Tom Lyon

THE C PROGRAMMING LANGUAGE

Brian W. Kernighan

Dennis M. Ritchie

Bell Telephone Laboratories, Incorporated
Murray Hill, New Jersey

DRAFT VERSION 1.
Comments on this book are solicited.

-0-: PREFACE

C is a general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators. C is not a 'very high level' language, nor a 'big' one and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.

This book aims to teach how to program in C. Most of the treatment is based on reading, writing and revising examples, rather than on bald statements of rules. For the most part, the examples are complete, real programs, rather than artificial ones, although they are of necessity not very big. Besides howing how to make effective use of the language, we have also tried where possible to show useful algorithms and principles of good style and sound design.

The book is not an introductory programming manual; it assumes some familiarity with basic programming concepts like variables, assignment statements, loops, and subroutines. Nonetheless, an absolutely novice programmer should be able to read along and pick up the language, although access to a more knowledgeable colleague will help.

C was originally designed for and implemented on the UNIX operating system on the DEC PDP-11; the operating system, the compiler, and essentially all UNIX applications programs are written in C. Production compilers also exist for the IBM System/370, the Honeywell 6000, and the Interdata 8-32, with preliminary versions on several other machines. C is not tied to any particular system and it is readily possible to write portable C programs.

Since C is an evolving language that exists on a variety of systems, some of the material in this book may be incorrect for a particular system. We have tried to steer clear of such problems, and to warn of the inevitable difficulties. When in doubt, however, we have generally chosen to describe the situation on UNIX, since that is the environment of the majority of C programmers.

CHAPTER 1: A TUTORIAL INTRODUCTION

Let us begin with a quick introduction to C. Our aim is to show the essential elements of the language in real programs, but without getting bogged down in details, formal 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 quite intentionally leaving out of this chapter features of C which are of vital importance for writing bigger programs. These include pointers, structures, most of C's rich set of operators, several control flow statements, and myriad details.

This approach has its drawbacks, of course. Most notable is that the complete story on any particular language feature is not found in a single place, and the tutorial, by being brief, may also mislead. We have tried to minimize this effect, 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 really learn a new programming language is by writing programs in it. And the first program is the same for all languages:

> Print upon the normal output the words
> hello, world

This is the basic 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

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

Just how to run this program depends on the system you are using. As an specific example, on Unix you must create the source program on a file whose name ends in ".c", such as *hello.c*, then compile it with the *cc* 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*. Running that by the command

```
a.out
```

will produce

```
hello, world
```

as its output.

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

Exercise 1-1: Run this program on your system. Experiment with leaving out parts of the program, to see what error messages you get. □

Now for some explanations about the program itself. A C program, whatever its size, consists of one or more "functions" which specify the actual computing operations that are to be done. C functions are similar to the functions and subroutines of a Fortran program or the procedures of PL/I, Pascal, etc. main is such a function. Normally you are at liberty to give functions whatever names you like, but main is a special name — your program begins executing at the beginning of main. This means that every program *must* have a main somewhere. main will usually invoke other functions to perform its job, some coming from the same program, and others from libraries of previously written functions.

One method of communicating data between functions is by arguments. The parentheses following the function name surround the argument list; here main is a function of no arguments, indicated by ( ). The braces { } enclose the statements that make up the function. A function is invoked by naming it, followed by a parenthesized list of arguments. There is no CALL statement as there is in Fortran or PL/I. The parentheses must be present even if there are no arguments.

The line that says

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

is a function call, which calls a function named printf, with the argument "hello, world\n". printf is a library function which prints output on the

terminal (unless some other destination is specified). In this case it prints the string of characters that make up its argument.

A sequence of characters enclosed in the double quotes "..." is called a *character string*. For the moment our only use of character strings will be as arguments for printf and other functions. A string may contain any number of any characters.

The sequence \n in the string is C shorthand for the *newline character*, which when printed advances the terminal to the next line. If you leave out the \n (a worthwhile experiment), you will find that your output is not terminated properly by a line feed. The only way to get a newline character into the printf argument is with \n; if you try something like

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

the C compiler will print unfriendly diagnostics about missing quotes.

printf never supplies a newline automatically, so multiple calls may be used to build up an output line in stages. Our first program could just as well have been written

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

to produce an identical output.

Notice that \n represents only a single character. There are several other "escape sequences" like \n for representing hard-to-get or invisible characters, such as \t for tab, \b for backspace, \" for the quote, and \\ for the backslash itself.

Exercise 1-2: Experiment to find out what happens when printf's argument string contains \x, where x is some character not listed above. □

1.2 Variables and Arithmetic

The next program prints the following table of fahrenheit temperatures and their centigrade equivalents, using the formula $c=(5/9)(f-32)$.

```
  0 -17.8
 20  -6.7
 40   4.4
 60  15.6
 80  26.7
100  37.8
120  48.9
140  60.0
160  71.1
180  82.2
```

```
200  93.3
220 104.4
240 115.6
260 126.7
280 137.8
300 148.9
```

Here is the program itself.

```
/* print fahrenheit-centigrade table
      for f = 0, 20, ..., 300 */
main( )
{
      int lower, upper, step;
      float fahr, cent;

      lower = 0;   /* lower limit of temperature table */
      upper = 300;      /* upper limit */
      step = 20;   /* step size */

      fahr = lower;
      while (fahr <= upper) {
            cent = 5.0/9.0 * (fahr-32);
            printf("%4.0f %6.1f\n", fahr, cent);
            fahr = fahr + step;
      }
}
```

The first two lines

```
/* print fahrenheit-centigrade table
      for f = 0, 20, ..., 300 */
```

are a *comment,* which in this case explains briefly what the program does. Any characters between /* and */ are ignored; they may be used freely to make a program easier to understand.

In C, *all* variables must be declared before use, usually at the beginning of the function before any executable statements. If you forget a declaration, you will get a diagnostic from the compiler. A declaration consists of a *type* and a list of variables which have that type, as in

```
int lower, upper, step;
float fahr, cent;
```

The type int implies that the variables listed are *integers*; float stands for *floating point,* i.e., numbers which may have a fractional part. The precision of both int and float depends on the particular machine you are using; on the PDP-11, for instance, an int is a 16 bit number, that is one which lies between −32768 and +32767. A float number is a 32 bit quantity, which amounts to about seven significant digits, with exponents between −38 and +38.

C provides several other basic data types besides int and float, the most common of which are

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

There are also *arrays* and *structures* of these basic types, *pointers* to them, and *functions* that return them, all of which we will meet in due course.

The executable statements in the temperature conversion program begin with the assignments

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

etc., which set the variables to their starting values. Individual statements are terminated by semicolons; no semicolon follows a brace.

To produce a table of many lines, we need a loop; this is the function of the while statement

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

The condition in parentheses is tested. If it is true, the body of the loop (all of the statements enclosed in the braces { and }) is executed. Then the condition is re-tested, and if true, the body is executed again. Eventually when the test becomes false (fahr > 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 a single C statement, like

```
while (...)
        printf(...);
```

or one or more statements enclosed in braces, as in the temperature converter.

The statements controlled by the while are indented by one tab stop so you can see at a glance what the scope of the while is, that is, what statements are inside the loop. The indentation emphasizes the logical structure of the program. Although C is quite permissive about statement positioning, proper indentation and use of white space are critical in making programs easy for people to read. The position of the braces is less important; 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 centigrade temperature is computed and assigned to cent by the statement

```
cent = 5.0/9.0 * (fahr-32);
```

The reason for using 5.0/9.0 instead of the simpler looking 5/9 is that in C, as in many other languages, integer division *truncates*, so any fractional part is discarded. Thus 5/9 is zero and of course so would be all the temperatures. A decimal point in a constant indicates that it is floating point, so 5.0/9.0 is 0.555..., as we want.

Why not use 32.0 instead of 32? Since fahr is a float, 32 is automatically converted to float (to 32.0) before the subtraction can be done. As a matter of style, it's wise to write floating point constants with explicit decimal points even when they have integral values; it emphasizes their floating point nature for human readers, and ensures that the compiler will see things your way too.

This example shows a bit more of how printf works. printf is actually a general-purpose format conversion function, which we'll describe completely in Chapter 7. Its first argument is a string of characters to be printed, with each % sign indicating where one of the other (second, third, ...) arguments is to be substituted, and what form it is to be printed in. In particular, %4.0f says that a floating point number is to be printed in a space at least four characters wide, with no digits after the decimal point. %6.1f describes another number to occupy six spaces, with 1 digit after the decimal point. (printf also recognizes %d for decimal integer, %o for octal, %x for hexadecimal, %c for character, %s for character string, and %% for % itself.)

Each % construction in the first argument of printf is paired with its corresponding second, third, etc., argument; they must line up properly by number and type, or you'll get meaningless answers.

By the way, printf is *not* part of the C language; there is no input or output defined in C itself. There is nothing magic about printf; it is just a useful function which is part of the standard library of routines that are normally accessible to C programs. In order to concentrate on C itself, we won't talk much about I/O until Chapter 7. In particular, we will defer formatted input until then.

Exercise 1-3: Modify the temperature conversion program to print a heading above the table. □

Exercise 1-4: Write the program which does the corresponding centigrade to fahrenheit table. □

Some Variations

As you might expect, there are plenty of different ways to write a program; let's try a variation on the temperature converter.

```
main( )        /* fahrenheit-centigrade table */
{
        int fahr;

        for (fahr = 0; fahr <= 300; fahr = fahr + 20)
                printf("%4d %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, as 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 used to produce cent now appears as the third argument of printf instead of a separate assignment statement.

This last change is an instance of a quite general rule in C — in any context where it is permissible to use the value of a variable of some type, you can use an expression of that type. Since the third argument of printf has to be a floating point value, any floating point expression can occur there.

The for itself is a loop, a generalization of the while. If you compare it to the earlier while, its operation should be clear. The first part

```
fahr = 0
```

is done once, before the loop proper is entered. The condition

```
fahr <= 300
```

is evaluated, and if true, the body of the loop (here a single printf) is executed. Then the re-initialization step

```
fahr = fahr + 20
```

is done, and the condition re-evaluated. The loop terminates when the condition becomes 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 and re-initialization parts can be any single expression.

The choice between while and for is arbitrary, based on what seems clearest. The for is usually appropriate for loops when the loop initialization and re-initialization are single statements and logically related.

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 no information to someone who might have to read the program later, and they are hard to change in a systematic way. Fortunately, C provides a way to avoid such magic numbers. With the #define construction, you can define a *symbolic name* with a particular value at the beginning of a program, then use the name whenever the value is needed. The compiler replaces all occurrences of the name by the value.

```
#define     LOWER     0
#define     UPPER     300
#define     STEP 20

main( )     /* fahrenheit-centigrade table */
{
        int fahr;

        for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
                printf("%4d %6.1f\n", fahr, 5.0/9.0 * (fahr-32));
}
```

The replacement for the name can actually be any text at all; it is not limited to numbers. Notice that there is no semicolon at the end of a definition. Since the whole line after the defined name is substituted, there would be too many semicolons in the **for**.

Exercise 1-5: Verify that the construction

```
#define     LOWER     0;
```

causes an error. □

1.3 A Collection of Useful Programs

We are now going to consider a family of related programs for doing simple operations on character data. You will find that many programs are just expanded versions of the prototypes that we write here.

The standard library provides functions for reading and writing a character at a time. **getchar( )** fetches the *next input character* each time it is called, and returns that character as its value. That is, after

```
c = getchar( )
```

then **c** contains the next character of input. The characters normally come from the terminal, but that need not concern us right now. (More on that topic in Chapter 7.)

The function **putchar(c)** is the complement of **getchar**:

```
putchar(c)
```

prints the character **c** on some output medium, again usually the terminal.

As with **printf**, there is nothing special about **getchar** and **putchar**. They are not part of the C language, but they are universally available.

File Copying

Given **getchar** and **putchar**, you can write a surprising amount of useful code without knowing anything more about I/O. The simplest example is a program which copies its input to its output a character at a time. In outline,

```
get a character
while (character is not end of file signal)
        putchar (the character just read)
        get a new character
```

Converting this into C gives

```
main( )        /* copy input to output */
{
        int c;

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

The relational operator != is "not equal to."

The main problem is detecting the end of the input. By convention, **getchar** returns a value which is not a valid character when it encounters the end of the input; in this way, programs can detect when they run out of input. The only complication is that there are two conventions in common use about what that end of file value really is. We have deferred the issue by using the symbolic name **EOF** for the value, whatever it might be. In practice, **EOF** will be either −1 or the null character '\0', so the program will have to be preceded by the appropriate one of

```
#define    EOF   -1
```

or

```
#define    EOF   '\0'
```

in order to work properly.

