The C programming Language

Steve Summit

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 and 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 )
    {
    statement1
    statement2
    statement3
    }
```

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 )
    statement1
else
    statement2
```

(where both *statement*$_1$ and *statement*$_2$ 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( expr1 ; expr2 ; expr3 )
    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: *expr₁* sets up the initial condition, *expr₂* tests whether another trip through the loop should be taken, and *expr₃* 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 *expr₁*, `i < 10` as *expr₂*, `i = i + 1` as *expr₃*, 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, *expr₁* is evaluated. Then, *expr₂* is evaluated, and if it is true, the body of the loop (*statement*) is executed. Then, *expr₃* is evaluated to go to the next step, and *expr₂* is evaluated again, to see if there *is* a next step. During the execution of a `for` loop, the sequence is:

expr₁
expr₂
statement
expr₃
expr₂
statement
expr₃
...
expr₂
statement
expr₃
expr₂

The first thing executed is *expr₁*. *expr₃* is evaluated after *every* trip through the loop. The last thing executed is always *expr₂*, because when *expr₂* evaluates false, the loop exits.

All three expressions of a `for` loop are optional. If you leave out *expr₁*, there simply is no initialization step, and the variable(s) used with the loop had better have been initialized already. If you leave out *expr₂*, 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 *expr₃*, 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(expr1; expr2; expr3)
    statement
```

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

```
expr1 ;
while(expr2)
    {
    statement
    expr3 ;
    }
```

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*$_2$) 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 (and Initialization)

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.

4.1 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

4.1.2 Arrays of Arrays ("Multidimensional" Arrays)

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        static local
    variables        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 and 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 library; 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

   ```
   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.
3. 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.

Chapter 8
Strings

Strings in C are represented by arrays of characters. The end of the string is marked with a special character, the null character, which is simply the character with the value 0. (The null character has no relation except in name to the null pointer. In the ASCII character set, the null character is named NUL.) The null or string-terminating character is represented by another character escape sequence, `\0`. (We've seen it once already, in the `getline` function of chapter 6.)

Because C has no built-in facilities for manipulating entire arrays (copying them, comparing them, etc.), it also has very few built-in facilities for manipulating strings.

In fact, C's only truly built-in string-handling is that it allows us to use string constants (also called string literals) in our code. Whenever we write a string, enclosed in double quotes, C automatically creates an array of characters for us, containing that string, terminated by the `\0` character. For example, we can declare and define an array of characters, and initialize it with a string constant:

```
char string[] = "Hello, world!";
```

In this case, we can leave out the dimension of the array, since the compiler can compute it for us based on the size of the initializer (14, including the terminating `\0`). This is the only case where the compiler sizes a string array for us, however; in other cases, it will be necessary that *we* decide how big the arrays and other data structures we use to hold strings are.

To do anything else with strings, we must typically call functions. The C library contains a few basic string manipulation functions, and to learn more about strings, we'll be looking at how these functions might be implemented.

Since C never lets us assign entire arrays, we use the `strcpy` function to copy one string to another:

```
#include <string.h>

char string1[] = "Hello, world!";
char string2[20];

strcpy(string2, string1);
```

The destination string is `strcpy`'s first argument, so that a call to `strcpy` mimics an assignment expression (with the destination on the left-hand side). Notice that we had to allocate `string2` big enough to hold the string that would be copied to it. Also, at the top of any source file where we're using the standard library's string-handling functions (such as `strcpy`) we must include the line

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

which contains external declarations for these functions.

Since C won't let us compare entire arrays, either, we must call a function to do that, too. The standard library's `strcmp` function compares two strings, and returns 0 if they are identical, or a negative number if the first string is alphabetically "less than" the second string, or a positive number if the first string is "greater." (Roughly speaking, what it means for one string to be "less than" another is that it would come first in a dictionary or telephone book, although there are a few anomalies.) Here is an example:

```
char string3[] = "this is";
char string4[] = "a test";

if(strcmp(string3, string4) == 0)
    printf("strings are equal\n");
else    printf("strings are different\n");
```

This code fragment will print "strings are different". Notice that `strcmp` does *not* return a Boolean, true/false, zero/nonzero answer, so it's not a good idea to write something like

```
if(strcmp(string3, string4))
    ...
```

because it will behave backwards from what you might reasonably expect. (Nevertheless, if you start reading other people's code, you're likely to come across conditionals like `if(strcmp(a, b))` or even `if(!strcmp(a, b))`. The first does something if the strings are unequal; the second does something if they're equal. You can read these more easily if you pretend for a moment that `strcmp`'s name were `strdiff`, instead.)

Another standard library function is `strcat`, which concatenates strings. It does *not* concatenate two strings together and give you a third, new string; what it really does is append one string onto the end of another. (If it gave you a new string, it would have to allocate memory for it somewhere, and the standard library string functions generally never do that for you automatically.) Here's an example:

```
char string5[20] = "Hello, ";
char string6[] = "world!";

printf("%s\n", string5);

strcat(string5, string6);

printf("%s\n", string5);
```

The first call to `printf` prints "Hello, ", and the second one prints "Hello, world!", indicating that the contents of `string6` have been tacked on to the end of `string5`. Notice that we declared `string5` with extra space, to make room for the appended characters.

If you have a string and you want to know its length (perhaps so that you can check whether it will fit in some other array you've allocated for it), you can call `strlen`, which returns the length of the string (i.e. the number of characters in it), not including the `\0`:

```
char string7[] = "abc";
int len = strlen(string7);
printf("%d\n", len);
```

Finally, you can print strings out with `printf` using the `%s` format specifier, as we've been doing in these examples already (e.g. `printf("%s\n", string5);`).

Since a string is just an array of characters, all of the string-handling functions we've just seen can be written quite simply, using no techniques more complicated than the ones we already know. In fact, it's quite instructive to look at how these functions might be implemented. Here is a version of `strcpy`:

```
mystrcpy(char dest[], char src[])
{
int i = 0;

while(src[i] != '\0')
    {
    dest[i] = src[i];
    i++;
    }

dest[i] = '\0';
}
```

We've called it `mystrcpy` instead of `strcpy` so that it won't clash with the version that's already in the standard library. Its operation is simple: it looks at characters in the `src` string one at a time, and as long as they're not `\0`, assigns them, one by one, to the corresponding positions in the `dest` string. When it's done, it terminates the `dest` string by appending a `\0`. (After exiting the `while` loop, `i` is guaranteed to have a value one greater than the subscript of the last character in `src`.) For comparison, here's a way of writing the same code, using a `for` loop:

```
for(i = 0; src[i] != '\0'; i++)
    dest[i] = src[i];

dest[i] = '\0';
```

Yet a third possibility is to move the test for the terminating `\0` character out of the `for` loop header and into the body of the loop, using an explicit `if` and `break` statement, so that we can perform the test after the assignment and therefore use the assignment inside the loop to copy the `\0` to `dest`, too:

```
for(i = 0; ; i++)
    {
    dest[i] = src[i];
    if(src[i] == '\0')
        break;
    }
```

(There are in fact many, many ways to write `strcpy`. Many programmers like to combine the assignment and test, using an expression like `(dest[i] = src[i]) != '\0'`. This is actually the same sort of combined operation as we used in our `getchar` loop in chapter 6.)

Here is a version of `strcmp`:

```
mystrcmp(char str1[], char str2[])
{
int i = 0;

while(1)
    {
    if(str1[i] != str2[i])
        return str1[i] - str2[i];
    if(str1[i] == '\0' || str2[i] == '\0')
        return 0;
    i++;
    }
}
```

Characters are compared one at a time. If two characters in one position differ, the strings are different, and we are supposed to return a value less than zero if the first string (`str1`) is alphabetically less than the second string. Since characters in C are represented by their numeric character set values, and since most reasonable character sets assign values to characters in alphabetical order, we can simply subtract the two differing characters from each other: the expression `str1[i] - str2[i]` will yield a negative result if the `i`'th character of `str1` is less than the corresponding character in `str2`. (As it turns out, this will behave a bit strangely when comparing upper- and lower-case letters, but it's the traditional approach, which the standard versions of `strcmp` tend to use.) If the characters are the same, we continue around the loop, *unless* the characters we just compared were (both) `\0`, in which case we've reached the end of both strings, and they were both equal. Notice that we used what may at first appear to be an infinite loop–the controlling expression is the constant 1, which is always true. What actually happens is that the loop runs until one of the two `return` statements breaks out of it (and the entire function). Note also that when one string is longer than the other, the first test will notice this (because one string will contain a real character at the `[i]` location, while the other will contain `\0`, and these are not equal) and the return value will be computed by subtracting the real character's value from 0, or vice versa. (Thus the shorter string will be treated as "less than" the longer.)

Finally, here is a version of `strlen`:

```
int mystrlen(char str[])
{
int i;

for(i = 0; str[i] != '\0'; i++)
    {}

return i;
}
```

In this case, all we have to do is find the `\0` that terminates the string, and it turns out that the three control expressions of the `for` loop do all the work; there's nothing left to do in the body. Therefore, we use an empty pair of braces {} as the loop body. Equivalently, we could use a null statement, which is simply a semicolon:

```
for(i = 0; str[i] != '\0'; i++)
    ;
```

Empty loop bodies can be a bit startling at first, but they're not unheard of.

Everything we've looked at so far has come out of C's standard libraries. As one last example, let's write a `substr` function, for extracting a substring out of a larger string. We might call it like this:

```
char string8[] = "this is a test";
char string9[10];
substr(string9, string8, 5, 4);
printf("%s\n", string9);
```

The idea is that we'll extract a substring of length 4, starting at character 5 (0-based) of `string8`, and copy the substring to `string9`. Just as with `strcpy`, it's our responsibility to declare the destination string (`string9`) big enough. Here is an implementation of `substr`. Not surprisingly, it's quite similar to `strcpy`:

```
substr(char dest[], char src[], int offset, int len)
{
int i;
for(i = 0; i < len && src[offset + i] != '\0'; i++)
    dest[i] = src[i + offset];
dest[i] = '\0';
}
```

If you compare this code to the code for `mystrcpy`, you'll see that the only differences are that characters are fetched from `src[offset + i]` instead of `src[i]`, and that the loop stops when `len` characters have been copied (or when the `src` string runs out of characters, whichever comes first).

In this chapter, we've been careless about declaring the return types of the string functions, and (with the exception of `mystrlen`) they haven't returned values. The real string functions do return values, but they're of type "pointer to character," which we haven't discussed yet.

When working with strings, it's important to keep firmly in mind the differences between characters and strings. We must also occasionally remember the way characters are represented, and about the relation between character values and integers.

As we have had several occasions to mention, a character is represented internally as a small integer, with a value depending on the character set in use. For example, we might find that `'A'` had the value 65, that `'a'` had the value 97, and that `'+'` had the value 43. (These are, in fact, the values in the ASCII character set, which most computers use. However, you don't need to learn these values, because the vast majority of the time, you use character constants to refer to characters, and the compiler worries about the values for you. Using character constants in preference to raw numeric values also makes your programs more portable.)

As we may also have mentioned, there is a big difference between a character and a string, even a string which contains only one character (other than the `\0`). For example, `'A'` is *not* the same as `"A"`. To drive home this point, let's illustrate it with a few examples.

If you have a string:

```
char string[] = "hello, world!";
```

you can modify its first character by saying

```
string[0] = 'H';
```

(Of course, there's nothing magic about the first character; you can modify any character in the string in this way. Be aware, though, that it is not always safe to modify strings in-place like this; we'll say more about the modifiability of strings in a later chapter on pointers.) Since you're replacing a character, you want a character constant, `'H'`. It would *not* be right to write

```
string[0] = "H";        /* WRONG */
```

because `"H"` is a string (an array of characters), not a single character. (The destination of the assignment, `string[0]`, is a `char`, but the right-hand side is a string; these types don't match.)

On the other hand, when you need a string, you must use a string. To print a single newline, you could call

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

It would *not* be correct to call

```
printf('\n');            /* WRONG */
```

`printf` always wants a string as its first argument. (As one final example, `putchar` wants a single character, so `putchar('\n')` would be correct, and `putchar("\n")` would be incorrect.)

We must also remember the difference between strings and integers. If we treat the character `'1'` as an integer, perhaps by saying

```
int i = '1';
```

we will probably *not* get the value 1 in `i`; we'll get the value of the character `'1'` in the machine's character set. (In ASCII, it's 49.) When we do need to find the numeric value of a digit character (or to go the other way, to get the digit character with a particular value) we can make use of the fact that, in any character set used by C, the values for the digit characters, whatever they are, are contiguous. In other words, no matter what values `'0'` and `'1'` have, `'1' - '0'` will be 1 (and, obviously, `'0' - '0'` will be 0). So, for a variable `c` holding some digit character, the expression

```
c - '0'
```

gives us its value. (Similarly, for an integer value `i`, `i + '0'` gives us the corresponding digit character, as long as $0 <= i <= 9$.)

