The C Programming Language

July 25, 2025

Contents

Chapter 1

A Tutorial Introduction

Let us begin with a quick introduction in C. Our aim is to show the essential elements of the language in real programs, but without getting bogged down in details, rules, and exceptions. At this point, we are not trying to be complete or even precise (save that the examples are meant to be correct). We want to get you as quickly as possible to the point where you can write useful programs, and to do that we have to concentrate on the basics: variables and constants, arithmetic, control flow, functions, and the rudiments of input and output. We are intentionally leaving out of this chapter features of C that are important for writing bigger programs. These include pointers, structures, most of C's rich set of operators, several control-flow statements, and the standard library.

This approach has its drawbacks. Most notable is that the complete story on any particular feature is not found here, and the tutorial, by being brief, may also be misleading. And because the examples do not use the full power of C, they are not as concise and elegant as they might be. We have tried to minimize these effects, but be warned. Another drawback is that later chapters will necessarily repeat some of this chapter. We hope that the repetition will help you more than it annoys.

In any case, experienced programmers should be able to extrapolate from the material in this chapter to their own programming needs. Beginners should supplement it by writing small, similar programs of their own. Both groups can use it as a framework on which to hang the more detailed descriptions that begin in Chapter 2.

1.1 Getting Started

The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:

```
Print the words
hello, world
```

This is a big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy.

In C, the program to print "hello, world" is

```
#include <stdio.h>

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

Just how to run this program depends on the system you are using. As a specific example, on the UNIX operating system you must create the program in a file whose name ends in "`.c`", such as `hello.c`, then compile it with the command

```
cc hello.c
```

If you haven't botched anything, such as omitting a character or misspelling something, the compilation will proceed silently, and make an executable file called `a.out`. If you run `a.out` by typing the command

```
a.out
```

it will print

```
hello, world
```

On other systems, the rules will be different; check with a local expert.

Now, for some explanations about the program itself. A C program, whatever its size, consists of functions and variables. A function contains statements that specify the computing operations to be done, and variables store values used during the computation. C functions are like the subroutines and functions in Fortran or the procedures and functions of Pascal. Our example is a function named `main`. Normally you are at liberty to give functions whatever names you like, but "`main`" is special - your program begins executing at the beginning of `main`. This means that every program must have a `main` somewhere.

`main` will usually call other functions to help perform its job, some that you wrote, and others from libraries that are provided for you. The first line of the program,

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

tells the compiler to include information about the standard input/output library; the line appears at the beginning of many C source files. The standard library is described in Chapter 7 and Appendix B.

One method of communicating data between functions is for the calling function to provide a list of values, called arguments, to the function it calls. The parentheses after the function name surround the argument list. In this example, `main` is defined to be a function that expects no arguments, which is indicated by the empty list `()`.

The statements of a function are enclosed in braces . The function `main` contains only one statement,

```
printf("hello, world\n");
```

A function is called by naming it, followed by a parenthesized list of arguments, so this calls the function `printf` with the argument `"hello, world\n"`. `printf` is a library function that prints output, in this case the string of characters between the quotes.

A sequence of characters in double quotes, like `"hello, world\n"`, is called a character string or string constant. For the moment our only use of character strings will be as arguments for `printf` and other functions.

The sequence `\n` in the string is C notation for the newline character, which when printed advances the output to the left margin on the next line. If you leave out the `\n` (a worthwhile experiment), you will find that there is no line advance after the output is printed. You must use `\n` to include a newline character in the `printf` argument; if you try something like

```
printf("hello, world
");
```

the C compiler will produce an error message `printf` never supplies a newline character automatically, so several calls may be used to build up an output line in stages. Our first program could just as well have been written

```
#include <stdio.h>

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

to produce identical output.

Notice that `\n` represents only a single character. An escape sequence like `\n` provides a general and extensible mechanism for representing hard-to-type or invisible characters. Among the others that C provides are `\t` for tab, `\b` for backspace, `\"` for the double quote and `\\` for the backslash itself. There is a complete list in Section 2.3.

Exercise 1.1 — Run the "hello, world" program on your system. Experiment with leaving out parts of the program, to see what error messages you get.

Exercise 1.2 — Experiment to find out what happens when `prints`'s argument string contains `\c`, where `c` is some character not listed above.

1.2 Variables and Arithmetic Expressions

The next program uses the formula `°C=(5/9)(°F-32)` to print the following table of Fahrenheit temperatures and their centigrade or Celsius equivalents:

```
1   -17
20  -6
40  4
60  15
80  26
100 37
120 48
140 60
160 71
180 82
200 93
```

```
220 104
240 115
260 126
280 137
300 148
```

The program itself still consists of the definition of a single function named `main`. It is longer than the one that printed `"hello, world"`, but not complicated. It introduces several new ideas, including comments, declarations, variables, arithmetic expressions, loops , and formatted output.

```
#include <stdio.h>

/* print Fahrenheit-Celsius table
for fahr = 0, 20, ..., 300 */

main()
{
    int fahr, celsius;
    int lower, upper, step;
    lower = 0;   /* lower limit of temperature scale */
    upper = 300; /* upper limit */
    step = 20;   /* step size */
    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

The two lines

```
/* print Fahrenheit-Celsius table
for fahr = 0, 20, ..., 300 */
```

are a comment, which in this case explains briefly what the program does. Any characters between `/*` and `*/` are ignored by the compiler; they may be used freely to make a program easier to understand. Comments may appear anywhere where a blank, tab or newline can. In C, all variables must be declared before they are used, usually at the beginning of the function before any executable statements. A declaration announces the properties of variables; it consists of a name and a list of variables, such as

```
int fahr, celsius;
int lower, upper, step;
```

The type int means that the variables listed are integers; by contrast with float, which means floating point, i.e., numbers that may have a fractional part. The range of both `int` and `float` depends on the machine you are using; 16-bits `ints`, which lie between -32768 and +32767, are common, as are 32-bit `ints`. A float number is typically a 32-bit quantity, with at least six significant digits and magnitude generally between about 10^{-38} and 10^38.

C provides several other data types besides `int` and `float`, including:

```
char    character - a single byte
short   short integer
long    long integer
double  double-precision floating point
```

The size of these objects is also machine-dependent. There are also arrays, structures and unions of these basic types, pointers to them, and functions that return them, all of which we will meet in due course.

Computation in the temperature conversion program begins with the assignment statements

```
lower = 0;
upper = 300;
step = 20;
```

which set the variables to their initial values. Individual statements are terminated by semicolons.

Each line of the table is computed the same way, so we use a loop that repeats once per output line; this is the purpose of the while loop

```
while (fahr <= upper) {
    ...
}
```

The `while` loop operates as follows: The condition in parentheses is tested. If it is true (fahr is less than or equal to upper), the body of the loop (the three statements enclosed in braces) is executed. Then the condition is re-tested, and if true, the body is executed again. When the test becomes false (fahr exceeds upper) the loop ends, and execution continues at the statement that follows the loop. There are no further statements in this program, so it terminates.

The body of a while can be one or more statements enclosed in braces, as in the temperature converter, or a single statement without braces, as in

```
while (i < j)
    i = 2 * i;
```

In either case, we will always indent the statements controlled by the while by one tab stop (which we have shown as four spaces) so you can see at a glance which statements are inside the loop. The indentation emphasizes the logical structure of the program. 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.

Most of the work gets done in the body of the loop. The Celsius temperature is computed and assigned to the variable celsius by the statement

```
celsius = 5 * (fahr-32) / 9;
```

The reason for multiplying by `5` and dividing by `9` instead of just multiplying by `5/9` is that in C, as in many other languages, integer division truncates: any fractional part is discarded. Since `5` and `9` are integers. `5/9` would be truncated to zero and so all the Celsius temperatures would be reported as zero.

This example also shows a bit more of how `printf` works. `printf` is a general-purpose output formatting function, which we will describe in detail in Chapter 7. Its first argument is a string of characters to be printed, with each `%` indicating where one of the other (second, third, ...) arguments is to be substituted, and in what form it is to be printed. For instance, `%d` specifies an integer argument, so the statement

```
printf("%d\t%d\n", fahr, celsius);
```

causes the values of the two integers fahr and celsius to be printed, with a tab (`\t`) between them.

Each `%` construction in the first argument of `printf` is paired with the corresponding second argument, third argument, etc.; they must match up properly by number and type, or you will get wrong answers.

By the way, `printf` is not part of the C language; there is no input or output defined in C itself. `printf` is just a useful function from the standard library of functions that are normally accessible to C programs. The behaviour of `printf` is defined in the ANSI standard, however, so its properties should be the same with any compiler and library that conforms to the standard.

In order to concentrate on C itself, we don't talk much about input and output until chapter 7. In particular, we will defer formatted input until then. If you have to input numbers, read the discussion of the function `scanf` in Section 7.4. `scanf` is like `printf`, except that it reads input instead of writing output.

There are a couple of problems with the temperature conversion program. The simpler one is that the output isn't very pretty because the numbers are not right-justified. That's easy to fix; if we augment each `%d` in the `printf` statement with a width, the numbers printed will be right-justified in their fields.

For instance, we might say

```
printf("%3d %6d\n", fahr, celsius);
```

to print the first number of each line in a field three digits wide, and the second in a field six digits wide, like this:

```
0   -17
20   -6
40    4
60   15
80   26
100  37
...
```

The more serious problem is that because we have used integer arithmetic, the Celsius temperatures are not very accurate; for instance, `0F` is actually about `-17.8C`, not `-17`. To get more accurate answers, we should use floating-point arithmetic instead of integer. This requires some changes in the program. Here is the second version:

```
#include <stdio.h>
/* print Fahrenheit-Celsius table
for fahr = 0, 20, ..., 300; floating-point version */
main()
```

```
{
    float fahr, celsius;
    float lower, upper, step;
    lower = 0;   /* lower limit of temperatuire scale */
    upper = 300; /* upper limit */
    step = 20;   /* step size */
    fahr = lower;
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%3.0f %6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

This is much the same as before, except that fahr and celsius are declared to be `float` and the formula for conversion is written in a more natural way. We were unable to use 5/9 in the previous version because integer division would truncate it to zero. A decimal point in a constant indicates that it is floating point, however, so `5.0/9.0` is not truncated because it is the ratio of two floating-point values.

If an arithmetic operator has integer operands, an integer operation is performed. If an arithmetic operator has one floating-point operand and one integer operand, however, the integer will be converted to floating point before the operation is done. If we had written `(fahr-32)`, the 32 would be automatically converted to floating point. Nevertheless, writing floating-point constants with explicit decimal points even when they have integral values emphasizes their floating-point nature for human readers.

The detailed rules for when integers are converted to floating point are in Chapter 2. For now, notice that the assignment

```
fahr = lower;
```

and the test

```
while (fahr <= upper)
```

also work in the natural way – the `int` is converted to `float` before the operation is done.

The `printf` conversion specification `%3.0f` says that a floating-point number (here fahr) is to be printed at least three characters wide, with no decimal point and no fraction digits. `%6.1f` describes another number (celsius) that is to be printed at least six characters wide, with `1` digit after the decimal point. The output looks like this:

```
0   -17.8
20   -6.7
40    4.4
...
```

Width and precision may be omitted from a specification: `%6f` says that the number is to be at least six characters wide; `%.2f` specifies two characters after the decimal point, but the width is not constrained; and `%f` merely says to print the number as floating point.

Among others, `printf` also recognizes `%o` for octal, `%x` for hexadecimal, `%c` for character, `%s` for character string and `%%` for itself

Exercise 1.3 — Modify the temperature conversion program to print a heading above the table.

Exercise 1.4 — Write a program to print the corresponding Celsius to Fahrenheit table.

1.3 The for statement

There are plenty of different ways to write a program for a particular task. Let's try a variation on the temperature converter.

```
#include <stdio.h>

/* print Fahrenheit-Celsius table */
main()
{
    int fahr;
    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

This produces the same answers, but it certainly looks different. One major change is the elimination of most of the variables; only fahr remains, and we have made it an `int`. The lower and upper limits and the step size appear only as constants in the `for` statement, itself a new construction, and the expression that computes the Celsius temperature now appears as the third argument of printf instead of a separate assignment statement.

This last change is an instance of a general rule – in any context where it is permissible to use the value of some type, you can use a more complicated expression of that type. Since the third argument of `printf` must be a floating-point value to match the `%6.1f`, any floating-point expression can occur here.

The `for` statement is a loop, a generalization of the `while`. If you compare it to the earlier `while`, its operation should be clear. Within the parentheses, there are three parts, separated by semicolons. The first part, the initialization

```
fahr = 0
```

is done once, before the loop proper is entered. The second part is the test or condition that controls the loop:

```
fahr <= 300
```

This condition is evaluated; if it is true, the body of the loop (here a single `printf`) is executed. Then the increment step

```
fahr = fahr + 20
```

is executed, and the condition re-evaluated. The loop terminates if the condition has become false. As with the `while`, the body of the loop can be a single statement or a group of statements enclosed in braces. The initialization, condition and increment can be any expressions.

The choice between `while` and `for` is arbitrary, based on which seems clearer. The `for` is usually appropriate `for` loops in which the initialization and increment are single statements and logically related, since it is more compact than `while` and it keeps the loop control statements together in one place.

Exercise 1.5 — Modify the temperature conversion program to print the table in reverse order, that is, from 300 degrees to 0.

1.4 Symbolic Constants

A final observation before we leave temperature conversion forever. It's bad practice to bury "magic numbers" like 300 and 20 in a program; they convey little information to someone who might have to read the program later, and they are hard to change in a systematic way.

One way to deal with magic numbers is to give them meaningful names. A `#define` line defines a symbolic name or symbolic constant to be a particular string of characters:

```
#define name replacement list
```

Thereafter, any occurrence of name (not in quotes and not part of another name) will be replaced by the corresponding replacement text. The name has the same form as a variable name: a sequence of letters and digits that begins with a letter. The replacement text can be any sequence of characters; it is not limited to numbers.

```
#include <stdio.h>

#define LOWER 0 /* lower limit of table */
#define UPPER 300 /* upper limit */
#define STEP 20 /* step size */

/* print Fahrenheit-Celsius table */
main()
{
    int fahr;
    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

The quantities `LOWER`, `UPPER` and `STEP` are symbolic constants, not variables, so they do not appear in declarations. Symbolic constant names are conventionally written in upper case so they can ber readily distinguished from lower case variable names. Notice that there is no semicolon at the end of a `#define` line.

1.5 Character Input and Output

We are going to consider a family of related programs for processing character data. You will find that many programs are just expanded versions of the prototypes that we discuss here.

The model of input and output supported by the standard library is very simple. Text input or output, regardless of where it originates or where it goes to, is dealt with as streams of characters. A text stream is a sequence of characters divided into lines; each line consists of zero or more characters followed by a newline character. It is the responsibility of the library to make each input or output stream confirm this model; the C programmer using the library need not worry about how lines are represented outside the program.

The standard library provides several functions for reading or writing one character at a time, of which `getchar` and `putchar` are the simplest. Each time it is called, `getchar` reads the next input character from a text stream and returns that as its value. That is, after `c = getchar();` the variable `c` contains the next character of input. The characters normally come from the keyboard; input from files is discussed in Chapter 7. The function `putchar` prints a character each time it is called: `putchar(c);` prints the contents of the integer variable c as a character, usually on the screen. Calls to `putchar` and `printf` may be interleaved; the output will appear in the order in which the calls are made.

1.5.1 File Copying

Given `getchar` and `putchar`, you can write a surprising amount of useful code without knowing anything more about input and output. The simplest example is a program that copies its input to its output one character at a time:

```
read a character
while (charater is not end-of-file indicator)
    output the character just read
    read a character
```

Converting this into C gives:

```
#include <stdio.h>
/* copy input to output; 1st version */
main()
{
    int c;
    c = getchar();
    while (c != EOF) {
        putchar(c);
```

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

The relational operator `!=` means "not equal to".

What appears to be a character on the keyboard or screen is of course, like everything else, stored internally just as a bit pattern. The type char is specifically meant for storing such character data, but any integer type can be used. We used int for a subtle but important reason.

The problem is distinguishing the end of input from valid data. The solution is that `getchar` returns a distinctive value when there is no more input, a value that cannot be confused with any real character. This value is called `EOF`, for "end of file". We must declare `c` to be a type big enough to hold any value that `getchar` returns. We can't use char since `c` must be big enough to hold `EOF` in addition to any possible char. Therefore we use int.