You should distinguish the value **EOF** that **getchar** returns when end of file is encountered from whatever mechanism is used by the operating system that the program runs on. For example, on Unix, end of file is implicit when a file is being read; from a terminal it can be entered by typing the character EOT ("control-D"), but on other systems this convention will probably be different.

Assignment Expression

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

```
c = getchar( )
```

can be used in an expression; its value is the value of the right hand side. If the assignment of a character to **c** is put inside the test part of a **while**, the file copy program can be written

```
main( )        /* copy input to output */
{
        int c;

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

The program gets a character, assigns it to c, and then tests if it 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 call to getchar — and shrinks the program. Nesting an assignment statement in a test is one of the places where C permits a valuable conciseness. (It's possible to get carried away and create impenetrable code, though, a tendency that we will try to curb.)

It's important to recognize that the parentheses around the assignment statement within the conditional are really necessary. The "precedence" of = is lower than that of !=, which means that the relational test != is done before the assignment =. So in the absence of parentheses, 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 the character fetched was an end of file or not.

Character Counting

The next program counts characters; it is a small elaboration of the copy program.

```
main( )        /* count characters in input */
{
        long nc;

        nc = 0;
        while (getchar( ) != EOF)
                nc++;
        printf("%ld\n", nc);
}
```

The line

```
nc++
```

shows a new C operator. ++ means *increment* by one. You could write nc = nc + 1 but nc++ is most concise and often most efficient. There is a

corresponding decrement operator −−. ++ and −− can be either prefix operators (++nc) or postfix (nc++); these have different values in expressions, as we will discuss in Chapter 2, but both increment nc.

The character counting program accumulates its count in a long variable instead of an int because on a PDP-11 the maximum value of an int is 32767, and it would take relatively little input to overflow the counter if it were declared int. (In Honeywell and IBM C, long and int are synonymous and much larger.) The %ld in the printf signals a long integer.

Some early versions of C do not support long variables; in that case, use float or even double (double length float) to cope with bigger numbers. Here is the character counting program with double. We will also use a for statement instead of a while, to illustrate an alternative construction.

```
main( )      /* count characters in input */
{
        double nc;

        for (nc = 0; getchar( ) != EOF; nc = nc + 1)
                ;
        printf("%f\n", nc);
}
```

Some versions of C do not permit ++ and −− to be applied to float or double. In that case, you must write out the increment as we did here.

printf uses %f for both float and double. You could also use %.0f to suppress printing the non-existent fraction part.

The body of the for loop here is *empty*, because all of the work is done in the test and re-initialization parts. But the grammatical rules of C require that a for statement have a body. The isolated semicolon, technically a *null statement*, is there to satisfy that requirement. We put it on a separate line to make it more 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 so the program produces zero, the right answer. This is an important observation. 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 handed input like "no characters". The while and for help ensure that they do reasonable things with extreme cases.

Line Counting

The next program counts *lines* in its input. Input lines are assumed to be terminated by the newline character \n that has been religiously appended to every line written out. The program is again simple and familiar.

```
main( )        /* count lines in input */
{
        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++. if tests the parenthesized condition, and if it is true, does 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 Fortran's .EQ.); it is pronounced "equals". A separate symbol is used to distinguish the equality test from the single = used for assignment. Since assignment is about twice as frequent as equality testing in typical C programs, it's appropriate that the operator be half as long.

Exercise 1-6: Write a program to count blanks, tabs, and newlines. □

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.

```
#define     YES  1
#define     NO   0

main( )     /* count words, lines, chars in input */
{
        int c, nc, nl, nw, inword;

        inword = NO;
        nc = nl = nw = 0;
        while ((c = getchar( )) != EOF) {
                nc++;
                if (c == '\n')
                        nl++;
                if (c == ' ' || c == '\n' || c == '\t')
                        inword = NO;
                else if (inword == NO) {
                        inword = YES;
                        nw++;
                }
        }
        printf("%d %d %d\n", nl, nw, nc);
}
```

Logically, this is a more complicated program. Every time the program makes the transition from not being in a word to being in a word, it counts one more word. The variable inword records which state the program is in; initially it is "not in a word", which is assigned the value NO. We prefer the symbolic constants YES and NO to the literal values 1 and 0 because they make the program more readable. Of course 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 originally. You'll also find that it's easier to make extensive changes in programs where numbers appear only as symbolic constants.

The line

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

sets all three variables to zero. This is not a special case, but a consequence of the fact that the assignment statement has a value. It's really as if we had written

```
nc = (nl = (nw = 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 ...". (The character \t is a tab; it is best written this way so it is visible on a listing.) There is also && for AND. Expressions connected by && and || are evaluated left to right, and it is

guaranteed that evaluation will stop as soon as the answer is known. Thus if c contains a blank, there is no need to test whether it contains a newline or tab, so these tests are *not* made.

The example also shows the C **else** statement, which specifies an alternative action to be done if the condition part of an if statement is false. One and only one of the two statements associated with an if-**else** is done. Either statement can in fact be quite complicated. In the word count program, the one after the **else** is another if.

Exercise 1-7: Write a program which prints only the words in its input, one per line. All non-alphabetic characters should be discarded. As a matter of design, what should happen to digits? What about contractions like don't? □

1.4 Arrays

Let us write a program to count the number of occurrences of each digit, of white space (blank, tab, newline), and all other characters. This is artificial, of course, but it permits us to illustrate several aspects of C in one program.

There are twelve categories of input, which is too many to use separate variables for each, so we must at least use an array to hold the number of occurrences of digits. Here is one version of the program:

```
main( )        /* count digits, white space, others */
{
        int c, i, nwhite, nother, 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("\nwhite space = %d, other = %d\n", nwhite, nother);
}
```

The declaration

```
int ndigit[10]
```

declares ndigit to be an array of 10 integers. Array subscripts start at zero in C

(rather than 1 as in Fortran or PL/I), so the elements are ndigit[0], ndigit[1], ... ndigit[9]. This is reflected in the for loops which initialize and print the array.

A subscript can be any integer expression; this includes as special cases integer variables like i, and integer constants.

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

```
c >= '0' && c <= '9'
```

for a digit and the character-to-integer conversion

```
c - '0'
```

work only if the digits are ordered and if there is nothing but digits between '0' and '9'. Fortunately, this is universally true.

char variables and constants are essentially identical to int's in arithmetic contexts. Thus c-'0' is an integer expression with a value between 0 and 9 corresponding to the character '0' to '9' stored in c. By definition, arithmetic involving char's and/or int's converts everything to int before proceeding, so things work out quite naturally and conveniently. In fact, single char's are often just declared int's.

The pattern

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

occurs frequently in programs as a way to express a multi-way decision; here the decision is digit, white space, or other. The code is simply read from the top until some *condition* is satisfied; at that point the corresponding *statement* part is executed, and the entire construction exited. (Of course *statement* can be several statements enclosed in braces.) If none of the conditions is satisfied, the *statement* after the final else is done. If that is omitted (as in the word count program), no action takes place. There can be an arbitrary 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, so that long decisions do not march off the right side of the page.

The switch statement, to be discussed in Chapter 3, provides another way to write a multi-way branch that is particularly suitable when the condition being tested is simply whether some integer or character expression matches one of a set of constants. We will present a switch version of this program in Chapter 3.

Exercise 1-8: Write a program to remove any trailing blanks or tabs from each line of input. □

1.5 Functions

A function is just a way to encapsulate some part of a computation in a black box, which can then be used without worrying about its innards. So far we have used only functions that have been provided for us: **printf**, **getchar** and **putchar**. Now it's time to write a few of our own.

To illustrate the points, let's write a program to print the longest line in the input. The basic outline is simple enough:

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

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

Since things divide so nicely, it would be well to write them that way too; that is the purpose of functions. Accordingly, let us first write a separate function to fetch the next *line* of input; this is a generalization of **getchar**. Since we want to make the function useful in other contexts, we'll try to make it as flexible as possible. Let us say that the newline at the end of the line will be trimmed off, so that the resulting line can be used as a character string. At the minimum, **getline** has to return a signal about possible end of file; a more generally useful design would be to return the length of the line, or -1 if end of file is encountered.

When we find a longer line than the previous longest, 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 whole program at once, so you can see what it looks like.

```
#define     MAXLINE     1000  /* maximum input line size */

main( )     /* find longest line */
{
        int n, max;
        char line[MAXLINE], save[MAXLINE];

        max = -1;  /* longest length seen so far */
        while ((n = getline(line, MAXLINE)) >= 0)
                if (n > max) {
                        max = n;
                        copy(line, save);
                }
        if (max > -1)       /* there was a line */
                printf("%s\n", save);
}

getline(s, lim)         /* read line into s; return length */
char s[ ];
int lim;        /* maximum line length, including \0 */
{
        int i, c;

        for (i = 0; i < lim-1 && (c = getchar( )) != '\n' && c != EOF; i++)
                s[i] = c;
        s[i] = '\0';
        if (c == EOF)
                return(-1);
        else
                return(i);
}

copy(s1, s2) /* copy s1 to s2; assumes s2 big enough */
char s1[ ], s2[ ];
{
        int i;

        for (i = 0; (s2[i] = s1[i]) != '\0'; i++)
                ;
}
```

Each function has the same form:

```
name(argument list, if any)
argument declarations, if any
{
        declarations
        statements
}
```

The functions can appear in any convenient order, in one source file or in several. (Whatever the allocation to source files, the definition of EOF must be accessible to getline.) Of course if the input appears in several files, you will have to do more work to compile and load it than if it all appears on one file, but that is an operating system matter more than a property of the C language. For the moment, we will assume that the three functions are all in one file.

main and getline communicate through both a pair of arguments and a returned value. In getline, the arguments have to be declared appropriately; this is done by the lines

```
char s[ ];
int lim;
```

which specify that the first argument is an array of indeterminate length, and the second is an integer. The declaration of arguments goes between the function argument list and the opening left brace. The names used by getline for the arguments are purely local to getline, and not accessible to anyone else.

getline uses a return statement to send a value back to the caller, just as in PL/I. Any expression may occur in the parentheses. A return with no expression causes control, but no value, to be returned to the caller. The same is true of "falling off the end" of a function, as in copy.

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 a string of characters. This convention is followed throughout C. For example, when a string constant like

```
"hello, world\n"
```

is written in a C program, the compiler terminates the array of characters representing that string with a \0 so that functions such as printf can detect the end.

If you examine copy closely, you will discover that it relies on the fact that its input argument s1 is terminated by \0, and it copies this character onto the output argument s2.

Exercise 1-9: Write a program to "fold" long input lines at the first blank or tab before the *n*th column of input. Make sure that n is a parameter in your program. Do something intelligent if there are no blanks or tabs before the specified column. □

1.6 Arguments — Call by Value

One aspect of function usage can trap programmers used to other languages, particularly Fortran and PL/I. 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 (actually on a stack) rather than their addresses. This leads to quite different properties than are seen with "call by reference" languages like Fortran and PL/I, where the called routine is handed the address of the argument variable, not its value.

The main distinction is that in C the called function *cannot* alter the original arguments in the calling function; it can only alter its private, temporary copy.

Call by value is generally an asset, however, not a liability. Arguments can be treated as conveniently initialized variables in the called routine. For example, consider the function log2(n), which returns the base 2 logarithm of n (that is, the number of bits needed to hold n), where n must be a positive int.

```
log2(n)       /* base 2 log of n */
int n;
{
        int log;

        log = 1;
        while ((n = n / 2) > 0)
                log++;
        return(log);
}
```

The argument n is used as a temporary variable, which is repeatedly divided by two until it becomes zero; there is no need for another variable in log2 to hold the same value. And whatever is done to n inside log2 has no effect on the argument that log2 was originally called with.

With call by value, array arguments are processed without special effort, since the name of an array is in effect the address of the first element, and once you know the address of something, it's easy to work your will on it. We have already done this with functions like getline and copy that store into character arrays.

It is possible to arrange for a function to modify a variable in a calling routine when necessary. The caller must provide the *address* of the variable to be set; and the callee must declare it to be a pointer and reference it indirectly. We will cover this in detail in Chapter 5.

1.7 Scope

The variables in **main** (**line**, **save**, etc.) are private or *local* to **main**; because they are declared within **main**, no other function can have direct access to them. The same is true of the variables in the other functions; for example, the variable **i** in **getline** is unrelated to the **i** in **copy**. Each local variable in a routine comes into existence only when the function is called, and *disappears* when the function is exited. Accordingly, local variables have no memory from one call to the next and must be explicitly initialized upon each entry. It is for this reason that local variables are also known as *automatic* variables, following terminology in other languages. (Chapter 4 discusses the **static** storage class, in which local variables do retain their values between function invocations.)

There is a need to share data between routines, however, so C provides two ways to achieve it. The first is function arguments and return values, which we have used in all our examples so far. The second is a way to define variables which are *external* to all functions, that is, global variables which can be accessed by name by any function that cares to. (This mechanism is rather like Fortran COMMON or PL/I EXTERNAL.) External variables remain in permanent existence, rather than appearing and disappearing as functions are called and exited.

To make a variable external, we have to *define* it outside of any function, and make an **extern** declaration in each function that wants to use it. To make the discussion concrete, let us rewrite the longest-line program with **line**, **save** and **max** as external variables. This requires changing the calls, declarations, and bodies of **getline** and **copy**.