Just as the character `'1'` is not the integer 1, the string `"123"` is not the integer 123. When we have a string of digits, we can convert it to the corresponding integer by calling the standard function `atoi`:

```
char string[] = "123";
int i = atoi(string);
int j = atoi("456");
```

Later we'll learn how to go in the other direction, to convert an integer into a string. (One way, as long as what you want to do is print the number out, is to call `printf`, using `%d` in the format string.)

Chapter 9
The C Preprocessor

Conceptually, the "preprocessor" is a translation phase that is applied to your source code before the compiler proper gets its hands on it. (Once upon a time, the preprocessor was a separate program, much as the compiler and linker may still be separate programs today.) Generally, the preprocessor performs textual substitutions on your source code, in three sorts of ways:

- File inclusion: inserting the contents of another file into your source file, as if you had typed it all in there.
- Macro substitution: replacing instances of one piece of text with another.
- Conditional compilation: Arranging that, depending on various circumstances, certain parts of your source code are seen or not seen by the compiler at all.

The next three sections will introduce these three preprocessing functions.

The syntax of the preprocessor is different from the syntax of the rest of C in several respects. First of all, the preprocessor is "line based." Each of the preprocessor directives we're going to learn about (all of which begin with the `#` character) must begin at the beginning of a line, and each ends at the end of the line. (The rest of C treats line ends as just another whitespace character, and doesn't care how your program text is arranged into lines.) Secondly, the preprocessor does not know about the structure of C–about functions, statements, or expressions. It is possible to play strange tricks with the preprocessor to turn something which does not look like C into C (or vice versa). It's also possible to run into problems when a preprocessor substitution does not do what you expected it to, because the preprocessor does not respect the structure of C statements and expressions (but you expected it to). For the simple uses of the preprocessor we'll be discussing, you shouldn't have any of these problems, but you'll want to be careful before doing anything tricky or outrageous with the preprocessor. (As it happens, playing tricky and outrageous games with the preprocessor is considered sporting in some circles, but it rapidly gets out of hand, and can lead to bewilderingly impenetrable programs.)

9.1 File Inclusion

[This section corresponds to K&R Sec. 4.11.1]

A line of the form

```
#include <filename.h>
```

or

```
#include "filename.h"
```

causes the contents of the file `filename.h` to be read, parsed, and compiled at that point. (After `filename.h` is processed, compilation continues on the line following the `#include` line.) For example, suppose you got tired of retyping external function prototypes such as

```
extern int getline(char [], int);
```

at the top of each source file. You could instead place the prototype in a header file, perhaps `getline.h`, and then simply place

```
#include "getline.h"
```

at the top of each source file where you called `getline`. (You might not find it worthwhile to create an entire header file for a single function, but if you had a package of several related function, it might be very useful to place all of their declarations in one header file.) As we may have mentioned, that's exactly what the Standard header files such as `stdio.h` are–collections of declarations (including external function prototype declarations) having to do with various sets of Standard library functions. When you use `#include` to read in a header file, you automatically get the prototypes and other declarations it contains, and you *should* use header files, precisely so that you will get the prototypes and other declarations they contain.

The difference between the `<>` and `""` forms is where the preprocessor searches for `filename.h`. As a general rule, it searches for files enclosed in `<>` in central, standard directories, and it searches for files enclosed in `""` in the "current directory," or the directory containing the source file that's doing the including. Therefore, `""` is usually used for header files you've written, and `<>` is usually used for headers which are provided for you (which someone else has written).

The extension "`.h`", by the way, simply stands for "header," and reflects the fact that `#include` directives usually sit at the top (head) of your source files, and contain global declarations and definitions which you would otherwise put there. (That extension is not mandatory–you can theoretically name your own header files anything you wish–but `.h` is traditional, and recommended.)

As we've already begun to see, the reason for putting something in a header file, and then using `#include` to pull that header file into several different source files, is when the something (whatever it is) must be declared or defined consistently in all of the source files. If, instead of using a header file, you typed the something in to each of the source files directly, and the something ever changed, you'd have to edit all those source files, and if you missed one, your program could fail in subtle (or serious) ways due to the mismatched declarations (i.e. due to the incompatibility between the new declaration in one source file and the old one in a source file you forgot to change). Placing common declarations and definitions into header files means that if they ever change, they only have to be changed in one place, which is a much more workable system.

What should you put in header files?

- External declarations of global variables and functions. We said that a global variable must have exactly one *defining instance*, but that it can have external declarations in many places. We said that it was a grave error to issue an external declaration in one place saying that a variable or function has one type, when the defining instance in some other place actually defines it with another type. (If the two places are two source files, separately compiled, the compiler will probably not even catch the discrepancy.) If you put the external declarations in a header file, however, and include the header wherever it's needed, the declarations are virtually guaranteed to be consistent. It's a good idea to include the header in the source file where the defining instance appears, too, so that the compiler can check that the declaration and definition match. (That is, if you ever change the type, you do still have to change it in two places: in the source file where the defining instance occurs, and in the header file where the external declaration appears. But at least you don't have to change it in an arbitrary number

of places, and, if you've set things up correctly, the compiler can catch any remaining mistakes.)

- Preprocessor macro definitions (which we'll meet in the next section).
- Structure definitions (which we haven't seen yet).
- Typedef declarations (which we haven't seen yet).

However, there are a few things *not* to put in header files:

- Defining instances of global variables. If you put these in a header file, and include the header file in more than one source file, the variable will end up multiply defined.
- Function bodies (which are also defining instances). You don't want to put these in headers for the same reason–it's likely that you'll end up with multiple copies of the function and hence "multiply defined" errors. People sometimes put commonly-used functions in header files and then use `#include` to bring them (once) into each program where they use that function, or use `#include` to bring together the several source files making up a program, but both of these are poor ideas. It's much better to learn how to use your compiler or linker to combine together separately-compiled object files.

Since header files typically contain only external declarations, and should *not* contain function bodies, you have to understand just what does and doesn't happen when you `#include` a header file. The header file may provide the declarations for some functions, so that the compiler can generate correct code when you call them (and so that it can make sure that you're calling them correctly), but the header file does *not* give the compiler the functions themselves. The actual functions will be combined into your program at the end of compilation, by the part of the compiler called the linker. The linker may have to get the functions out of libraries, or you may have to tell the compiler/linker where to find them. In particular, if you are trying to use a third-party library containing some useful functions, the library will often come with a header file describing those functions. Using the library is therefore a two-step process: you must `#include` the header in the files where you call the library functions, *and* you must tell the linker to read in the functions from the library itself.

9.2 Macro Definition and Substitution

[This section corresponds to K&R Sec. 4.11.2]

A preprocessor line of the form

`#define` *name text*

defines a macro with the given name, having as its value the given replacement text. After that (for the rest of the current source file), wherever the preprocessor sees that name, it will replace it with the replacement text. The name follows the same rules as ordinary identifiers (it can contain only letters, digits, and underscores, and may not begin with a digit). Since macros behave quite differently from normal variables (or functions), it is customary to give them names which are all capital letters (or at least which begin with a capital letter). The replacement text can be absolutely anything–it's not restricted to numbers, or simple strings, or anything.

The most common use for macros is to propagate various constants around and to make them more self-documenting. We've been saying things like

```
char line[100];
...
getline(line, 100);
```

but this is neither readable nor reliable; it's not necessarily obvious what all those 100's scattered around the program are, and if we ever decide that 100 is too small for the size of the array to hold lines, we'll have to remember to change the number in two (or more) places. A much better solution is to use a macro:

```
#define MAXLINE 100
char line[MAXLINE];
...
getline(line, MAXLINE);
```

Now, if we ever want to change the size, we only have to do it in one place, and it's more obvious what the words `MAXLINE` sprinkled through the program mean than the magic numbers 100 did.

Since the replacement text of a preprocessor macro can be anything, it can also be an expression, although you have to realize that, as always, the text is substituted (and perhaps evaluated) later. No evaluation is performed when the macro is defined. For example, suppose that you write something like

```
#define A 2
#define B 3
#define C A + B
```

(this is a pretty meaningless example, but the situation does come up in practice). Then, later, suppose that you write

```
int x = C * 2;
```

If `A`, `B`, and `C` were ordinary variables, you'd expect `x` to end up with the value 10. But let's see what happens.

The preprocessor always substitutes text for macros exactly as you have written it. So it first substitites the replacement text for the macro `C`, resulting in

```
int x = A + B * 2;
```

Then it substitutes the macros `A` and `B`, resulting in

```
int x = 2 + 3 * 2;
```

Only when the preprocessor is done doing all this substituting does the compiler get into the act. But when it evaluates that expression (using the normal precedence of multiplication over addition), it ends up initializing `x` with the value 8!

To guard against this sort of problem, it is always a good idea to include explicit parentheses in the definitions of macros which contain expressions. If we were to define the macro `C` as

```
#define C (A + B)
```

then the declaration of `x` would ultimately expand to

```
int x = (2 + 3) * 2;
```

and `x` would be initialized to 10, as we probably expected.

Notice that there does not have to be (and in fact there usually is *not*) a semicolon at the end of a `#define` line. (This is just one of the ways that the syntax of the preprocessor is different from the rest of C.) If you accidentally type

```
#define MAXLINE 100;            /* WRONG */
```

then when you later declare

```
char line[MAXLINE];
```

the preprocessor will expand it to

```
char line[100;];            /* WRONG */
```

which is a syntax error. This is what we mean when we say that the preprocessor doesn't know much of anything about the syntax of C–in this last example, the value or replacement text for the macro `MAXLINE` was the 4 characters `1 0 0 ;` , and that's exactly what the preprocessor substituted (even though it didn't make any sense).

Simple macros like `MAXLINE` act sort of like little variables, whose values are constant (or constant expressions). It's also possible to have macros which look like little functions (that is, you invoke them with what looks like function call syntax, and they expand to replacement text which is a function of the actual arguments they are invoked with) but we won't be looking at these yet.

9.3 Conditional Compilation

[This section corresponds to K&R Sec. 4.11.3]

The last preprocessor directive we're going to look at is `#ifdef`. If you have the sequence

```
#ifdef name
program text
#else
more program text
#endif
```

in your program, the code that gets compiled depends on whether a preprocessor macro by that *name* is defined or not. If it is (that is, if there has been a `#define` line for a macro called *name*), then "*program text*" is compiled and "*more program text*" is ignored. If the macro is not defined, "*more program text*" is compiled and "*program text*" is ignored. This looks a lot like an `if` statement, but it behaves completely differently: an `if` statement controls which statements of your program are executed at run time, but `#ifdef` controls which parts of your program actually get compiled.

Just as for the `if` statement, the `#else` in an `#ifdef` is optional. There is a companion directive `#ifndef`, which compiles code if the macro is *not* defined (although the "`#else` clause" of an `#ifndef` directive will then be compiled if the macro *is* defined). There is also an `#if` directive which compiles code depending on whether a compile-time expression is true or false. (The expressions which are allowed in an `#if` directive are somewhat restricted, however, so we won't talk much about `#if` here.)

Conditional compilation is useful in two general classes of situations:

- You are trying to write a portable program, but the way you do something is different depending on what compiler, operating system, or computer you're using. You place different versions of your code, one for each situation, between suitable `#ifdef` directives, and when you compile the progam in a particular environment, you arrange to have the macro names defined which select the variants you need in that environment. (For this reason, compilers usually have ways of letting you define macros from the invocation command line or in a configuration file, and many also predefine certain macro names related to the operating system, processor, or compiler in use. That way,

you don't have to change the code to change the `#define` lines each time you compile it in a different environment.)
For example, in ANSI C, the function to delete a file is `remove`. On older Unix systems, however, the function was called `unlink`. So if `filename` is a variable containing the name of a file you want to delete, and if you want to be able to compile the program under these older Unix systems, you might write

```
#ifdef unix
    unlink(filename);
#else
    remove(filename);
#endif
```

Then, you could place the line

```
#define unix
```

at the top of the file when compiling under an old Unix system. (Since all you're using the macro `unix` for is to control the `#ifdef`, you don't need to give it any replacement text at all. *Any* definition for a macro, even if the replacement text is empty, causes an `#ifdef` to succeed.)
(In fact, in this example, you wouldn't even need to define the macro `unix` at all, because C compilers on old Unix systems tend to predefine it for you, precisely so you can make tests like these.)