`EOF` is an integer defined in `<stdio.h>`, but the specific numeric value doesn't matter as long as it is not the same as any char value. By using the symbolic constant, we are assured that nothing in the program depends on the specific numeric value.

The program for copying would be written more concisely by experienced C programmers. In C, any assignment, such as

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

is an expression and has a value, which is the value of the left hand side after the assignment. This means that a assignment can appear as part of a larger expression. If the assignment of a character to `c` is put inside the test part of a while loop, the copy program can be written this way:

```
    #include <stdio.h>

    /* copy input to output; 2nd version */
    main()
    {
        int c;
        while ((c = getchar()) != EOF)
            putchar(c);
    }
```

The while gets a character, assigns it to `c`, and then tests whether the character was the end-of-file signal. If it was not, the body of the while is executed, printing the character. The `while` then repeats. When the end of the input is finally reached, the `while` terminates and so does `main`.

This version centralizes the input – there is now only one reference to `getchar` – and shrinks the program. The resulting program is more compact, and, once the idiom is mastered, easier to read. You'll see this style often. (It's possible to get carried away and create impenetrable code, however, a tendency that we will try to curb.)

The parentheses around the assignment, within the condition are necessary. The precedence of `!=` is higher than that of `=`, which means that in the absence of parentheses the relational test `!=` would be done before the assignment `=`. So the statement

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

is equivalent to

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

This has the undesired effect of setting `c` to `0` or `1`, depending on whether or not the call of getchar returned end of file. (More on this in Chapter 2.)

Exercise 1.6 — Verify that the expression `getchar() != EOF` is `0` or `1`.

Exercise 1.7 — Write a program to print the value of `EOF`.

1.5.2 Character Counting

The next program counts characters; it is similar to the copy program.

```
#include <stdio.h>

/* count characters in input; 1st version */
main()
{
    long nc;
    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

The statement

```
    ++nc;
```

presents a new operator, `++`, which means *increment by one*. You could instead write `nc = nc + 1` but `++nc` is more concise and often more efficient. There is a corresponding operator `--` to decrement by 1. The operators `++` and `--` can be either prefix operators (`++nc`) or postfix operators (`nc++`); these two forms have different values in expressions, as will be shown in Chapter 2, but `++nc` and `nc++` both increment `nc`. For the moment we will will stick to the prefix form.

The character counting program accumulates its count in a long variable instead of an `int`. `long` integers are at least 32 bits. Although on some machines, `int` and `long` are the same size, on others an `int` is 16 bits, with a maximum value of 32767, and it would take relatively little input to overflow an `int` counter. The conversion specification `%ld` tells `printf` that the corresponding argument is a `long` integer.

It may be possible to cope with even bigger numbers by using a `double` (double precision `float`). We will also use a `for` statement instead of a `while`, to illustrate another way to write the loop.

```
#include <stdio.h>

/* count characters in input; 2nd version */
main()
{
    double nc;
    for (nc = 0; getchar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

`printf` uses `%f` for both `float` and `double`; `%.0f` suppresses the printing of the decimal point and the fraction part, which is zero.

The body of this for loop is empty, because all the work is done in the test and increment parts. But the grammatical rules of C require that a `for` statement have a body. The isolated semicolon, called a `null statement`, is there to satisfy that requirement. We put it on a separate line to make it visible.

Before we leave the character counting program, observe that if the input contains no characters, the `while` or `for` test fails on the very first call to `getchar`, and the program produces zero, the right answer. This is important. One of the nice things about `while` and `for` is that they test at the top of the loop, before proceeding with the body. If there is nothing to do, nothing is done, even if that means never going through the loop body. Programs should act intelligently when given zero-length input. The `while` and `for` statements help ensure that programs do reasonable things with boundary conditions.

1.5.3 Line Counting

The next program counts input lines. As we mentioned above, the standard library ensures that an input text stream appears as a sequence of lines, each terminated by a newline. Hence, counting lines is just counting newlines:

```
#include <stdio.h>

/* count lines in input */
main()
{
    int c, nl;
    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

The body of the `while` now consists of an `if`, which in turn controls the increment `++nl`. The `if` statement tests the parenthesized condition, and `if` the condition is true, executes the statement (or group of statements in braces) that follows. We have again indented to show what is controlled by what.

The double equals sign `==` is the C notation for “is equal to” (like Pascal’s single `=` or Fortran’s `.EQ.`). This symbol is used to distinguish the equality test from the single `=` that C uses for assignment. A word of caution: newcomers to C occasionally write `=` when they mean `==`. As we will see in Chapter 2, the result is usually a legal expression, so you will get no warning.

A character written between single quotes represents an integer value equal to the numerical value of the character in the machine's character set. This is called a *character constant*, although it is just another way to write a small integer. So, for example, `'A'` is a character constant; in the ASCII character set its value is 65, the internal representation of the character A. Of course, `'A'` is to be preferred over 65: its meaning is obvious, and it is independent of a particular character set.

The escape sequences used in string constants are also legal in character constants, so `'\n'` stands for the value of the newline character, which is 10 in ASCII. You should note carefully that `'\n'` is a single character, and in expressions is just an integer; on the other hand, `'\n'` is a string constant that happens to contain only one character. The topic of strings versus characters is discussed further in Chapter 2.

Exercise 1.8 — Write a program to count blanks, tabs, and newlines.

Exercise 1.9 — Write a program to copy its input to its output, replacing each string of one or more blanks by a single blank.

Exercise 1.10 — Write a program to copy its input to its output, replacing each tab by `\t`, each backspace by `\b`, and each backslash by `\\`. This makes tabs and backspaces visible in an unambiguous way.

1.5.4 Word Counting

The fourth in our series of useful programs counts lines, words, and characters, with the loose definition that a word is any sequence of characters that does not contain a blank, tab or newline. This is a bare-bones version of the UNIX program `wc`.

```
#include <stdio.h>
#define IN 1 /* inside a word */
#define OUT 0 /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;
    state = OUT;
    nl = nw = nc = 0;
```

```
        while ((c = getchar()) != EOF) {
            ++nc;
            if (c == '\n')
                ++nl;
            if (c == ' ' || c == '\n' || c == '\t')
                state = OUT;
            else if (state == OUT) {
                state = IN;
                ++nw;
            }
        }
        printf("%d %d %d\n", nl, nw, nc);
    }
```

Every time the program encounters the first character of a word, it counts one more word. The variable state records whether the program is currently in a word or not; initially it is "not in a word", which is assigned the value `OUT`. We prefer the symbolic constants `IN` and `OUT` to the literal values `1` and `0` because they make the program more readable. In a program as tiny as this, it makes little difference, but in larger programs, the increase in clarity is well worth the modest extra effort to write it this way from the beginning. You'll also find that it's easier to make extensive changes in programs where magic numbers appear only as symbolic constants.

The line

```
    nl = nw = nc = 0;
```

sets all three variables to zero. This is not a special case, but a consequence of the fact that an assignment is an expression with the value and assignments associated from right to left. It's as if we had written

```
    nl = (nw = (nc = 0));
```

The operator `||` means OR, so the line

```
    if (c == ' ' || c == '\n' || c = '\t')
```

says "if `c` is a blank or `c` is a newline or `c` is a tab". (Recall that the escape sequence `\t` is a visible representation of the tab character.) There is a corresponding operator `&&` for AND; its precedence is just higher than `||`. Expressions connected by `&&` or `||` are evaluated left to right, and it is guaranteed that evaluation will stop as soon as the truth or falsehood is known. If `c` is a blank, there is no need to test whether it is a newline or tab, so these tests are not made. This isn't particularly important here, but is significant in more complicated situations, as we will soon see. The example also shows an else, which specifies an alternative action if the condition part of an if statement is false. The general form is

```
if (expression)
    statement1
else
    statement2
```

One and only one of the two statements associated with an `if-else` is performed. If the expression is true, statement1 is executed; if not, statement2 is executed. Each statement can be a single statement or several in braces. In the word count program, the one after the `else` is an `if` that controls two statements in braces.

Exercise 1.11 — How would you test the word count program? What kinds of input are most likely to uncover bugs if there are any?

Exercise 1.12 — Write a program that prints its input one word per line.

1.6 Arrays

Let us write a program to count the number of occurrences of each digit, of white space characters (blank, tab, newline), and of all other characters. This is artificial, but it permits us to illustrate several aspects of C in one program. There are twelve categories of input, so it is convenient to use an array to hold the number of occurrences of each digit, rather than ten individual variables. Here is one version of the program:

```
#include <stdio.h>

/* count digits, white space, others */
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];
```

```
        nwhite = nother = 0;
        for (i = 0; i < 10; ++i)
            ndigit[i] = 0;
        while ((c = getchar()) != EOF)
            if (c >= '0' && c <= '9')
                ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;

        printf("digits =");
        for (i = 0; i < 10; ++i)
            printf(" %d", ndigit[i]);
        printf(", white space = %d, other = %d\n", nwhite, ↩
   nother);
    }
```

The output of this program on itself is

```
    digits = 9 3 0 0 0 0 0 0 0 1, white space = 123, other = ↩
   345
```

The declaration

```
int ndigit[10];
```

declares `ndigit` to be an array of `10` integers. Array subscripts always start at zero in C, so the elements are `ndigit[0]`, `ndigit[1]`, ..., `ndigit[9]`. This is reflected in the for loops that initialize and print the array.

A subscript can be any integer expression, which includes integer variables like `i`, and integer constants.

This particular program relies on the properties of the character representation of the digits. For example, the test

```
if (c >= '0' && c <= '9')
```

determines whether the character in `c` is a digit. If it is, the numeric value of that digit is

```
c - '0'
```

This works only if `'0'`, `'1'`, ..., `'9'` have consecutive increasing values. Fortunately, this is true for all character sets.

By definition, `char`s are just small integers, so `char` variables and constants are identical to `int`s in arithmetic expressions. This is natural and convenient; for example `c - '0'` is an integer expression with a value between 0 and 9 corresponding to the character `'0'` to `'9'` stored in `c`, and thus a valid subscript for the array `ndigit`.

The decision as to whether a character is a digit, white space, or something else is made with the sequence

```
if (c >= '0' && c <= '9')
    ++ndigit[c-'0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++nwhite;
else
    ++nother;
```

The pattern

```
if (condition1)
    statement1
else if (condition2)
    statement2
...
...
else
    statementn
```

occurs frequently in programs as a way to express a multi-way decision.

The conditions are evaluated in order from the top until some condition is satisfied; at that point the corresponding statement part is executed, and the entire construction is finished. (Any statement can be several statements enclosed in braces.) If none of the conditions is satisfied, the statement after the final else is executed if it is present. If the final else and statement are omitted, as in the word count program, no action takes place. There can be any number of

```
else if(condition)
    statement
```

groups between the initial `if` and the final `else`.

As a matter of style, it is advisable to format this construction as we have shown; if each `if` were indented past the previous `else`, a long sequence of decisions would march off the right side of the page. The `switch` statement, to be discussed in Chapter 4, provides another way to write a multiway branch that is particulary suitable when the condition is whether some integer or character expression matches one of a set of constants. For contrast, we will present a `switch` version of this program in Section 3.4.

Exercise 1.13 — Write a program to print a histogram of the lengths of words in its input. It is easy to draw the histogram with the bars horizontal; a vertical orientation is more challenging.

Exercise 1.14 — Write a program to print a histogram of the frequencies of different characters in its input.

1.7 Functions

In C, a function is equivalent to a subroutine or function in Fortran, or a procedure or function in Pascal. A function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. With properly designed functions, it is possible to ignore how a job is done; knowing what is done is sufficient. C makes the use of functions easy, convinient and efficient; you will often see a short function defined and called only once, just because it clarifies some piece of code.

So far we have used only functions like `printf`, `getchar` and `putchar` that have been provided for us; now it's time to write a few of our own. Since C has no exponentiation operator like the `**` of Fortran, let us illustrate the mechanics of function definition by writing a function `power(m,n)` to raise an integer `m` to a positive integer power `n`. That is, the value of `power(2,5)` is `32`. This function is not a practical exponentiation routine, since it handles only positive powers of small integers, but it's good enough for illustration. (The standard library contains a function `pow(x,y)` that computes `x ** y`.)

Here is the function `power` and a `main` program to exercise it, so you can see the whole structure at once.

```
#include <stdio.h>

int power(int m, int n);

/* test power function */
main()
```

```
{
    int i;
    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0;
}

/* power: raise base to n-th power; n >= 0 */
int power(int base, int n)
{
    int i, p;
    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

A function definition has this form:

```
return-type function-name(parameter declarations, if ↩
any)
{
    declarations
    statements
}
```

Function definitions can appear in any order, and in one source file or several, although no function can be split between files. If the source program appears in several files, you may have to say more to compile and load it than if it all appears in one, but that is an operating system matter, not a language attribute. For the moment, we will assume that both functions are in the same file, so whatever you have learned about running C programs will still work.

The function `power` is called twice by main, in the line

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

Each call passes two arguments to power, which each time returns an integer to be formatted and printed. In an expression, `power(2,i)` is an integer just as `2` and `i` are. (Not all functions produce an integer value; we will take this up in Chapter 4.)

The first line of power itself,

```
int power(int base, int n)
```

declares the parameter types and names, and the type of the result that the function returns.

The names used by power for its parameters are local to power, and are not visible to any other function: other routines can use the same names without conflict. This is also true of the variables `i` and `p`: the `i` in power is unrelated to the `i` in `main`.

We will generally use *parameter* for a variable named in the parenthesized list in a function. The terms *formal argument* and *actual argument* are sometimes used for the same distinction.

The value that power computes is returned to `main` by the `return` statement. Any expression may follow return:

```
return expression;
```

A function need not return a value; a return statement with no expression causes control, but no useful value, to be returned to the caller, as does "falling off the end" of a function by reaching the terminating right brace. And the calling function can ignore a value returned by a function.

You may have noticed that there is a `return` statement at the end of `main`. Since `main` is a function like any other, it may return a value to its caller, which is in effect the environment in which the program was executed. Typically, a return value of zero implies normal termination; non-zero values signal unusual or erroneous termination conditions.

In the interests of simplicity, we have omitted `return` statements from our `main` functions up to this point, but we will include them hereafter, as a reminder that programs should return status to their environment.

The declaration

```
int power(int base, int n);
```

just before `main` says that `power` is a function that expects two int arguments and returns an `int`.

This declaration, which is called a *function prototype*, has to agree with the definition and uses of `power`. It is an error if the definition of a function or any uses of it do not agree with its prototype.

Parameter names need not agree. Indeed, parameter names are optional in a function prototype, so for the prototype we could have written

```
int power(int, int);
```

Well-chosen names are good documentation however, so we will often use them.

A note of history: the biggest change between ANSI C and earlier versions is how functions are declared and defined. In the original definition of C, the power function would have been written like this:

```
/* power: raise base to n-th power; n >= 0 */
/* (old-style version) */
power(base, n)
int base, n;
{
    int i, p;
    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

The parameters are named between the parentheses, and their types are declared before opening the left brace; undeclared parameters are taken as `int`. (The body of the function is the same as before.) The declaration of power at the beginning of the program would have looked like this:

```
int power();
```

No parameter list was permitted, so the compiler could not readily check that `power` was being called correctly. Indeed, since by default `power` would have been assumed to return an `int`, the entire declaration might well have been omitted. The new syntax of function prototypes makes it much easier for a compiler to detect errors in the number of arguments or their types. The old style of declaration and definition still works in ANSI C, at least for a transition period, but we strongly recommend that you use the new form when you have a compiler that supports it.

Exercise 1.15 — Rewrite the temperature conversion program of Section 1.2 to use a function for conversion.

1.8 Arguments – Call by Value

One aspect of C functions may be unfamiliar to programmers who are used to some other languages, particulary Fortran. In C, all function arguments are passed "by value". This means that the called function is given the values of its arguments in temporary variables rather than the originals. This leads to some different properties than are seen with "call by reference" languages like Fortran or with var parameters in Pascal, in which the called routine has access to the original argument, not a local copy.

Call by value is an asset, however, not a liability. It usually leads to more compact programs with fewer extraneous variables, because parameters can be treated as conveniently initialized local variables in the called routine. For example, here is a version of power that makes use of this property.

```
/* power: raise base to n-th power; n >= 0; version 2 */
int power(int base, int n)
{
    int p;
    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

The parameter `n` is used as a temporary variable, and is counted down (a for loop that runs backwards) until it becomes zero; there is no longer a need for the variable `i`. Whatever is done to `n` inside power has no effect on the argument that power was originally called with.