```
#define     MAXLINE     1000  /* maximum input line size */

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

main( )     /* find longest line */
{
        int n;

        max = -1;
        while ((n = getline( )) >= 0)
                if (n > max) {
                        max = n;
                        copy( );
                }
        if (max >= 0)
                printf("%s\n", save);
}

getline( )     /* specialized version */
{
        int i, c;
        extern char line[ ];

        for (i = 0; i<MAXLINE-1 && (c = getchar( )) != '\n' && c != EOF; i++)
                line[i] = c;
        line[i] = '\0';
        if (c == EOF)
                return(-1);
        else
                return(i);
}

copy( )     /* specialized version */
{
        int i;
        extern char line[ ], save[ ];

        for (i = 0; (save[i] = line[i]) != '\0'; i++)
                ;
}
```

Roughly speaking, any function that wishes to access an external variable must contain an **extern** declaration for it, although this can sometimes be done by context instead of explicitly. The declaration is the same as others, except for the added keyword **extern**. Furthermore, there must appear a *definition* of

these variables which is external to all functions, as in the first lines of the example above.

In certain circumstances, the **extern** declaration can be omitted: if the external definition for a variable occurs *before* its use in some function, then there is no need for an **extern** declaration. The declarations in **getline** and **copy** are redundant.

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. This style of coding is fraught with peril, though, since it leads to programs whose data connections are not at all obvious — variables can be changed in unexpected and even inadvertent ways. The second version of the longest line program is inferior to the first, partly for these reasons, and partly because we have destroyed the generality of two quite useful functions by wiring into them the names of the variables they will manipulate.

1.8 Summary

At this point we have covered what might be called the conventional core of C. Variables and constants are the basic objects that a program processes. Each variable must be declared to be of one of the types that C supports. A variable may be an array, with subscripts running from zero to one less than the size specified in the declaration. Arithmetic operators include the usual +, −, *, and /, and the increment and decrement operators ++ and −−. Control flow operations include **if**, perhaps with an **else** part; loops with **while** or **for**; and statements grouped with braces. Functions communicate with arguments, returned values, and external variables.

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. The exercises that follow are intended to give you some suggestions for programs of about the level of complexity that we have written here. All can be handled with small programs, no more than say 20 lines.

Exercise 1-10: Write a program to replace all strings of blanks and tabs by a single blank. □

Exercise 1-11: Write a program to replace each tab by the sequence > and each backspace by <. □

Exercise 1-12: Write a program to remove all comments from a C program. Don't forget to handle quoted strings properly. □

CHAPTER 2: TYPES, OPERATORS AND EXPRESSIONS

Variables and constants are the basic data objects manipulated in a program. Declarations state what the variables are, what type they have, and perhaps what their initial values are. Operators specify what is to be done to them. Expressions combine variables and constants to produce new values. These are the topics of this chapter.

It's hard to present these basic objects entirely in the context of complete examples, so there will be fewer programs and more snippets and fragments and rules than we like. This can make dry reading, but bear with us; significant examples will start again in Chapter 3.

2.1 Variable Names

Although we didn't come right out and say so, there are some restrictions on variable names. Names are made up of letters and digits; the first must be a letter. The underscore '_' counts as a letter. Upper and lower case are different; traditional C practice is to use lower case for variable names, and upper case for symbolic constants.

Only the first eight characters of a name are significant; for external names such as function names, only seven are significant. (These numbers may vary from machine to machine.) Furthermore, words like **if**, **else**, **int**, **float**, etc., are *reserved;* that is, you can't use them as variable names. (They must be in lower case.)

Naturally it's wise to choose variable names that mean something, that are related to the function of the variable, and that are unlikely to get mixed up typographically. Anyone who uses l1 and ll in the same program deserves what will inevitably happen.

2.2 Data Types and Sizes

There are only a few basic data types in C.

char a single byte, capable of holding one character in the local character set.

int an integer, typically reflecting the natural size of integers on the host machine.

float single precision floating point.
double double precision floating point.

In addition, there are a number of qualifiers which can be applied to int's: **short**, **long**, and **unsigned**. **short** and **long** refer to different sizes of integers; **unsigned** implies that the number is to be treated as a logical quantity, not an arithmetic one. The declarations for the qualifiers look like

```
short int x;
long int y;
unsigned int z;
```

The word **int** can be omitted in such situations, and typically is.

The precision of these objects depends on the machine at hand; the table below shows some representative values.

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8-32
char	8 bits ascii	9 bits ascii	8 bits ebcdic	8 bits ascii
int	16	36	32	32
short	16	36	16	16
long	32	36	32	32
float	32	36	32	32
double	64	72	64	64

The intent is that **short** and **long** should provide different lengths of integers where practical; **int** will normally reflect the most "natural" size for a particular machine. As you can see, each compiler is free to interpret **short** and **long** as appropriate for its own hardware. About all you should count on is that **short** is no longer than **long**.

2.3 Constants

Constants of these various types can be specified. **int** and **float** constants have already been disposed of, except to note that the usual

```
123.456e-7
```

or

```
0.12E3
```

scientific notation for **float**'s is also legal. Every floating point constant is taken to be **double**, so the "**e**" notation serves for both **float** and **double**.

Long constants are written in the style **123L**. A lower case **l** can also be used, but it's hard to distinguish from the digit **1**. An ordinary integer constant that is too long to fit in an **int** is also taken to be a **long**.

There is a notation for octal and hexadecimal constants: a leading **0** (zero) or **0x** on an **int** or **long** constant implies octal or hex respectively. For example, decimal 31 can be written as **037** in octal and **0x1f** in hex.

A *character constant* is a single character written within single quotes, as in 'x'. The internal value of a character constant is the numeric value of the character in the machine's character set. These values can participate in numeric operations just as any other numbers, although they are most often used in strings and in comparisons with other characters. A later section talks about conversion rules.

Certain non-graphic characters, like newline, tab, etc., can be represented in character constants by escape sequences like \n, \t, \0, \\ (backslash), \' (single quote), etc., which look like two characters, but are actually only one.

A *constant expression* is an arithmetic expression that involves only constants. Such expressions can be evaluated at compile time, rather than run time, and accordingly may be used in any place that a constant may be. Examples are

```
char  line[MAXLINE+1];

      twopi = 2 * 3.141592654;
```

A *string constant* is a sequence of zero or more characters surrounded by double quotes, as in

```
"I am a string"
```

or

```
""      /* a null string */
```

The quotes are not part of the string, but serve only to delimit it. The same escape sequences used for character constants apply in strings; of course \" represents the double quote.

Technically, a string is an array whose elements are single characters. The compiler automatically places the null character \0 at the end of each such string, so programs can conveniently find its end. This representation means that there is no real limit to how long a string can be, but programs have to scan one completely to determine its length. The physical storage required is thus one more than the number of characters written between the quotes. The function **strlen** computes the length of a character string, excluding the terminal \0.

```
strlen(s)       /* return length of s */
char s[ ];
{
        int i;

        i = 0;
        while (s[i] != '\0')
                i++;
        return(i);
}
```

Be careful to distinguish between a character constant and a string that contains a single character: "\n" is not the same as '\n'.

2.4 Declarations

All variables must be declared before use. A declaration names the type of variable, and is followed by a list of one or more variables of that type, as in

```
int lower, upper, step;
char  c, line[1000];
```

Variables can be distributed among declarations in any fashion; the lists above could equally well be written as

```
int     lower;
int     upper;
int     step;
char    c;
char    line[1000];
```

This latter form takes more room, but is convenient for adding a comment to each declaration or for subsequent modifications.

Variables may also be initialized in their declaration, although there are some restrictions. If the name is followed by an equals sign '=' and a constant, that serves as an initializer, as in

```
int     i = 0;
float   twopi = 2 * 3.141592654;
char    quote = '\'';
```

Some early versions of the compiler allow the '=' to be omitted; some only allow external variables to be initialized.

If the variable in question is external or static, the initialization is done once only, conceptually before the program starts executing. Local variables are initialized each time the function they are in is called. Unitialized variables have undefined values, usually garbage.

Whether to initialize a variable in its declaration or in the code which follows is not always clear-cut. Consider

```
int     i = 0;

while (s[i] != '\0')
        i++;
```

and

```
int     i;

i = 0;
while (s[i] != '\0')
        i++;
```

We have a mild preference for the second version because the initialization of i is closer to the code that manipulates it.

We will discuss initialization more as new objects are introduced.

2.5 Arithmetic Operators

The arithmetic operators are +, −, *, /, and the modulus operator %. Integer division truncates. The expression

```
a % b
```

produces the remainder when a is divided by b, and thus is zero when b divides a exactly. For example, a year is a leap year if it is divisible by 4 but not by 100, except that years divisible by 400 *are* leap years. Therefore

```
if ((year%400 == 0) || (year%100 != 0 && year%4 == 0))
        it's a leap year
else
        it's not
```

% cannot be applied to float or double. For both / and %, if either operand is negative, the results are machine dependent.

The + and − operators have the same precedence, which is lower than the (identical) precedence of *, /, and %. Arithmetic operators group left to right. The order of evaluation is not specified for associative and commutative operators like * and +; the compiler is free to rearrange even parenthesized computations as it wishes. Thus a+(b+c) may well be evaluated as (a+b)+c.

Any underflow or overflow that occurs as an expression is being evaluated is silently ignored.

2.6 Relational Operators

The relational operators are, in order of decreasing precedence,

```
>  >=  <  <=
==  !=
```

Operators on the same line have the same precedence. Relationals have lower precedence than arithmetic operators, so expressions like `i < lim-1` are taken as `i < (lim-1)`, as would be expected.

More interesting are the logical operators `&&` and `||`. Expressions connected by `&&` or `||` are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known. These properties are both critical to writing programs that work. For example, here is a loop from the input function `getline` which we wrote in Chapter 1.

```
for (i = 0; i < lim-1 && (c = getchar( )) != '\n' && c != EOF; i++)
    s[i] = c;
```

Clearly, it is necessary to check *first* that there is room in the array `s` to store any new character, so the test `i < lim-1` must be made first. Not only that, but if this test fails, we must not go on and read another character.

Similarly, it would be unfortunate if `c` were tested against `EOF` before `getchar` was called: the call *must* be made before the character in `c` is tested.

The precedence of `&&` is greater than that of `||`, and both are lower than the relationals, so that expressions like

```
i < lim-1 && (c = getchar( )) != '\n' && c != EOF
```

need no extra parentheses. But notice again that the precedence of `!=` is higher than assignment, so parentheses are needed around

```
(c = getchar( )) != '\n'
```

Exercise 2-1: Write the `for` loop above without `&&`. □

2.7 Type Conversions

What happens if you say

```
int i;

i = 'x' + 3e5
```

Not that anyone would, perhaps, but the question of type conversions requires some discussion.

First, `char`'s and `int`'s may be freely intermixed in arithmetic expressions: every `char` in an expression is automatically converted to an `int`. This permits considerable flexibility in certain kinds of character transformations. One is exemplified by the routine `atoi` which converts a string of ASCII digits into its numeric equivalent.