- You want to compile several different versions of your program, with different features present in the different versions. You bracket the code for each feature with `#ifdef` directives, and (as for the previous case) arrange to have the right macros defined or not to build the version you want to build at any given time. This way, you can build the several different versions from the same source code. (One common example is whether you turn debugging statements on or off. You can bracket each debugging printout with `#ifdef DEBUG` and `#endif`, and then turn on debugging only when you need it.)
For example, you might use lines like this:

```
#ifdef DEBUG
printf("x is %d\n", x);
#endif
```

to print out the value of the variable `x` at some point in your program to see if it's what you expect. To enable debugging printouts, you insert the line

```
#define DEBUG
```

at the top of the file, and to turn them off, you delete that line, but the debugging printouts quietly remain in your code, temporarily deactivated, but ready to reactivate if you find yourself needing them again later. (Also, instead of inserting and deleting the `#define` line, you might use a compiler flag such as `-DDEBUG` to define the macro `DEBUG` from the compiler invocatin line.)

Conditional compilation can be very handy, but it can also get out of hand. When large chunks of the program are completely different depending on, say, what operating system the program is being compiled for, it's often better to place the different versions in separate source files, and then only use one of the files (corresponding to one of the versions) to build the program on any given system. Also, if you are using an ANSI Standard compiler and you are writing ANSI-compatible code, you usually won't need so much conditional

compilation, because the Standard specifies exactly how the compiler must do certain things, and exactly which library functions it much provide, so you don't have to work so hard to accommodate the old variations among compilers and libraries.

Chapter 10
Pointers

Pointers are often thought to be the most difficult aspect of C. It's true that many people have various problems with pointers, and that many programs founder on pointer-related bugs. Actually, though, many of the problems are not so much with the pointers *per se* but rather with the memory they point to, and more specifically, when there *isn't* any valid memory which they point to. As long as you're careful to ensure that the pointers in your programs always point to valid memory, pointers can be useful, powerful, and relatively trouble-free tools. (We'll talk about memory allocation in the next chapter.)

[This chapter is the only one in this series that contains any graphics. If you are using a text-only browser, there are a few figures you won't be able to see.]

A pointer is a variable that points at, or refers to, another variable. That is, if we have a pointer variable of type "pointer to `int`," it might point to the `int` variable `i`, or to the third cell of the `int` array `a`. Given a pointer variable, we can ask questions like, "What's the value of the variable that this pointer points to?"

Why would we want to have a variable that refers to another variable? Why not just use that other variable directly? The answer is that a level of indirection can be very useful. (Indirection is just another word for the situation when one variable refers to another.)

Imagine a club which elects new officers each year. In its clubroom, it might have a set of mailboxes for each member, along with special mailboxes for the president, secretary, and treasurer. The bank doesn't mail statements to the treasurer under the treasurer's name; it mails them to "treasurer," and the statements go to the mailbox marked "treasurer." This way, the bank doesn't have to change the mailing address it uses every year. The mailboxes labeled "president," "treasurer," and "secretary" are a little bit like pointers–they don't refer to people directly.

If we make the analogy that a mailbox holding letters is like a variable holding numbers, then mailboxes for the president, secretary, and treasurer aren't quite like pointers, because they're still mailboxes which in principle could hold letters directly. But suppose that mail is never actually put in those three mailboxes: suppose each of the officers' mailboxes contains a little marker listing the name of the member currently holding that office. When you're sorting mail, and you have a letter for the treasurer, you first go to the treasurer's mailbox, but rather than putting the letter there, you read the name on the marker there, and put the mail in the mailbox for that person. Similarly, if the club is poorly organized, and the treasurer stops doing his job, and you're the president, and one day you get a call from the bank saying that the club's account is in arrears and the treasurer hasn't done anything about it and asking if you, the president, can look into it; and if the club is so poorly organized that you've forgotten who the treasurer is, you can go to the treasurer's mailbox, read the name on the marker there, and go to *that* mailbox (which is probably overflowing) to find all the treasury-related mail.

We could say that the markers in the mailboxes for the president, secretary, and treasurer were pointers to other mailboxes. In an analogous way, pointer variables in C contain pointers to other variables or memory locations.

10.1 Basic Pointer Operations

[This section corresponds to K&R Sec. 5.1]

The first things to do with pointers are to declare a pointer variable, set it to point somewhere, and finally manipulate the value that it points to. A simple pointer declaration looks like this:

```
int *ip;
```

This declaration looks like our earlier declarations, with one obvious difference: that asterisk. The asterisk means that `ip`, the variable we're declaring, is not of type `int`, but rather of type pointer-to-`int`. (Another way of looking at it is that `*ip`, which as we'll see is the value pointed to by `ip`, will be an `int`.)

We may think of setting a pointer variable to point to another variable as a two-step process: first we generate a pointer to that other variable, then we assign this new pointer to the pointer variable. We can say (but we have to be careful when we're saying it) that a pointer variable has a value, and that its value is "pointer to that other variable". This will make more sense when we see how to generate pointer values.

Pointers (that is, pointer values) are generated with the "address-of" operator `&`, which we can also think of as the "pointer-to" operator. We demonstrate this by declaring (and initializing) an `int` variable `i`, and then setting `ip` to point to it:

```
int i = 5;
ip = &i;
```

The assignment expression `ip = &i;` contains both parts of the "two-step process": `&i` generates a pointer to `i`, and the assignment operator assigns the new pointer to (that is, places it "in") the variable `ip`. Now `ip` "points to" `i`, which we can illustrate with this picture:

`i` is a variable of type `int`, so the value in its box is a number, 5. `ip` is a variable of type pointer-to-`int`, so the "value" in its box is an arrow pointing at another box. Referring once again back to the "two-step process" for setting a pointer variable: the `&` operator draws us the arrowhead pointing at `i`'s box, and the assignment operator `=`, with the pointer variable `ip` on its left, anchors the other end of the arrow in `ip`'s box.

We discover the value pointed to by a pointer using the "contents-of" operator, `*`. Placed in front of a pointer, the `*` operator accesses the value pointed to by that pointer. In other words, if `ip` is a pointer, then the expression `*ip` gives us whatever it is that's in the variable or location pointed to by `ip`. For example, we could write something like

```
printf("%d\n", *ip);
```

which would print 5, since `ip` points to `i`, and `i` is (at the moment) 5.

(You may wonder how the asterisk `*` can be the pointer contents-of operator when it is also the multiplication operator. There is no ambiguity here: it is the multiplication operator when it sits between two variables, and it is the contents-of operator when it sits in front of a single variable. The situation is analogous to the minus sign: between two variables or expressions it's the subtraction operator, but in front of a single operator or expression it's the negation operator. Technical terms you may hear for these distinct roles are unary and binary: a binary operator applies to two operands, usually on either side of it, while a unary operator applies to a single operand.)

The contents-of operator `*` does not merely fetch values through pointers; it can also *set* values through pointers. We can write something like

```
*ip = 7;
```

which means "set whatever `ip` points to to 7." Again, the `*` tells us to go to the location pointed to by `ip`, but this time, the location isn't the one to fetch from–we're on the left-hand sign of an assignment operator, so `*ip` tells us the location to store *to*. (The situation is no different from array subscripting expressions such as `a[3]` which we've already seen appearing on both sides of assignments.)

The result of the assignment `*ip = 7` is that `i`'s value is changed to 7, and the picture changes to:

If we called `printf("%d\n", *ip)` again, it would now print 7.

At this point, you may be wondering why we're going through this rigamarole–if we wanted to set `i` to 7, why didn't we do it directly? We'll begin to explore that next, but first let's notice the difference between changing a pointer (that is, changing what variable it points to) and changing the value at the location it points to. When we wrote `*ip = 7`, we changed the value pointed to by `ip`, but if we declare another variable `j`:

```
int j = 3;
```

and write

```
ip = &j;
```

we've changed `ip` itself. The picture now looks like this:

We have to be careful when we say that a pointer assignment changes "what the pointer points to." Our earlier assignment

```
*ip = 7;
```

changed the value pointed to by `ip`, but this more recent assignment

```
ip = &j;
```

has changed what *variable* `ip` points to. It's true that "what `ip` points to" has changed, but this time, it has changed for a different reason. Neither `i` (which is still 7) nor `j` (which is still 3) has changed. (What has changed is `ip`'s value.) If we again call

```
printf("%d\n", *ip);
```

this time it will print 3.

We can also assign pointer values to other pointer variables. If we declare a second pointer variable:

```
int *ip2;
```

then we can say

```
ip2 = ip;
```

Now `ip2` points where `ip` does; we've essentially made a "copy" of the arrow:

Now, if we set `ip` to point back to `i` again:

```
ip = &i;
```

the two arrows point to different places:

We can now see that the two assignments

```
ip2 = ip;
```

and

```
*ip2 = *ip;
```

do two very different things. The first would make `ip2` again point to where `ip` points (in other words, back to `i` again). The second would store, at the location pointed to by `ip2`, a copy of the value pointed to by `ip`; in other words (if `ip` and `ip2` still point to `i` and `j` respectively) it would set `j` to `i`'s value, or 7.

It's important to keep very clear in your mind the distinction between *a pointer* and *what it points to*. The two are like apples and oranges (or perhaps oil and water); you can't mix them. You can't "set `ip` to 5" by writing something like

```
ip = 5;             /* WRONG */
```

5 is an integer, but `ip` is a pointer. You probably wanted to "set *the value pointed to by* `ip` to 5," which you express by writing

```
*ip = 5;
```

Similarly, you can't "see what `ip` is" by writing

```
printf("%d\n", ip);    /* WRONG */
```

Again, `ip` is a pointer-to-`int`, but `%d` expects an `int`. To print *what* `ip` *points to*, use

```
printf("%d\n", *ip);
```

Finally, a few more notes about pointer declarations. The `*` in a pointer declaration is related to, but different from, the contents-of operator `*`. After we declare a pointer variable

```
int *ip;
```

the expression

```
ip = &i
```

sets what `ip` points to (that is, which location it points to), while the expression

```
*ip = 5
```

sets the value of the location pointed to by `ip`. On the other hand, if we declare a pointer variable and include an initializer:

```
int *ip3 = &i;
```

we're setting the initial value for `ip3`, which is where `ip3` will point, so that initial value is a pointer. (In other words, the `*` in the declaration `int *ip3 = &i;` is not the contents-of operator, it's the indicator that `ip3` is a pointer.)

If you have a pointer declaration containing an initialization, and you ever have occasion to break it up into a simple declaration and a conventional assignment, do it like this:

```
int *ip3;
ip3 = &i;
```

Don't write

```
int *ip3;
*ip3 = &i;
```

or you'll be trying to mix oil and water again.

Also, when we write

```
int *ip;
```

although the asterisk affects `ip`'s type, it goes with the identifier name `ip`, not with the type `int` on the left. To declare two pointers at once, the declaration looks like

```
int *ip1, *ip2;
```

Some people write pointer declarations like this:

```
int* ip;
```

This works for one pointer, because C essentially ignores whitespace. But if you ever write

```
int* ip1, ip2;        /* PROBABLY WRONG */
```

it will declare one pointer-to-`int` `ip1` and one *plain* `int` `ip2`, which is probably not what you meant.

What is all of this good for? If it was just for changing variables like `i` from 5 to 7, it would not be good for much. What it's good for, among other things, is when for various reasons we don't know exactly which variable we want to change, just like the bank didn't know exactly which club member it wanted to send the statement to.

10.2 Pointers and Arrays; Pointer Arithmetic

[This section corresponds to K&R Sec. 5.3]

Pointers do not have to point to single variables. They can also point at the cells of an array. For example, we can write

```
int *ip;
int a[10];
ip = &a[3];
```

and we would end up with `ip` pointing at the fourth cell of the array `a` (remember, arrays are 0-based, so `a[0]` is the first cell). We could illustrate the situation like this:

We'd use this `ip` just like the one in the previous section: `*ip` gives us what `ip` points to, which in this case will be the value in `a[3]`.

Once we have a pointer pointing into an array, we can start doing pointer arithmetic. Given that `ip` is a pointer to `a[3]`, we can add 1 to `ip`:

```
ip + 1
```

What does it mean to add one to a pointer? In C, it gives a pointer to the cell one farther on, which in this case is `a[4]`. To make this clear, let's assign this new pointer to another pointer variable:

```
ip2 = ip + 1;
```

Now the picture looks like this:

If we now do

```
*ip2 = 4;
```

we've set `a[4]` to 4. But it's not necessary to assign a new pointer value to a pointer variable in order to use it; we could also compute a new pointer value and use it immediately:

```
*(ip + 1) = 5;
```

In this last example, we've changed `a[4]` again, setting it to 5. The parentheses are needed because the unary "contents of" operator `*` has higher precedence (i.e., binds more tightly than) the addition operator. If we wrote `*ip + 1`, without the parentheses, we'd be fetching the value pointed to by `ip`, and adding 1 to that value. The expression `*(ip + 1)`, on the other hand, accesses the value one past the one pointed to by `ip`.