When necessary, it is possible to arrange for a function to modify a variable in a calling routine. The caller must provide the address of the variable to be set (technically a pointer to the variable), and the called function must declare the parameter to be a pointer and access the variable indirectly through it. We will cover pointers in Chapter 5.

The story is different for arrays. When the name of an array is used as an argument, the value passed to the function is the location or address of the beginning of the array – there is no copying of array elements. By subscripting this value, the function can access and alter any argument of the array. This is the topic of the next section.

1.9 Character Arrays

The most common type of array in C is the array of characters. To illustrate the use of character arrays and functions to manipulate them, let's write a program that reads a set of text lines and prints the longest. The outline is simple enough:

```
while (there's another line)
    if (it's longer than the previous longest)
        (save it)
        (save its length)
print longest line
```

This outline makes it clear that the program divides naturally into pieces. One piece gets a new line, another saves it, and the rest controls the process.

Since things divide so nicely, it would be well to write them that way too. Accordingly, let us first write a separate function `getline` to fetch the next line of input. We will try to make the function useful in other contexts. At the minimum, `getline` has to return a signal about possible end of file; a more useful design would be to return the length of the line, or zero if end of file is encountered. Zero is an acceptable end-of-file return because it is never a valid line length. Every text line has at least one character; even a line containing only a newline has length 1.

When we find a line that is longer than the previous longest line, it must be saved somewhere. This suggests a second function, copy, to copy the new line to a safe place. Finally, we need a `main` program to control `getline` and copy. Here is the result.

```
#include <stdio.h>
#define MAXLINE 1000 /* maximum input line length */

int getline(char line[], int maxline);
void copy(char to[], char from[]);

/* print the longest input line */
main()
{
    int len;                    /* current line length */
    int max;                /* maximum length seen so far */
```

```
        char line[MAXLINE];         /* current input line */
        char longest[MAXLINE]; /* longest line saved here */
        max = 0;
        while ((len = getline(line, MAXLINE)) > 0)
            if (len > max) {
            max = len;
            copy(longest, line);
            }
        if (max > 0)     /* there was a line */
            printf("%s", longest);
        return 0;
    }

    /* getline: read a line into s, return length */
    int getline(char s[],int lim)
    {
        int c, i;
        for (i=0; i < lim-1 && (c=getchar())!=EOF && c!='\n';↩
    ++i)
        s[i] = c;
        if (c == '\n') {
            s[i] = c;
            ++i;
        }
        s[i] = '\0';
        return i;
    }

    /* copy: copy from into to; assume to is big enough */
    void copy(char to[], char from[])
    {
        int i;
        i = 0;
        while ((to[i] = from[i]) != '\0')
            ++i;
    }
```

The functions `getline` and copy are declared at the beginning of the program, which we assume is contained in one file. `main` and `getline` communicate through a pair of arguments and a returned value. In `getline`, the arguments are declared by the line

```
int getline(char s[], int lim);
```

which specifies that the first argument, `s`, is an array, and the second, `lim`, is an integer.

The purpose of supplying the size of an array in a declaration is to set aside storage. The length of an array `s` is not necessary in `getline` since its size is set in `main`. `getline` uses return to send a value back to the caller, just as the function `power` did. This line also declares that `getline` returns an `int`; since `int` is the default return type, it could be omitted.

Some functions return a useful value; others, like `copy`, are used only for their effect and return no value. The return type of `copy` is `void`, which states explicitly that no value is returned. `getline` puts the character `'\0'` (the null character, whose value is zero) at the end of the array it is creating, to mark the end of the string of characters. This conversion is also used by the C language: when a string constant like

```
"hello\n"
```

appears in a C program, it is stored as an array of characters containing the characters in the string and terminated with a `'\0'` to mark the end.

The `%s` format specification in `printf` expects the corresponding argument to be a string represented in this form. `copy` also relies on the fact that its input argument is terminated with a `'\0'`, and copies this character into the output.

It is worth mentioning in passing that even a program as small as this one presents some sticky design problems. For example, what should `main` do if it encounters a line which is bigger than its limit? `getline` works safely, in that it stops collecting when the array is full, even if no newline has been seen. By testing the length and the last character returned, `main` can determine whether the line was too long, and then cope as it wishes. In the interests of brevity, we have ignored this issue.

There is no way for a user of `getline` to know in advance how long an input line might be, so `getline` checks for overflow. On the other hand, the user of `copy` already knows (or can find out) how big the strings are, so we have chosen not to add error checking to it.

Exercise 1.16 — Revise the `main` routine of the longest-line program so it will correctly print the length of arbitrary long input lines, and as much as possible of the text.

Exercise 1.17 — Write a program to print all input lines that are longer than 80 characters.

Exercise 1.18 — Write a program to remove trailing blanks and tabs from each line of input, and to delete entirely blank lines.

Exercise 1.19 — Write a function `reverse(s)` that reverses the character string `s`. Use it to write a program that reverses its input a line at a time.

1.10 External Variables and Scope

Each local variable in a function comes into existence only when the function is called, and disappears when the function is exited. This is why such variables are usually known as automatic variables, following terminology in other languages. We will use the term *automatic* henceforth to refer to these local variables. Chapter 4 discusses the `static` storage class, in which local variables do retain their values between calls.

Because automatic variables come and go with function invocation, they do not retain their values from one call to the next, and must be explicitly set upon each entry. If they are not set, they will contain garbage.

As an alternative to automatic variables, it is possible to define variables that are external to all functions, that is, variables that can be accessed by name by any function. (This mechanism is rather like Fortran `COMMON` or Pascal variables declared in the outermost block.) Because external variables are globally accessible, they can be used instead of argument lists to communicate data between functions. Furthermore, because external variables remain in existence permanently, rather than appearing and disappearing as functions are called and exited, they retain their values even after the functions that set them have returned.

An external variable must be `defined`, exactly once, outside of any function; this sets aside storage for it. The variable must also be `declared` in each function that wants to access it; this states the type of the variable. The declaration may be an explicit extern statement or may be implicit from context. To make the discussion concrete, let us rewrite the longest-line program with `line`, `longest`, and `max` as external variables. This requires changing the calls, declarations, and bodies of all three functions.

```
#include <stdio.h>
#define MAXLINE 1000    /* maximum input line size */

int max;                /* maximum length seen so far */
char line[MAXLINE];     /* current input line */
char longest[MAXLINE];  /* longest line saved here */

int getline(void);
void copy(void);

/* print longest input line; specialized version */
main()
{
    int len;
    extern int max;
    extern char longest[];
    max = 0;
    while ((len = getline()) > 0)
        if (len > max) {
            max = len;
            copy();
        }
    if (max > 0)    /* there was a line */
    printf("%s", longest);
    return 0;
}

/* getline: specialized version */
int getline(void)
{
    int c, i;
    extern char line[];
    for (i = 0; i < MAXLINE - 1 && (c=getchar()) != EOF ↩
&& c != '\n'; ++i)
        line[i] = c;
    if (c == '\n') {
        line[i] = c;
        ++i;
    }
    line[i] = '\0';
    return i;
}

/* copy: specialized version */
void copy(void)
{
    int i;
    extern char line[], longest[];
        i = 0;
    while ((longest[i] = line[i]) != '\0')
        ++i;
}
```

The external variables in `main`, `getline` and `copy` are defined by the first lines of the example above, which state their type and cause storage to be allocated for them. Syntactically, external definitions are just like definitions of local variables, but since they occur outside of functions, the variables are external. Before a function can use an external variable, the name of the variable must be made known to the function; the declaration is the same as before except for the added keyword extern.

In certain circumstances, the extern declaration can be omitted. If the definition of the external variable occurs in the source file before its use in a particular function, then there is no need for an extern declaration in the function. The extern declarations in `main`, `getline` and `copy` are thus redundant. In fact, common practice is to place definitions of all external variables at the beginning of the source file, and then omit all extern declarations.

If the program is in several source files, and a variable is defined in *file1* and used in *file2* and *file3*, then extern declarations are needed in *file2* and *file3* to connect the occurrences of the variable. The usual practice is to collect extern declarations of variables and functions in a separate file, historically called a header, that is included by `#include` at the front of each source file. The suffix `.h` is conventional for header names. The functions of the standard library, for example, are declared in headers like `<stdio.h>`.

This topic is discussed at length in Chapter 4, and the library itself in Chapter 7 and Appendix B.

Since the specialized versions of `getline` and `copy` have no arguments, logic would suggest that their prototypes at the beginning of the file should be `getline()` and `copy()`. But for compatibility with older C programs the standard takes an empty list as an old-style declaration, and turns off all argument list checking; the word void must be used for an explicitly empty list. We will discuss this further in Chapter 4.

You should note that we are using the words *definition* and *declaration* carefully when we refer to external variables in this section. "Definition" refers to the place where the variable is created or assigned storage; "declaration" refers to places where the nature of the variable is stated but no storage is allocated.

By the way, there is a tendency to make everything in sight an `extern` variable because it appears to simplify communications – argument lists are short and variables are always there when you want them. But external variables are always there even when you don't want them. Relying too heavily on external variables is fraught with peril since it leads to programs whose data connections are not all obvious – variables can be changed in unexpected and even inadvertent ways, and the program is hard to modify. The second version of the longest-line program is inferior to the first, partly for these reasons, and partly because it destroys the generality of two useful functions by writing into them the names of the variables they manipulate.

At this point we have covered what might be called the conventional core of C. With this handful of building blocks, it's possible to write useful programs of considerable size, and it would probably be a good idea if you paused long enough to do so. These exercises suggest programs of somewhat greater complexity than the ones earlier in this chapter.

Exercise 1.20 — Write a program `detab` that replaces tabs in the input with the proper number of blanks to space to the next tab stop. Assume a fixed set of tab stops, say every `n` columns. Should `n` be a variable or a symbolic parameter?

Exercise 1.21 — Write a program `entab` that replaces strings of blanks by the minimum number of tabs and blanks to achieve the same spacing. Use the same tab stops as for `detab`. When either a tab or a single blank would suffice to reach a tab stop, which should be given preference?

Exercise 1.22 — Write a program to "fold" long input lines into two or more shorter lines after the last non-blank character that occurs before the `n`-th column of input. Make sure your program does something intelligent with very long lines, and if there are no blanks or tabs before the specified column.

Exercise 1.23 — Write a program to remove all comments from a C program. Don't forget to handle quoted strings and character constants properly. C comments don't nest.

Exercise 1.24 — Write a program to check a C program for rudimentary syntax errors like unmatched parentheses, brackets and braces. Don't forget about quotes, both single and double, escape sequences, and comments. (This program is hard if you do it in full generality.)

Chapter 4

Functions and Program Structure

Chapter 5

Pointers and Arrays

A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways. Pointers and arrays are closely related; this chapter also explores this relationship and shows how to exploit it.

Pointers have been lumped with the `goto` statement as a marvelous way to create impossible-to-understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can also be used to achieve clarity and simplicity. This is the aspect that we will try to illustrate.

The main change in ANSI C is to make explicit the rules about how pointers can be manipulated, in effect mandating what good programmers already practice and good compilers already enforce. In addition, the type `void *` (pointer to void) replaces `char *` as the proper type for a generic pointer.

5.1 Pointers and Addresses

Let us begin with a simplified picture of how memory is organized. A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups. One common situation is that any byte can be a `char`, a pair of one-byte cells can be treated as a `short` integer, and four adjacent bytes form a `long`. A pointer is a group of cells (often two or four) that can hold an address. So if `c` is a `char` and p is a pointer that points to it, we could represent the situation this way:

The unary operator `&` gives the address of an object, so the statement

```
p = &c;
```

assigns the address of `c` to the variable `p`, and `p` is said to "point to" `c`. The `&` operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or `register` variables.

The unary operator `*` is the *indirection* or *dereferencing* operator; when applied to a pointer, it accesses the object the pointer points to. Suppose that `x` and `y` are integers and `ip` is a pointer to `int`. This artificial sequence shows how to declare a pointer and how to use `1&` and `*`:

```
int x = 1, y =2, z[10];
int *ip;        /* ip is a pointer to int */
ip = &x;        /* ip now points to x */
y = *ip;        /* y is now 1 */
*ip = 0;        /* x is now 0 */
ip = &z[0];     /* ip now points to z[0] */
```

The declaration of `x`, `y`, and `z` are what we've seen all along. The declaration of the pointer `ip`,

```
int *ip;
```

is intended as a mnemonic; it says that the expression `*ip` is an `int`. The syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear. This reasoning applies to function declarations as well. For example,

```
double *dp, atof(char *);
```

says that in an expression `*dp` and `atof(s)` have values of `double`, and that the argument of `atof` is a pointer to `char`.

You should also note the implication that a pointer is constrained to point to a particular kind of object: every pointer points to a specific data type. (There is one exception: a "pointer to void" is used to hold any type of pointer but cannot be dereferenced itself. We'll come back to it in Section 5.11.)

If `ip` points to the integer `x`, then `*ip` can occur in any context where `x` could, so

```
    *ip = *ip + 10;
```

increments `*ip` by `10`.

The unary operators `*` and `&` bind more tightly than arithmetic operators, so the assignment

```
    y = *ip + 1
```

takes whatever `ip` points at, adds `1`, and assigns the result to `y`, while

```
    *ip += 1
```

increments what `ip` points to, as do

```
    ++*ip
```

and

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

The parentheses are necessary in this last example; without them, the expression would increment `ip` instead of what it points to, because unary operators like `*` and `++` associate right to left.

Finally, since pointers are variables, they can be used without dereferencing. For example, if `iq` is another pointer to `int`,

```
    iq = ip
```

copies the contents of `ip` into `iq`, thus making `iq` point to whatever `ip` pointed to.

5.2 Pointers and Function Arguments

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two out-of-order arguments with a function called `swap`. It is not enough to write

```
    swap(a, b);
```

where the `swap` function is defined as

```
    void swap(int x, int y)     /* WRONG */
    {
        int temp;
        temp = x;
        x = y;
        y = temp;
    }
```

Because of *call by value*, `swap` can't affect the arguments `a` and `b` in the routine that called it. The function above swaps copies of `a` and `b`.

The way to obtain the desired effect is for the calling program to pass *pointers* to the values to be changed:

```
    swap(&a, &b);
```

Since the operator `&` produces the address of a variable, `&a` is a pointer to `a`. In `swap` itself, the parameters are declared as pointers, and the operands are accessed indirectly through them.

```
    void swap(int *px, int *py) /* interchange *px and *py */
    {
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
    }
```

Pictorially:

Pointer arguments enable a function to access and change objects in the function that called it. As an example, consider a function `getint` that performs free-format input conversion by breaking a stream of characters into integer values, one integer per call. `getint` has to return the value it found and also signal end of file when there is no more input. These values have to be passed back by separate paths, for no matter what value is used for `EOF`, that could also be the value of an input integer.

One solution is to have `getint` return the end of file status as its function value, while using a pointer argument to store the converted integer back in the calling function. This is the scheme used by `scanf` as well; see Section 7.4.

The following loop fills an array with integers by calls to `getint`:

```
    int n, array[SIZE], getint(int *);
    for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++)
        ;
```

Each call sets `array[n]` to the next integer found in the input and increments `n`. Notice that it is essential to pass the address of `array[n]` to `getint`. Otherwise there is no way for `getint` to communicate the converted integer back to the caller.