```
atoi(s)/* convert s to integer */
char s[ ];
{
        int i, n;

        n = 0;
        for (i = 0; s[i] >= '0' && s[i] <= '9'; i++)
                n = 10 * n + s[i] - '0';
        return(n);
}
```

The expression

```
s[i] - '0'
```

gives the numeric value of the character stored in s[i] only if the digits are contiguous and in increasing order as characters; fortunately this is true for all known character sets.

Exercise 2-2: Are the assignments

```
n *= 10 + s[i] - '0'
```

and

```
n = 10 * n + s[i] - '0'
```

equivalent? □

Another example is the routine lower which converts a single character to lower case *for the ASCII character set only.*

```
lower(c)
char c;
{
        if (c >= 'A' && c <= 'Z')
                return(c - 'A' + 'a')
        else
                return(c);
}
```

This works for ASCII because corresponding upper case and lower case letters are a fixed distance apart as numeric values and are contiguous — there is nothing but letters between 'a' and 'z'. This latter observation is *not* true of the EBCDIC alphabet (IBM), so this code fails on such systems — it converts more than letters.

Exercise 2-3: Write the corresponding function upper, which converts lower case letters to upper case. □

There is one subtle point about the conversion of characters to integers. If a character is negative does it become a negative integer ("sign extension"), or is it positive? Regrettably, this varies from machine to machine. It is true that by definition, any character in the machine's standard character set will never be negative, so the problem won't arise there. But for arbitrary bit

patterns stored in character variables, it can.

Another useful form of automatic type conversion is that relational expressions like i > j and logical expressions connected by && and || are defined to have value 1 if true, and 0 if false. Thus the assignment

```
leap = (year%400 == 0) || (year%100 != 0 && year%4 == 0)
```

has the value 1 for leap years and 0 for non-leap years.

Implicit arithmetic conversions work much as expected. We have already seen **char**'s become **int**'s, and **int**'s become **float**'s, as in

```
cent = 5.0/9.0 * (fahr - 32)
```

All **float**ing point arithmetic in C is done in double precision, so all **float**'s in an expression are converted to **double**.

In general, if an operator like + or * which takes two operands (a "binary operator") has operands of different types, the "lower" type is *promoted* to the "higher" type before the operation proceeds. The result is of the higher type. More formally, for all arithmetic operators, the following sequence of conversion rules is applied.

> **char** is converted to **int** and **float** is converted to **double**.
> Then if either operand is **double**, the other is converted to **double**, and the result is **double**.
> Then if either operand is **long**, the other is converted to **long**, and the result is **long**.
> Then if either operand is **unsigned**, the other is converted to **unsigned**, and the result is **unsigned**.
> Otherwise the operands must be **int**, and the result is **int**.

Conversions also take place across assignment statements; the value of the right side is converted to the type of the left. If **x** is **float** and **i** is **int**, then

```
x = i;
```

and

```
i = x;
```

both cause conversions; **float** to **int** causes truncation of any fractional part.

Finally, explicit type conversions can be put in any expression with a construct called a *cast*. In the construction

```
( type ) expression
```

the expression is converted to the named type. For example, the library routine **sqrt** expects a **double** argument, and will produce nonsense if inadvertently handed something else. So if **n** is an integer,

```
sqrt((double) n)
```

converts **n** to **double** before passing it to **sqrt**.

Early versions of C do not provide the cast operation; in that case, you must do your own conversions by assigning to explicit temporary variables of the proper type.

2.8 Increment and Decrement Operators

C provides two unusual operators for incrementing and decrementing variables. The increment operator ++ adds 1 to its operand; the decrement operator -- subtracts 1. We have frequently used ++ to increment variables, as in

```
if (c == '\n')
        nl++;
```

The unusual aspect is that ++ and -- may be either prefix (before the variable, as in ++i), or postfix (after the variable: i++). In both cases, the effect is to increment i, but the *value* of the expression ++i is i *after* it is incremented, while the value of i++ is i *before* it is incremented. Thus if i is 5, then

```
x = i++
```

sets x to 5, but

```
x = ++i
```

sets x to 6. In both cases, i becomes 6.

In a context where no value is wanted, just the incrementing effect, as in

```
if (c == '\n')
        nl++;
```

choose prefix or postfix according to taste. But there are situations where one or the other is specifically called for. For instance, consider the function **squeeze(s, c)** which removes all occurrences of the character c from the string s.

```
squeeze(s, c)        /* delete all c from s */
char s[ ], c;
{
        int i, j;

        for (i = j = 0; s[i] != '\0'; i++)
                if (s[i] != c)
                        s[j++] = s[i];
        s[j] = '\0';
}
```

Each time a non-c occurs, it is copied into the current j position, and only then is j incremented to be ready for the next character.

Of course, we could have written the longer form

```
        if (s[i] != c) {
                s[j] = s[i];
                j++;
        }
```

but the shorter form is better, and, after a bit of experience, just as easy to read.

As another example, perhaps more compelling, here is a function called **strcat(s, t)**, which concatenates the string **t** to the end of the string **s**. **strcat** believes that there is enough space in **s** to hold the combination.

```
strcat(s, t)    /* concatenate t to end of s */
char s[ ], t[ ];
{
        int i, j;

        i = j = 0;
        while (s[i])    /* find end of s */
                i++;
        while (s[i++] = t[j++])  /* copy t */
                ;
}
```

The postfix ++ is applied to both i and j to make sure that they are both in position for the next pass through the loop.

The test could be written as

```
        while (s[i] = t[i]) != '\0')  /* copy t */
```

but since \0 is zero, and since **while** tests whether the parenthesized expression is zero or not, the != '\0' is redundant and can be omitted. Although you might argue that this is bad form, the \0 is very often elided.

2.9 Bitwise Logical Operators

C provides a full set of bitwise logical operators in addition to the usual arithmetic ones. These include & (bitwise AND), | (bitwise inclusive OR), and ^ (bitwise exclusive OR). The classic application of & is masking off everything but the last 7 bits, to make an integer into an ASCII character:

```
c = n & 0177;
```

You should carefully distinguish the bitwise operators & and | from && and ||, which imply left-to-right evaluation of a truth value. For example, if x is 1 and y is 2, then x & y is zero while x && y is one.

The shift operators << and >> perform left and right shifts of their left operand by the number of bit positions given by the right operand. Thus x << 2 shifts x left by two positions, equivalent to multiplication by 4. Whether a shift is logical or arithmetic depends: if the variable in question is **unsigned**, then the shift is logical; otherwise it is arithmetic on some

machines, so beware.

The unary negation operator ! converts a non-zero or true operand into 0, and a zero or false operand into 1. A common use of ! is in constructions like

```
if (!inword)
```

rather than

```
if (inword == 0)
```

It's hard to generalize about which form is better. Constructions like !inword read quite nicely ("if not in word"), but more complicated ones can be hard to understand.

The unary operator ~ gives the one's complement of integers; that is, it converts each 1 bit into a 0 bit and vice versa. This operator typically finds use in expressions like

```
x &= ~077
```

which masks the last six bits of x to zero.

2.10 Assignment Operators

Expressions like

```
i = i + 10
```

abound in typical programs. Expressions like these, where the left hand side is repeated on the right, can be written in the compressed form

```
i += 10
```

using an *assignment operator* like +=.

Any binary operator op, that is, one that has a left and right operand, has a corresponding assignment operator op=. op is one of

```
+  -  *  /  %  <<  >>  &  ^  |
```

If e1 and e2 are expressions, then

```
e1 op= e2
```

is equivalent to

```
e1 = e1 op (e2)
```

except that e1 is computed only once. The parentheses around e2 should be noted — for example,

```
x *= y + 1
```

is actually

```
x = x * (y + 1)
```

rather than

```
x = x * y + 1
```

The type of an assignment expression like `i += 10` is the type of its left operand.

The conciseness of an assignment operator is not much saving for an expression as simple as `i += 10`, but for something like

```
yyval[yypv[p3+p4] + yypv[p1+p2]] += 2
```

it is well worthwhile. It makes the code easier to understand, since the reader doesn't have to check painstakingly that two long expressions are indeed the same. And it may help the compiler to produce more efficient code.

Exercise 2-4: Rewrite the `log2` function of Chapter 1 with an assignment operator. □

As an example that combines logical and assignment operators, here is a function called `bitcount` which counts the number of 1 bits in its integer argument.

```
bitcount(n)   /* count 1 bits in n */
unsigned n;
{
        int b;

        for (b = 0; n != 0; n >>= 1)
                if (n & 01)
                        b++;
        return(b);
}
```

Declaring the argument to be `unsigned` ensures that when `n` is right-shifted, vacated bits will be filled with zeros, not with sign bits; regardless of the machine the program is run on.

The loop can also be written as

```
        for (b = 0; n != 0; n >>= 1)
                b += n % 2;
```

since `n % 2` is 1 if `n` is odd, that is, if its bottom bit is turned on, and zero otherwise.

Early versions of C used the form `=op` instead of `op=` for assignment operators. This leads to some nasty ambiguities, typified by

```
x=-1
```

Does this decrement `x` or set it to −1? The only way to be sure is to add a space at the right place, to make either

```
x =- 1
```

or

```
x = -1
```

If your C compiler supports the new form, use it.

Exercise 2-5: AND'ing a value n with n−1 deletes the rightmost 1 bit in n. Use this fact to write a faster version of bitcount. □

2.11 Assignments as Values

We have already used the fact that the assignment statement has a value and can occur in expressions; the most common example is

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

The other assignment operators (+=, −=, etc.) can also be used in this way, although it is a less frequent occurrence. As an illustration, here is a line from a program that we will discuss in detail in Chapter 4:

```
val += (c - '0') / (den *= 10.0);
```

den is multiplied by 10 before it in turn is used to divide c − '0'; the result is then added to val.

Increment and decrement operators and nested assignment statements cause "side effects" — some variable is changed as a by-product of the evaluation of an expression. In any expression involving side effects, there can be subtle dependencies on the order in which parts of the expression are evaluated. One unhappy situation is typified by

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

The question is whether i is incremented before or after it's assigned to a. The compiler can obviously do this in different ways, and generate different answers depending on its interpretation. Normally this kind of code is only written inadvertently, but you should be aware of the possibility.

Exercise 2-6: Write a program to make *all* ASCII non-printing characters visible. □

CHAPTER 3: CONTROL FLOW

The control flow statements of a language specify the order in which things get done. We have already met the most common control flow constructions of C in earlier examples; here we will complete the set, and be more precise about the ones discussed before.

3.1 Braces

The braces { and } are used to group statements together so that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are one obvious example; braces around multiple statements after an **if** or **else** or **while** or **for** are the other. Braces can also be used to delimit a "block" in which local variables can be declared; we will talk about this in Chapter 4.

3.2 If-Else

if-else is used to make decisions. Formally, the syntax is either

```
if (expression)
        statement
```

or

```
if (expression)
        statement
else
        statement
```

The *expression* is evaluated; if it is true (that is, if the *expression* has a non-zero value), the first statement is done. If it is false (zero) ("false") and if there is an **else** part, the second statement is done.

Since an **if** simply tests the numeric value of an expression, certain coding shortcuts are possible. The most obvious is writing

```
if (expression)
```

instead of

```
if (expression != 0)
```

Sometimes this is natural and clear; at other times it is cryptic and unsafe, so watch out.

Because the **else** part of an **if-else** is optional, there is an ambiguity when an **else** is omitted from a nested **if** sequence. This is resolved in the usual way — the **else** is associated with the previous un-**else**'ed **if**. For example, in

```
if (n > 0)
        if (a > b)
                z = a;
        else
                z = b;
```

the **else** goes with the inner **if**, as we have shown by indentation. If that isn't what you want, braces must be used to force the proper association:

```
if (n > 0) {
        if (a > b)
                z = a;
}
else
        z = b;
```

The ambiguity is especially pernicious in situations like:

```
if (n > 0)
        for (i = 0; i < n; i++)
                if (s[i] > 0) {
                        printf("...");
                        return(i);
                }
else
        printf("error - n is zero\n");
```

The indentation shows unequivocally what you want, but the compiler doesn't get the message. This kind of bug can be very hard to find.

By the way, notice that there is a semicolon after `z = a` in

```
if (a > b)
        z = a;
else
        z = b;
```

This is because grammatically, a *statement* follows the **if**, and statements must be terminated by a semicolon.

3.3 Conditional Expressions

This last example of course computes in z the maximum of a and b. The ternary operator ? : provides an alternate way to write this and similar constructions:

```
z = (a > b) ? a : b;        /* z = max(a, b) */
```

In the expression

```
p ? q : r
```

the expression p is evaluated. If it is non-zero (true), then the value of the conditional expression is the value of the expression q; otherwise it is r. Only one of q and r is evaluated. If q and r are of different types, the type of the result is based on the conversion rules discussed in the previous chapter. For example, if f is a float, and n is an int, then the expression

```
(n > 0) ? f : n
```

is of type float regardless of whether n is positive or not.

Parentheses are not necessary around the first expression of a conditional expression, since the precedence of ? : is very low. They are advisable anyway, however, since they make the condition part of the expression easier to see.

The conditional expression often leads to succinct code. For example, this loop prints N elements of an array, 10 per line, separated by three blanks, with no extra characters, and each line (even the last) properly terminated by a single newline.