Given that we can add 1 to a pointer, it's not surprising that we can add and subtract other numbers as well. If `ip` still points to `a[3]`, then

```
*(ip + 3) = 7;
```

sets `a[6]` to 7, and

```
*(ip - 2) = 4;
```

sets `a[1]` to 4.

Up above, we added 1 to `ip` and assigned the new pointer to `ip2`, but there's no reason we can't add one to a pointer, and change the same pointer:

```
ip = ip + 1;
```

Now `ip` points one past where it used to (to `a[4]`, if we hadn't changed it in the meantime). The shortcuts we learned in a previous chapter all work for pointers, too: we could also increment a pointer using

```
ip += 1;
```

or

```
ip++;
```

Of course, pointers are not limited to `int`s. It's quite common to use pointers to other types, especially `char`. Here is the innards of the `mystrcmp` function we saw in a previous chapter, rewritten to use pointers. (`mystrcmp`, you may recall, compares two strings, character by character.)

```
char *p1 = &str1[0], *p2 = &str2[0];

while(1)
    {
    if(*p1 != *p2)
        return *p1 - *p2;
    if(*p1 == '\0' || *p2 == '\0')
        return 0;
    p1++;
    p2++;
    }
```

The autoincrement operator `++` (like its companion, `--`) makes it easy to do two things at once. We've seen idioms like `a[i++]` which accesses `a[i]` and simultaneously increments `i`, leaving it referencing the next cell of the array `a`. We can do the same thing with pointers: an expression like `*ip++` lets us access what `ip` points to, while simultaneously incrementing `ip` so that it points to the next element. The preincrement form works, too: `*++ip` increments `ip`, then accesses what it points to. Similarly, we can use notations like `*ip--` and `*--ip`.

As another example, here is the `strcpy` (string copy) loop from a previous chapter, rewritten to use pointers:

```
char *dp = &dest[0], *sp = &src[0];
while(*sp != '\0')
    *dp++ = *sp++;
*dp = '\0';
```

(One question that comes up is whether the expression `*p++` increments `p` or what it points to. The answer is that it increments `p`. To increment what `p` points to, you can use `(*p)++`.)

When you're doing pointer arithmetic, you have to remember how big the array the pointer points into is, so that you don't ever point outside it. If the array `a` has 10 elements, you can't access `a[50]` or `a[-1]` or even `a[10]` (remember, the valid subscripts for a 10-element array run from 0 to 9). Similarly, if `a` has 10 elements and `ip` points to `a[3]`, you can't compute or access `ip + 10` or `ip - 5`. (There is one special case: you can, in this case, compute, but not access, a pointer to the nonexistent element just beyond the end of the array, which in this case is `&a[10]`. This becomes useful when you're doing pointer comparisons, which we'll look at next.)

10.3 Pointer Subtraction and Comparison

As we've seen, you can add an integer to a pointer to get a new pointer, pointing somewhere beyond the original (as long as it's in the same array). For example, you might write

```
ip2 = ip1 + 3;
```

Applying a little algebra, you might wonder whether

```
ip2 - ip1 = 3
```

and the answer is, yes. When you subtract two pointers, as long as they point into the same array, the result is the number of elements separating them. You can also ask (again, as long as they point into the same array) whether one pointer is greater or less than another: one pointer is "greater than" another if it points beyond where the other one points. You can also compare pointers for equality and inequality: two pointers are equal if they point to the same variable or to the same cell in an array, and are (obviously) unequal if they don't. (When testing for equality or inequality, the two pointers do not have to point into the same array.)

One common use of pointer comparisons is when copying arrays using pointers. Here is a code fragment which copies 10 elements from `array1` to `array2`, using pointers. It uses an end pointer, `ep`, to keep track of when it should stop copying.

```
int array1[10], array2[10];
int *ip1, *ip2 = &array2[0];
int *ep = &array1[10];
for(ip1 = &array1[0]; ip1 < ep; ip1++)
    *ip2++ = *ip1;
```

As we mentioned, there is no element `array1[10]`, but it is legal to compute a pointer to this (nonexistent) element, as long as we only use it in pointer comparisons like this (that is, as long as we never try to fetch or store the value that it points to.)

10.4 Null Pointers

We said that the value of a pointer variable is a pointer to some other variable. There is one other value a pointer may have: it may be set to a null pointer. A null pointer is a special pointer value that is known not to point anywhere. What this means that no other valid pointer, to any other variable or array cell or anything else, will ever compare equal to a null pointer.

The most straightforward way to "get" a null pointer in your program is by using the predefined constant `NULL`, which is defined for you by several standard header files, including `<stdio.h>`, `<stdlib.h>`, and `<string.h>`. To initialize a pointer to a null pointer, you might use code like

```
#include <stdio.h>

int *ip = NULL;
```

and to test it for a null pointer before inspecting the value pointed to you might use code like

```
if(ip != NULL)
    printf("%d\n", *ip);
```

It is also possible to refer to the null pointer by using a constant `0`, and you will see some code that sets null pointers by simply doing

```
int *ip = 0;
```

(In fact, `NULL` is a preprocessor macro which typically has the value, or replacement text, `0`.)

Furthermore, since the definition of "true" in C is a value that is not equal to 0, you will see code that tests for non-null pointers with abbreviated code like

```
if(ip)
    printf("%d\n", *ip);
```

This has the same meaning as our previous example; `if(ip)` is equivalent to `if(ip != 0)` and to `if(ip != NULL)`.

All of these uses are legal, and although I recommend that you use the constant `NULL` for clarity, you will come across the other forms, so you should be able to recognize them.

You can use a null pointer as a placeholder to remind yourself (or, more importantly, to help your program remember) that a pointer variable does not point anywhere at the moment and that you should not use the "contents of" operator on it (that is, you should not try to inspect what it points to, since it doesn't point to anything). A function that returns pointer values can return a null pointer when it is unable to perform its task. (A null pointer used in this way is analogous to the `EOF` value that functions like `getchar` return.)

As an example, let us write our own version of the standard library function `strstr`, which looks for one string within another, returning a pointer to the string if it can, or a null pointer if it cannot. Here is the function, using the obvious brute-force algorithm: at every character of the input string, the code checks for a match there of the pattern string:

```
#include <stddef.h>

char *mystrstr(char input[], char pat[])
{
    char *start, *p1, *p2;
    for(start = &input[0]; *start != '\0'; start++)
        {       /* for each position in input string... */
        p1 = pat;    /* prepare to check for pattern string there */
        p2 = start;
        while(*p1 != '\0')
            {
            if(*p1 != *p2)    /* characters differ */
                break;
            p1++;
            p2++;
            }
        if(*p1 == '\0')        /* found match */
            return start;
        }

    return NULL;
}
```

The `start` pointer steps over each character position in the `input` string. At each character, the inner loop checks for a match there, by using `p1` to step over the pattern string (`pat`), and `p2` to step over the input string (starting at `start`). We compare successive characters until either (a) we reach the end of the pattern string (`*p1 == '\0'`), or (b) we find two characters which differ. When we're done with the inner loop, if we reached the end of the pattern string (`*p1 == '\0'`), it means that all preceding characters matched, and we found a complete match for the pattern starting at `start`, so we return `start`. Otherwise, we go around the outer loop again, to try another starting position. If we run out of those (if `*start == '\0'`), without finding a match, we return a null pointer.

Notice that the function is declared as returning (and does in fact return) a pointer-to-`char`.

We can use `mystrstr` (or its standard library counterpart `strstr`) to determine whether one string contains another:

```
    if(mystrstr("Hello, world!", "lo") == NULL)
        printf("no\n");
    else    printf("yes\n");
```

In general, C does not initialize pointers to null for you, and it never tests pointers to see if they are null before using them. If one of the pointers in your programs points somewhere some of the time but not all of the time, an excellent convention to use is to set it to a null pointer when it doesn't point anywhere valid, and to test to see if it's a null pointer before using it. But you must use explicit code to set it to `NULL`, and to test it against `NULL`. (In other words, just setting an unused pointer variable to `NULL` doesn't guarantee safety; you also have to check for the null value before using the pointer.) On the other hand, if you know that a particular pointer variable is always valid, you don't have to insert a paranoid test against `NULL` before using it.

10.5 "Equivalence" between Pointers and Arrays

There are a number of similarities between arrays and pointers in C. If you have an array

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

you can refer to `a[0]`, `a[1]`, `a[2]`, etc., or to `a[i]` where `i` is an `int`. If you declare a pointer variable `ip` and set it to point to the beginning of an array:

```
int *ip = &a[0];
```

you can refer to `*ip`, `*(ip+1)`, `*(ip+2)`, etc., or to `*(ip+i)` where `i` is an `int`.

There are also differences, of course. You cannot assign two arrays; the code

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

is illegal. As we've seen, though, you *can* assign two pointer variables:

```
int *ip1, *ip2;
ip1 = &a[0];
ip2 = ip1;
```

Pointer assignment is straightforward; the pointer on the left is simply made to point wherever the pointer on the right does. We haven't copied the data pointed to (there's still just one copy, in the same place); we've just made two pointers point to that one place.

The similarities between arrays and pointers end up being quite useful, and in fact C builds on the similarities, leading to what is called "the equivalence of arrays and pointers in C." When we speak of this "equivalence" we do not mean that arrays and pointers are the same thing (they are in fact quite different), but rather that they can be used in related ways, and that certain operations may be used between them.

The first such operation is that it is possible to (apparently) assign an array to a pointer:

```
int a[10];
int *ip;
ip = a;
```

What can this mean? In that last assignment `ip = a`, aren't we mixing apples and oranges again? It turns out that we are not; C defines the result of this assignment to be that `ip` receives a pointer to the first element of `a`. In other words, it is as if you had written

```
ip = &a[0];
```

The second facet of the equivalence is that you can use the "array subscripting" notation `[i]` on pointers, too. If you write

```
ip[3]
```

it is just as if you had written

```
*(ip + 3)
```

So when you have a pointer that points to a block of memory, such as an array or a part of an array, you can treat that pointer "as if" it *were* an array, using the convenient `[i]` notation. In other words, at the beginning of this section when we talked about `*ip`, `*(ip+1)`, `*(ip+2)`, and `*(ip+i)`, we could have written `ip[0]`, `ip[1]`, `ip[2]`, and `ip[i]`. As we'll see, this can be quite useful (or at least convenient).

The third facet of the equivalence (which is actually a more general version of the first one we mentioned) is that *whenever* you mention the name of an array in a context where the "value" of the array would be needed, C automatically generates a pointer to the first element of the array, as if you had written `&array[0]`. When you write something like

```
int a[10];
int *ip;
ip = a + 3;
```

it is as if you had written

```
ip = &a[0] + 3;
```

which (and you might like to convince yourself of this) gives the same result as if you had written

```
ip = &a[3];
```

For example, if the character array

```
char string[100];
```

contains some string, here is another way to find its length:

```
int len;
char *p;

for(p = string; *p != '\0'; p++)
    ;

len = p - string;
```

After the loop, `p` points to the `'\0'` terminating the string. The expression `p - string` is equivalent to `p - &string[0]`, and gives the length of the string. (Of course, we could also call `strlen`; in fact here we've essentially written another implementation of `strlen`.)

10.6 Arrays and Pointers as Function Arguments

[This section corresponds to K&R Sec. 5.2]

Earlier, we learned that functions in C receive copies of their arguments. (This means that C uses call by value; it means that a function can modify one of its arguments without modifying the value in the caller.) We didn't say so at the time, but when a function is called, the copies of the arguments are made as if by assignment. But since arrays can't be assigned, how can a function receive an array as an argument? The answer will explain why arrays are an apparent exception to the rule that functions cannot modify their arguments.

We've been regularly calling a function `getline` like this:

```
char line[100];
getline(line, 100);
```

with the intention that `getline` read the next line of input into the character array `line`. But in the previous paragraph, we learned that when we mention the name of an array in an expression, the compiler generates a pointer to its first element. So the call above is as if we had written

```
char line[100];
getline(&line[0], 100);
```

In other words, the `getline` function does *not* receive an array of `char` at all; it actually receives a pointer to `char`!

