF)
```

right away, or in order to be an ``expert C programmer.'' But, for better or worse, most experienced C programmers do like to use these idioms (whether they're justified or not), so you'll need to be able to at least recognize and understand them when you're reading other peoples' code.

6.3 Reading Lines

It's often convenient for a program to process its input not a character at a time but rather a line at a time, that is, to read an entire line of input and then act on it all at once. The standard C library has a couple of functions for reading lines, but they have a few awkward features, so we're going to learn more about character input (and about writing functions in general) by writing our own function to read one line. Here it is:

```
#include <stdio.h>

/* Read one line from standard input, */
/* copying it to line array (but no more than max chars). */
/* Does not place terminating \n in line array. */
/* Returns line length, or 0 for empty line, or EOF for end-of-file. */

int getline(char line[], int max)
{
int nch = 0;
int c;
max = max - 1;                  /* leave room for '\0' */

while((c = getchar()) != EOF)
        {
        if(c == '\n')
                break;

        if(nch < max)
                {
                line[nch] = c;
                nch = nch + 1;
                }
        }

if(c == EOF && nch == 0)
        return EOF;

line[nch] = '\0';
return nch;
}
```

As the comment indicates, this function will read one line of input from the standard input, placing it into the `line` array. The size of the `line` array is given by the `max` argument; the function will never write more than `max` characters into `line`.

The main body of the function is a `getchar` loop, much as we used in the character-copying program. In the body of this loop, however, we're storing the characters in an array (rather than immediately printing them out). Also, we're only reading one line of characters, then stopping and returning.

There are several new things to notice here.

First of all, the `getline` function accepts an array as a parameter. As we've said, array parameters are an exception to the rule that functions receive copies of their arguments--in the case of arrays, the function *does* have access to the actual array passed by the caller, and *can* modify it. Since the function is accessing the caller's array, not creating a new one to hold a copy, the function does not have to declare the argument array's size;

it's set by the caller. (Thus, the brackets in ``char line[]'' are empty.) However, so that we won't overflow the caller's array by reading too long a line into it, we allow the caller to pass along the size of the array, which we promise not to exceed.

Second, we see an example of the break statement. The top of the loop looks like our earlier character-copying loop--it stops when it reaches EOF--but we only want this loop to read one line, so we also stop (that is, break out of the loop) when we see the \n character signifying end-of-line. An equivalent loop, without the break statement, would be

```
	while((c = getchar()) != EOF && c != '\n')
		{
		if(nch < max)
			{
			line[nch] = c;
			nch = nch + 1;
			}
		}
```

We haven't learned about the internal representation of strings yet, but it turns out that strings in C are simply arrays of characters, which is why we are reading the line into an array of characters. The end of a string is marked by the special character, '\0'. To make sure that there's always room for that character, on our way in we subtract 1 from max, the argument that tells us how many characters we may place in the line array. When we're done reading the line, we store the end-of-string character '\0' at the end of the string we've just built in the line array.

Finally, there's one subtlety in the code which isn't too important for our purposes now but which you may wonder about: it's arranged to handle the possibility that a few characters (i.e. the apparent beginning of a line) are read, followed immediately by an EOF, without the usual \n end-of-line character. (That's why we return EOF only if we received EOF *and* we hadn't read any characters first.)

In any case, the function returns the length (number of characters) of the line it read, not including the \n. (Therefore, it returns 0 for an empty line.) Like getchar, it returns EOF when there are no more lines to read. (It happens that EOF is a negative number, so it will never match the length of a line that getline has read.)

Here is an example of a test program which calls getline, reading the input a line at a time and then printing each line back out:

```
#include <stdio.h>

extern int getline(char [], int);

main()
{
char line[256];
while(getline(line, 256) != EOF)
        printf("you typed \"%s\"\n", line);

return 0;
}
```

The notation `char []` in the function prototype for `getline` says that `getline` accepts as its first argument an array of `char`. When the program calls `getline`, it is careful to pass along the actual size of the array. (You might notice a potential problem: since the number 256 appears in two places, if we ever decide that 256 is too small, and that we want to be able to read longer lines, we could easily change one of the instances of 256, and forget to change the other one. Later we'll learn ways of solving--that is, avoiding--this sort of problem.)

6.4 Reading Numbers

The `getline` function of the previous section reads one line from the user, as a *string*. What if we want to read a number? One straightforward way is to read a string as before, and then immediately convert the string to a number. The standard C library contains a number of functions for doing this. The simplest to use are `atoi()`, which converts a string to an integer, and `atof()`, which converts a string to a floating-point number. (Both of these functions are declared in the header `<stdlib.h>`, so you should `#include` that header at the top of any file using these functions.) You could read an integer from the user like this:

```
        #include <stdlib.h>

        char line[256];
        int n;
        printf("Type an integer:\n");
        getline(line, 256);
        n = atoi(line);
```
Now the variable `n` contains the number typed by the user. (This assumes that the user *did* type a valid number, and that `getline` did not return `EOF`.)

Reading a floating-point number is similar:

```
        #include <stdlib.h>

        char line[256];
        double x;
        printf("Type a floating-point number:\n");
        getline(line, 256);
        x = atof(line);
```
(`atof` is actually declared as returning type `double`, but you could also use it with a variable of type `float`, because in general, C automatically converts between `float` and `double` as needed.)

Another way of reading in numbers, which you're likely to see in other books on C, involves the `scanf` function, but it has several problems, so we won't discuss it for now. (Superficially, `scanf` seems simple enough, which is why it's often used, especially in textbooks. The trouble is that to perform input reliably using `scanf` is not nearly as easy as it looks, especially when you're not sure what the user is going to type.)

Chapter 7: More Operators

In this chapter we'll meet some (though still not all) of C's more advanced arithmetic operators. The ones we'll meet here have to do with making common patterns of operations easier.

It's extremely common in programming to have to *increment* a variable by 1, that is, to add 1 to it. (For example, if you're processing each element of an array, you'll typically write a loop with an index or pointer variable stepping through the elements of the array, and you'll increment the variable each time through the loop.) The classic way to increment a variable is with an assignment like

```
        i = i + 1
```

Such an assignment is perfectly common and acceptable, but it has a few slight problems:

1. As we've mentioned, it looks a little odd, especially from an algebraic perspective.
2. If the object being incremented is not a simple variable, the idiom can become cumbersome to type, and correspondingly more error-prone. For example, the expression
3.
   ```
           a[i+j+2*k] = a[i+j+2*k] + 1
   ```

   is a bit of a mess, and you may have to look closely to see that the similar-looking expression

   ```
     a[i+j+2*k] = a[i+j+2+k] + 1
   ```

   probably has a mistake in it.

4. Since incrementing things is *so* common, it might be nice to have an easier way of doing it.

In fact, C provides not one but two other, simpler ways of incrementing variables and performing other similar operations.

7.1 Assignment Operators

[This section corresponds to K&R Sec. 2.10]

The first and more general way is that any time you have the pattern

```
        v = v op e
```

where *v* is any variable (or anything like `a[i]`), *op* is any of the binary arithmetic operators we've seen so far, and *e* is any expression, you can replace it with the simplified

```
        v op= e
```

For example, you can replace the expressions

```
        i = i + 1
```

```
        j = j - 10
        k = k * (n + 1)
        a[i] = a[i] / b
```
with
```
        i += 1
        j -= 10
        k *= n + 1
        a[i] /= b
```

In an example in a previous chapter, we used the assignment

```
        a[d1 + d2] = a[d1 + d2] + 1;
```
to count the rolls of a pair of dice. Using `+=`, we could simplify this expression to
```
        a[d1 + d2] += 1;
```

As these examples show, you can use the ``op='' form with any of the arithmetic operators (and with several other operators that we haven't seen yet). The expression, *e*, does not have to be the constant 1; it can be any expression. You don't always need as many explicit parentheses when using the *op*`=` operators: the expression