```
for (i = 0; i < N; i++)
        printf("%d%s", a[i], (i==N-1 || i%10==9) ? "\n" : "   ");
```

A newline is printed every 10 elements, and after the Nth. All other elements are followed by three blanks. Although this might look tricky, it's instructive to try to write it without the conditional expression.

Exercise 3-1: Write the printing loop without a conditional expression. □

Exercise 3-2: Rewrite the function lower (see Chapter 2) to convert upper case letters to lower case, using a conditional expression instead of if-else. □

3.4 Else-If

The construction

```
if (expression)
        statement
else if (expression)
        statement
else if (expression)
        statement
else
        statement
```

occurs so often in programming that it is worth a brief separate discussion. This sequence of if's is the most general way of writing a multi-way decision. The *expression*'s are evaluated in order; if any *expression* is true, the *statement* associated with it is executed, and then the whole chain is exited. The code for each *statement* is either a single statement, or a group in braces.

The last **else** part handles the "none of the above" or default case where none of the other conditions was satisfied. Sometimes there is no explicit action for the default; in that case it can be omitted.

To illustrate a three-way decision, here is a binary search function that decides if a particular value x occurs in the sorted array v. It returns the position (a number between 0 and n−1) if x occurs in v, and −1 if not.

```
binary(x, v, n)        /* find x in v[0] ... v[n-1] */
int x, v[ ], n;
{
        int low, high, mid;

        low = 0;
        high = n - 1;
        while (low <= high) {
                mid = (low+high) / 2;
                if (x == v[mid])
                        return(mid);
                else if (x < v[mid])
                        high = mid - 1;
                else
                        low = mid + 1;
        }
        return(-1);
}
```

The fundamental decision is whether x is less than, equal to, or greater than v[mid] at each step; this is a natural for **else-if**. Since the first case ends with a **return**, there is no need for the **else** that follows; we could remove it. But the version presented is preferable, since it better shows the three-way nature of the decision.

3.5 Switch

The **switch** statement is a special multi-way decision maker that tests whether an expression matches one of a number of *constant* values, and branches accordingly. In Chapter 1 we wrote a program to count the number of each digit, white space, and all other characters, using an **if** ... **else if** ... **else**. Here is the same program with **switch**.

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

        nwhite = nother = 0;
        for (i = 0; i < 10; i++)
                ndigit[i] = 0;

        while ((c = getchar( )) != EOF)
                switch (c) {
                        case '0':
                        case '1':
                        case '2':
                        case '3':
                        case '4':
                        case '5':
                        case '6':
                        case '7':
                        case '8':
                        case '9':
                                ndigit[c-'0']++;
                                break;
                        case ' ':
                        case '\n':
                        case '\t':
                                nwhite++;
                                break;
                        default:
                                nother++;
                                break;
                }

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

The **switch** compares the expression in parentheses (in this program the character **c**) to all the cases. Each case must be labelled by an *integer constant*,

which includes character constants like '0'. If a case matches, execution starts at that case. The case labelled **default** is executed if none of the other cases is satisfied. (A **default** is optional; if it isn't there, and none of the cases matches, no action at all takes place.) Cases and default can occur in any order.

The **break** statement causes an immediate exit from the **switch**. Because cases are just labels, after the code for one case is done, execution *falls through* to the next unless you take explicit action to escape. **break** and **return** are the most common ways to leave a **switch**. **break** also causes an immediate exit from **while** and **for** loops as well, as will be discussed later in this chapter.

Falling through cases is a mixed blessing. On the positive side, it allows multiple cases for a single action, as with the blank, tab or newline in this example. But it also implies that normally each case must end with a **break** to prevent falling through to the next. Falling through from one case to another is not robust, being prone to disintegration when the program is modified. With the exception of multiple labels for a single computation, fall-throughs should be used sparingly.

As a matter of good form, put a **break** after the last case (the **default** here) even though it's logically unnecessary. Some day when another case gets added at the end, this bit of defensive programming will save you.

Exercise 3-3: Write a function **expand(s, t)** which converts characters like newline into visible escape sequences like \n as it copies **s** to **t**. Use a **switch**. □

3.6 Loops — while and for

We have already encountered the **while** and **for** loops. In

while (*expression*)
    statement

the *expression* is evaluated. If it is non-zero, *statement* is executed and *expression* is re-evaluated. This cycle continues until *expression* becomes false, at which point execution resumes after *statement.*

The **for** statement

for (*init*; *expression*; *re-init*)
    statement

is equivalent to

```
init;
while (expression) {
        statement
        re-init;
}
```

init and *re-init* are expressions. Any of the three parts can be omitted. If *init* or *re-init* is left out, it is simply dropped from the expansion. If *expression* is not present, it is taken as permanently true, so

```
for ( ; ; ) {
        ...
}
```

is an "infinite" loop, presumably to be broken by other means (such as a **break** or **return**).

Whether to use **while** or **for** is largely a matter of taste. For example, in

```
/* skip white space characters */
while ((c = getchar( )) == ' ' || c == '\n' || c == '\t')
        ;
```

there is no initialization or re-initialization, so the **while** seems most natural.

The **for** is clearly superior when there is a simple initialization and re-initialization, since it keeps the loop control statements close together and visible at the top of the loop. This is obvious in

```
for (i = 0; i < N; i++)
```

which is the C idiom for processing the first **N** elements of an array, the analog of the Fortran or PL/I DO loop.

As a larger example (which also makes use of some other constructs we've discussed), here is another version of **atoi** for converting a string to its numeric equivalent. This one is more general; it copes with optional leading white space and an optional + or − sign. (Chapter 4 shows **atof**, which does the same conversion for floating point numbers.)

The basic structure of the program reflects the form of the input:

skip blanks, if any
get sign, if any
get integer part

Each step processes its part if present, and leaves things in a clean state for the next part. If the whole process terminates on a character that is not white space, then there was some sort of error, although we won't do anything about that yet.

```
atoi(s)/* convert s to integer */
char s[ ];
{
        int i, n, sign = 1;

        for (i = 0; s[i] == ' ' || s[i] == '\n' || s[i] == '\t'; i++)
                ;       /* skip white space */
        if (s[i] == '+' || s[i] == '-')     /* sign */
                sign = (s[i++]=='+') ? 1 : -1;
        for (n = 0; s[i] >= '0' && s[i] <= '9'; i++)  /* integer part */
                n = 10 * n + s[i] - '0';
        return(sign * n);
}
```

The advantages of keeping loop control centralized are even more obvious for nested loops. The following function is a Shell sort, for sorting an array of integers. The basic idea of the Shell sort is that in early stages, far apart elements are compared, rather than adjacent ones, as in simple interchange sorts. This tends to eliminate large amounts of disorder quickly, so later stages have less work to do. The interval between compared elements is gradually decreased to one, at which point the sort effectively becomes an adjacent interchange method.

```
shell(v, n)     /* sort v[0] ... v[n-1] into increasing order */
int v[ ], n;
{
        int gap, i, j, k;

        for (gap = n/2; gap > 0; gap /= 2)
                for (i = gap; i < n; i++)
                        for (j=i-gap; j>=0 && v[j]>v[j+gap]; j -= gap) {
                                k = v[j];
                                v[j] = v[j+gap];
                                v[j+gap] = k;
                        }
}
```

The outermost loop controls the gap between compared elements, shrinking it from n/2 by a factor of two each pass until it becomes zero. The middle loop compares elements separated by gap; the innermost loop reverses any that are out of order. Since gap is eventually reduced to one, all elements are eventually ordered correctly.

Writing this code with while expands it by *nine* lines; the conciseness of the for adds clarity.

3.7 Loops — do-while

The while and for share the desirable attribute of testing the loop at the top, rather than at the bottom, as we discussed in Chapter 1. The third loop in C, the do-while, tests at the bottom *after* making one pass through the body. The syntax is

```
do
        statement
while (expression)
```

statement is executed, then *expression* is evaluated. If it is true, *statement* is evaluated again, and so on. If the expression becomes false, the loop terminates.

As might be expected, the do-while is much less used than while and for. Nonetheless, it is from time to time valuable, as in this function for converting a number to a character string, the inverse of atoi. The job is slightly more complicated than might be thought at first, because the easy methods of generating the digits generate them in the wrong order. We have chosen here to generate the string backwards, then reverse it.

```
itoa(n, s)      /* convert n to characters in s */
char s[];
int n;
{
        int c, i, j, sign = 1;

        if (n < 0) {    /* record sign */
                sign = -1;
                n = -n;
        }
        i = 0;
        do {            /* generate digits in reverse order */
                s[i++] = n % 10 + '0';
        } while ((n /= 10) > 0);
        if (sign < 0)
                s[i++] = '-';
        s[i] = '\0';
        for (i--, j = 0; j < i; i--, j++) {     /* reverse s */
                c = s[i];
                s[i] = s[j];
                s[j] = c;
        }
}
```

COMMAS!

The do-while is necessary, or at least convenient, since at least one character must be installed in the array s, regardless of the value of n. We also used braces around the single statement that makes up the body of the do-while, even though it is unnecessary, so the hasty reader will not mistake the while

part for the *beginning* of a **while** loop.

Exercise 3-4: Our version of **itoa** does not handle "negative infinity", that is, the value of n equal to $-2^{wordsize}$. Explain why not. Modify it to print that value correctly. □

3.8 Break and Continue

It is useful to be able to control loop exit by other means than simply testing at the top. The **break** statement provides an early exit from **for**, **while**, and **do**, just as from **switch**.

The following program trims blanks and tabs from each line of input, using a **break** to exit from a loop when the last non-blank, non-tab is found.

```
main( )       /* remove trailing blanks and tabs */
{
        int n;
        char line[MAXLINE];

        while ((n = getline(line, MAXLINE)) >= 0) {
                while (--n >= 0)
                        if (line[n] != ' ' && line[n] != '\t')
                                break;
                line[n+1] = '\0';
                printf("%s\n", line);
        }
}
```

getline returns the length of the line with the terminating \n removed. The inner **while** loop starts at the last character of **line** (recall that **--n** decrements **n** before using the value), and scans backwards looking for a non-blank, non-tab. The loop is broken when one is found, or when **n** becomes negative (that is, when the beginning of the string is reached). This is correct behavior even when an empty line is encountered, for which **n** is zero.

An alternative to **break** is to put the testing in the loop itself:

```
while ((n = getline(line, MAXLINE)) >= 0) {
        while (--n >= 0 && (line[n] == ' ' || line[n] == '\t'))
                ;
        ...
}
```

This is inferior to the previous version, because the test is much harder to understand. Compound tests with parentheses and different operators should be avoided.

The **continue** statement is related to **break** (although empirically much less used); it causes the *next iteration* of the enclosing loop (**for**, **while**, **do**) to begin. In the **while** and **do**, this means that the test part is executed immediately; in the **for**, control passes to the re-initialization step. (**continue** applies

only to loops, not to **switch**.)

As an example, this fragment processes only positive elements in the array **a**; negative values are skipped.

```
for (i = 0; i < N; i++) {
        if (a[i] < 0)  /* skip negative */
                continue;
        ...     /* do positive */
}
```

continue seems most often used in situations like this, when the part of the loop that follows is complicated, so that turning a logical condition around and indenting another level would complicate the program too much.

3.9 goto's and Labels

C provides the infinitely-abusable **goto** statement, and labels to branch to. Formally, the **goto** is never necessary, and in practice it is almost always easy to write code without it. Certainly we don't use **goto** in this book.

Nonetheless, we will suggest a few situations where **goto**'s may find a place. The most common use is in error-handling code when it is necessary to abandon processing in some deeply nested structure.

```
        for ( ... )
                for ( ... )
                        ...
                        if (disaster)
                                goto error;
...

error:
        clean up the mess
```

This organization is handy if the error-handling code is non-trivial, and if errors can occur in several deeply nested places.

As another example, consider the problem of finding the first negative element in a two-dimensional array. One possibility is

```
        for (i = 0; i < N; i++)
                for (j = 0; j < M; j++)
                        if (v[i][j] < 0)
                                goto found;
        /* didn't find */

found:
        /* found one at position i, j */
```

This can be written without a **goto**, at the price of some repeated tests or an extra variable; this is always the case.