As we've seen throughout this chapter, it's straightforward to manipulate the elements of an array using pointers, so there's no particular insurmountable difficulty if `getline` receives a pointer. One question remains, though: we had been defining `getline` with its `line` parameter declared as an array:

```
int getline(char line[], int max)
{
...
}
```

We mentioned that we didn't have to specify a size for the `line` parameter, with the explanation that `getline` really used the array in its caller, where the actual size was specified. But that declaration certainly does look like an array–how can it work when `getline` actually receives a pointer?

The answer is that the C compiler does a little something behind your back. It knows that whenever you mention an array name in an expression, it (the compiler) generates a pointer to the array's first element. Therefore, it knows that a function can never actually receive an array as a parameter. Therefore, whenever it sees you defining a function that seems to accept an array as a parameter, the compiler quietly pretends that you had declared it as accepting a pointer, instead. The definition of `getline` above is compiled exactly as if it had been written

```
int getline(char *line, int max)
{
...
}
```

Let's look at how `getline` might be written if we thought of its first parameter (argument) as a pointer, instead:

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

#ifndef FGETLINE
while((c = getchar()) != EOF)
#else
while((c = getc(fp)) != EOF)
#endif
    {
    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;
}
```

But, as we've learned, we can also use "array subscript" notation with pointers, so we could rewrite the pointer version of `getline` like this:

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

#ifndef FGETLINE
while((c = getchar()) != EOF)
#else
while((c = getc(fp)) != EOF)
#endif
    {
    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;
}
```

But this is exactly what we'd written before (see chapter 6, Sec. 6.3), except that the declaration of the `line` parameter is different. In other words, within the body of the function, it hardly matters whether we thought `line` was an array or a pointer, since we can use array subscripting notation with both arrays and pointers.

These games that the compiler is playing with arrays and pointers may seem bewildering at first, and it may seem faintly miraculous that everything comes out in the wash when you declare a function like `getline` that seems to accept an array. The equivalence in C between arrays and pointers can be confusing, but it *does* work and is one of the central features of C. If the games which the compiler plays (pretending that you declared a parameter as a pointer when you thought you declared it as an array) bother you, you can do two things:

1. Continue to pretend that functions can receive arrays as parameters; declare and use them that way, but remember that unlike other arguments, a function can modify the copy in its caller of an argument that (seems to be) an array.
2. Realize that arrays are always passed to functions as pointers, and always declare your functions as accepting pointers.

10.7 Strings

Because of the "equivalence" of arrays and pointers, it is extremely common to refer to and manipulate strings as character pointers, or `char *`'s. It is so common, in fact, that it is easy to forget that strings are arrays, and to imagine that they're represented by pointers. (Actually, in the case of strings, it may not even matter that much if the distinction gets a little blurred; there's certainly nothing wrong with referring to a character pointer, suitably initialized, as a "string.") Let's look at a few of the implications:

1. Any function that manipulates a string will actually accept it as a `char *` argument. The caller may pass an array containing a string, but the function will receive a pointer to the array's (string's) first element (character).
2. The `%s` format in `printf` expects a character pointer.
3. Although you have to use `strcpy` to copy a string from one array to another, you can use simple pointer assignment to assign a string to a pointer. The string being assigned might either be in an array or pointed to by another pointer. In other words, given

   ```
   char string[] = "Hello, world!";
   char *p1, *p2;
   ```

   both

   ```
   p1 = string
   ```

   and

   ```
   p2 = p1
   ```

   are legal. (Remember, though, that when you assign a pointer, you're making a copy of the pointer but *not* of the data it points to. In the first example, `p1` ends up pointing to the string in `string`. In the second example, `p2` ends up pointing to the same string as `p1`. In any case, after a pointer assignment, if you ever change the string (or other data) pointed to, the change is "visible" to *both* pointers.
4. Many programs manipulate strings exclusively using character pointers, never explicitly declaring any actual arrays. As long as these programs are careful to allocate appropriate memory for the strings, they're perfectly valid and correct.

When you start working heavily with strings, however, you have to be aware of one subtle fact.

When you initialize a character array with a string constant:

```
char string[] = "Hello, world!";
```

you end up with an array containing the string, and you can modify the array's contents to your heart's content:

```
string[0] = 'J';
```

However, it's possible to use string constants (the formal term is string literals) at other places in your code. Since they're arrays, the compiler generates pointers to their first elements when they're used in expressions, as usual. That is, if you say

```
char *p1 = "Hello";
int len = strlen("world");
```

it's almost as if you'd said

```
char internal_string_1[] = "Hello";
char internal_string_2[] = "world";
char *p1 = &internal_string_1[0];
int len = strlen(&internal_string_2[0]);
```

Here, the arrays named `internal_string_1` and `internal_string_2` are supposed to suggest the fact that the compiler is actually generating little temporary arrays every time you use a string constant in your code. *However*, the subtle fact is that the arrays which are "behind" the string constants are *not* necessarily modifiable. In particular, the compiler may store them in read-only-memory. Therefore, if you write

```
char *p3 = "Hello, world!";
p3[0] = 'J';
```

your program may crash, because it may try to store a value (in this case, the character `'J'`) into nonwritable memory.

The moral is that whenever you're building or modifying strings, you have to make sure that the memory you're building or modifying them in is writable. That memory should either be an array you've allocated, or some memory which you've dynamically allocated by the techniques which we'll see in the next chapter. Make sure that no part of your program will ever try to modify a string which is actually one of the unnamed, unwritable arrays which the compiler generated for you in response to one of your string constants. (The only exception is array initialization, because if you write to such an array, you're writing to the array, not to the string literal which you used to initialize the array.)

10.8 Example: Breaking a Line into "Words"

In an earlier assignment, an "extra credit" version of a problem asked you to write a little checkbook balancing program that accepted a series of lines of the form

```
deposit 1000
check 10
check 12.34
deposit 50
check 20
```

It was a surprising nuisance to do this in an *ad hoc* way, using only the tools we had at the time. It was easy to read each line, but it was cumbersome to break it up into the word ("deposit" or "check") and the amount.

I find it very convenient to use a more general approach: first, break lines like these into a series of whitespace-separated words, then deal with each word separately. To do this, we will use an *array of pointers to* `char`, which we can also think of as an "array of strings," since a string is an array of `char`, and a pointer-to-`char` can easily point at a string. Here is the declaration of such an array:

```
char *words[10];
```

This is the first complicated C declaration we've seen: it says that `words` is an array of 10 pointers to `char`. We're going to write a function, `getwords`, which we can call like this:

```
int nwords;
nwords = getwords(line, words, 10);
```

where `line` is the line we're breaking into words, `words` is the array to be filled in with the (pointers to the) words, and `nwords` (the return value from `getwords`) is the number of words which the function finds. (As with `getline`, we tell the function the size of the array so that if the line should happen to contain more words than that, it won't overflow the array).

Here is the definition of the `getwords` function. It finds the beginning of each word, places a pointer to it in the array, finds the end of that word (which is signified by at least one whitespace character) and terminates the word by placing a `'\0'` character after it. (The `'\0'` character will overwrite the first whitespace character following the word.) Note that the original input string is therefore modified by `getwords`: if you were to try to print the input line after calling `getwords`, it would appear to contain only its first word (because of the first inserted `'\0'`).

```
#include <stddef.h>
#include <ctype.h>

getwords(char *line, char *words[], int maxwords)
{
char *p = line;
int nwords = 0;

while(1)
    {
    while(isspace(*p))
        p++;

    if(*p == '\0')
        return nwords;

    words[nwords++] = p;

    while(!isspace(*p) && *p != '\0')
        p++;

    if(*p == '\0')
        return nwords;

    *p++ = '\0';

    if(nwords >= maxwords)
        return nwords;
    }
}
```

Each time through the outer `while` loop, the function tries to find another word. First it skips over whitespace (which might be leading spaces on the line, or the space(s) separating this word from the previous one). The `isspace` function is new: it's in the standard library, declared in the header file `<ctype.h>`, and it returns nonzero ("true") if the character you hand it is a space character (a space or a tab, or any other whitespace character there might happen to be).

When the function finds a non-whitespace character, it has found the beginning of another word, so it places the pointer to that character in the next cell of the `words` array. Then it steps though the word, looking at non-whitespace characters, until it finds another whitespace character, or the `\0` at the end of the line. If it finds the `\0`, it's done with the entire line; otherwise, it changes the whitespace character to a `\0`, to terminate the word

it's just found, and continues. (If it's found as many words as will fit in the `words` array, it returns prematurely.)

Each time it finds a word, the function increments the number of words (`nwords`) it has found. Since arrays in C start at `[0]`, the number of words the function has found so far is also the index of the cell in the `words` array where the next word should be stored. The function actually assigns the next word and increments `nwords` in one expression:

```
words[nwords++] = p;
```

You should convince yourself that this arrangement works, and that (in this case) the preincrement form

```
words[++nwords] = p;        /* WRONG */
```

would *not* behave as desired.

When the function is done (when it finds the `\0` terminating the input line, or when it runs out of cells in the `words` array) it returns the number of words it has found.

Here is a complete example of calling `getwords`:

```
char line[] = "this is a test";
int i;

nwords = getwords(line, words, 10);
for(i = 0; i < nwords; i++)
    printf("%s\n", words[i]);
```

Chapter 11
Memory Allocation

In this chapter, we'll meet `malloc`, C's dynamic memory allocation function, and we'll cover dynamic memory allocation in some detail.

As we begin doing dynamic memory allocation, we'll begin to see (if we haven't seen it already) what pointers can really be good for. Many of the pointer examples in the previous chapter (those which used pointers to access arrays) didn't do all that much for us that we couldn't have done using arrays. However, when we begin doing dynamic memory allocation, pointers are the only way to go, because what `malloc` returns is a pointer to the memory it gives us. (Due to the equivalence between pointers and arrays, though, we will still be able to think of dynamically allocated regions of storage as if they were arrays, and even to use array-like subscripting notation on them.)

You have to be careful with dynamic memory allocation. `malloc` operates at a pretty "low level"; you will often find yourself having to do a certain amount of work to manage the memory it gives you. If you don't keep accurate track of the memory which `malloc` has given you, and the pointers of yours which point to it, it's all too easy to accidentally use a pointer which points "nowhere", with generally unpleasant results. (The basic problem is that if you assign a value to the location pointed to by a pointer:

```
*p = 0;
```

and if the pointer `p` points "nowhere", well actually it can be construed to point somewhere, just not where you wanted it to, and that "somewhere" is where the 0 gets written. If the "somewhere" is memory which is in use by some other part of your program, or even worse, if the operating system has not protected itself from you and "somewhere" is in fact in use by the operating system, things could get ugly.)

11.1 Allocating Memory with `malloc`

[This section corresponds to parts of K&R Secs. 5.4, 5.6, 6.5, and 7.8.5]

A problem with many simple programs, including in particular little teaching programs such as we've been writing so far, is that they tend to use fixed-size arrays which may or may not be big enough. We have an array of 100 `int`s for the numbers which the user enters and wishes to find the average of–what if the user enters 101 numbers? We have an array of 100 `char`s which we pass to `getline` to receive the user's input–what if the user types a line of 200 characters? If we're lucky, the relevant parts of the program check how much of an array they've used, and print an error message or otherwise gracefully abort before overflowing the array. If we're not so lucky, a program may sail off the end of an array, overwriting other data and behaving quite badly. In either case, the user doesn't get his job done. How can we avoid the restrictions of fixed-size arrays?

The answers all involve the standard library function `malloc`. Very simply, `malloc` returns a pointer to n bytes of memory which we can do anything we want to with. If we didn't want to read a line of input into a fixed-size array, we could use `malloc`, instead. Here's the first step:

```
#include <stdlib.h>

char *line;
int linelen = 100;
line = malloc(linelen);
/* incomplete -- malloc's return value not checked */
getline(line, linelen);
```

`malloc` is declared in `<stdlib.h>`, so we `#include` that header in any program that calls `malloc`. A "byte" in C is, by definition, an amount of storage suitable for storing one character, so the above invocation of `malloc` gives us exactly as many `char`s as we ask for. We could illustrate the resulting pointer like this:

The 100 bytes of memory (not all of which are shown) pointed to by `line` are those allocated by `malloc`. (They are brand-new memory, conceptually a bit different from the memory which the compiler arranges to have allocated automatically for our conventional variables. The 100 boxes in the figure don't have a name next to them, because they're not storage for a variable we've declared.)