Our version of `getint` returns `EOF` for end of file, zero if the next input is not a number, and a positive value if the input contains a valid number.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getint: get next integer from input into *pn */
int getint(int *pn)
{
    int c, sign;
    while (isspace(c = getch())) /* skip white space */
        ;
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c); /* it is not a number */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c), c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}
```

Throughout `getint`, `*pn` is used as an ordinary `int` variable. We have also used `getch` and `ungetch` (described in Section 4.3) so the one extra character that must be read can be pushed back onto the input.

Exercise 5.1 — As written, `getint` treats a + or - not followed by a digit as a valid representation of zero. Fix it to push such a character back on the input.

Exercise 5.2 — Write `getfloat`, the floating-point analog of `getint`. What type does `getfloat` return as its function value?

5.3 Pointers and Arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand. The declaration

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

defines an array of size 10, that is, a block of 10 consecutive objects named `a[0]`, `a[1]`, ..., `a[9]`.

The notation `a[i]` refers to the i-th element of the array. If `pa` is a pointer to an integer, declared as

```
int *pa;
```

then the assignment

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

sets `pa` to point to element zero of `a`; that is, `pa` contains the address of `a[0]`.

Now the assignment

```
x = *pa;
```

will copy the contents of `a[0]` into `x`. If `pa` points to a particular element of an array, then by definition `pa+1` points to the next element, `pa+i` points i elements after `pa`, and `pa-i` points `i` elements before. Thus, if `pa` points to `a[0]`,

```
*(pa+1)
```

refers to the contents of `a[1]`, `pa+i` is the address of `a[i]`, and `*(pa+i)` is the contents of `a[i]`.

These remarks are true regardless of the type or size of the variables in the array `a`. The meaning of "adding 1 to a pointer", and by extension, all pointer arithmetic, is that `pa+1` points to the next object, and `pa+i` points to the i-th object beyond `pa`.

The correspondence between indexing and pointer arithmetic is very close. By definition, the value of a variable or expression of type array is the address of element zero of the array. Thus after the assignment

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

`pa` and `a` have identical values.

Since the name of an array is a synonym for the location of the initial element, the assignment `pa=&a[0]` can also be written as

```
pa = a;
```

Rather more surprising, at first sight, is the fact that a reference to `a[i]` can also be written as `*(a+i)`. In evaluating `a[i]`, C converts it to `*(a+i)` immediately; the two forms are equivalent.

Applying the operator `&` to both parts of this equivalence, it follows that `&a[i]` and `a+i` are also identical: `a+i` is the address of the i-th element beyond `a`.

As the other side of this coin, if pa is a pointer, expressions might use it with a subscript; pa[i] is identical to

```
*(pa+i).
```

In short, an array-and-index expression is equivalent to one written as a pointer and offset. There is one difference between an array name and a pointer that must be kept in mind. A pointer is a variable, so `pa=a` and `pa++` are legal. But an array name is not a variable; constructions like `a=pa` and `a++` are illegal.

When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address. We can use this fact to write another version of `strlen`, which computes the length of a string.

```
/* strlen: return length of string s */
int strlen(char *s)
{
    int n;
    for (n = 0; *s != '\0', s++)
        n++;
    return n;
}
```

Since `s` is a pointer, incrementing it is perfectly legal; `s++` has no effect on the character string in the function that called `strlen`, but merely increments `strlen`'s private copy of the pointer. That means that calls like

```
    strlen("hello, world"); /* string constant */
    strlen(array);          /* char array[100]; */
    strlen(ptr);            /* char *ptr; */
```

all work.

As formal parameters in a function definition,

```
    char s[];
```

and

```
    char *s;
```

are equivalent; we prefer the latter because it says more explicitly that the variable is a pointer. When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or a pointer, and manipulate it accordingly. It can even use both notations if it seems appropriate and clear.

It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray. For example, if `a` is an array,

```
    f(&a[2])
```

and

```
    f(a+2)
```

both pass to the function `f` the address of the subarray that starts at `a[2]`. Within `f`, the parameter declaration can read

```
    f(int arr[]) { ... }
```

or

```
    f(int *arr) { ... }
```

So as far as `f` is concerned, the fact that the parameter refers to part of a larger array is of no consequence.

If one is sure that the elements exist, it is also possible to index backwards in an array; `p[-1]`, `p[-2]`, and so on are syntactically legal, and refer to the elements that immediately precede `p[0]`. Of course, it is illegal to refer to objects that are not within the array bounds.

5.4 Address Arithmetic

If `p` is a pointer to some element of an array, then `p++` increments `p` to point to the next element, and `p += i` increments it to point `i` elements beyond where it currently does. These and similar constructions are the simplest forms of pointer or address arithmetic.

C is consistent and regular in its approach to address arithmetic; its integration of pointers, arrays, and address arithmetic is one of the strengths of the language. Let us illustrate by writing a rudimentary storage allocator. There are two routines. The first, `alloc(n)`, returns a pointer `p` to n-consecutive character positions, which can be used by the caller of `alloc` for storing characters. The second, `afree(p)`, releases the storage thus acquired so it can be reused later. The routines are "rudimentary" because the calls to `afree` must be made in the opposite order to the calls made on `alloc`. That is, the storage managed by `alloc` and `afree` is a stack, or last-in, first-out. The standard library provides analogous functions called `malloc` and `free` that have no such restrictions; in Section 8.7 we will show how they can be implemented.

The easiest implementation is to have `alloc` hand out pieces of a large character array that we will call `allocbuf`. This array is private to `alloc` and `afree`. Since they deal in pointers, not array indices, no other routine need know the name of the array, which can be declared `static` in the source file containing `alloc` and `afree`, and thus be invisible outside it. In practical implementations, the array may well not even have a name; it might instead be obtained by calling `malloc` or by asking the operating system for a pointer to some unnamed block of storage.

The other information needed is how much of `allocbuf` has been used. We use a pointer, called `allocp`, that points to the next free element. When `alloc` is asked for n characters, it checks to see if there is enough room left in `allocbuf`. If so, `alloc` returns the current value of `allocp` (i.e., the beginning of the free block), then increments it by `n` to point to the next free area. If there is no room, `alloc` returns zero. `afree(p)` merely sets `allocp` to `p` if `p` is inside `allocbuf`.

```
#define ALLOCSIZE 10000       /* size of available space */

static char allocbuf[ALLOCSIZE];    /* storage for alloc */
static char *allocp = allocbuf;    /* next free position */

char *alloc(int n)     /* return pointer to n characters */
{
    if (allocbuf + ALLOCSIZE - allocp >= n) { /* it fits */
        allocp += n;
        return allocp - n;                          /* old p */
    } else                                 /* not enough room */
        return 0;
}

void afree(char *p)      /* free storage pointed to by p */
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
    allocp = p;
}
```

In general a pointer can be initialized just as any other variable can, though normally the only meaningful values are zero or an expression involving the address of previously defined data of appropriate type. The declaration

```
    static char *allocp = allocbuf;
```

defines `allocp` to be a character pointer and initializes it to point to the beginning of `allocbuf`, which is the next free position when the program starts. This could also have been written

```
    static char *allocp = &allocbuf[0];
```

since the array name is the address of the zeroth element. The test

```
    if (allocbuf + ALLOCSIZE - allocp >= n) { /* it fits */
```

checks if there's enough room to satisfy a request for `n` characters. If there is, the new value of `allocp` would be at most one beyond the end of `allocbuf`. If the request can be satisfied, `alloc` returns a pointer to the beginning of a block of characters (notice the declaration of the function itself). If not, `alloc` must return some signal that there is no space left. C guarantees that zero is never a valid address for data, so a return value of zero can be used to signal an abnormal event, in this case no space.

Pointers and integers are not interchangeable. Zero is the sole exception: the constant zero may be assigned to a pointer, and a pointer may be compared with the constant zero. The symbolic constant `NULL` is often used in place of zero, as a mnemonic to indicate more clearly that this is a special value for a pointer. `NULL` is defined in `<stdio.h>`. We will use `NULL` henceforth.

Tests like

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* it fits */
```

and

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

show several important facets of pointer arithmetic.

First, pointers may be compared under certain circumstances. If `p` and `q` point to members of the same array, then relations like `==`, `!=`, `<`, `>=`, etc., work properly. For example,

```
p < q
```

is true if `p` points to an earlier element of the array than `q` does. Any pointer can be meaningfully compared for equality or inequality with zero. But the behavior is undefined for arithmetic or comparisons with pointers that do not point to members of the same array. (There is one exception: the address of the first element past the end of an array can be used in pointer arithmetic.)

Second, we have already observed that a pointer and an integer may be added or subtracted. The construction

```
p + n
```

means the address of the `n`-th object beyond the one `p` currently points to. This is true regardless of the kind of object `p` points to; `n` is scaled according to the size of the objects `p` points to, which is determined by the declaration of `p`. If an `int` is four bytes, for example, the int will be scaled by four.

Pointer subtraction is also valid: if `p` and `q` point to elements of the same array, and `p < q`, then `q-p+1` is the number of elements from `p` to `q` inclusive. This fact can be used to write yet another version of `strlen`:

```
/* strlen: return length of string s */
int strlen(char *s)
```

```
    {
        char *p = s;
        while (*p != '\0')
        p++;
        return p - s;
    }
```

In its declaration, p is initialized to s, that is, to point to the first character of the string. In the while loop, each character in turn is examined until the '\0' at the end is seen. Because p points to characters, p++ advances p to the next character each time, and p-s gives the number of characters advanced over, that is, the string length. (The number of characters in the string could be too large to store in an int. The header <stddef.h> defines a type ptrdiff_t that is large enough to hold the signed difference of two pointer values. If we were being cautious, however, we would use size_t for the return value of strlen, to match the standard library version. size_t is the unsigned integer type returned by the sizeof operator.)

Pointer arithmetic is consistent: if we had been dealing with floats, which occupy more storage that chars, and if p were a pointer to float, p++ would advance to the next float. Thus we could write another version of alloc that maintains floats instead of chars, merely by changing char to float throughout alloc and afree. All the pointer manipulations automatically take into account the size of the objects pointed to.

The valid pointer operations are assignment of pointers of the same type, adding or subtracting a pointer and an integer, subtracting or comparing two pointers to members of the same array, and assigning or comparing to zero. All other pointer arithmetic is illegal. It is not legal to add two pointers, or to multiply or divide or shift or mask them, or to add float or double to them, or even, except for void *, to assign a pointer of one type to a pointer of another type without a cast.

5.5 Character Pointers and Functions

A string constant, written as “I am a string” is an array of characters. In the internal representation, the array is terminated with the null character '\0' so that programs can find the end. The length in storage is thus one more than the number of characters between the double quotes.

Perhaps the most common occurrence of string constants is as arguments to functions, as in

```
printf("hello, world\n");
```

When a character string like this appears in a program, access to it is through a character pointer; `printf` receives a pointer to the beginning of the character array. That is, a string constant is accessed by a pointer to its first element.

String constants need not be function arguments. If `pmessage` is declared as

```
char *pmessage;
```

then the statement

```
pmessage = "now is the time";
```

assigns to `pmessage` a pointer to the character array. This is not a string copy; only pointers are involved. C does not provide any operators for processing an entire string of characters as a unit.

There is an important difference between these definitions:

```
char amessage[] = "now is the time"; /* an array */
char *pmessage = "now is the time";  /* a pointer */
```

`amessage` is an array, just big enough to hold the sequence of characters and `'\0'` that initializes it. Individual characters within the array may be changed but `amessage` will always refer to the same storage. On the other hand, `pmessage` is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.

We will illustrate more aspects of pointers and arrays by studying versions of two useful functions adapted from the standard library. The first function is `strcpy(s,t)`, which copies the string `t` to the string `s`. It would be nice just to say `s=t` but this copies the pointer, not the characters. To copy the characters, we need a loop. The array version first:

```
/* strcpy: copy t to s; array subscript version */
void strcpy(char *s, char *t)
{
    int i;
    i = 0;
```

```
        while ((s[i] = t[i]) != '\0')
            i++;
    }
```

For contrast, here is a version of `strcpy` with pointers:

```
    /* strcpy: copy t to s; pointer version */
    void strcpy(char *s, char *t)
    {
        int i;
        i = 0;
        while ((*s = *t) != '\0') {
            s++;
            t++;
        }
    }
```

Because arguments are passed by value, `strcpy` can use the parameters `s` and `t` in any way it pleases. Here they are conveniently initialized pointers, which are marched along the arrays a character at a time, until the `'\0'` that terminates `t` has been copied into `s`.

In practice, `strcpy` would not be written as we showed it above. Experienced C programmers would prefer

```
    /* strcpy: copy t to s; pointer version 2 */
    void strcpy(char *s, char *t)
    {
        while ((*s++ = *t++) != '\0')
            ;
    }
```

This moves the increment of s and t into the test part of the loop. The value of `*t++` is the character that `t` pointed to before `t` was incremented; the postfix `++` doesn't change t until after this character has been fetched. In the same way, the character is stored into the old `s` position before `s` is incremented. This character is also the value that is compared against `'\0'` to control the loop. The net effect is that characters are copied from `t` to `s`, up and including the terminating `'\0'`.

As the final abbreviation, observe that a comparison against `'\0'` is redundant, since the question is merely whether the expression is zero. So the function would likely be written as

```
/* strcpy: copy t to s; pointer version 3 */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

Although this may seem cryptic at first sight, the notational convenience is considerable, and the idiom should be mastered, because you will see it frequently in C programs.

The `strcpy` in the standard library (`<string.h>`) returns the target string as its function value.

The second routine that we will examine is `strcmp(s,t)`, which compares the character strings `s` and `t`, and returns negative, zero or positive if `s` is lexicographically less than, equal to, or greater than `t`. The value is obtained by subtracting the characters at the first position where `s` and `t` disagree.

```
/* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
    int i;
    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

The pointer version of strcmp:

```
/* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

Since `++` and `--` are either prefix or postfix operators, other combinations of `*`and `++` and `--` occur, although less frequently. For example,

```
*--p
```

decrements `p` before fetching the character that `p` points to. In fact, the pair of expressions

```
    *p++ = val;     /* push val onto stack */
    val = *--p;     /* pop top of stack into val */
```

are the standard idiom for pushing and popping a stack; see Section 4.3. The header `<string.h>` contains declarations for the functions mentioned in this section, plus a variety of other string-handling functions from the standard library.

Exercise 5.3 — Write a pointer version of the function `strcat` that we showed in Chapter 2: `strcat(s,t)` copies the string `t` to the end of `s`.

Exercise 5.4 — Write the function `strend(s,t)`, which returns 1 if the string `t` occurs at the end of the string `s`, and zero otherwise.

Exercise 5.5 — Write versions of the library functions `strncpy`, `strncat`, and `strncmp`, which operate on at most the first `n` characters of their argument strings. For example, `strncpy(s,t,n)` copies at most `n` characters of `t` to `s`. Full descriptions are in Appendix B.

Exercise 5.6 — Rewrite appropriate programs from earlier chapters and exercises with pointers instead of array indexing. Good possibilities include `getline` (Chapters 1 and 4), `atoi`, `itoa`, and their variants (Chapters 2, 3, and 4), `reverse` (Chapter 3), and `strindex` and `getop` (Chapter 4).

5.6 Pointer Arrays; Pointers to Pointers

Since pointers are variables themselves, they can be stored in arrays just as other variables can. Let us illustrate by writing a program that will sort a set of text lines into alphabetic order, a stripped-down version of the UNIX program sort.

In Chapter 3, we presented a Shell sort function that would sort an array of integers, and in Chapter 4 we improved on it with a quicksort. The same algorithms will work, except that now we have to deal with lines of text, which are of different lengths, and which, unlike integers, can't be compared or moved in a single operation. We need a data representation that will cope efficiently and conveniently with variable-length text lines.

This is where the array of pointers enters. If the lines to be sorted are stored end-to-end in one long character array, then each line can be accessed by a pointer to its first character. The pointers themselves can bee stored in an array. Two lines can be compared by passing their pointers to `strcmp`. When two out-of-order lines have to be exchanged, the pointers in the pointer array are exchanged, not the text lines themselves.

This eliminates the twin problems of complicated storage management and high overhead that would go with moving the lines themselves.

The sorting process has three steps:

```
read all the lines of input
sort them
print them in order
```

As usual, it's best to divide the program into functions that match this natural division, with the main routine controlling the other functions. Let us defer the sorting step for a moment, and concentrate on the data structure and the input and output.

The input routine has to collect and save the characters of each line, and build an array of pointers to the lines. It will also have to count the number of input lines, since that information is needed for sorting and printing. Since the input function can only cope with a finite number of input lines, it can return some illegal count like -1 if too much input is presented.

The output routine only has to print the lines in the order in which they appear in the array of pointers.

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

#define MAXLINES 5000       /* max #lines to be sorted */
char *lineptr[MAXLINES];    /* pointers to text lines */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);
void qsort(char *lineptr[], int left, int right);

/* sort input lines */
main()
{
    int nlines;     /* number of input lines read */
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort(lineptr, 0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("error: input too big to sort\n");
        return 1;
    }
}

#define MAXLEN 1000 /* max length of any input line */
int getline(char *, int);
char *alloc(int);

/* readlines: read input lines */
int readlines(char *lineptr[], int maxlines)
{
    int len, nlines;
    char *p, line[MAXLEN];
    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines || p = alloc(len) == NULL)
            return -1;
    else {
        line[len-1] = '\0'; /* delete newline */
        strcpy(p, line);
        lineptr[nlines++] = p;
    }
    return nlines;
}

/* writelines: write output lines */
void writelines(char *lineptr[], int nlines)
{
    int i;
    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
```

```
}
```

The function `getline` is from Section 1.9.