```
found = 0;
for (i = 0; i < N && !found; i++)
        for (j = 0; j < M && !found; j++)
                found = v[i][j] < 0;
if (found)
        /* it was at i-1, j-1 */
else
        /* not found */
```

A label has the same form as a variable name. It can be attached to any statement in the same function as the **goto**. It is currently possible to branch into a set of statements in braces, or to the **else** part of an **if**, etc., but these are bad practices indeed.

CHAPTER 4: FUNCTIONS AND PROGRAM STRUCTURE

Functions break large computing tasks into smaller ones, and enable people to build on what others have done instead of starting over from scratch. Appropriate functions can often hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole, and easing the pain of making changes.

C has been designed to make the use of functions easy and efficient. C programs generally consist of numerous small functions rather than a few big ones.

Most programmers are familiar with "library" functions for input and output (**getchar**, **putchar**) and numerical computations (**sin**, **cos**, **sqrt**). In this chapter we will show more about writing new functions.

4.1 Basics

To begin, let us design and write a program to print each line of its input that contains a particular "pattern" or string of characters. (This is a special case of the Unix utility program *grep*). For example, searching for the pattern "the" in the set of lines

```
Now is the time
for all good
men to come to the aid
of their party.
```

will produce the output

```
Now is the time
men to come to the aid
of their party.
```

The basic structure of the job falls neatly into three pieces:

```
while (there's a new line)
        if (the line matches the pattern)
                print it
```

If you put the code for all of this in the main routine, you'll create an unbelievable mess. (Try it.) A better way is to use the natural structure to

advantage, by making each part a separate function. Irrelevant details can be buried in the functions, and the chance of unwanted interactions minimized. Three small pieces are easier to deal with than one big one.

"while there's a new line" is **getline**, a function that we wrote in Chapter 1, and "print it" is **printf**, which someone has already provided for us. This means we need only write a routine which decides if the line contains an occurrence of the pattern.

We can solve that problem by stealing a design from PL/I: the function **index(s, t)** returns the position or index in the string **s** where the string **t** begins. We use 0 rather than 1 as the starting position in **s**, since arrays begin at position zero, and then **index** can return −1 if **s** doesn't contain a **t**. The **index** function centralizes some fairly messy logic in a single place, and provides a routine that may well prove useful in other contexts as well. (**index** wasn't written for this book — we wrote it for something else quite a while ago.) If we later need more sophisticated pattern matching we can **index** by a more general pattern matcher; the rest of the code remains the same.

Given this much design, filling in the details of the program is straightforward. Here it is in its entirety, including **getline**, so you can see how the pieces fit together.

```
main( )        /* find all lines matching a pattern */
{
        char line[MAXLINE];

        while (getline(line, MAXLINE) >= 0)
                if (index(line, "the") >= 0)
                        printf("%s\n", line);
}

getline(s, lim)        /* read line into s; return length */
char s[ ];
int lim;
{
        int i, c;

        for (i = 0; i < lim-1 && (c = getchar( )) != '\n' && c != EOF; i++)
                s[i] = c;
        s[i] = '\0';
        return(c == EOF ? -1 : i);
}
```

```
index(s, t)    /* return index of t in s, -1 if none */
char s[ ], t[ ];
{
        int i, j, k;

        for (i = 0; s[i] != '\0'; i++) {
                for (j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++)
                        ;
                if (t[k] == '\0')
                        return(i);
        }
        return(-1);
}
```

More commas

Burying the pattern to be searched for ("the") in the middle of main is not the most general of mechanisms, but we haven't yet discussed how to initialize character arrays; we'll return to this topic in due course.

Each function has the form

```
name(argument list, if any)
argument declarations, if any
{
        statements of function, if any
}
```

As noted, the various parts may be absent; a minimal function is

```
dummy( ) { }
```

which does nothing. (A do-nothing function is sometimes useful as a place holder during program development.)

A program with multiple functions is just a set of individual function declarations, as shown. Each function is self-contained, and the only communication between the pieces is (in this case) by arguments and values returned by the functions. The functions can occur in any order, and the source program can be split into multiple files, so long as no function is split. And of course the definition of EOF has to be accessible as getline is compiled.

The return statement is the mechanism for returning a value from the called function to its caller. Any expression can follow return, as in

```
return (expression)
```

It is common practice, though not required, to put parentheses around the *expression*, as we have done.

The calling function is free to ignore the returned value if it wishes. Furthermore, there need be no expresion after return; in that case, no value is returned to the caller. If there is no return statement in a function, control returns to the caller with no value when the function "falls off the end" by reaching the closing right brace. It is not illegal, but probably a sign of trouble, if a function returns a value from one place and no value from another.

4.2 Functions Returning Non-Integers

So far, none of our programs has contained any declaration of the type of a function. This is because by default a function is implicitly declared by its appearance in an expression or statement, such as

```
while (getline(line, MAXLINE) >= 0)
```

In such cases, the context of a name followed by a left parenthesis is sufficient declaration. Furthermore, by default the function is assumed to return an int. Since char promotes to int in expressions, there is no need to declare functions that return char. These assumptions cover the majority of cases, including all of our examples so far.

But what happens if a function must return some other type? Many numerical functions like sqrt, sin, and cos return double; other specialized functions return other types.

To illustrate how this is done, let us write and use the function atof(s), which converts the string s to its floating point equivalent. atof is an extension of atoi, which we wrote versions of in Chapter 2 and 3; it handles an optional sign and decimal point, and the presence or absence of either integer part or fractional part. (This is *not* a high-quality input conversion routine; that takes more space than we care to use.)

To communicate the fact that atof returns float, the function itself must declare the type of value it returns, since it is not int. The type precedes the function name, like this:

```
float atof(s)  /* convert s to float */
char s[ ];
{
        float den = 1.0, val = 0.0;
        int i, sign = 1;

        for (i = 0; s[i] == ' ' || s[i] == '\n' || s[i] == '\t'; i++)
                ;       /* skip white space */
        if (s[i] == '+' || s[i] == '-')     /* sign */
                sign = (s[i++]=='+') ? 1 : -1;
        for (val = 0; s[i] >= '0' && s[i] <= '9'; i++)        /* integer part */
                val = 10 * val + s[i] - '0';
        if (s[i] == '.')        /* decimal point */
                i++;
        for ( ; s[i] >= '0' && s[i] <= '9'; i++) /* fraction part */
                val += (s[i] - '0') / (den *= 10.0);
        return(sign * val);
}
```

Second, and just as important, the *calling* routine must state that this function returns a non-int value. The declaration is shown in the following rudimentary desk calculator (barely adequate for check-book balancing), which reads one number per line, optionally preceded by a sign, and adds them all

up, printing the sum after each input.

```
main( )        /* rudimentary desk calculator */
{
        float sum, atof( );
        char line[MAXLINE];

        sum = 0;
        while (getline(line, MAXLINE) >= 0) {
                sum += atof(line);
                printf("\t%f\n", sum);
        }
}
```

The declaration

```
        float sum, atof( );
```

says that sum is a float variable, and that atof is a function that returns a float value. Unless atof is properly declared in both places, C assumes that it returns an integer, and you'll get nonsense answers.

If atof and main are in the same source file, a type inconsistency will be detected by the compiler. But (as is more likely) if atof were compiled separately. the mismatch would not be detected, atof would return a float, which main would treat as an int, and garbage would result.

Given atof, we can write atoi (convert a string to int) in terms of it:

```
atoi(s) /* convert s to integer */
char s[ ];
{
        float atof( );

        return(atof(s));
}
```

Notice the structure of the declarations. Type conversion in return is a general rule — the value of the expression in

return(*expression*)

is converted to the type of the function before the return is taken. Therefore, the value of atof, a float, is converted automatically to int when it appears in a return, since the function atoi returns an int. (The conversion of float to int truncates any fractional part, as discussed in Chapter 2.)

As a further example, the following program computes the average and standard deviation of a set of one-per-line numbers. If a is the average of n numbers $x_1, \ldots, x_n$, the standard deviation is

$$\left(\frac{\Sigma(x_i)^2}{n-1} - a^2 \right)^{1/2}$$

The corresponding program is

```
main( )        /* avg and standard deviation */
{
        float atof( ), val;
        double dmax( ), sqrt( ), sum, sumsq, avg, std_dev;
        char line[MAXLINE];
        int n;

        sum = sumsq = n = avg = std_dev = 0;
        while (getline(line, MAXLINE) >= 0) {
                n++;
                val = atof(line);
                sum += val;
                sumsq += val * val;
        }

        if (n > 0)
                avg = sum / n;
        if (n > 1)
                std_dev = sqrt(dmax(sumsq/(n-1) - avg*avg, 0.0));
        printf("%d numbers: avg = %f, std dev = %f\n",
                n, avg, std_dev);
}
```

Here we have to declare **sqrt** to be a function that returns a **double**. The function **dmax**, which computes the maximum of two **double**'s or **float**'s, also has to be declared in **main** and written:

```
double dmax(x, y)
double x, y;
{
        return(x > y ? x : y);
}
```

The second argument passed to **dmax** is 0.0, not 0. Since **dmax** expects a **double**, passing it the **int** 0 could lead to disaster, or at least a wrong answer.

By the way, there is no entirely satisfactory way to write a function that accepts a variable number of arguments, because there is no portable way for the callee to determine how many arguments were actually used in a given call. Thus, you can't write a version of **dmax** that will take an arbitrary number of arguments, as will the MAX functions of Fortran and PL/I.

It is generally safe to deal with a variable number of arguments if the called function doesn't use an argument which was not actually supplied, and if the types are consistent. **printf**, the most common C function with a variable number of arguments, uses information from the first argument to determine how many other arguments are present, and what their types are. It fails badly if the types are not what the first argument says. Alternatively, it is possible to

mark the end of the argument list in some agreed-upon way, such as an "end of list" marker.

4.3 Arguments — Call by Value

In Chapter 1 we discussed the fact that function arguments are passed by value, that is, the called function receives a private, temporary copy of each argument, not its address. This means that the function cannot affect the original argument in the calling function (although if the argument was an array name, it can certainly affect array elements). Thus each argument is in effect a local variable initialized to the value the function was called with.

In Chapter 5 we will discuss the use of pointers to permit functions to affect non-arrays in calling functions.

4.4 External Variables

Speaking formally, a C program consists of a set of external objects, which are either functions or variables. Like functions, external variables are "global," that is, they are potentially accessible to any function. In this sense, external variables are analogous to Fortran COMMON or PL/I EXTERNAL.

Because external variables are globally accessible, they provide an alternative to function arguments and return values for communicating data between functions. Any function may read or write data in an external variable merely by referring to it by name. For example, in the pattern-finding example shown earlier in this chapter, we could make the line buffer and the pattern external, and refer to them in **main** by including an appropriate **extern** declaration, like this:

```
char  line[MAXLINE];     /* line buffer */
char  pattern[ ]    = "the";     /* pattern to search for */

main( )
{
        extern char line[ ], pattern[ ];

        while (getline(line, MAXLINE) >= 0)
                if (index(line, pattern) >= 0)
                        printf("%s\n", line);
}
```

There are circumstances under which the **extern** declaration may be omitted, which we will discuss in a moment.

If a large number of variables must be shared among functions, external variables are more convenient than long argument lists, and will be rather more efficient. As pointed out in Chapter 1, however, this reasoning should be applied with some caution, for it can have a bad effect on program structure, and lead to program with many data connections between functions. Consider the pattern finding program again. Making **line** and **pattern** external is reasonable. It would be unreasonable, though, to write **getline** to store the input line

in a specific external variable; an argument is more flexible, since then **getline** can read into different arrays with different calls. And it would be a grave error to write **index** with the names of **line** and **pattern** built in instead of passed as arguments, since this would severely limit the utility of a general purpose routine.

A second reason for using external variables is that there are fewer restrictions on how they may be initialized. In particular, automatic arrays may not be initialized, but external ones may. In the pattern finding program, the array **pattern** can be initialized by the declaration

```
char  pattern[ ]  = "the";
```

if it is an external variable but not if automatic. We will treat initialization near the end of this chapter.

The third reason for using external variables is their lifetime. Automatic variables are internal to a function; they come into existence when the routine is entered, and disappear when it is left. External variables, on the other hand, are permanent. They do not come and go, so they retain values from one function invocation to the next. Thus if two functions must share some data, yet neither calls the other, the shared data *must* be kept as external variables; there is no other way for it to be both permanent and accessible.

Let us examine this issue further in the context of another example. It is often the case that a program reading input cannot determine that it has read enough until it has read too much. One instance is collecting the characters that make up a name: until the first non-alphabetic character is seen, the name is not complete. But then the program has read a character that it is not really ready to deal with.

Dealing with this situation can tremendously complicate a program if we let it. Each time we need another character, we much check whether to read a new character or use the one we already have. Tangling this up with the logic of what to *do* with each character makes an unreadable mess.

The problem would be solved if it were possible to "un-read" the unwanted character. Then, every time the program reads one character too many, it could push it back on the input, so the rest of the code could behave as if it had never been read.

Fortunately, it's easy to simulate un-getting a character, by writing a pair of cooperating functions. **getc** delivers the next input character to be considered. **ungetc** puts a character back on the input, so that the next call to **getc** will return it again.

How they work together is simple. **ungetc** puts the pushed-back characters into a shared buffer — a character array. **getc** reads from the buffer if there is anything there; it calls **getchar** if the buffer is empty. There must also be an index variable which records the position of the current character in the buffer.

Since the buffer and the index are shared by **getc** and **ungetc** and must retain their values between calls, they must be external to both routines. As we saw in Chapter 1, a variable is external if it is defined outside of the body

of any function. Thus we can write getc, ungetc, and their shared variables as:

```
char  buf[BUFSIZE];      /* pushback buffer for getc and ungetc */
int   bufp  = -1;        /* current character position in buf */

getc( )      /* get a (possibly pushed back) character */
{
      extern char buf[ ];
      extern int bufp;

      return(bufp >= 0 ? buf[bufp--] : getchar( ));
}

ungetc(c)    /* push character back on input */
char c;
{
      extern char buf[ ];
      extern int bufp;

      if (bufp >= BUFSIZE)
            printf("Pushback overflow\n");
      else
            return(buf[++bufp] = c);
}
```

We have used an array for the pushback, rather than a single character, since the generality may come in handy later.

Exercise 4-1: Write a routine ungets(s) which will push back an entire string onto the input. Should ungets know about buf and bufp, or should it just use ungetc? □

Exercise 4-2: Suppose that there will never be more than one character of pushback. Modify getc and ungetc accordingly. □

4.5 Scope Rules

The use of external variables raises a number of questions about *scope*, that is, when variables are known to functions implicitly, and when declarations are needed.

The functions and external variables that make up a C program need not all be compiled at the same time: the source text of the program may be kept in several files, and previously compiled routines may be loaded from libraries. The two questions of interest are

(1) How are declarations written so that variables are properly declared during compilation?

(2) How are declarations set up so that all the pieces are properly connected when the program is loaded?

C determines the scope of external variables by where they are defined in the source file. If an external variable is declared at some point in a file, then it may be referenced thereafter as if it had appeared in an **extern** declaration. For example, if **buf**, **bufp**, **getc**, and **ungetc** are declared in a single file, in the order shown above, that is,

```
char  buf[BUFSIZE];
int   bufp  = -1;

getc( ) ...

ungetc( ) ...
```

then there is no need for any **extern** declarations with **getc** and **ungetc**, and it is common practice to omit them.

On the other hand, if an external variable is to be used before it is defined, or if it is defined in a *different* file, then an **extern** declaration is mandatory.

There can only be one external *definition* of a variable among all the files that make up the source program; other files may contain **extern** declarations to access it. (There may also be an **extern** declaration in the file that defines it.) Any initialization of such a variable must go with the definition.

Thus, as an unlikely arrangement, but typical of larger programs, **buf** and **bufp** could be defined and initialized in one file, and the functions **getc** and **ungetc** defined in another. Then these definitions and declarations are necessary to tie them together:

In file 1:

```
char  buf[BUFSIZE];     /* pushback buffer for getc and ungetc */
int   bufp  = -1;       /* current character position in buf */
```

In file 2:

```
extern char buf[ ];
extern int bufp;

getc( )
{
        ...
}

ungetc(c)
char c;
{
        ...
}
```

Because the **extern** declarations lie outside both **getc** and **ungetc**, they apply to both; one declaration suffices for all of file 2.

4.6 Static Variables

Static variables are a third class of storage, in addition to the **extern** and **auto** (automatic) that we have already met.

static variables may be either internal or external. Internal **static** variables are local to a particular function just as **auto** variables are, but unlike **auto**'s, they remain in existence rather than coming and going each time the function is activated. This means that internal **static** variables provide private but permanent storage in a function. Character strings declared within a function, such as the arguments of **printf**, are internal static.

An external static variable is known within the *file* in which it is declared, but not any other file. External **static** provides a way to hide names like **buf** and **bufp** in the **getc-ungetc** combination, which must be external, yet which should not visible to users of **getc** and **ungetc**. If the two routines and the two variables are compiled in one file, as

```
static char   buf[BUFSIZE];     /* pushback buffer for getc and ungetc */
static int    bufp  = -1;       /* current character position in buf */

getc( )
{
        ...
}

ungetc(c)
char c;
{
        ...
}
```

then no other routine will be able to access **buf** and **bufp**; in fact, they will not conflict with the same names in other files.

Static storage, whether internal or external, is specified by prefixing the normal declaration with the word **static**:

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

The variable will be external or internal according to where the declaration occurs.

As a final note, functions may also be declared **static**; this makes them inaccessible outside of the file in which they are declared.

4.7 Register Variables

The fourth and final storage class is called **register**, which may be used to advise the compiler that the variables declared will be heavily used. The declaration is

```
register int x;
register char c;
```

and so on; the **int** part may be omitted.

In practice, only a few variables and only variables of certain types can be accomodated in registers on most machines. The specific restrictions vary from machine to machine; on the PDP-11, three register variables are allowed; they must be **int**, **char**, or pointer.

4.8 Block Structure

C is not a block structured language in the sense of PL/I or Algol and its derivatives. In particular, functions may not be defined within other functions; every function is external.

On the other hand, variables can be used in a block-structured way. Declarations of variables may occur after *any* left brace, not just the one that brackets the function. Variables declared in this way supersede any identically named variables in outer blocks, and remain in existence until the matching right brace.

The declaration of variables in a function is one instance of this. Given the declarations

```
int x;

f( )
{
        int x;
        ...
}
```

then within the function f, occurrences of x refer to the internal variable; outside of f, they refer to the external x. (Be sure that this is what you want; it can be hard to see this kind of error if it isn't.)

4.9 Initialization

Initialization has been mentioned in passing many times so far, but always peripherally to some other topic. This section summarizes some of the rules, now that we have discussed the various storage classes.

In the absence of explicit initialization, external and static variables are initialized to zero; automatic and register variables have undefined (i.e., garbage) values.

Ordinary variables (not arrays) may be initialized when they are declared, by following the name with an equals sign and a constant value:

```
int     x = 1;
char    c = 'X';
float   twopi = 2 * 3.141592654;
```

For external and static variables, the initialization is done once, conceptually at compile time. For automatic and register variables, it is done each time the function is entered.

For automatic and register variables, the initializer is not restricted to a constant: it may in fact be any valid expression involving previously defined values. For example, the initializations of the binary search program in Chapter 3 could be written as

```
binary(x, v, n)
int x, v[ ], n;
{
        int low = 1, high = n-1; mid;
        ...
}
```

instead of

```
binary(x, v, n)
int x, v[ ], n;
{
        int low, high, mid;

        low = 1;
        high = n - 1;
        ...
}
```

In effect, initializations of automatic variables are just shorthand for assignment statements.

"Aggregates", that is, arrays and structures, may not be initialized if they are automatic. External and static aggregates may be initialized, by following the declaration by a list of initializers enclosed in braces and separated by commas. For example, we can rewrite the character counting program of Chapter 1 from the original

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

        nwhite = nother = 0;
        for (i = 0; i < 10; i++)
                ndigit[i] = 0;
        ...
}
```

to

```
int     nwhite          = 0;
int     nother= 0;
int     ndigit[10] ={0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

main( )        /* count digits, white space, others */
{
        int c, i;
        ...
}
```

The initializations are actually unnecesary since all are zero, but it's good form to make them explicit anyway.

Character arrays are a special case of initialization. Since they occur so frequently, a string may be used instead of the braces and commas notation:

```
char  pattern[ ]     = "the";
```

It is also true that in all cases, the compiler will fill in the length of an array if there is a list of initializers. If there are fewer initalizers than the specified size, the others will be zero.

4.10 Recursion

C functions may be used recursively; that is, a function may call *itself* either directly or indirectly. One traditional example involves printing a number as a character string. The trouble is that the digits are generated in the wrong order: low-order digits are available before high-order digits, but they have to be printed the other way around.

There are two solutions to this problem. One is to store the digits in an array as they are generated, then print them in the reverse order, as we did in Chapter 3 with itoa.

```
printd(n)       /* print n in decimal */
int n;
{
        char s[10];
        int i;

        if (n < 0) {
                putchar('-');
                n = -n;
        }
        i = 0;
        do {
                s[i++] = n % 10 + '0';   /* convert last char to digit */
        } while ((n /= 10) > 0);   /* discard last digit */
        while (--i >= 0)
                putchar(s[i]);
}
```

The alternative is a recursive solution, in which each call of printd first calls itself to cope with any leading digits, then prints the trailing digit. The resulting code:

```
printd(n)       /* print n in decimal (recursive) */
int n;
{
        int i;

        if (n < 0) {
                putchar('-');
                n = -n;
        }
        if ((i = n/10) != 0)
                printd(i);
        putchar(n % 10 + '0');
}
```

Recursion generally provides no saving in storage, since somewhere a stack of the values being processed has to be maintained, nor will it be faster.

But recursive code is more compact, and often easier to write and understand. Recursion is especially convenient for recursively defined data structures like trees; we will see a nice example in Chapter 6.

Exercise 4-3: Adapt the ideas of **printd** to write a recursive version of **itoa**; that is, convert an integer into a string with a recursive routine. □

Exercise 4-4: Write recursive and non-recursive versions of the function **reverse(s)**, which reverses the string **s**. □

4.11 The C Preprocessor

C provides certain language extensionss by means of a preprocessor which is actually a rather simple macro processor. The **#define** capability which we have used is the most common of these extensions. The other aspect of the c preprocessor is the ability to include during compilation the contents of other files.

#include

To facilitate handling large collections of declarations (among other things) C provides a file inclusion feature. Any line that begins with

#include "*filename*"

is replaced by the contents of the file *filename.* Commonly a line or two of this form appears at the beginning of a file, to include common **#define** statements and **extern** declarations for global variables.

#include is definitely the preferred way to tie the declarations together for a large program. It pretty well guarantees that all the source files will be supplied with the same version of definitions and variable declarations, and thus eliminates a particularly nasty kind of bug.

Macros

A definition of the form

```
#define    YES  1
```

calls for a macro substitution of the simplest kind — replacing one fixed string by another. It is possible, however, to specify a macro with arguments, so the replacement text depends on the way the macro is called. As an example, we could define a macro called **max** like this:

```
#define    max(a, b)    ((a > b) ? a : b)
```

Then a line in a program like

```
x = max(p+q, r+s);
```

will be replaced by the text

```
x = ((p+q > r+s) ? p+q : r+s);
```

This provides a maximum "function" that expands into in-line code rather than a function call. So long as the arguments are treated consistently, this

macro will serve for any data type; there is no need for **max** and **dmax**, as there would be with functions.

Of course, if you examine the expansion of **max** above, you will notice some pitfalls. The expressions are evaluated twice; this is bad if they involve side effects. Some care has to be taken with parentheses to make sure the order of evaluation is preserved. And there are even some purely lexical problems: there can be no space between the macro name and the left parenthesis that introduces its argument list.

Nonetheless, macros are quite valuable. One practical example is the standard I/O library to be described in Chapter 7, in which commonly used functions like **getchar** and **putchar** are defined as macros (obviously **putchar** needs an argument), thus avoiding the overhead of a function call per character processed.

Exercise 4-5: Write macros for the functions **lower** and **upper** discussed in Chapter 2. □

Exercise 4-6: Write macros for **getc** and **ungetc** from this chapter. □

CHAPTER 5: POINTERS AND ARRAYS

A pointer is a variable that contains the address of another object. Pointers are very much used in C, partly because they are sometimes the only way to express some computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways.

Pointers have been lumped with the **goto** statement as a marvelous way to create impossible-to-understand programs. This is certainly true for careless programming, 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.

5.1 Pointers and Addresses

Since a pointer contains the address of an object, it is possible to access the object "indirectly" through the pointer. Suppose that x is a variable, say an int, and that px is a pointer, created in some as yet unspecified way. Then the variable px can be set to the *address* of x by the assignment

```
px = &x;
```

The unary operator & gives the address of an object (it can be applied only to variables and functions; constructs like &(x+1) and &3 are illegal). So now px contains the address of x.

The unary operator * is the inverse of &; it treats its operand as a variable that contains the address of the ultimate target, and accesses indirectly through that address to fetch the contents. Thus if y is also an int,

```
y = *px;
```

assigns to y the contents of whatever px points to. So the sequence

```
px = &x;
y = *px;
```

is equivalent to

```
y = x;
```

It is also necessary to declare the variables participating in all of this:

```
int x, y;
int *px;
```

The declaration of x and y is what we've seen all along. The declaration of the pointer px is new.

```
int *px;
```

is intended as a mnemonic; it says that the combination *px is an int, that is, if px occurs in the context *px, it is equivalent to a variable of type int. This reasoning is useful in all cases involving complicated declarations. It is quite analogous to a declaration like

```
float atof( );
```

which says that atof( ) is an object of type float.