```
        k *= n + 1
```
is interpreted as
```
        k = k * (n + 1)
```

7.2 Increment and Decrement Operators

[This section corresponds to K&R Sec. 2.8]

The assignment operators of the previous section let us replace *v* `=` *v op e* with *v op*`=` *e*, so that we didn't have to mention *v* twice. In the most common cases, namely when we're adding or subtracting the constant 1 (that is, when *op* is `+` or `-` and *e* is 1), C provides another set of shortcuts: the *autoincrement* and *autodecrement* operators. In their simplest forms, they look like this:

```
        ++i             add 1 to i
        --j             subtract 1 from j
```
These correspond to the slightly longer `i += 1` and `j -= 1`, respectively, and also to the fully ``longhand'' forms `i = i + 1` and `j = j - 1`.

The `++` and `--` operators apply to one operand (they're *unary* operators). The expression `++i` adds 1 to `i`, and stores the incremented result back in `i`. This means that these operators don't just compute new values; they also modify the value of some variable. (They share this property--modifying some variable--with the assignment operators; we can say that these operators all have *side effects*. That is, they have some effect, on the side, other than just computing a new value.)

The incremented (or decremented) result is also made available to the rest of the expression, so an expression like

```
        k = 2 * ++i
```
means ``add one to `i`, store the result back in `i`, multiply it by 2, and store *that* result in `k`.'' (This is a pretty meaningless expression; our actual uses of `++` later will make more sense.)

Both the `++` and `--` operators have an unusual property: they can be used in two ways, depending on whether they are written to the left or the right of the variable they're operating on. In either case, they increment or decrement the variable they're operating on; the difference concerns whether it's the old or the new value that's ``returned'' to the surrounding expression. The *prefix* form `++i` increments `i` and returns the incremented value. The *postfix* form `i++` increments `i`, but returns the *prior*, non-incremented value. Rewriting our previous example slightly, the expression

```
        k = 2 * i++
```
means ``take `i`'s old value and multiply it by 2, increment `i`, store the result of the multiplication in `k`.''

The distinction between the prefix and postfix forms of `++` and `--` will probably seem strained at first, but it will make more sense once we begin using these operators in more realistic situations.

For example, our `getline` function of the previous chapter used the statements

```
        line[nch] = c;
        nch = nch + 1;
```
as the body of its inner loop. Using the `++` operator, we could simplify this to
```
        line[nch++] = c;
```
We wanted to increment `nch` *after* deciding which element of the `line` array to store into, so the postfix form `nch++` is appropriate.

Notice that it only makes sense to apply the `++` and `--` operators to variables (or to other ``containers,'' such as `a[i]`). It would be meaningless to say something like

```
        1++
```
or
```
        (2+3)++
```
The `++` operator doesn't just mean ``add one''; it means ``add one *to a variable*'' or ``make a variable's value one more than it was before.'' But `(1+2)` is not a variable, it's an expression; so there's no place for `++` to store the incremented result.

Another unfortunate example is

```
        i = i++;
```
which some confused programmers sometimes write, presumably because they want to be extra sure that `i` is incremented by 1. But `i++` all by itself is sufficient to increment `i` by 1; the extra (explicit) assignment to `i` is unnecessary and in fact counterproductive,

meaningless, and incorrect. If you want to increment `i` (that is, add one to it, and store the result back in `i`), either use

```
        i = i + 1;
or
        i += 1;
or
        ++i;
or
        i++;
```

Don't try to use some bizarre combination.

Did it matter whether we used `++i` or `i++` in this last example? Remember, the difference between the two forms is what value (either the old or the new) is passed on to the surrounding expression. If there is no surrounding expression, if the `++i` or `i++` appears all by itself, to increment `i` and do nothing else, you can use either form; it makes no difference. (Two ways that an expression can appear ``all by itself,'' with ``no surrounding expression,'' are when it is an expression statement terminated by a semicolon, as above, or when it is one of the controlling expressions of a `for` loop.) For example, both the loops