As a second example, we might have occasion to allocate a piece of memory, and to copy a string into it with `strcpy`:

```
char *p = malloc(15);
/* incomplete -- malloc's return value not checked */
strcpy(p, "Hello, world!");
```

When copying strings, remember that all strings have a terminating `\0` character. If you use `strlen` to count the characters in a string for you, that count will *not* include the trailing `\0`, so you must add one before calling `malloc`:

```
char *somestring, *copy;
...
copy = malloc(strlen(somestring) + 1);        /* +1 for \0 */
/* incomplete -- malloc's return value not checked */
strcpy(copy, somestring);
```

What if we're not allocating characters, but integers? If we want to allocate 100 `int`s, how many bytes is that? If we know how big `int`s are on our machine (i.e. depending on whether we're using a 16- or 32-bit machine) we could try to compute it ourselves, but it's much safer and more portable to let C compute it for us. C has a `sizeof` operator, which computes the size, in bytes, of a variable or type. It's just what we need when calling `malloc`. To allocate space for 100 `int`s, we could call

```
int *ip = malloc(100 * sizeof(int));
```

The use of the `sizeof` operator tends to look like a function call, but it's really an operator, and it does its work at compile time.

Since we can use array indexing syntax on pointers, we can treat a pointer variable after a call to `malloc` almost exactly as if it were an array. In particular, after the above call to `malloc` initializes `ip` to point at storage for 100 `int`s, we can access `ip[0]`, `ip[1]`, ... up to `ip[99]`. This way, we can get the effect of an array even if we don't know until run time how big the "array" should be. (In a later section we'll see how we might deal with the case where we're not even sure at the point we begin using it how big an "array" will eventually have to be.)

Our examples so far have all had a significant omission: they have not checked `malloc`'s return value. Obviously, no real computer has an infinite amount of memory available, so there is no guarantee that `malloc` will be able to give us as much memory as we ask for. If we call `malloc(100000000)`, or if we call `malloc(10)` 10,000,000 times, we're probably going to run out of memory.

When `malloc` is unable to allocate the requested memory, it returns a null pointer. A null pointer, remember, points definitively nowhere. It's a "not a pointer" marker; it's not a pointer you can use. (As we said in section 9.4, a null pointer can be used as a failure return from a function that returns pointers, and `malloc` is a perfect example.) Therefore, whenever you call `malloc`, it's vital to check the returned pointer before using it! If you call `malloc`, and it returns a null pointer, and you go off and use that null pointer as if it pointed somewhere, your program probably won't last long. Instead, a program should immediately check for a null pointer, and if it receives one, it should at the very least print an error message and exit, or perhaps figure out some way of proceeding without the memory it asked for. But it cannot go on to use the null pointer it got back from `malloc` in any way, because that null pointer by definition points nowhere. ("It cannot use a null pointer in any way" means that the program cannot use the `*` or `[]` operators on such a pointer value, or pass it to any function that expects a valid pointer.)

A call to `malloc`, with an error check, typically looks something like this:

```
int *ip = malloc(100 * sizeof(int));
if(ip == NULL)
    {
    printf("out of memory\n");
    exit or return
    }
```

After printing the error message, this code should return to its caller, or exit from the program entirely; it cannot proceed with the code that would have used `ip`.

Of course, in our examples so far, we've still limited ourselves to "fixed size" regions of memory, because we've been calling `malloc` with fixed arguments like 10 or 100. (Our call to `getline` is still limited to 100-character lines, or whatever number we set the `linelen` variable to; our `ip` variable still points at only 100 `int`s.) However, since the sizes are now values which can in principle be determined at run-time, we've at least moved beyond having to recompile the program (with a bigger array) to accommodate longer lines, and with a little more work, we could arrange that the "arrays" automatically grew to be as large as required. (For example, we could write something like `getline` which could read the longest input line actually seen.) We'll begin to explore this possibility in a later section.

11.2 Freeing Memory

Memory allocated with `malloc` lasts as long as you want it to. It does not automatically disappear when a function returns, as automatic-duration variables do, but it does not have to remain for the entire duration of your program, either. Just as you can use `malloc` to control exactly when and how much memory you allocate, you can also control exactly when you deallocate it.

In fact, many programs use memory on a transient basis. They allocate some memory, use it for a while, but then reach a point where they don't need that particular piece any more. Because memory is not inexhaustible, it's a good idea to deallocate (that is, release or free) memory you're no longer using.

Dynamically allocated memory is deallocated with the `free` function. If `p` contains a pointer previously returned by `malloc`, you can call

```
free(p);
```

which will "give the memory back" to the stock of memory (sometimes called the "arena" or "pool") from which `malloc` requests are satisfied. Calling `free` is sort of the ultimate in recycling: it costs you almost nothing, and the memory you give back is immediately usable by other parts of your program. (Theoretically, it may even be usable by other programs.)

(Freeing unused memory is a good idea, but it's not mandatory. When your program exits, any memory which it has allocated but not freed should be automatically released. If your computer were to somehow "lose" memory just because your program forgot to free it, that would indicate a problem or deficiency in your operating system.)

Naturally, once you've freed some memory you must remember not to use it any more. After calling

```
free(p);
```

it is probably the case that `p` still points at the same memory. However, since we've given it back, it's now "available," and a later call to `malloc` might give that memory to some other part of your program. If the variable `p` is a global variable or will otherwise stick around for a while, one good way to record the fact that it's not to be used any more would be to set it to a null pointer:

```
free(p);
p = NULL;
```

Now we don't even have the pointer to the freed memory any more, and (as long as we check to see that `p` is non-`NULL` before using it), we won't misuse any memory via the pointer `p`.

When thinking about `malloc`, `free`, and dynamically-allocated memory in general, remember again the distinction between a pointer and what it points to. If you call `malloc` to allocate some memory, and store the pointer which `malloc` gives you in a local pointer variable, what happens when the function containing the local pointer variable returns? If the local pointer variable has automatic duration (which is the default, unless the variable is declared `static`), it will disappear when the function returns. But for the pointer variable to disappear says nothing about the memory pointed to! That memory still exists and, as far as `malloc` and `free` are concerned, is still allocated. The only thing that has disappeared is the pointer variable you had which pointed at the allocated memory. (Furthermore, if it contained the only copy of the pointer you had, once it disappears, you'll have no way of freeing the memory, and no way of using it, either. Using memory and freeing memory both require that you have at least one pointer to the memory!)

11.3 Reallocating Memory Blocks

Sometimes you're not sure at first how much memory you'll need. For example, if you need to store a series of items you read from the user, and if the only way to know how many there are is to read them until the user types some "end" signal, you'll have no way of knowing, as you begin reading and storing the first few, how many you'll have seen by the time you do see that "end" marker. You might want to allocate room for, say, 100 items, and if the user enters a 101st item before entering the "end" marker, you might wish for a way to say "uh, `malloc`, remember those 100 items I asked for? Could I change my mind and have 200 instead?"

In fact, you can do exactly this, with the `realloc` function. You hand `realloc` an old pointer (such as you received from an initial call to `malloc`) and a new size, and `realloc` does what it can to give you a chunk of memory big enough to hold the new size. For example, if we wanted the `ip` variable from an earlier example to point at 200 `ints` instead of 100, we could try calling

```
ip = realloc(ip, 200 * sizeof(int));
```

Since you always want each block of dynamically-allocated memory to be contiguous (so that you can treat it as if it were an array), you and `realloc` have to worry about the case where `realloc` can't make the old block of memory bigger "in place," but rather has to relocate it elsewhere in order to find enough contiguous space for the new requested size. `realloc` does this by returning a new pointer. If `realloc` was able to make the old block of memory bigger, it returns the same pointer. If `realloc` has to go elsewhere to get enough contiguous memory, it returns a pointer to the new memory, after copying your old data there. (In this case, after it makes the copy, it frees the old block.) Finally, if `realloc` can't find enough memory to satisfy the new request at all, it returns a null pointer. Therefore, you usually don't want to overwrite your old pointer with `realloc`'s return value until you've tested it to make sure it's not a null pointer. You might use code like this:

```
int *newp;
newp = realloc(ip, 200 * sizeof(int));
if(newp != NULL)
    ip = newp;
else    {
    printf("out of memory\n");
    /* exit or return */
    /* but ip still points at 100 ints */
    }
```

If `realloc` returns something other than a null pointer, it succeeded, and we set `ip` to what it returned. (We've either set `ip` to what it used to be or to a new pointer, but in either case, it points to where our data is now.) If `realloc` returns a null pointer, however, we hang on to our old pointer in `ip` which still points at our original 100 values.

Putting this all together, here is a piece of code which reads lines of text from the user, treats each line as an integer by calling `atoi`, and stores each integer in a dynamically-allocated "array":

```
#define MAXLINE 100

char line[MAXLINE];
int *ip;
int nalloc, nitems;

nalloc = 100;
ip = malloc(nalloc * sizeof(int));        /* initial allocation */
if(ip == NULL)
    {
    printf("out of memory\n");
    exit(1);
    }

nitems = 0;
```

```
while(getline(line, MAXLINE) != EOF)
    {
    if(nitems >= nalloc)
        {               /* increase allocation */
        int *newp;
        nalloc += 100;
        newp = realloc(ip, nalloc * sizeof(int));
        if(newp == NULL)
            {
            printf("out of memory\n");
            exit(1);
            }
        ip = newp;
        }
    ip[nitems++] = atoi(line);
    }
```

We use two different variables to keep track of the "array" pointed to by `ip`. `nalloc` is now many elements we've allocated, and `nitems` is how many of them are in use. Whenever we're about to store another item in the "array," if `nitems >= nalloc`, the old "array" is full, and it's time to call `realloc` to make it bigger.

Finally, we might ask what the return type of `malloc` and `realloc` is, if they are able to return pointers to `char` or pointers to `int` or (though we haven't seen it yet) pointers to any other type. The answer is that both of these functions are declared (in `<stdlib.h>`) as returning a type we haven't seen, `void *` (that is, pointer to `void`). We haven't really seen type `void`, either, but what's going on here is that `void *` is specially defined as a "generic" pointer type, which may be used (strictly speaking, assigned to or from) any pointer type.

11.4 Pointer Safety

At the beginning of the previous chapter, we said that the hard thing about pointers is not so much manipulating them as ensuring that the memory they point to is valid. When a pointer doesn't point where you think it does, if you inadvertently access or modify the memory it points to, you can damage other parts of your program, or (in some cases) other programs or the operating system itself!

When we use pointers to simple variables, as in section 10.1, there's not much that can go wrong. When we use pointers into arrays, as in section 10.2, and begin moving the pointers around, we have to be more careful, to ensure that the roving pointers always stay within the bounds of the array(s). When we begin passing pointers to functions, and especially when we begin returning them from functions (as in the `strstr` function of section 10.4) we have to be more careful still, because the code using the pointer may be far removed from the code which owns or allocated the memory.

One particular problem concerns functions that return pointers. Where is the memory to which the returned pointer points? Is it still around by the time the function returns? The `strstr` function returns either a null pointer (which points definitively nowhere, and which the caller presumably checks for) or it returns a pointer which points into the input string, which the caller supplied, which is pretty safe. One thing a function must *not* do, however, is return a pointer to one of its own, local, automatic-duration arrays. Remember that automatic-duration variables (which includes all non-static local variables), including

automatic-duration arrays, are deallocated and disappear when the function returns. If a function returns a pointer to a local array, that pointer will be invalid by the time the caller tries to use it.

Finally, when we're doing dynamic memory allocation with `malloc`, `realloc`, and `free`, we have to be most careful of all. Dynamic allocation gives us a lot more flexibility in how our programs use memory, although with that flexibility comes the responsibility that we manage dynamically allocated memory carefully. The possibilities for misdirected pointers and associated mayhem are greatest in programs that make heavy use of dynamic memory allocation. You can reduce these possibilities by designing your program in such a way that it's easy to ensure that pointers are used correctly and that memory is always allocated and deallocated correctly. (If, on the other hand, your program is designed in such a way that meeting these guarantees is a tedious nuisance, sooner or later you'll forget or neglect to, and maintenance will be a nightmare.)

Chapter 12
Input and Output

So far, we've been calling `printf` to print formatted output to the "standard output" (wherever that is). We've also been calling `getchar` to read single characters from the "standard input," and `putchar` to write single characters to the standard output. "Standard input" and "standard output" are two predefined I/O streams which are implicitly available to us. In this chapter we'll learn how to take control of input and output by opening our own streams, perhaps connected to data files, which we can read from and write to.