The main new thing is the declaration for `lineptr`:

```
char *lineptr[MAXLINES]
```

says that `lineptr` is an array of MAXLINES elements, each element of which is a pointer to a char. That is, `lineptr[i]` is a character pointer, and `*lineptr[i]` is the character it points to, the first character of the `i`-th saved text line.

Since `lineptr` is itself the name of an array, it can be treated as a pointer in the same manner as in our earlier examples, and writelines can be written instead as

```
/* writelines: write output lines */
void writelines(char *lineptr[], int nlines)
{
    while (nlines-- > 0)
    printf("%s\n", *lineptr++);
}
```

Initially, `*lineptr` points to the first line; each element advances it to the next line pointer while `nlines` is counted down.

With input and output under control, we can proceed to sorting. The `quicksort` from Chapter 4 needs minor changes: the declarations have to be modified, and the comparison operation must be done by calling `strcmp`. The algorithm remains the same, which gives us some confidence that it will still work.

```
/* qsort: sort v[left]...v[right] into increasing order*/
void qsort(char *v[], int left, int right)
{
    int i, last;
    void swap(char *v[], int i, int j);

    if (left >= right) /* do nothing if array contains */
        return;        /* fewer than two elements */

    swap(v, left, (left + right)/2);
```

```
        last = left;
        for (i = left+1; i <= right; i++)
            if (strcmp(v[i], v[left]) < 0)
                swap(v, ++last, i);
        swap(v, left, last);

        qsort(v, left, last-1);
        qsort(v, last+1, right);
    }
```

Similarly, the swap routine needs only trivial changes:

```
    /* swap: interchange v[i] and v[j] */
    void swap(char *v[], int i, int j)
    {
        char *temp;
        temp = v[i];
        v[i] = v[j];
        v[j] = temp;
    }
```

Since any individual element of `v` (alias `lineptr`) is a character pointer, `temp` must be also, so one can be copied to the other.

Exercise 5.7 — Rewrite `readlines` to store lines in an array supplied by `main`, rather than calling `alloc` to maintain storage. How much faster is the program?

5.7 Multi-dimensional Arrays

C provides rectangular multi-dimensional arrays, although in practice they are much less used than arrays of pointers. In this section, we will show some of their properties.

Consider the problem of date conversion, from day of the month to day of the year and vice versa. For example, March 1 is the 60th day of a non-leap year, and the 61st day of a leap year. Let us define two functions to do the conversions: `day_of_year` converts the month and day into the day of the year, and `month_day` converts the day of the year into the month and day.

Since this latter function computes two values, the month and day arguments will be pointers:

```
month_day(1988, 60, &m, &d)
```

sets `m` to 2 and `d` to 29 (February 29th).

These functions both need the same information, a table of the number of days in each month ("thirty days hath September ..."). Since the number of days per month differs for leap years and non-leap years, it's easier to separate them into two rows of a two-dimensional array than to keep track of what happens to February during computation. The array and the functions for performing the transformations are as follows:

```
static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year: set day of year from month & day */
int day_of_year(int year, int month, int day)
{
    int i, leap;
    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; i < month; i++)
        day += daytab[leap][i];
    return day;
}

/* month_day: set month, day from day of year */
void month_day(int year, int yearday, int *pmonth, int *↩
pday)
{
    int i, leap;
    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; yearday > daytab[leap][i]; i++)
        yearday -= daytab[leap][i];
    *pmonth = i;
    *pday = yearday;
}
```

Recall that the arithmetic value of a logical expression, such as the one for leap, is either zero (false) or one (true), so it can be used as a subscript of the array `daytab`. The array daytab has to be external to both `day_of_year` and `month_day`, so they can both use it. We made it `char` to illustrate a legitimate use of `char` for storing small non-character integers. `daytab` is the first two-dimensional array we have dealt with. In C, a two-dimensional array is really a one-dimensional array, each of whose elements is an array. Hence subscripts are written as

```
daytab[i][j]    /* [row][col] */
```

rather than

```
daytab[i,j]     /* WRONG */
```

Other than this notational distinction, a two-dimensional array can be treated in much the same way as in other languages. Elements are stored by rows, so the rightmost subscript, or column, varies fastest as elements are accessed in storage order.

An array is initialized by a list of initializers in braces; each row of a two-dimensional array is initialized by a corresponding sub-list. We started the array `daytab` with a column of zero so that month numbers can run from the natural `1` to `12` instead of `0` to `11`. Since space is not at a premium here, this is clearer than adjusting the indices.

If a two-dimensional array is to be passed to a function, the parameter declaration in the function must include the number of columns; the number of rows is irrelevant, since what is passed is, as before, a pointer to an array of rows, where each row is an array of 13 `int`s. In this particular case, it is a pointer to objects that are arrays of 13 `int`s. Thus if the array daytab is to be passed to a function `f`, the declaration of `f` would be:

```
f(int daytab[2][13]) { ... }
```

It could also be

```
f(int daytab[][13]) { ... }
```

since the number of rows is irrelevant, or it could be

```
f(int (*daytab)[13]) { ... }
```

which says that the parameter is a pointer to an array of 13 integers.

The parentheses are necessary since brackets `[]` have higher precedence than `*`. Without parentheses, the declaration

```
int *daytab[13]
```

is an array of 13 pointers to integers.

More generally, only the first dimension (subscript) of an array is free; all the others have to be specified. Section 5.12 has a further discussion of complicated declarations.

Exercise 5.8 — There is no error checking in `day_of_year` or `month_day`. Remedy this defect.

5.8 Initialization of Pointer Arrays

Consider the problem of writing a function `month_name(n)`, which returns a pointer to a character string containing the name of the `n`-th month. This is an ideal application for an internal `static` array. `month_name` contains a private array of character strings, and returns a pointer to the proper one when called. This section shows how that array of names is initialized.

The syntax is similar to previous initializations:

```
/* month_name: return name of n-th month */
char *month_name(int n)
{
    static char *name[] = {
        "Illegal month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };
    return (n < 1 || n > 12) ? name[0] : name[n];
}
```

The declaration of `name`, which is an array of character pointers, is the same as `lineptr` in the sorting example. The initializer is a list of character strings; each is assigned to the corresponding position in the array. The characters of the `i`-th string are placed somewhere, and a pointer to them is stored in `name[i]`. Since the size of the array `name` is not specified, the compiler counts the initializers and fills in the correct number.

5.9 Pointers vs. Multi-dimensional Arrays

Newcomers to C are sometimes confused about the difference between a two-dimensional array and an array of pointers, such as name in the example above. Given the definitions

```
int a[10][20];
int *b[10];
```

then `a[3][4]` and `b[3][4]` are both syntactically legal references to a single `int`. But a is a true two-dimensional array: 200 `int`-sized locations have been set aside, and the conventional rectangular subscript calculation `20 * row +col` is used to find the element `a[row,col]`. For `b`, however, the definition only allocates 10 pointers and does not initialize them; initialization must be done explicitly, either statically or with code. Assuming that each element of `b` does point to a twenty-element array, then there will be 200 `int`s set aside, plus ten cells for the pointers. The important advantage of the pointer array is that the rows of the array may be of different lengths. That is, each element of `b` need not point to a twenty-element vector; some may point to two elements, some to fifty, and some to none at all.

Although we have phrased this discussion in terms of integers, by far the most frequent use of arrays of pointers is to store character strings of diverse lengths, as in the function `month_name`. Compare the declaration and picture for an array of pointers:

```
char *name[] = {"Illegal month", "Jan", "Feb", "Mar"};
char aname[][15] = {"Illegal month", "Jan", "Feb", "Mar"↩
};
```

Exercise 5.9 — Rewrite the routines `day_of_year` and `month_day` with pointers instead of indexing.

5.10 Command-line Arguments

In environments that support C, there is a way to pass command-line arguments or parameters to a program when it begins executing. When `main` is called, it is called with two arguments. The first (conventionally called `argc`, for argument count) is the number of command-line arguments the program was invoked with; the second (`argv`, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string.

We customarily use multiple levels of pointers to manipulate these character strings.

The simplest illustration is the program echo, which echoes its command-line arguments on a single line, separated by blanks. That is, the command

```
echo hello, world
```

prints the output

```
hello, world
```

By convention, `argv[0]` is the name by which the program was invoked, so `argc` is at least 1. If `argc` is `1`, there are no command-line arguments after the program name. In the example above, `argc` is `3`, and `argv[0]`, `argv[1]`, and `argv[2]` are ``‘‘echo’’``, ``‘‘hello,’’``, and ``‘‘world’’`` respectively. The first optional argument is `argv[1]` and the last is `argv[argc-1]`; additionally, the standard requires that `argv[argc]` be a null pointer.

The first version of echo treats `argv` as an array of character pointers:

```
#include <stdio.h>

/* echo command-line arguments; 1st version */
main(int argc, char *argv[])
{
    int i;
    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    return 0;
}
```

Since `argv` is a pointer to an array of pointers, we can manipulate the pointer rather than index the array. This next variant is based on incrementing `argv`, which is a pointer to pointer to char, while `argc` is counted down:

```
#include <stdio.h>

/* echo command-line arguments; 2nd version */
main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}
```

Since `argv` is a pointer to the beginning of the array of argument strings, incrementing it by 1 (`++argv`) makes it point at the original `argv[1]` instead of `argv[0]`. Each successive increment moves it along to the next argument; `*argv` is then the pointer to that argument. At the same time, `argc` is decremented; when it becomes zero, there are no arguments left to print. Alternatively, we could write the `printf` statement as

```
    printf((argc > 1) ? "%s " : "%s", *++argv);
```

This shows that the format argument of `printf` can be an expression too.

As a second example, let us make some enhancements to the pattern-finding program from Section 4.1. If you recall, we wired the search pattern deep into the program, an obviously unsatisfactory arrangement. Following the lead of the UNIX program `grep`, let us enhance the program so the pattern to be matched is specified by the first argument on the command line.

```
    #include <stdio.h>
    #include <string.h>
    #define MAXLINE 1000

    int getline(char *line, int max);

    /* find: print lines that match pattern from 1st arg */
    main(int argc, char *argv[])
    {
        char line[MAXLINE];
        int found = 0;
        if (argc != 2)
            printf("Usage: find pattern\n");
        else
            while (getline(line, MAXLINE) > 0)
        if (strstr(line, argv[1]) != NULL) {
            printf("%s", line);
            found++;
        }
        return found;
    }
```

The standard library function `strstr(s,t)` returns a pointer to the first occurrence of the string `t` in the string `s`, or `NULL` if there is none. It is declared in `<string.h>`.

The model can now be elaborated to illustrate further pointer constructions. Suppose we want to allow two optional arguments. One says "print all the lines except those that match the pattern;" the second says "precede each printed line by its line number".

A common convention for C programs on UNIX systems is that an argument that begins with a minus sign introduces an optional flag or parameter. If we choose `-x` (for "except") to signal the inversion, and `-n` ("number") to request line numbering, then the command

```
    find -x -n pattern
```

will print each line that doesn't match the pattern, preceded by its line number.

Optional arguments should be permitted in any order, and the rest of the program should be independent of the number of arguments that we present. Furthermore, it is convenient for users if option arguments can be combined, as in

```
    find -nx pattern
```

Here is the program:

```
    #include <stdio.h>
    #include <string.h>
    #define MAXLINE 1000

    int getline(char *line, int max);

    /* find: print lines that match pattern from 1st arg */
    main(int argc, char *argv[])
    {
        char line[MAXLINE];
        long lineno = 0;
        int c, except = 0, number = 0, found = 0;
        while (--argc > 0 && (*++argv)[0] == '-')
            while (c = *++argv[0])
                switch (c) {
                    case 'x':
                        except = 1;
                        break;
                    case 'n':
                        number = 1;
                        break;
                    default:
                        printf("find: illegal option %c\n", c↩
);
                        argc = 0;
                        found = -1;
                    break;
        }

        if (argc != 1)
            printf("Usage: find -x -n pattern\n");
        else
            while (getline(line, MAXLINE) > 0) {
                lineno++;
                if ((strstr(line, *argv) != NULL) != except) ↩
{
                    if (number)
                        printf("%ld:", lineno);
                    printf("%s", line);
                    found++;
                }
            }
        return found;
 }
```

argc is decremented and argv is incremented before each optional argument. At the end of the loop, if there are no errors, argc tells how many arguments remain unprocessed and argv points to the first of these. Thus argc should be 1 and *argv should point at the pattern. Notice that *++argv is a pointer to an argument string, so `(*++argv)[0]` is its first character. (An alternate valid form would be `**++argv`.) Because `[]` binds tighter than `*` and `++`, the parentheses are necessary; without them the expression would be taken as `*++(argv[0])`. In fact, that is what we have used in the inner loop, where the task is to walk along a specific argument string. In the inner loop, the expression `*++argv[0]` increments the pointer `argv[0]`! It is rare that one uses pointer expressions more complicated than these; in such cases, breaking them into two or three steps will be more intuitive.

Exercise 5.10 — Write the program `expr`, which evaluates a reverse Polish expression from the command line, where each operator or operand is a separate argument. For example,

```
expr 2 3 4 + *
```

evaluates 2 * (3+4).

Exercise 5.11 — Modify the program `entab` and `detab` (written as exercises in Chapter 1) to accept a list of tab stops as arguments. Use the default tab settings if there are no arguments.

Exercise 5.12 — Extend `entab` and `detab` to accept the shorthand

```
entab -m +n
```

to mean tab stops every `n` columns, starting at column `m`. Choose convenient (for the user) default behavior.

Exercise 5.13 — Write the program `tail`, which prints the last `n` lines of its input. By default, `n` is set to `10`, let us say, but it can be changed by an optional argument so that

```
tail -n
```

prints the last `n` lines. The program should behave rationally no matter how unreasonable the input or the value of `n`. Write the program so it makes the best use of available storage; lines should be stored as in the sorting program of Section 5.6, not in a two-dimensional array of fixed size.

5.11 Pointers to Functions

In C, a function itself is not a variable, but it is possible to define pointers to functions, which can be assigned, placed in arrays, passed to functions, returned by functions, and so on. We will illustrate this by modifying the sorting procedure written earlier in this chapter so that if the optional argument `-n` is given, it will sort the input lines numerically instead of lexicographically.

A sort often consists of three parts – a comparison that determines the ordering of any pair of objects, an exchange that reverses their order, and a sorting algorithm that makes comparisons and exchanges until the objects are in order. The sorting algorithm is independent of the comparison and exchange operations, so by passing different comparison and exchange functions to it, we can arrange to sort by different criteria. This is the approach taken in our new sort.

Lexicographic comparison of two lines is done by `strcmp`, as before;we will also need a routine `numcmp` that compares two lines on the basis of numeric value and returns the same kind of condition indication as `strcmp` does. These functions are declared ahead of `main` and a pointer to the appropriate one is passed to `qsort`. We have skimped on error processing for arguments, so as to concentrate on the `main` issues.

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

#define MAXLINES 5000 /* max #lines to be sorted */
char *lineptr[MAXLINES]; /* pointers to text lines */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);
void qsort(void *lineptr[], int left, int right,
int (*comp)(void *, void *));
int numcmp(char *, char *);

/* sort input lines */
main(int argc, char *argv[])
{
    int nlines; /* number of input lines read */
    int numeric = 0; /* 1 if numeric sort */
    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort((void**) lineptr, 0, nlines-1,
            (int (*)(void*,void*))(numeric ? numcmp : ↩
strcmp));
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("input too big to sort\n");
        return 1;
    }
}
```

In the call to qsort, strcmp and numcmp are addresses of functions. Since they are known to be functions, the & is not necessary, in the same way that it is not needed before an array name.

We have written qsort so it can process any data type, not just character strings. As indicated by the function prototype, qsort expects an array of pointers, two integers, and a function with two pointer arguments. The generic pointer type void * is used for the pointer arguments. Any pointer can be cast to void * and back again without loss of information, so we can call qsort by casting arguments to void *. The elaborate cast of the function argument casts the arguments of the comparison function. These will generally have no effect on actual representation, but assure the compiler that all is well.