```
        for(i = 0; i < 10; ++i)
                printf("%d\n", i);
```

and

```
        for(i = 0; i < 10; i++)
                printf("%d\n", i);
```

will behave exactly the same way and produce exactly the same results. (In real code, postfix increment is probably more common, though prefix definitely has its uses, too.)

In the preceding section, we simplified the expression

```
        a[d1 + d2] = a[d1 + d2] + 1;
```

from a previous chapter down to

```
        a[d1 + d2] += 1;
```

Using `++`, we could simplify it still further to

```
        a[d1 + d2]++;
```

or

```
        ++a[d1 + d2];
```

(Again, in this case, both are equivalent.)

We'll see more examples of these operators in the next section and in the next chapter.

7.3 Order of Evaluation

[This section corresponds to K&R Sec. 2.12]

When you start using the `++` and `--` operators in larger expressions, you end up with expressions which do several things at once, i.e., they modify several different variables at more or less the same time. When you write such an expression, you must be careful not to have the expression ``pull the rug out from under itself'' by assigning two different values to the same variable, or by assigning a new value to a variable at the same time that another part of the expression is trying to use the value of that variable.

Actually, we had already started writing expressions which did several things at once even before we met the `++` and `--` operators. The expression

```
        (c = getchar()) != EOF
```

assigns `getchar`'s return value to `c`, *and* compares it to `EOF`. The `++` and `--` operators make it much easier to cram a lot into a small expression: the example

```
        line[nch++] = c;
```

from the previous section assigned `c` to `line[nch]`, *and* incremented `nch`. We'll eventually meet expressions which do *three* things at once, such as

```
        a[i++] = b[j++];
```

which assigns `b[j]` to `a[i]`, and increments `i`, *and* increments `j`.

If you're not careful, though, it's easy for this sort of thing to get out of hand. Can you figure out exactly what the expression

```
        a[i++] = b[i++];                /* WRONG */
```

should do? I can't, and here's the important part: *neither can the compiler*. We know that the definition of postfix `++` is that the former value, before the increment, is what goes on to participate in the rest of the expression, but the expression `a[i++] = b[i++]` contains *two* `++` operators. Which of them happens first? Does this expression assign the old `i`th element of `b` to the new `i`th element of `a`, or vice versa? No one knows.

When the order of evaluation matters but is not well-defined (that is, when we can't say for sure which order the compiler will evaluate the various dependent parts in) we say that the meaning of the expression is *undefined*, and if we're smart we won't write the expression in the first place. (Why would anyone ever write an ``undefined'' expression? Because sometimes, the compiler happens to evaluate it in the order a programmer wanted, and the programmer assumes that since it works, it must be okay.)

For example, suppose we carelessly wrote this loop:

```
        int i, a[10];
        i = 0;
        while(i < 10)
                a[i] = i++;                     /* WRONG */
```

It looks like we're trying to set `a[0]` to 0, `a[1]` to 1, etc. But what if the increment `i++` happens before the compiler decides which cell of the array `a` to store the (unincremented) result in? We might end up setting `a[1]` to 0, `a[2]` to 1, etc., instead. Since, in this case, we can't be sure which order things would happen in, we simply shouldn't write code like this. In this case, what we're doing matches the pattern of a `for` loop, anyway, which would be a better choice:

```
        for(i = 0; i < 10; i++)
                a[i] = i;
```

Now that the increment `i++` isn't crammed into the same expression that's setting `a[i]`, the code is perfectly well-defined, and is guaranteed to do what we want.

In general, you should be wary of ever trying to second-guess the order an expression will be evaluated in, with two exceptions:

1. You can obviously assume that precedence will dictate the order in which binary operators are applied. This typically says more than just what order things happens in, but also what the expression actually *means*. (In other words, the precedence of `*` over `+` says more than that the multiplication ``happens first'' in `1 + 2 * 3`; it says that the answer is 7, not 9.)
2. Although we haven't mentioned it yet, it is guaranteed that the logical operators `&&` and `||` are evaluated left-to-right, and that the right-hand side is not evaluated at all if the left-hand side determines the outcome.

To look at one more example, it might seem that the code