12.1 File Pointers and `fopen`

[This section corresponds to K&R Sec. 7.5]

How will we specify that we want to access a particular data file? It would theoretically be possible to mention the name of a file each time it was desired to read from or write to it. But such an approach would have a number of drawbacks. Instead, the usual approach (and the one taken in C's stdio library) is that you mention the name of the file once, at the time you open it. Thereafter, you use some little token–in this case, the file pointer–which keeps track (both for your sake and the library's) of which file you're talking about. Whenever you want to read from or write to one of the files you're working with, you identify that file by using its file pointer (that is, the file pointer you obtained when you opened the file). As we'll see, you store file pointers in variables just as you store any other data you manipulate, so it is possible to have several files open, as long as you use distinct variables to store the file pointers.

You declare a variable to store a file pointer like this:

```
FILE *fp;
```

The type `FILE` is predefined for you by `<stdio.h>`. It is a data structure which holds the information the standard I/O library needs to keep track of the file for you. For historical reasons, you declare a variable which is a pointer to this `FILE` type. The name of the variable can (as for any variable) be anything you choose; it is traditional to use the letters `fp` in the variable name (since we're talking about a `f`ile `p`ointer). If you were reading from two files at once you'd probably use two file pointers:

```
FILE *fp1, *fp2;
```

If you were reading from one file and writing to another you might declare and input file pointer and an output file pointer:

```
FILE *ifp, *ofp;
```

Like any pointer variable, a file pointer isn't any good until it's initialized to point to something. (Actually, no variable of any type is much good until you've initialized it.) To actually open a file, and receive the "token" which you'll store in your file pointer variable, you call `fopen`. `fopen` accepts a file name (as a string) and a mode value indicating among other things whether you intend to read or write this file. (The mode variable is also a string.) To open the file `input.dat` for reading you might call

```
ifp = fopen("input.dat", "r");
```

The mode string `"r"` indicates reading. Mode `"w"` indicates writing, so we could open `output.dat` for output like this:

```
ofp = fopen("output.dat", "w");
```

The other values for the mode string are less frequently used. The third major mode is `"a"` for append. (If you use `"w"` to write to a file which already exists, its old contents will be discarded.) You may also add a `+` character to the mode string to indicate that you want to both read and write, or a `b` character to indicate that you want to do "binary" (as opposed to text) I/O.

One thing to beware of when opening files is that it's an operation which may fail. The requested file might not exist, or it might be protected against reading or writing. (These possibilities ought to be obvious, but it's easy to forget them.) `fopen` returns a null pointer if it can't open the requested file, and it's important to check for this case before going off and using `fopen`'s return value as a file pointer. Every call to `fopen` will typically be followed with a test, like this:

```
ifp = fopen("input.dat", "r");
if(ifp == NULL)
    {
    printf("can't open file\n");
    exit or return
    }
```

If `fopen` returns a null pointer, and you store it in your file pointer variable and go off and try to do I/O with it, your program will typically crash.

It's common to collapse the call to `fopen` and the assignment in with the test:

```
if((ifp = fopen("input.dat", "r")) == NULL)
    {
    printf("can't open file\n");
    exit or return
    }
```

You don't have to write these "collapsed" tests if you're not comfortable with them, but you'll see them in other people's code, so you should be able to read them.

12.2 I/O with File Pointers

For each of the I/O library functions we've been using so far, there's a companion function which accepts an additional file pointer argument telling it where to read from or write to. The companion function to `printf` is `fprintf`, and the file pointer argument comes first. To print a string to the `output.dat` file we opened in the previous section, we might call

```
fprintf(ofp, "Hello, world!\n");
```

The companion function to `getchar` is `getc`, and the file pointer is its only argument. To read a character from the `input.dat` file we opened in the previous section, we might call

```
int c;
c = getc(ifp);
```

The companion function to `putchar` is `putc`, and the file pointer argument comes last. To write a character to `output.dat`, we could call

```
    putc(c, ofp);
```

Our own `getline` function calls `getchar` and so always reads the standard input. We could write a companion `fgetline` function which reads from an arbitrary file pointer:

```
#include <stdio.h>

/* Read one line from fp, */
/* 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 fgetline(FILE *fp, char line[], int max)
{
int nch = 0;
int c;
max = max - 1;            /* leave room for '\0' */

while((c = getc(fp)) != 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;
}
```

Now we could read one line from `ifp` by calling

```
    char line[MAXLINE];
    ...
    fgetline(ifp, line, MAXLINE);
```

12.3 Predefined Streams

Besides the file pointers which we explicitly open by calling `fopen`, there are also three predefined streams. `stdin` is a constant file pointer corresponding to standard input, and `stdout` is a constant file pointer corresponding to standard output. Both of these can be used anywhere a file pointer is called for; for example, `getchar()` is the same as `getc(stdin)` and `putchar(c)` is the same as `putc(c, stdout)`. The third predefined stream is `stderr`. Like `stdout`, `stderr` is typically connected to the screen by default. The difference is that `stderr` is not redirected when the standard output is redirected. For example, under Unix or MS-DOS, when you invoke

```
    program > filename
```

anything printed to `stdout` is redirected to the file `filename`, but anything printed to `stderr` still goes to the screen. The intent behind `stderr` is that it is the "standard error output"; error messages printed to it will not disappear into an output file. For example, a more realistic way to print an error message when a file can't be opened would be

```
if((ifp = fopen(filename, "r")) == NULL)
    {
    fprintf(stderr, "can't open file %s\n", filename);
    exit or return
    }
```

where `filename` is a string variable indicating the file name to be opened. Not only is the error message printed to `stderr`, but it is also more informative in that it mentions the name of the file that couldn't be opened. (We'll see another example in the next chapter.)

12.4 Closing Files

Although you can open multiple files, there's a limit to how many you can have open at once. If your program will open many files in succession, you'll want to close each one as you're done with it; otherwise the standard I/O library could run out of the resources it uses to keep track of open files. Closing a file simply involves calling `fclose` with the file pointer as its argument:

```
fclose(fp);
```

Calling `fclose` arranges that (if the file was open for output) any last, buffered output is finally written to the file, and that those resources used by the operating system (and the C library) for this file are released. If you forget to close a file, it will be closed automatically when the program exits.

12.5 Example: Reading a Data File

Suppose you had a data file consisting of rows and columns of numbers:

```
1    2    34
5    6    78
9    10    112
```

Suppose you wanted to read these numbers into an array. (Actually, the array will be an array of arrays, or a "multidimensional" array; see section 4.1.2.) We can write code to do this by putting together several pieces: the `fgetline` function we just showed, and the `getwords` function from chapter 10. Assuming that the data file is named `input.dat`, the code would look like this:

```
#define MAXLINE 100
#define MAXROWS 10
#define MAXCOLS 10

int array[MAXROWS][MAXCOLS];
char *filename = "input.dat";
FILE *ifp;
char line[MAXLINE];
char *words[MAXCOLS];
int nrows = 0;
int n;
int i;
```

```
ifp = fopen(filename, "r");
if(ifp == NULL)
    {
    fprintf(stderr, "can't open %s\n", filename);
    exit(EXIT_FAILURE);
    }
while(fgetline(ifp, line, MAXLINE) != EOF)
    {
    if(nrows >= MAXROWS)
        {
        fprintf(stderr, "too many rows\n");
        exit(EXIT_FAILURE);
        }
    n = getwords(line, words, MAXCOLS);
    for(i = 0; i < n; i++)
        array[nrows][i] = atoi(words[i]);
    nrows++;
    }
```

Each trip through the loop reads one line from the file, using `fgetline`. Each line is broken up into "words" using `getwords`; each "word" is actually one number. The numbers are however still represented as strings, so each one is converted to an `int` by calling `atoi` before being stored in the array. The code checks for two different error conditions (failure to open the input file, and too many lines in the input file) and if one of these conditions occurs, it prints an error message, and exits. The `exit` function is a Standard library function which terminates your program. It is declared in `<stdlib.h>`, and accepts one argument, which will be the exit status of the program. `EXIT_FAILURE` is a code, also defined by `<stdlib.h>`, which indicates that the program failed. Success is indicated by a code of `EXIT_SUCCESS`, or simply 0. (These values can also be returned from `main()`; calling `exit` with a particular status value is essentially equivalent to returning that same status value from `main`.)

Chapter 13
Reading the Command Line

[This section corresponds to K&R Sec. 5.10]

We've mentioned several times that a program is rarely useful if it does exactly the same thing every time you run it. Another way of giving a program some variable input to work on is by invoking it with command line arguments.

(We should probably admit that command line user interfaces are a bit old-fashioned, and currently somewhat out of favor. If you've used Unix or MS-DOS, you know what a command line is, but if your experience is confined to the Macintosh or Microsoft Windows or some other Graphical User Interface, you may never have seen a command line. In fact, if you're learning C on a Mac or under Windows, it can be tricky to give your program a command line at all. Think C for the Macintosh provides a way; I'm not sure about other compilers. If your compilation environment doesn't provide an easy way of simulating an old-fashioned command line, you may skip this chapter.)

C's model of the command line is that it consists of a sequence of words, typically separated by whitespace. Your main program can receive these words as an array of strings, one word per string. In fact, the C run-time startup code is always willing to pass you this array, and all you have to do to receive it is to declare `main` as accepting two parameters, like this:

```
int main(int argc, char *argv[])
{
...
}
```

When `main` is called, `argc` will be a count of the number of command-line arguments, and `argv` will be an array ("vector") of the arguments themselves. Since each word is a string which is represented as a pointer-to-`char`, `argv` is an array-of-pointers-to-`char`. Since we are not defining the `argv` array, but merely declaring a parameter which references an array somewhere else (namely, in `main`'s caller, the run-time startup code), we do not have to supply an array dimension for `argv`. (Actually, since functions never receive arrays as parameters in C, `argv` can also be thought of as a pointer-to-pointer-to-`char`, or `char **`. But multidimensional arrays and pointers to pointers can be confusing, and we haven't covered them, so we'll talk about `argv` as if it were an array.) (Also, there's nothing magic about the names `argc` and `argv`. You can give `main`'s two parameters any names you like, as long as they have the appropriate types. The names `argc` and `argv` are traditional.)

The first program to write when playing with `argc` and `argv` is one which simply prints its arguments:

```
#include <stdio.h>

main(int argc, char *argv[])
{
int i;
```

```
for(i = 0; i < argc; i++)
    printf("arg %d: %s\n", i, argv[i]);
return 0;
}
```

(This program is essentially the Unix or MS-DOS `echo` command.)

If you run this program, you'll discover that the set of "words" making up the command line includes the command you typed to invoke your program (that is, the name of your program). In other words, `argv[0]` typically points to the name of your program, and `argv[1]` is the first argument.

There are no hard-and-fast rules for how a program should interpret its command line. There is one set of conventions for Unix, another for MS-DOS, another for VMS. Typically you'll loop over the arguments, perhaps treating some as option flags and others as actual arguments (input files, etc.), interpreting or acting on each one. Since each argument is a string, you'll have to use `strcmp` or the like to match arguments against any patterns you might be looking for. Remember that `argc` contains the number of words on the command line, and that `argv[0]` is the command name, so if `argc` is 1, there are no arguments to inspect. (You'll never want to look at `argv[i]`, for `i >= argc`, because it will be a null or invalid pointer.)

As another example, also illustrating `fopen` and the file I/O techniques of the previous chapter, here is a program which copies one or more input files to its standard output. Since "standard output" is usually the screen by default, this is therefore a useful program for displaying files. (It's analogous to the obscurely-named Unix `cat` command, and to the MS-DOS `type` command.) You might also want to compare this program to the character-copying program of section 6.2.

```
#include <stdio.h>

main(int argc, char *argv[])
{
int i;
FILE *fp;
int c;

for(i = 1; i < argc; i++)
    {
    fp = fopen(argv[i], "r");
    if(fp == NULL)
        {
        fprintf(stderr, "cat: can't open %s\n", argv[i]);
        continue;
        }

    while((c = getc(fp)) != EOF)
        putchar(c);

    fclose(fp);
    }

return 0;
}
```

As a historical note, the Unix `cat` program is so named because it can be used to con**cat**enate two files together, like this:

```
cat a b > c
```

This illustrates why it's a good idea to print error messages to `stderr`, so that they don't get redirected. The "can't open file" message in this example also includes the name of the program as well as the name of the file.

Yet another piece of information which it's usually appropriate to include in error messages is the reason why the operation failed, if known. For operating system problems, such as inability to open a file, a code indicating the error is often stored in the global variable `errno`. The standard library function `strerror` will convert an `errno` value to a human-readable error message string. Therefore, an even more informative error message printout would be

```
fp = fopen(argv[i], "r");
if(fp == NULL)
    fprintf(stderr, "cat: can't open %s: %s\n",
            argv[i], strerror(errno));
```

If you use code like this, you can `#include <errno.h>` to get the declaration for `errno`, and `<string.h>` to get the declaration for `strerror()`.