```
/* qsort: sort v[left]...v[right] into increasing order*/
void qsort(void *v[], int left, int right,
           int (*comp)(void *, void *))
{
    int i, last;
    void swap(void *v[], int, int);
    if (left >= right) /* do nothing if array contains */
        return;        /* fewer than two elements */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
```

```
            if ((*comp)(v[i], v[left]) < 0)
                swap(v, ++last, i);
        swap(v, left, last);
        qsort(v, left, last-1, comp);
        qsort(v, last+1, right, comp);
    }
```

The declarations should be studied with some care. The fourth parameter of `qsort` is

```
int (*comp)(void *, void *)
```

which says that `comp` is a pointer to a function that has two `void *` arguments and returns an `int`.

The use of `comp` in the line

```
if ((*comp)(v[i], v[left]) < 0)
```

is consistent with the declaration: `comp` is a pointer to a function, `*comp` is the function, and

```
(*comp)(v[i], v[left])
```

is the call to it. The parentheses are needed so the components are correctly associated; without them,

```
int *comp(void *, void *) /* WRONG */
```

says that `comp` is a function returning a pointer to an int, which is very different.

We have already shown strcmp, which compares two strings. Here is `numcmp`, which compares two strings on a leading numeric value, computed by calling `atof`:

```
#include <stdlib.h>

/* numcmp: compare s1 and s2 numerically */
int numcmp(char *s1, char *s2)
{
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
```

```
        if (v1 < v2)
            return -1;
        else if (v1 > v2)
            return 1;
        else
            return 0;
    }
```

The `swap` function, which exchanges two pointers, is identical to what we presented earlier in the chapter, except that the declarations are changed to `void *`.

```
    void swap(void *v[], int i, int j;)
    {
        void *temp;
        temp = v[i];
        v[i] = v[j];
        v[j] = temp;
    }
```

A variety of other options can be added to the sorting program; some make challenging exercises.

Exercise 5.14 — Modify the sort program to handle a `-r` flag, which indicates sorting in reverse (decreasing) order. Be sure that `-r` works with `-n`.

Exercise 5.15 — Add the option `-f` to fold upper and lower case together, so that case distinctions are not made during sorting; for example, a and A compare equal.

Exercise 5.16 — Add the `-d` ("directory order") option, which makes comparisons only on letters, numbers and blanks. Make sure it works in conjunction with `-f`.

Exercise 5.17 — Add a field-searching capability, so sorting may bee done on fields within lines, each field sorted according to an independent set of options. (The index for this book was sorted with `-df` for the index category and `-n` for the page numbers.)

5.12 Complicated Declarations

C is sometimes castigated for the syntax of its declarations, particularly ones that involve pointers to functions. The syntax is an attempt to make the declaration and the use agree; it works well for simple cases, but it can be confusing for the harder ones, because declarations cannot be read left to right, and because parentheses are over-used.

The difference between

```
int *f();    /* f: function returning pointer to int */
```

and

```
int (*pf)() /* pf: pointer to function returning int */
```

illustrates the problem: `*` is a prefix operator and it has lower precedence than `()`, so parentheses are necessary to force the proper association.

Although truly complicated declarations rarely arise in practice, it is important to know how to understand them, and, if necessary, how to create them. One good way to synthesize declarations is in small steps with typedef, which is discussed in Section 6.7. As an alternative, in this section we will present a pair of programs that convert from valid C to a word description and back again. The word description reads left to right.

The first, `dcl`, is the more complex. It converts a C declaration into a word description, as in these examples:

```
char **argv
    argv: pointer to pointer to char

int (*daytab)[13]
    daytab: pointer to array[13] of int

int *daytab[13]
    daytab: array[13] of pointer to int

void *comp()
    comp: function returning pointer to void

void (*comp)()
    comp: pointer to function returning void
```

```
char (*(*x())[])()
    x: function returning pointer to array[] of
    pointer to function returning char

char (*(*x[3])())[5]
    x: array[3] of pointer to function returning
    pointer to array[5] of char
```

`dcl` is based on the grammar that specifies a declarator, which is spelled out precisely in Appendix A, Section 8.5; this is a simplified form:

```
dcl:        optional *'s direct-dcl
direct-dcl: name
            (dcl)
            direct-dcl()
            direct-dcl[optional size]
```

In words, a dcl is a `direct-dcl`, perhaps preceded by `*`'s. A `direct-dcl` is a name, or a parenthesized `dcl`, or a `direct-dcl` followed by parentheses, or a `direct-dcl` followed by brackets with an optional size.

This grammar can be used to parse functions. For instance, consider this declarator:

```
(*pfa[])()
```

`pfa` will be identified as a name and thus as a `direct-dcl`. Then `pfa[]` is also a `direct-dcl`. Then `*pfa[]` is recognized as a `dcl`, so `(*pfa[])` is a `direct-dcl`. Then `(*pfa[])()` is a `direct-dcl` and thus a `dcl`. We can also illustrate the parse with a tree like this (where `direct-dcl` has been abbreviated to `dir-dcl`):

The heart of the `dcl` program is a pair of functions, `dcl` and `dirdcl`, that parse a declaration according to this grammar. Because the grammar is recursively defined, the functions call each other recursively as they recognize pieces of a declaration; the program is called a recursive-descent parser.

```
/* dcl: parse a declarator */
void dcl(void)
{
    int ns;
    for (ns = 0; gettoken() == '*'; )   /* count *'s */
```

```
            ns++;
        dirdcl();
        while (ns-- > 0)
            strcat(out, " pointer to");
    }

    /* dirdcl: parse a direct declarator */
    void dirdcl(void)
    {
        int type;
        if (tokentype == '(') {          /* ( dcl ) */
            dcl();
            if (tokentype != ')')
                printf("error: missing )\n");
        } else if (tokentype == NAME)   /* variable name */
            strcpy(name, token);
        else
            printf("error: expected name or (dcl)\n");

        while ((type = gettoken()) == PARENS || type == ↩
   BRACKETS)
            if (type == PARENS)
                strcat(out, " function returning");
            else {
                strcat(out, " array");
                strcat(out, token);
                strcat(out, " of");
            }
    }
```

Since the programs are intended to be illustrative, not bullet-proof, there are significant restrictions on `dcl`. It can only handle a simple data type line `char` or `int`. It does not handle argument types in functions, or qualifiers like `const`. Spurious blanks confuse it. It doesn't do much error recovery, so invalid declarations will also confuse it. These improvements are left as exercises.

Here are the global variables and the main routine:

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

#define MAXTOKEN 100

enum { NAME, PARENS, BRACKETS };

void dcl(void);
void dirdcl(void);
int gettoken(void);

int tokentype;           /* type of last token */
char token[MAXTOKEN];    /* last token string */
char name[MAXTOKEN];     /* identifier name */
char datatype[MAXTOKEN]; /* data type = char, int,... */
char out[1000];

int main() /* convert declaration to words */
{
    while (gettoken() != EOF) {  /* 1st token on line */
        strcpy(datatype, token);   /* is the datatype */
        out[0] = '\0';
        dcl();                    /* parse rest of line */
        if (tokentype != '\n')
            printf("syntax error\n");
        printf("%s: %s %s\n", name, out, datatype);
    }
    return 0;
}
```

The function `gettoken` skips blanks and tabs, then finds the next token in the input; a "token" is a name, a pair of parentheses, a pair of brackets perhaps including a number, or any other single character.

```
int gettoken(void) /* return next token */
{
    int c, getch(void);
    void ungetch(int);
```

```
        char *p = token;

        while ((c = getch()) == ' ' || c == '\t')
            ;

        if (c == '(') {
            if ((c = getch()) == ')') {
                strcpy(token, "()");
                return tokentype = PARENS;
            } else {
                ungetch(c);
                return tokentype = '(';
            }
        } else if (c == '[') {
            for (*p++ = c; (*p++ = getch()) != ']'; )
                ;
            *p = '\0';
            return tokentype = BRACKETS;
        } else if (isalpha(c)) {
            for (*p++ = c; isalnum(c = getch()); )
                *p++ = c;
            *p = '\0';
            ungetch(c);
            return tokentype = NAME;
        } else
            return tokentype = c;
    }
```

`getch` and `ungetch` are discussed in Chapter 4.

Going in the other direction is easier, especially if we do not worry about generating redundant parentheses. The program `undcl` converts a word description like “x is a function returning a pointer to an array of pointers to functions returning char,” which we will express as

```
    x () * [] * () char
```

to

```
    char (*(*x())[])()
```

The abbreviated input syntax lets us reuse the `gettoken` function. `undcl` also uses the same external variables as `dcl` does.

```
/* undcl: convert word descriptions to declarations */
int main()
{

    int type;
    char temp[MAXTOKEN];

    while (gettoken() != EOF) {

        strcpy(out, token);

        while ((type = gettoken()) != '\n')
            if (type == PARENS || type == BRACKETS)
                strcat(out, token);
            else if (type == '*') {
                sprintf(temp, "(*%s)", out);
                strcpy(out, temp);
            } else if (type == NAME) {
                sprintf(temp, "%s %s", token, out);
                strcpy(out, temp);
            } else
                printf("invalid input at %s\n", token);
    }
    return 0;
}
```

Exercise 5.18 — Make `dcl` recover from input errors.

Exercise 5.19 — Modify `undcl` so that it does not add redundant parentheses to declarations.

Exercise 5.20 — Expand `dcl` to handle declarations with function argument types, qualifiers like `const`, and so on.

Chapter 6

Structures

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called “records” in some languages, notably Pascal.) Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

One traditional example of a structure is the payroll record: an employee is described by a set of attributes such as name, address, social security number, salary, etc. Some of these in turn could be structures: a name has several components, as does an address and even a salary. Another example, more typical for C, comes from graphics: a point is a pair of coordinate, a rectangle is a pair of points, and so on.

The main change made by the ANSI standard is to define structure assignment – structures may be copied and assigned to, passed to functions, and returned by functions. This has been supported by most compilers for many years, but the properties are now precisely defined. Automatic structures and arrays may now also be initialized.

6.1 Basics of Structures

Let us create a few structures suitable for graphics. The basic object is a point, which we will assume has an `x` coordinate and a `y` coordinate, both integers. The two components can be placed in a structure declared like this:

```
    struct point {
        int x;
        int y;
    };
```

The keyword `struct` introduces a structure declaration, which is a list of declarations enclosed in braces. An optional name called a structure tag may follow the word `struct` (as with point here). The tag names this kind of structure, and can be used subsequently as a shorthand for the part of the declaration in braces. The variables named in a structure are called members. A structure member or tag and an ordinary (i.e., non-member) variable can have the same name without conflict, since they can always be distinguished by context. Furthermore, the same member names may occur in different structures, although as a matter of style one would normally use the same names only for closely related objects.

A `struct` declaration defines a type. The right brace that terminates the list of members may be followed by a list of variables, just as for any basic type. That is,

```
    struct { ... } x, y, z;
```

is syntactically analogous to

```
    int x, y, z;
```

in the sense that each statement declares `x`, `y` and `z` to be variables of the named type and causes space to be set aside for them. A structure declaration that is not followed by a list of variables reserves no storage; it merely describes a template or shape of a structure. If the declaration is tagged, however, the tag can be used later in definitions of instances of the structure. For example, given the declaration of `point` above,

```
    struct point pt;
```

defines a variable `pt` which is a structure of type `struct point`. A structure can be initialized by following its definition with a list of initializers, each a constant expression, for the members:

```
    struct maxpt = { 320, 200 };
```

An automatic structure may also be initialized by assignment or by calling a function that returns a structure of the right type. A member of a particular structure is referred to in an expression by a construction of the form

```
    structure-name.member
```

The structure member operator "." connects the structure name and the member name. To print the coordinates of the point `pt`, for instance,

```
    printf("%d,%d", pt.x, pt.y);
```

or to compute the distance from the origin `(0,0)` to `pt`,

```
    double dist, sqrt(double);
    dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Structures can be nested. One representation of a rectangle is a pair of points that denote the diagonally opposite corners:

```
    struct rect {
        struct point pt1;
        struct point pt2;
    };
```

The `rect` structure contains two point structures. If we declare `screen` as

```
    struct rect screen;
```

then

```
    screen.pt1.x
```

refers to the `x` coordinate of the `pt1` member of `screen`.

6.2 Structures and Functions

The only legal operations on a structure are copying it or assigning to it as a unit, taking its address with `&`, and accessing its members. Copy and assignment include passing arguments to functions and returning values from functions as well. Structures may not be compared. A structure may be initialized by a list of constant member values; an automatic structure may also be initialized by an assignment.

Let us investigate structures by writing some functions to manipulate points and rectangles. There are at least three possible approaches: pass components separately, pass an entire structure, or pass a pointer to it. Each has its good points and bad points.

The first function, `makepoint`, will take two integers and return a `point` structure:

```
/* makepoint: make a point from x and y components */
struct point makepoint(int x, int y)
{
struct point temp;
temp.x = x;
temp.y = y;
return temp;
}
```

Notice that there is no conflict between the argument name and the member with the same name; indeed the re-use of the names stresses the relationship. `makepoint` can now be used to initialize any structure dynamically, or to provide structure arguments to a function:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);
screen.pt1 = makepoint(0,0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                   (screen.pt1.y + screen.pt2.y)/2);
```

The next step is a set of functions to do arithmetic on points. For instance,

```
/* addpoints: add two points */
struct addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

Here both the arguments and the return value are structures. We incremented the components in `p1` rather than using an explicit temporary variable to emphasize that structure parameters are passed by value like any others. As another example, the function `ptinrect` tests whether a point is inside a rectangle, where we have adopted the convention that a rectangle includes its left and bottom sides but not its top and right sides:

```
/* ptinrect: return 1 if p in r, 0 if not */
int ptinrect(struct point p, struct rect r)
```

```
{
    return p.x >= r.pt1.x && p.x < r.pt2.x
    && p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

This assumes that the rectangle is presented in a standard form where the pt1 coordinates are less than the pt2 coordinates. The following function returns a rectangle guaranteed to be in canonical form:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))
/* canonrect: canonicalize coordinates of rectangle */
struct rect canonrect(struct rect r)
{
    struct rect temp;
    temp.pt1.x = min(r.pt1.x, r.pt2.x);
    temp.pt1.y = min(r.pt1.y, r.pt2.y);
    temp.pt2.x = max(r.pt1.x, r.pt2.x);
    temp.pt2.y = max(r.pt1.y, r.pt2.y);
    return temp;
}
```

If a large structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure. Structure pointers are just like pointers to ordinary variables. The declaration

```
struct point *pp;
```

says that pp is a pointer to a structure of type struct point. If pp points to a point structure, *pp is the structure, and (*pp).x and (*pp).y are the members. To use pp, we might write, for example,

```
struct point origin, *pp;
pp = &origin;
printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
```

The parentheses are necessary in (*pp).x because the precedence of the structure member operator . is higher then *. The expression *pp.x means *(pp.x), which is illegal here because x is not a pointer. Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If p is a pointer to a structure, then

```
p->member-of-structure
```

refers to the particular member. So we could write instead

```
printf("origin is (%d,%d)\n", pp->x, pp->y);
```

Both `.` and `->` associate from left to right, so if we have

```
struct rect r, *rp = &r;
```

then these four expressions are equivalent:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

The structure operators `.` and `->`, together with `()` for function calls and `[]` for subscripts, are at the top of the precedence hierarchy and thus bind very tightly. For example, given the declaration

```
struct {
    int len;
    char *str;
} *p;
```

then

```
++p->len
```

increments `len`, not `p`, because the implied parenthesization is `++(p->len)`. Parentheses can be used to alter binding: `(++p)->len` increments `p` before accessing len, and `(p++)->len` increments `p` afterward. (This last set of parentheses is unnecessary.) In the same way, `*p->str` fetches whatever `str` points to; `*p->str++` increments `str` after accessing whatever it points to (just like `*s++`); `(*p->str)++` increments whatever `str` points to; and `*p++->str` increments `p` after accessing whatever `str` points to.

6.3 Arrays of Structures

Consider writing a program to count the occurrences of each C keyword. We need an array of character strings to hold the names, and an array of integers for the counts. One possibility is to use two parallel arrays, `keyword` and `keycount`, as in

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

But the very fact that the arrays are parallel suggests a different organization, an array of structures. Each `keyword` is a pair:

```
char *word;
int cout;
```

and there is an array of pairs. The structure declaration

```
struct key {
    char *word;
    int count;
} keytab[NKEYS];
```

declares a structure type `key`, defines an array `keytab` of structures of this type, and sets aside storage for them. Each element of the array is a structure. This could also be written