```
        int i = 7;
        printf("%d\n", i++ * i++);
```

would have to print 56, because no matter which order the increments happen in, 7`*`8 is 8`*`7 is 56. But `++` just says that the increment happens later, not that it happens immediately, so this code could print 49 (if the compiler chose to perform the multiplication first, and both increments later). And, it turns out that ambiguous expressions like this are such a bad idea that the ANSI C Standard does not require compilers to do anything reasonable with them at all. Theoretically, the above code could end up printing 42, or 8923409342, or 0, or crashing your computer.

Programmers sometimes mistakenly imagine that they can write an expression which tries to do too much at once and then predict exactly how it will behave based on ``order of evaluation.'' For example, we know that multiplication has higher *precedence* than addition, which means that in the expression

```
        i + j * k
```

`j` will be multiplied by `k`, and then `i` will be added to the result. Informally, we often say that the multiplication happens ``before'' the addition. That's true in this case, but it doesn't say as much as we might think about a more complicated expression, such as

```
        i++ + j++ * k++
```

In this case, besides the addition and multiplication, `i`, `j`, and `k` are all being incremented. We can *not* say which of them will be incremented first; it's the compiler's choice. (In particular, it is *not* necessarily the case that `j++` or `k++` will happen first; the compiler might choose to save `i`'s value somewhere and increment `i` first, even though it will have to keep the old value around until after it has done the multiplication.)

In the preceding example, it probably doesn't matter which variable is incremented first. It's not too hard, though, to write an expression where it does matter. In fact, we've seen one already: the ambiguous assignment `a[i++] = b[i++]`. We still don't know which `i++` happens first. (We can *not* assume, based on the right-to-left behavior of the `=` operator, that the right-hand `i++` will happen first.) But if we had to know what `a[i++] = b[i++]` really did, we'd have to know which `i++` happened first.

Finally, note that parentheses don't dictate overall evaluation order any more than precedence does. Parentheses override precedence and say which operands go with which operators, and they therefore affect the overall meaning of an expression, but they don't say anything about the order of subexpressions or side effects. We could not ``fix'' the evaluation order of any of the expressions we've been discussing by adding parentheses. If we wrote

```
	i++ + (j++ * k++)
```

we still wouldn't know which of the increments would happen first. (The parentheses would force the multiplication to happen before the addition, but precedence already would have forced that, anyway.) If we wrote

```
	(i++) * (i++)
```

the parentheses wouldn't force the increments to happen before the multiplication or in any well-defined order; this parenthesized version would be just as undefined as `i++ * i++` was.

There's a line from Kernighan & Ritchie, which I am fond of quoting when discussing these issues [Sec. 2.12, p. 54]:

The moral is that writing code that depends on order of evaluation is a bad programming practice in any language. Naturally, it is necessary to know what things to avoid, but if you don't know *how* they are done on various machines, you won't be tempted to take advantage of a particular implementation.

The first edition of K&R said

...if you don't know *how* they are done on various machines, that innocence may help to protect you.

I actually prefer the first edition wording. Many textbooks encourage you to write small programs to find out how your compiler implements some of these ambiguous expressions, but it's just one step from writing a small program to find out, to writing a real program which makes use of what you've just learned. But you *don't* want to write programs that work only under one particular compiler, that take advantage of the way that one compiler (but perhaps no other) happens to implement the undefined expressions. It's fine to be curious about what goes on ``under the hood,'' and many of you will be curious enough about what's going on with these ``forbidden'' expressions that you'll want to investigate them, but please keep very firmly in mind that, for real programs, the very easiest way of dealing with ambiguous, undefined expressions (which one compiler interprets one way and another interprets another way and a third crashes on) is not to write them in the first place.