Chapter 14
What's Next?

This last handout contains a brief list of the significant topics in C which we have not covered, and which you'll want to investigate further if you want to know all of C.

Types and Declarations

We have not talked about the `void`, `short int`, and `long double` types. `void` is a type with no values, used as a placeholder to indicate functions that do not return values or that accept no arguments, and in the "generic" pointer type `void *` that can point to anything. `short int` is an integer type that might use less space than a plain `int`; `long double` is a floating-point type that might have even more range or precision than plain `double`.

The `char` type and the various sizes of `int` also have "unsigned" versions, which are declared using the keyword `unsigned`. Unsigned types cannot hold negative values but have guaranteed properties on overflow. (Whether a plain `char` is signed or unsigned is implementation-defined; you can use the keyword `signed` to force a character type to contain signed characters.) Unsigned types are also useful when manipulating individual bits and bytes, when "sign extension" might otherwise be a problem.

Two additional type qualifiers `const` and `volatile` allow you to declare variables (or pointers to data) which you promise not to change, or which might change in unexpected ways behind the program's back.

There are user-defined structure and union types. A structure or `struct` is a "record" consisting of one or more values of one or more types concreted together into one entity which can be manipulated as a whole. A `union` is a type which, at any one time, can hold a value from one of a specified set of types.

There are user-defined enumeration types ("`enum`") which are like integers but which always contain values from some fixed, predefined set, and for which the values are referred to by name instead of by number.

Pointers can point to functions as well as to data types.

Types can be arbitrarily complicated, when you start using multiple levels of pointers, arrays, functions, structures, and/or unions. Eventually, it's important to understand the concept of a declarator: in the declaration

```
int i, *ip, *fpi();
```

we have the base type `int` and three declarators `i`, `*ip`, and `*fpi()`. The declarator gives the name of a variable (or function) and also indicates whether it is a simple variable or a pointer, array, function, or some more elaborate combination (array of pointers, function returning pointer, etc.). In the example, `i` is declared to be a plain `int`, `ip` is declared to be a pointer to `int`, and `fpi` is declared to be a function returning pointer to `int`. (Complicated declarators may also contain parentheses for grouping, since there's a precedence hierarchy in declarators as well as expressions: `[]` for arrays and `()` for functions have higher precedence than `*` for pointers.)

We have not said much about pointers to pointers, or arrays of arrays (i.e. multidimensional arrays), or the ramifications of array/pointer equivalence on multidimensional arrays. (In particular, a reference to an array of arrays does *not* generate a pointer to a pointer; it generates a pointer to an array. You cannot pass a multidimensional array to a function which accepts pointers to pointers.)

Variables can be declared with a hint that they be placed in high-speed CPU registers, for efficiency. (These hints are rarely needed or used today, because modern compilers do a good job of register allocation by themselves, without hints.)

A mechanism called `typedef` allows you to define user-defined aliases (i.e. new and perhaps more-convenient names) for other types.

Operators

The bitwise operators `&`, `|`, `^`, and `~` operate on integers thought of as binary numbers or strings of bits. The `&` operator is bitwise AND, the `|` operator is bitwise OR, the `^` operator is bitwise exclusive-OR (XOR), and the `~` operator is a bitwise negation or complement. (`&`, `|`, and `^` are "binary" in that they take two operands; `~` is unary.) These operators let you work with the individual bits of a variable; one common use is to treat an integer as a set of single-bit flags. You might define the 3rd (2**2) bit as the "verbose" flag bit by defining

```
#define VERBOSE 4
```

Then you can "turn the verbose bit on" in an integer variable `flags` by executing

```
flags = flags | VERBOSE;
```

or

```
flags |= VERBOSE;
```

and turn it off with

```
flags = flags & ~VERBOSE;
```

or

```
flags &= ~VERBOSE;
```

and test whether it's set with

```
if(flags & VERBOSE)
```

The left-shift and right-shift operators `<<` and `>>` let you shift an integer left or right by some number of bit positions; for example, `value << 2` shifts `value` left by two bits.

The `?:` or conditional operator (also called the "ternary operator") essentially lets you embed an `if/then` statement in an expression. The assignment

```
a = expr ? b : c;
```

is roughly equivalent to

```
if(expr)
    a = b;
else    a = c;
```

Since you can use `?:` anywhere in an expression, it can do things that `if/then` can't, or that would be cumbersome with `if/then`. For example, the function call

```
f(a, b, c ? d : e);
```

is roughly equivalent to

```
if(c)
    f(a, b, d);
else    f(a, b, e);
```

(Exercise: what would the call

```
g(a, b, c ? d : e, h ? i : j, k);
```

be equivalent to?)

The comma operator lets you put two separate expressions where one is required; the expressions are executed one after the other. The most common use for comma operators is when you want multiple variables controlling a `for` loop, for example:

```
for(i = 0, j = 10; i < j; i++, j--)
```

A cast operator allows you to explicitly force conversion of a value from one type to another. A cast consists of a type name in parentheses. For example, you could convert an `int` to a `double` by typing

```
int i = 10;
double d;
d = (double)i;
```

(In this case, though, the cast is redundant, since this is a conversion that C would have performed for you automatically, i.e. if you'd just said `d = i` .) You use explicit casts in those circumstances where C does not do a needed conversion automatically. One example is division: if you're dividing two integers and you want a floating-point result, you must explicitly force at least one of the operands to floating-point, otherwise C will perform an integer division and will discard the remainder. The code

```
int i = 1, j = 2;
double d = i / j;
```

will set `d` to 0, but

```
d = (double)i / j;
```

will set `d` to 0.5. You can also "cast to `void`" to explicitly indicate that you're ignoring a function's return value, as in

```
(void)fclose(fp);
```

or

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

(Usually, it's a bad idea to ignore return values, but in some cases it's essentially inevitable, and the `(void)` cast keeps some compilers from issuing warnings every time you ignore a value.)

There's a precise, mildly elaborate set of rules which C uses for converting values automatically, in the absence of explicit casts.

The `.` and `->` operators let you access the members (components) of structures and unions.

Statements

The `switch` statement allows you to jump to one of a number of numeric `case` labels depending on the value of an expression; it's more convenient than a long `if/else` chain. (However, you can use `switch` only when the expression is integral and all of the `case` labels are compile-time constants.)

The `do/while` loop is a loop that tests its controlling expression at the bottom of the loop, so that the body of the loop always executes once even if the condition is initially false. (C's `do/while` loop is therefore like Pascal's `repeat/until` loop, while C's `while` loop is like Pascal's `while/do` loop.)

Finally, when you really need to write "spaghetti code," C does have the all-purpose `goto` statement, and labels to go to.

Functions

Functions can't return arrays, and it's tricky to write a function as if it returns an array (perhaps by simulating the array with a pointer) because you have to be careful about allocating the memory that the returned pointer points to.

The functions we've written have all accepted a well-defined, fixed number of arguments. `printf` accepts a variable number of arguments (depending on how many `%` signs there are in the format string) but we haven't seen how to declare and write functions that do this.

C Preprocessor

If you're careful, it's possible (and can be useful) to use `#include` within a header file, so that you end up with "nested header files."

It's possible to use `#define` to define "function-like" macros that accept arguments; the expansion of the macro can therefore depend on the arguments it's "invoked" with.

Two special preprocessing operators `#` and `##` let you control the expansion of macro arguments in fancier ways.

The preprocessor directive `#if` lets you conditionally include (or, with `#else`, conditionally not include) a section of code depending on some arbitrary compile-time expression. (`#if` can also do the same macro-definedness tests as `#ifdef` and `#ifndef`, because the expression can use a `defined()` operator.)

Other preprocessing directives are `#elif`, `#error`, `#line`, and `#pragma`.

There are a few predefined preprocessor macros, some required by the C standard, others perhaps defined by particular compilation environments. These are useful for conditional compilation (`#ifdef`, `#ifndef`).

Standard Library Functions

C's standard library contains many features and functions which we haven't seen.

We've seen many of `printf`'s formatting capabilities, but not all. Besides format specifier characters for a few types we haven't seen, you can also control the width, precision, justification (left or right) and a few other attributes of `printf`'s format conversions. (In their full complexity, `printf` formats are about as elaborate and powerful as FORTRAN `format` statements.)

A `scanf` function lets you do "formatted input" analogous to `printf`'s formatted output. `scanf` reads from the standard input; a variant `fscanf` reads from a specified file pointer.

The `sprintf` and `sscanf` functions let you "print" and "read" to and from in-memory strings instead of files. We've seen that `atoi` lets you convert a numeric string into an integer; the inverse operation can be performed with `sprintf`:

```
int i = 10;
char str[10];
sprintf(str, "%d", i);
```

We've used `printf` and `fprintf` to write formatted output, and `getchar`, `getc`, `putchar`, and `putc` to read and write characters. There are also functions `gets`, `fgets`, `puts`, and `fputs` for reading and writing lines (though we rarely need these, especially if we're using our own `getline` and maybe `fgetline`), and also `fread` and `fwrite` for reading or writing arbitrary numbers of characters.

It's possible to "un-read" a character, that is, to push it back on an input stream, with `ungetc`. (This is useful if you accidentally read one character too far, and would prefer that some other part of your program read that character instead.)

You can use the `ftell`, `fseek`, and `rewind` functions to jump around in files, performing random access (as opposed to sequential) I/O.

The `feof` and `ferror` functions will tell you whether you got `EOF` due to an actual end-of-file condition or due to a read error of some sort. You can clear errors and end-of-file conditions with `clearerr`.

You can open files in "binary" mode, or for simultaneous reading and writing. (These options involve extra characters appended to `fopen`'s mode string: `b` for binary, `+` for read/write.)

There are several more string functions in `<string.h>`. A second set of string functions `strncpy`, `strncat`, and `strncmp` all accept a third argument telling them to stop after `n` characters if they haven't found the `\0` marking the end of the string. A third set of "`mem`" functions, including `memcpy` and `memcmp`, operate on blocks of memory which aren't necessarily strings and where `\0` is not treated as a terminator. The `strchr` and `strrchr` functions find characters in strings. There is a motley collection of "span" and "scan" functions, `strspn`, `strcspn`, and `strpbrk`, for searching out or skipping over sequences of characters all drawn from a specified set of characters. The `strtok` function aids in breaking up a string into words or "tokens," much like our own `getwords` function.

The header file `<ctype.h>` contains several functions which let you classify and manipulate characters: check for letters or digits, convert between upper- and lower-case, etc.

A host of mathematical functions are defined in the header file `<math.h>`. (As we've mentioned, besides including `<math.h>`, you may on some Unix systems have to ask for a special library containing the math functions while compiling/linking.)

There's a random-number generator, `rand`, and a way to "seed" it, `srand`. `rand` returns integers from 0 up to `RAND_MAX` (where `RAND_MAX` is a constant `#defined` in `<stdlib.h>`). One way of getting random integers from 1 to `n` is to call

```
(int)(rand() / (RAND_MAX + 1.0) * n) + 1
```

Another way is

```
rand() / (RAND_MAX / n + 1) + 1
```

It seems like it would be simpler to just say

```
rand() % n + 1
```

but this method is imperfect (or rather, it's imperfect if `n` is a power of two and your system's implementation of `rand()` is imperfect, as all too many of them are).

Several functions let you interact with the operating system under which your program is running. The `exit` function returns control to the operating system immediately, terminating your program and returning an "exit status." The `getenv` function allows you to read your operating system's or process's "environment variables" (if any). The `system` function allows you to invoke an operating-system command (i.e. another program) from within your program.

The `qsort` function allows you to sort an array (of any type); you supply a comparison function (via a function pointer) which knows how to compare two array elements, and `qsort` does the rest. The `bsearch` function allows you to search for elements in sorted arrays; it, too, operates in terms of a caller-supplied comparison function.

Several functions—`time`, `asctime`, `gmtime`, `localtime`, `asctime`, `mktime`, `difftime`, and `strftime`—allow you to determine the current date and time, print dates and times, and perform other date/time manipulations. For example, to print today's date in a program, you can write

```
#include <time.h>

time_t now;
now = time((time_t *)NULL);
printf("It's %.24s", ctime(&now));
```

The header file `<stdarg.h>` lets you manipulate variable-length function argument lists (such as the ones `printf` is called with). Additional members of the `printf` family of functions let you write your own functions which accept `printf`-like format specifiers and variable numbers of arguments but call on the standard `printf` to do most of the work.

There are facilities for dealing with multibyte and "wide" characters and strings, for use with multinational character sets.