```
struct key {
    char *word;
    int count;
};
struct key keytab[NKEYS];
```

Since the structure `keytab` contains a constant set of names, it is easiest to make it an external variable and initialize it once and for all when it is defined. The structure initialization is analogous to earlier ones – the definition is followed by a list of initializers enclosed in braces:

```
struct key {
    char *word;
    int count;
} keytab[] = {
    "auto", 0,
    "break", 0,
    "case", 0,
    "char", 0,
    "const", 0,
    "continue", 0,
    "default", 0,
    /* ... */
```

```
        "unsigned", 0,
        "void", 0,
        "volatile", 0,
        "while", 0
    };
```

The initializers are listed in pairs corresponding to the structure members. It would be more precise to enclose the initializers for each “row” or structure in braces, as in

```
    { "auto", 0 },
    { "break", 0 },
    { "case", 0 },
    ...
```

but inner braces are not necessary when the initializers are simple variables or character strings, and when all are present. As usual, the number of entries in the array `keytab` will be computed if the initializers are present and the `[]` is left empty.

The keyword counting program begins with the definition of `keytab`. The main routine reads the input by repeatedly calling a function `getword` that fetches one word at a time. Each word is looked up in `keytab` with a version of the binary search function that we wrote in Chapter 3. The list of keywords must be sorted in increasing order in the table.

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

    #define MAXWORD 100
    int getword(char *, int);
    int binsearch(char *, struct key *, int);

    /* count C keywords */
    main()
    {
        int n;
        char word[MAXWORD];
        while (getword(word, MAXWORD) != EOF)
```

```
            if (isalpha(word[0]))
                if ((n = binsearch(word, keytab, NKEYS)) >= ↩
0)
                    keytab[n].count++;
        for (n = 0; n < NKEYS; n++)
        if (keytab[n].count > 0)
            printf("%4d %s\n", keytab[n].count, keytab[n].↩
word);
        return 0;
    }

    /* binsearch: find word in tab[0]...tab[n-1] */
    int binsearch(char *word, struct key tab[], int n)
    {
        int cond;
        int low, high, mid;
        low = 0;
        high = n - 1;
        while (low <= high) {
            mid = (low+high) / 2;
            if ((cond = strcmp(word, tab[mid].word)) < 0)
                high = mid - 1;
            else if (cond > 0)
                low = mid + 1;
            else
                return mid;
            }
        return -1;
    }
```

We will show the function `getword` in a moment; for now it suffices to say that each call to `getword` finds a word, which is copied into the array named as its first argument.

The quantity `NKEYS` is the number of keywords in `keytab`. Although we could count this by hand, it's a lot easier and safer to do it by machine, especially if the list is subject to change. One possibility would be to terminate the list of initializers with a null pointer, then loop along `keytab` until the end is found.

But this is more than is needed, since the size of the array is completely determined at compile time. The size of the array is the size of one entry times the number of entries, so the number of entries is just

size of keytab / size of struct key

C provides a compile-time unary operator called `sizeof` that can be used to compute the size of any object. The expressions

sizeof object

and

sizeof (type name)

yield an integer equal to the size of the specified object or type in bytes. (Strictly, `sizeof` produces an unsigned integer value whose type, `size_t`, is defined in the header `<stddef.h>`.) An object can be a variable or array or structure. A type name can be the name of a basic type like `int` or `double`, or a derived type like a structure or a pointer.

In our case, the number of keywords is the size of the array divided by the size of one element. This computation is used in a `#define` statement to set the value of `NKEYS`:

```
#define NKEYS (sizeof keytab / sizeof(struct key))
```

Another way to write this is to divide the array size by the size of a specific element:

```
#define NKEYS (sizeof keytab / sizeof(keytab[0]))
```

This has the advantage that it does not need to be changed if the type changes.

A `sizeof` can not be used in a `#if` line, because the preprocessor does not parse type names. But the expression in the `#define` is not evaluated by the preprocessor, so the code here is legal.

Now for the function `getword`. We have written a more general `getword` than is necessary for this program, but it is not complicated. `getword` fetches the next "word" from the input, where a word is either a string of letters and digits beginning with a letter, or a single non-white space character. The function value is the first character of the word, or `EOF` for end of file, or the character itself if it is not alphabetic.

```
/* getword: get next word or character from input */
```

```
int getword(char *word, int lim)
{
    int c, getch(void);
    void ungetch(int);
    char *w = word;
    while (isspace(c = getch()))
        ;
    if (c != EOF)
        *w++ = c;
    if (!isalpha(c)) {
        *w = '\0';
        return c;
    }
    for ( ; --lim > 0; w++)
    if (!isalnum(*w = getch())) {
        ungetch(*w);
        break;
    }
    *w = '\0';
    return word[0];
}
```

getword uses the getch and ungetch that we wrote in Chapter 4. When the collection of an alphanumeric token stops, getword has gone one character too far. The call to ungetch pushes that character back on the input for the next call. getword also uses isspace to skip whitespace, isalpha to identify letters, and isalnum to identify letters and digits; all are from the standard header <ctype.h>.

Exercise 6.1 — Our version of getword does not properly handle underscores, string constants, comments, or preprocessor control lines. Write a better version.

6.4 Pointers to Structures

To illustrate some of the considerations involved with pointers to and arrays of structures, let us write the keyword-counting program again, this time using pointers instead of array indices. The external declaration of keytab need not change, but `main` and `binsearch` do need modification.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

int getword(char *, int);
struct key *binsearch(char *, struct key *, int);

/* count C keywords; pointer version */
main()
{
    char word[MAXWORD];
    struct key *p;
    while (getword(word, MAXWORD) != EOF)
    if (isalpha(word[0]))
        if ((p=binsearch(word, keytab, NKEYS)) != NULL)
            p->count++;
    for (p = keytab; p < keytab + NKEYS; p++)
        if (p->count > 0)
            printf("%4d %s\n", p->count, p->word);
    return 0;
}

/* binsearch: find word in tab[0]...tab[n-1] */
struct key *binsearch(char *word, struck key *tab, int n)
{
    int cond;
    struct key *low = &tab[0];
    struct key *high = &tab[n];
    struct key *mid;
    while (low < high) {
        mid = low + (high-low) / 2;
        if ((cond = strcmp(word, mid->word)) < 0)
            high = mid;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return NULL;
}
```

There are several things worthy of note here. First, the declaration of `binsearch` must indicate that it returns a pointer to `struct` key instead of an integer; this is declared both in the function prototype and in `binsearch`. If `binsearch` finds the word, it returns a pointer to it; if it fails, it returns `NULL`. Second, the elements of `keytab` are now accessed by pointers. This requires significant changes in `binsearch`.

The initializers for `low` and `high` are now pointers to the beginning and just past the end of the table. The computation of the middle element can no longer be simply

```
mid = (low+high) / 2  /* WRONG */
```

because the addition of pointers is illegal. Subtraction is legal, however, so high-low is the number of elements, and thus

```
mid = low + (high-low) / 2
```

sets `mid` to the element halfway between `low` and `high`.

The most important change is to adjust the algorithm to make sure that it does not generate an illegal pointer or attempt to access an element outside the array. The problem is that `&tab[-1]` and `&tab[n]` are both outside the limits of the array tab. The former is strictly illegal, and it is illegal to dereference the latter. The language definition does guarantee, however, that pointer arithmetic that involves the first element beyond the end of an array (that is, `&tab[n]`) will work correctly. In `main` we wrote

```
for (p = keytab; p < keytab + NKEYS; p++)
```

If `p` is a pointer to a structure, arithmetic on `p` takes into account the size of the structure, so `p++` increments `p` by the correct amount to get the next element of the array of structures, and the test stops the loop at the right time.

Don't assume, however, that the size of a structure is the sum of the sizes of its members. Because of alignment requirements for different objects, there may be unnamed "holes" in a structure. Thus, for instance, if a `char` is one byte and an `int` four bytes, the structure

```
struct {
    char c;
    int i;
};
```

might well require eight bytes, not five. The `sizeof` operator returns the proper value. Finally, an aside on program format: when a function returns a complicated type like a

```
structure pointer,
```

as in

```
struct key *binsearch(char *word, struct key *tab, int n)
```

the function name can be hard to see, and to find with a text editor. Accordingly an alternate style is sometimes used:

```
struct key *
binsearch(char *word, struct key *tab, int n)
```

This is a matter of personal taste; pick the form you like and hold to it.

6.5 Self-referential Structures

Suppose we want to handle the more general problem of counting the occurrences of all the words in some input. Since the list of words isn't known in advance, we can't conveniently sort it and use a binary search. Yet we can't do a linear search for each word as it arrives, to see if it's already been seen; the program would take too long. (More precisely, its running time is likely to grow quadratically with the number of input words.)

How can we organize the data to copy efficiently with a list or arbitrary words? One solution is to keep the set of words seen so far sorted at all times, by placing each word into its proper position in the order as it arrives. This shouldn't be done by shifting words in a linear array, though – that also takes too long. Instead we will use a data structure called a *binary tree.*

The tree contains one "node" per distinct word; each node contains

- A pointer to the text of the word,
- A count of the number of occurrences,
- A pointer to the left child node,
- A pointer to the right child node.

No node may have more than two children; it might have only zero or one.

The nodes are maintained so that at any node the left subtree contains only words that are lexicographically less than the word at the node, and the right subtree contains only words that are greater. This is the tree for the sentence "now is the time for all good men to come to the aid of their party", as built by inserting each word as it is encountered:

To find out whether a new word is already in the tree, start at the root and compare the new word to the word stored at that node. If they match, the question is answered affirmatively. If the new record is less than the tree word, continue searching at the left child, otherwise at the right child. If there is no child in the required direction, the new word is not in the tree, and in fact the empty slot is the proper place to add the new word. This process is recursive, since the search from any node uses a search from one of its children. Accordingly, recursive routines for insertion and printing will be most natural.

Going back to the description of a node, it is most conveniently represented as a structure with four components:

```
struct tnode {              /* the tree node: */
    char *word;             /* points to the text */
    int count;              /* number of occurrences */
    struct tnode *left;     /* left child */
    struct tnode *right;    /* right child */
};
```

This recursive declaration of a node might look chancy, but it's correct. It is illegal for a structure to contain an instance of itself, but

```
struct tnode *left;
```

declares left to be a pointer to a `tnode`, not a `tnode` itself

Occasionally, one needs a variation of self-referential structures: two structures that refer to each other. The way to handle this is:

```
struct t {
    ...
    struct s *p; /* p points to an s */
};
struct s {
    ...
    struct t *q; /* q points to a t */
    };
```

The code for the whole program is surprisingly small, given a handful of supporting routines like `getword` that we have already written. The `main` routine reads words with `getword` and installs them in the tree with `addtree`.

```
    #include <stdio.h>
    #include <ctype.h>
    #include <string.h>
    #define MAXWORD 100

    struct tnode *addtree(struct tnode *, char *);
    void treeprint(struct tnode *);
    int getword(char *, int);

    /* word frequency count */
    main()
    {
        struct tnode *root;
        char word[MAXWORD];
        root = NULL;
        while (getword(word, MAXWORD) != EOF)
            if (isalpha(word[0]))
                root = addtree(root, word);
        treeprint(root);
        return 0;
    }
```

The function `addtree` is recursive. A word is presented by main to the top level (the root) of the tree. At each stage, that word is compared to the word already stored at the node, and is percolated down to either the left or right subtree by a recursive call to `adtree`. Eventually, the word either matches something already in the tree (in which case the count is incremented), or a null pointer is encountered, indicating that a node must be created and added to the tree. If a new node is created, `addtree` returns a pointer to it, which is installed in the parent node.

```
    struct tnode *talloc(void);
    char *strdup(char *);
```

```
/* addtree: add a node with w, at or below p */
struct treenode *addtree(struct tnode *p, char *w)
{
    int cond;
    if (p == NULL) {    /* a new word has arrived */
        p = talloc();   /* make a new node */
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;     /* repeated word */
    else if (cond < 0)  /* less than into left subtree */
        p->left = addtree(p->left, w);
    else            /* greater than into right subtree */
        p->right = addtree(p->right, w);
    return p;
}
```

Storage for the new node is fetched by a routine `talloc`, which returns a pointer to a free space suitable for holding a tree node, and the new word is copied into a hidden space by `strdup`. (We will discuss these routines in a moment.) The count is initialized, and the two children are made null. This part of the code is executed only at the leaves of the tree, when a new node is being added. We have (unwisely) omitted error checking on the values returned by `strdup` and `talloc`.

`treeprint` prints the tree in sorted order; at each node, it prints the left subtree (all the words less than this word), then the word itself, then the right subtree (all the words greater). If you feel shaky about how recursion works, simulate `treeprint` as it operates on the tree shown above.

```
/* treeprint: in-order print of tree p */
void treeprint(struct tnode *p)
{
    if (p != NULL) {
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}
```

A practical note: if the tree becomes “unbalanced” because the words don’t arrive in random order, the running time of the program can grow too much. As a worst case, if the words are already in order, this program does an expensive simulation of linear search. There are generalizations of the binary tree that do not suffer from this worst-case behavior, but we will not describe them here.

Before leaving this example, it is also worth a brief digression on a problem related to storage allocators. Clearly it’s desirable that there be only one storage allocator in a program, even though it allocates different kinds of objects. But if one allocator is to process requests for, say, pointers to chars and pointers to struct tnodes, two questions arise. First, how does it meet the requirement of most real machines that objects of certain types must satisfy alignment restrictions (for example, integers often must be located at even addresses)? Second, what declarations can cope with the fact that an allocator must necessarily return different kinds of pointers?

Alignment requirements can generally be satisfied easily, at the cost of some wasted space, by ensuring that the allocator always returns a pointer that meets all alignment restrictions. The `alloc` of Chapter 5 does not guarantee any particular alignment, so we will use the standard library function `malloc`, which does. In Chapter 8 we will show one way to implement `malloc`.

The question of the type declaration for a function like `malloc` is a vexing one for any language that takes its type-checking seriously. In C, the proper method is to declare that `malloc` returns a pointer to `void`, then explicitly coerce the pointer into the desired type with a cast. `malloc` and related routines are declared in the standard header `<stdlib.h>`. Thus talloc can be written as

```
#include <stdlib.h>

/* talloc: make a tnode */
struct tnode *talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}
```

`strdup` merely copies the string given by its argument into a safe place, obtained by a call on `malloc`:

```
char *strdup(char *s) /* make a duplicate of s */
```

```
{
char *p;
p = (char *) malloc(strlen(s)+1); /* +1 for '\0' */
if (p != NULL)
strcpy(p, s);
return p;
}
```

`malloc` returns `NULL` if no space is available; `strdup` passes that value on, leaving errorhandling to its caller. Storage obtained by calling `malloc` may be freed for re-use by calling `free`; see Chapters 8 and 7.

Exercise 6.2 — Write a program that reads a C program and prints in alphabetical order each group of variable names that are identical in the first 6 characters, but different somewhere thereafter. Don't count words within strings and comments. Make 6 a parameter that can be set from the command line.

Exercise 6.3 — Write a cross-referencer that prints a list of all words in a document, and for each word, a list of the line numbers on which it occurs. Remove noise words like "the," "and," and so on.

Exercise 6.4 — Write a program that prints the distinct words in its input sorted into decreasing order of frequency of occurrence. Precede each word by its count.

6.6 Table Lookup

In this section we will write the innards of a table-lookup package, to illustrate more aspects of structures. This code is typical of what might be found in the symbol table management routines of a macro processor or a compiler. For example, consider the `#define` statement. When a line like

```
#define IN 1
```

is encountered, the name `IN` and the replacement text `1` are stored in a table. Later, when the name `IN` appears in a statement like

```
state = IN;
```

it must be replaced by 1.

There are two routines that manipulate the names and replacement texts. `install(s,t)` records the name `s` and the replacement text `t` in a table; `s` and `t` are just character strings. `lookup(s)` searches for `s` in the table, and returns a pointer to the place where it was found, or `NULL` if it wasn't there.

The algorithm is a hash-search – the incoming name is converted into a small non-negative integer, which is then used to index into an array of pointers. An array element points to the beginning of a linked list of blocks describing names that have that hash value. It is `NULL` if no names have hashed to that value.

A block in the list is a structure containing pointers to the name, the replacement text, and the next block in the list. A null next-pointer marks the end of the list.

```
struct nlist {      /* table entry: */
struct nlist *next; /* next entry in chain */
    char *name;     /* defined name */
    char *defn;     /* replacement text */
};
```

The pointer array is just

```
#define HASHSIZE 101
static struct nlist *hashtab[HASHSIZE]; /* pointer table ↩
*/
```

The hashing function, which is used by both `lookup` and `install`, adds each character value in the string to a scrambled combination of the previous ones and returns the remainder modulo the array size. This is not the best possible hash function, but it is short and effective.

```
/* hash: form hash value for string s */
unsigned hash(char *s)
{
    unsigned hashval;
    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + 31 * hashval;
    return hashval % HASHSIZE;
}
```

Unsigned arithmetic ensures that the hash value is non-negative. The hashing process produces a starting index in the array `hashtab`; if the string is to be found anywhere, it will be in the list of blocks beginning there. The search is performed by `lookup`. If `lookup` finds the entry already present, it returns a pointer to it; if not, it returns `NULL`.

```
/* lookup: look for s in hashtab */
struct nlist *lookup(char *s)
{
struct nlist *np;
for (np = hashtab[hash(s)]; np != NULL; np = np->next)
    if (strcmp(s, np->name) == 0)
        return np;  /* found */
return NULL;    /* not found */
}
```

The for loop in lookup is the standard idiom for walking along a linked list:

```
for (ptr = head; ptr != NULL; ptr = ptr->next)
...
```

`install` uses `lookup` to determine whether the name being installed is already present; if so, the new definition will supersede the old one. Otherwise, a new entry is created. `install` returns `NULL` if for any reason there is no room for a new entry.

```
struct nlist *lookup(char *);
char *strdup(char *);

/* install: put (name, defn) in hashtab */
struct nlist *install(char *name, char *defn)
{
    struct nlist *np;
    unsigned hashval;
    if ((np = lookup(name)) == NULL) { /* not found */
        np = (struct nlist *) malloc(sizeof(*np));
        if (np == NULL || (np->name = strdup(name)) == ↩
NULL)
            return NULL;
        hashval = hash(name);
        np->next = hashtab[hashval];
        hashtab[hashval] = np;
    } else          /* already there */
```

```
            free((void *) np->defn); /*free previous defn */
    if ((np->defn = strdup(defn)) == NULL)
        return NULL;
    return np;
    }
```

Exercise 6.5 — Write a function `undef` that will remove a name and definition from the table maintained by `lookup` and `install`.

Exercise 6.6 — Implement a simple version of the `#define` processor (i.e., no arguments) suitable for use with C programs, based on the routines of this section. You may also find `getch` and `ungetch` helpful.

6.7 Typedef

C provides a facility called typedef for creating new data type names. For example, the declaration

```
    typedef int Length;
```

makes the name `Length` a synonym for `int`. The type `Length` can be used in declarations, casts, etc., in exactly the same ways that the `int` type can be:

```
    Length len, maxlen;
    Length *lengths[];
```

Similarly, the declaration

```
    typedef char *String;
```

makes `String` a synonym for `char *` or character pointer, which may then be used in declarations and casts:

```
    String p, lineptr[MAXLINES], alloc(int);
    int strcmp(String, String);
    p = (String) malloc(100);
```

Notice that the type being declared in a `typedef` appears in the position of a variable name, not right after the word `typedef`. Syntactically, `typedef` is like the storage classes extern, static, etc. We have used capitalized names for `typedef`s, to make them stand out. As a more complicated example, we could make `typedef`s for the tree nodes shown earlier in this chapter:

```
typedef struct tnode *Treeptr;
typedef struct tnode {  /* the tree node: */
char *word;             /* points to the text */
int count;              /* number of occurrences */
struct tnode *left;     /* left child */
struct tnode *right;    /* right child */
} Treenode;
```

This creates two new type keywords called `Treenode` (a structure) and `Treeptr` (a pointer to the structure). Then the routine talloc could become

```
Treeptr talloc(void)
{
    return (Treeptr) malloc(sizeof(Treenode));
}
```

It must be emphasized that a typedef declaration does not create a new type in any sense; it merely adds a new name for some existing type. Nor are there any new semantics: variables declared this way have exactly the same properties as variables whose declarations are spelled out explicitly. In effect, `typedef` is like `#define`, except that since it is interpreted by the compiler, it can cope with textual substitutions that are beyond the capabilities of the preprocessor. For example,

```
typedef int (*PFI)(char *, char *);
```

creates the type `PFI`, for "pointer to function (of two `char *` arguments) returning `int`," which can be used in contexts like

```
PFI strcmp, numcmp;
```

in the sort program of Chapter 5.

Besides purely aesthetic issues, there are two main reasons for using `typedef`s. The first is to parameterize a program against portability problems. If `typedef`s are used for data types that may be machine-dependent, only the `typedef`s need change when the program is moved. One common situation is to use `typedef` names for various integer quantities, then make an appropriate set of choices of `short`, `int`, and `long` for each host machine. Types like `size_t` and `ptrdiff_t` from the standard library are examples.

The second purpose of `typedef`s is to provide better documentation for a program – a type called `Treeptr` may be easier to understand than one declared only as a pointer to a complicated structure.

6.8 Unions

A `union` is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements. Unions provide a way to manipulate different kinds of data in a single area of storage, without embedding any machinedependent information in the program. They are analogous to variant records in pascal.

As an example such as might be found in a compiler symbol table manager, suppose that a constant may be an `int`, a `float`, or a character pointer. The value of a particular constant must be stored in a variable of the proper type, yet it is most convenient for table management if the value occupies the same amount of storage and is stored in the same place regardless of its type. This is the purpose of a `union` – a single variable that can legitimately hold any of one of several types. The syntax is based on structures:

```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

The variable `u` will be large enough to hold the largest of the three types; the specific size is implementation-dependent. Any of these types may be assigned to `u` and then used in expressions, so long as the usage is consistent: the type retrieved must be the type most recently stored. It is the programmer's responsibility to keep track of which type is currently stored in a union; the results are implementation-dependent if something is stored as one type and extracted as another. Syntactically, members of a union are accessed as

`union-name.member`

or

`union-pointer->member`

just as `for` structures. If the variable `utype` is used to keep track of the current type stored in `u`, then one might see code such as

```
if (utype == INT)
    printf("%d\n", u.ival);
if (utype == FLOAT)
```

```
        printf("%f\n", u.fval);
    if (utype == STRING)
        printf("%s\n", u.sval);
    else
        printf("bad type %d in utype\n", utype);
```

Unions may occur within structures and arrays, and vice versa. The notation for accessing a member of a union in a structure (or vice versa) is identical to that for nested structures. For example, in the structure array defined by

```
    struct {
        char *name;
        int flags;
        int utype;
        union {
            int ival;
            float fval;
            char *sval;
        } u;
    } symtab[NSYM];
```

the member `ival` is referred to as

```
    symtab[i].u.ival
```

and the first character of the string `sval` by either of

```
    *symtab[i].u.sval
    symtab[i].u.sval[0]
```

In effect, a `union` is a structure in which all members have offset zero from the base, the structure is big enough to hold the "widest" member, and the alignment is appropriate for all of the types in the union. The same operations are permitted on unions as on structures: assignment to or copying as a unit, taking the address, and accessing a member.

A `union` may only be initialized with a value of the type of its first member; thus `union u` described above can only be initialized with an integer value. The storage allocator in Chapter 8 shows how a `union` can be used to force a variable to be aligned on a particular kind of storage boundary.

6.9 Bit-fields

When storage space is at a premium, it may be necessary to pack several objects into a single machine word; one common use is a set of single-bit flags in applications like compiler symbol tables. Externally-imposed data formats, such as interfaces to hardware devices, also often require the ability to get at pieces of a word.

Imagine a fragment of a compiler that manipulates a symbol table. Each identifier in a program has certain information associated with it, for example, whether or not it is a keyword, whether or not it is external and/or `static`, and so on. The most compact way to encode such information is a set of one-bit flags in a single `char` or `int`. The usual way this is done is to define a set of "masks" corresponding to the relevant bit positions, as in

```
#define KEYWORD 01
#define EXTRENAL 02
#define STATIC 04
```

or

```
enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

The numbers must be powers of two. Then accessing the bits becomes a matter of "bitfiddling" with the shifting, masking, and complementing operators that were described in Chapter 2. Certain idioms appear frequently:

```
flags |= EXTERNAL | STATIC;
```

turns on the `EXTERNAL` and `STATIC` bits in flags, while

```
flags &= ~(EXTERNAL | STATIC);
```

turns them off, and

```
if ((flags & (EXTERNAL | STATIC)) == 0) ...
```

is true if both bits are off.

Although these idioms are readily mastered, as an alternative C offers the capability of defining and accessing fields within a word directly rather than by bitwise logical operators. A *bit-field*, or *field* for short, is a set of adjacent bits within a single implementation-defined storage unit that we will call a "word." For example, the symbol table `#defines` above could be replaced by the definition of three fields:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern : 1;
    unsigned int is_static : 1;
} flags;
```

This defines a variable table called `flags` that contains three 1-bit fields. The number following the colon represents the field width in bits. The fields are declared `unsigned int` to ensure that they are unsigned quantities.

Individual fields are referenced in the same way as other structure members: `flags.is_keyword`, `flags.is_extern`, etc. Fields behave like small integers, and may participate in arithmetic expressions just like other integers. Thus the previous examples may be written more naturally as

```
flags.is_extern = flags.is_static = 1;
```

to turn the bits on;

```
flags.is_extern = flags.is_static = 0;
```

to turn them off; and

```
if (flags.is_extern == 0 && flags.is_static == 0)
...
```

to test them.

Almost everything about fields is implementation-dependent. Whether a field may overlap a word boundary is implementation-defined. Fields need not be names; unnamed fields (a colon and width only) are used for padding. The special width 0 may be used to force alignment at the next word boundary.

Fields are assigned left to right on some machines and right to left on others. This means that although fields are useful for maintaining internally-defined data structures, the question of which end comes first has to be carefully considered when picking apart externally-defined data; programs that depend on such things are not portable. Fields may be declared only as `int`s; for portability, specify `signed` or `unsigned` explicitly. They are not arrays and they do not have addresses, so the `&` operator cannot be applied on them.

Appendix A

Reference Manual

A.13 Grammar

```
translation-unit:
    external-declaration
    translation-unit external-declaration

external-declaration:
    function-definition
    declaration

function-definition:
    declaration-specifiers_{opt} declarator declaration↩
-list_{opt} compound-statement

declaration:
    declaration-specifiers init-declarator-list_{opt};

declaration-list:
    declaration
    declaration-list declaration

declaration-specifiers:
    storage-class-specifier declaration-specifiersopt
    type-specifier declaration-specifiersopt
```

```
     type-qualifier declaration-specifiersopt

storage-class specifier: one of
     auto register static extern typedef

type specifier: one of
     void char short int long float double signed
     unsigned struct-or-union-specifier enum-specifier ↩
typedef-name

type-qualifier: one of
     const volatile

struct-or-union-specifier:
     struct-or-union identifieropt { struct-declaration-↩
list }
     struct-or-union identifier

struct-or-union: one of
     struct union

struct-declaration-list:
     struct declaration
     struct-declaration-list struct declaration

init-declarator-list:
     init-declarator
     init-declarator-list, init-declarator

init-declarator:
     declarator
     declarator = initializer

struct-declaration:
     specifier-qualifier-list struct-declarator-list;

specifier-qualifier-list:
     type-specifier specifier-qualifier-listopt
```

```
    type-qualifier specifier-qualifier-listopt

struct-declarator-list:
    struct-declarator
    struct-declarator-list, struct-declarator

struct-declarator:
    declarator
    declaratoropt : constant-expression

enum-specifier:
    enum identifieropt { enumerator-list }
    enum identifier

enumerator-list:
    enumerator
    enumerator-list , enumerator

enumerator:
    identifier
    identifier = constant-expression

declarator:
pointeropt direct-declarator

direct-declarator:
    identifier
    (declarator)
    direct-declarator [ constant-expressionopt ]
    direct-declarator ( parameter-type-list )
    direct-declarator ( identifier-listopt )

pointer:
    * type-qualifier-listopt
    * type-qualifier-listopt pointer
    type-qualifier-list:
    type-qualifier
    type-qualifier-list type-qualifier
```

```
parameter-type-list:
    parameter-list
    parameter-list , ...

parameter-list:
    parameter-declaration
    parameter-list , parameter-declaration

parameter-declaration:
    declaration-specifiers declarator
    declaration-specifiers abstract-declaratoropt

identifier-list:
    identifier
    identifier-list , identifier

initializer:
    assignment-expression
    { initializer-list }
    { initializer-list , }

initializer-list:
    initializer
    initializer-list , initializer

type-name:
    specifier-qualifier-list abstract-declaratoropt

abstract-declarator:
    pointer
    pointeropt direct-abstract-declarator

direct-abstract-declarator:
    ( abstract-declarator )
    direct-abstract-declaratoropt [constant-↩
expressionopt]
    direct-abstract-declaratoropt (parameter-type-↩
listopt)

 typedef-name:
     identifier

 statement:
     labeled-statement
     expression-statement
     compound-statement
     selection-statement
     jump-statement

 labeled-statement:
     identifier : statement
     case constant-expression : statement
     default : statement

 expression-statement:
     expressionopt;

 compound-statement:
 { declaration-listopt statement-listopt }

 statement-list:
     statement
     statement-list statement

 selection-statement:
     if (expression) statement
     if (expression) statement else statement
     switch (expression) statement

 iteration-statement:
     while (expression) statement
     do statement while (expression);
     for (expressionopt; expressionopt; expressionopt) ↩
statement
```

```
 jump-statement:
     goto identifier;
     continue;
     break;
     return expressionopt;

 expression:
     assignment-expression
     expression , assignment-expression

 assignment-expression:
     conditional-expression
     unary-expression assignment-operator assignment-↩
expression
     assignment-operator: one of
     = *= /= %= += -= <<= >>= &= ^= |=

 conditional-expression:
     logical-OR-expression
     logical-OR-expression ? expression : conditional-↩
expression

 constant-expression:
     conditional-expression

 logical-OR-expression:
     logical-AND-expression
     logical-OR-expression || logical-AND-expression

 logical-AND-expression:
     inclusive-OR-expression
     logical-AND-expression && inclusive-OR-expression

 inclusive-OR-expression:
     exclusive-OR-expression
     inclusive-OR-expression | exclusive-OR-expression

 exclusive-OR-expression:
     AND-expression
     exclusive-OR-expression ^ AND-expression

 AND-expression:
     equality-expression
     AND-expression & equality-expression

 equality-expression:
     relational-expression
     equality-expression == relational-expression
     equality-expression != relational-expression

 relational-expression:
     shift-expression
     relational-expression < shift-expression
     relational-expression > shift-expression
     relational-expression <= shift-expression
     relational-expression >= shift-expression

 shift-expression:
     additive-expression
     shift-expression << additive-expression
     shift-expression >> additive-expression

 additive-expression:
     multiplicative-expression
     additive-expression + multiplicative-expression
     additive-expression - multiplicative-expression
     multiplicative-expression:
     multiplicative-expression * cast-expression
     multiplicative-expression / cast-expression
     multiplicative-expression % cast-expression

 cast-expression:
     unary expression
     (type-name) cast-expression

 unary-expression:
     postfix expression
     ++unary expression
     --unary expression
     unary-operator cast-expression
     sizeof unary-expression
     sizeof (type-name)
     unary operator: one of
     & * + - ~ !

postfix-expression:
     primary-expression
     postfix-expression[expression]
     postfix-expression(argument-expression-listopt)
     postfix-expression.identifier
     postfix-expression->+identifier
     postfix-expression++
     postfix-expression--

primary-expression:
     identifier
     constant
     string
     (expression)

argument-expression-list:
     assignment-expression
     assignment-expression-list , assignment-expression

constant:
     integer-constant
     character-constant
     floating-constant
     enumeration-constant

control-line:
# define identifier token-sequence
# define identifier(identifier, ... , identifier) token↩
-sequence
```

```
# undef identifier
# include <filename>
# include "filename"
# line constant "filename"
# line constant
# error token-sequenceopt
# pragma token-sequenceopt
#

preprocessor-conditional:
if-line text elif-parts else-partopt #endif
if-line:
# if constant-expression
# ifdef identifier
# ifndef identifier
elif-parts:
elif-line text
elif-partsopt
elif-line:
# elif constant-expression
else-part:
else-line text
else-line:
#else
```