You should also note the implication in the declaration that a pointer is constrained to point to a particular kind of object. It is fraught with peril to declare a variable to be a pointer to one type, then use it as a pointer to another.

Pointers can occur in expressions. For example, if px points to the integer x, then *px can occur in any context where x could.

```
y = *px + 1
```

sets y to 1 more than x;

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

prints the current value of x; and

```
d = sqrt( (double) *px);
```

produces in d the square root of x, which has to be coerced into a double before being passed to sqrt.

In expressions like

```
y = *px + 1
```

* and & bind more tightly than arithmetic operators, so this adds 1 to whatever px points at, and assigns it to y. We will return shortly to the meaning of

```
y = *(px+1)
```

Pointer references can also occur on the left side of assignments.

```
*px = 0;
```

sets x zero, and

```
*px += 1
```

increments it, as does

```
(*px)++
```

The parentheses are necessary here; without them, the expression would

increment px instead of what it points to, because unary operators like * and ++ are evaluated right to left.

Finally, since pointers are variables, they can be manipulated as other variables can. If py is another pointer to int, then

```
py = px
```

has the obvious meaning of making py point to whatever px points to.

5.2 Pointers and Function Arguments

Since C passes arguments to functions by "call by value", there is no direct way for the called function to alter a variable in the calling function. What do you do if you really have to change an ordinary argument? For example, a sorting routine might exchange two out-of-order elements with a function called swap. It is not enough to write

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

        temp = x;
        x = y;
        y = temp;
}
```

because swap *can't* affect the arguments x and y. Fortunately, pointers provide a way to obtain the desired effect. In the calling program, pass not the argument but a pointer to it:

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

Since the operator & gives the *address* of a variable, &a is a pointer to a.

In swap itself, declare the arguments to be pointers, and treat them as such:

```
swap(px, py) /* interchange *px with *py */
int *px, *py;
{
        int temp;

        temp = *px;
        *px = *py;
        *py = temp;
}
```

One common use of pointer arguments is in functions that must return more than a single value. (You might say that swap returns two values, the new values of its arguments.) As an example, a function which reads input and

converts it to a stream of integers has to return the value it found, and also some indication of whether or not end of file has occurred. And these values have to be returned in different places, for no matter what value is used for EOF, that could also be some legitimate integer from the input.

One solution, which is based on the input function **scanf** that we will talk about in Chapter 7, is to write a function **getint** which returns 1 if it really found a number, 0 if it found no number, and −1 if it found end of file. The actual numeric value is returned in the argument, which then must be a pointer to an integer. In this way end of file and error signals can be distinguished from legal numeric values.

The call

```
int n, stat;
...
stat = getint(&n);
```

sets n to the next integer found in the input and **stat** to the status returned by **getint**.

```
getint(pn)    /* get next integer from input */
int *pn;
{
        int c, sign = 1;

        while ((c = getchar( )) == ' ' || c == '\n' || c == '\t')
                ;       /* skip white space */
        if (c == '+' || c == '-') {       /* sign */
                sign = (c=='+') ? 1 : -1;
                c = getchar( );
        }
        for (*pn = 0; c >= '0' && c <= '9'; c = getchar( ))       /* integer part */
                *pn = 10 * *pn + c - '0';
        *pn *= sign;
        if (c == ' ' || c == '\n' || c == '\t')
                return(1);    /* normal integer found */
        else if (c == EOF)
                return(-1);
        else
                return(0);    /* some error */
}
```

Throughout **getint**, *pn is used as an ordinary **int** variable.

Exercise 5-1: Write **getfloat**, the floating point analog of **getint**. What type does **getfloat** return as its function value? □

5.3 Pointers and Arrays

There is a strong relationship between pointers and arrays, strong enough that pointers and arrays really should be treated simultaneously. Any operation which can be achieved by array subscripting can also be done with pointers. The pointer version will in general be rather more efficient but, at least to the uninitiated, somewhat harder to grasp immediately.

The declaration

```
int a[10]
```

defines an array **a** of size 10, that is a block of 10 consecutive objects named **a[0]**, **a[1]**, ..., **a[9]**. The notation **a[i]** means the i-th element of the array, that is, i positions from the beginning. If **pa** is a pointer to **int**, declared as

```
int *pa
```

then the assignment

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

sets **pa** to point to the zeroth element of **a**. Now the assignment

```
x = *pa
```

copies **a[0]** into **x**.

If **pa** points to a particular element of the array **a**, then *by definition* **pa + 1** points to the next element, and in general **pa ± i** points i elements before or after **pa**. Thus, if **pa** points to **a[0]**,

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

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

The definition of "adding 1 to a pointer," and by extension, all pointer arithmetic, is that the increment is scaled by the size in storage of the object that is pointed to. This makes the correspondence between indexing and pointer arithmetic very close.

In fact, the association is even closer. Again by definition, a reference to an array of a particular type is converted by the compiler to a pointer to the beginning of the array. In effect, an array name *is* a pointer expression. This has quite a few useful implications.

Since the name of an array is a synonym for the location of the zeroth element, the assignment

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

can also be written as

```
pa = a
```

The array name **a**, being of type "array", is a pointer expression, and can be assigned to a pointer of the same type.

Rather more surprising, at least at first sight, is the fact that a reference to **a[i]** can also be written as ***(a+i)**. By definition, in evaluating **a[i]**, C

converts it to *(a+i) immediately; the two forms are identical. 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 an object i elements beyond a.

There is one difference between an array name and a pointer that should be kept in mind. A pointer is a variable, so pa=0 or pa++ make perfect sense. But an array name is a name, not a variable: constructions like a=0 or a++ are illegal.

There is an exception to this rule, an important one. Within a function, C interprets arguments which are declared "array of ..." as actually being "pointer to ...", so that either subscripting or indexing can be used. 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 kinds of operations if it seems appropriate and clear.

As a consequence of this conversion from array to pointer, it is possible to pass a part of an array to a function. For example,

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

and

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

both pass to the function f the address of element a[2]. Within f, the declaration can read

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

or

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

So as far as f is concerned, the argument either an array or a pointer, and the fact that it is really part of a larger array is of no consequence.

5.4 Address Arithmetic

We have seen that p++ increments p by 1 to point to the next element of whatever kind of object p points to, and p += i increments p to point i elements beyond where it currently does. These and similar constructions are among the simplest and most common forms of pointer or address arithmetic; many others are also possible.

C is quite consistent and regular in its approach to address arithmetic. To illustrate some of its properties, let us build a rudimentary storage allocator (but useful in spite of its simplicity). There are two routines. alloc(n) returns a pointer to n consecutive character positions, which can be used by the receiver of the pointer for storing any characters it likes. free(p) releases the storage thus acquired so it can be later re-used. The routines are "rudimentary" because the calls to free must be made in the opposite order to the calls made on alloc. That is, the storage managed by alloc and free is a stack, or last-in, first-out queue. The standard C library provides an alloc and free which have no such restrictions. In the meantime, many applications really only need a trivial alloc to hand out little pieces of storage of unpredictable sizes at unpredictable times.

The simplest implementation is to have alloc and free hand out pieces of a large character array which we will call alloc_buf. This array is private to alloc and free; since they deal in pointers, not array indices, no other routine need know the name of the array, which can be declared external static. In fact, in practical implementations, the array may well not even have a name; rather it might be obtained by asking the operating system for a pointer to some unnamed free storage. (This is one place where pointers can serve and array indices can't.)

The other piece of data needed is some record of how much of alloc_buf has been used. A pointer to the next free element is convenient; we will call it alloc_p.

When alloc is asked for n bytes, it checks to see if there is enough room, and if so returns the current value of alloc_p, then increments it by n. free(p) merely sets alloc_p to p after a bit of error checking.

```
#define     NULL 0      /* illegal pointer value for error reporting */
#define     ALLOC       1000 /* size of available space */

static char  alloc_buf[ALLOC];  /* storage for alloc and free */
static char  *alloc_p       = alloc_buf; /* next free position */

char *alloc(n)      /* return pointer to n bytes */
int n;
{
        if (alloc_p + n <= alloc_buf + ALLOC)       /* it fits */
                return((alloc_p += n) - n);
        else            /* not enough room */
                return(NULL);
}

free(p)         /* free storage for alloc */
char *p;
{
        if (p >= alloc_buf && p < alloc_buf + ALLOC)
                alloc_p = p;
}
```

Some explanations. In general a pointer can be initialized just as any other variable, though normally the only meaningful values are NULL or some expression involving previously defined pointers of the same type. The statement

```
static char  *alloc_p       = alloc_buf; /* next free position */
```

defines alloc_p to be a character pointer and initializes it to point to alloc_buf, which is the next free position when the program starts. This could have also been written

```
char  *alloc_p      = &alloc_buf[0];
```

but since the array name *is* the address of the zeroth element, it's redundant to do so.

The test

```
if (alloc_p + n <= alloc_buf + ALLOC)       /* it fits */
```

checks if there's enough room to satisfy this request. If there is, the new value of alloc_p would be at most one beyond the end of alloc_buf. If the request can be satisfied, alloc returns a normal pointer (notice the declaration of the function itself). If not, alloc must return some signal that an error has happened. C guarantees that a pointer that really points at data will never be zero, so returning zero signals some abnormal event, in this case, no space. We write NULL instead of zero, however, to indicate more clearly that this is a special value for a pointer.

Tests like

```
if (alloc_p + n <= alloc_buf + ALLOC)        /* it fits */
```

and

```
if (p >= alloc_buf && p < alloc_buf + ALLOC)
```

also show several important facets of pointer arithmetic. First, a pointer and an integer may be added or subtracted. The construction

```
p + n
```

means the nth object beyond the one p currently points to. This is true regardless of the kind of object p is declared to point at; n is scaled as necessary to make it work.

As a corollary, pointer subtraction is valid: if p and q point to members of the same array, p−q is an integer equal to the number of elements between p and q (plus 1). This fact can be used to write a very efficient pointer version of the function **strlen**, which computes the length of a character string.

```
strlen(s)      /* length of s */
char *s;
{
        char *p = s;

        while (*p)
                p++;
        return(p-s);
}
```

As formal parameters in a function definition,

```
char s[ ];
```

and

```
char *s;
```

are exactly equivalent; which one to write is determined largely by how expressions will be written in the function.

p is initialized to s, that is, to point to the first character. In the **while** loop, each character in turn is examined until the \0 at the end is seen. Since \0 is zero, and since **while** tests only whether the expression is zero, we have omitted the explicit test. It could be written as

```
while (*p != '\0')
```

Since 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. Pointer arithmetic is consistent: if we had been dealing with **float**'s, which occupy more storage than **char**'s, and if p were a pointer to **float**, p++ would still advance to the next **float**.

You might find it instructive to compare this to the version in Chapter 2.

The second aspect of pointer arithmetic is that pointers may be compared under certain circumstances. Again, if p and q point to members of the same array, then relations like <, >=, etc., work properly.

```
p < q
```

is true, for example, if p points to an earlier member of the array than does q. The relations == and != also work. Any pointer can be meaningfully compared for equality or inequality with NULL. Other comparisons may not do what you expect, however. For instance, the innocent-looking code

```
int *p;

p = 1;
if (p == 1)
        printf("hello\n");
```

will *never* print "hello" because the 1 in p==1 is scaled to int before the comparison, and thus isn't 1 any more.

All bets are off if you do arithmetic or comparisons with pointers pointing to different arrays. The best that will happen is that your code will work on one machine but collapse mysteriously on another. The worst is that you'll get nonsense on all machines.

All of these considerations mean that we could write another version of alloc which maintains, let us say, float's instead of char's, merely by changing the declaration

```
char *alloc(n)
```

to

```
float *alloc(n)
```

Since all the pointer manipulations automatically take into account the size of the object pointed to, no other parts of alloc and free have to be altered.

5.5 Character Pointers and Functions

By definition, a "string constant", written as

```
"I am a string"
```

is an array of characters. The compiler terminates the array with the character \0 so that programs can find the end.

Although a variable may not hold a string, it may hold a pointer to one, as in the sequence

```
char *message;

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

This assigns the pointer to the string to **message**, which can be manipulated as desired. Note that this is *not* a string copy; only pointers are involved. C does not provide any operators for processing character strings as a unit.

Since one of the most common uses of pointers is accessing character arrays, we will illustrate some aspects of pointers and arrays by studying three useful functions from the standard I/O library to be discussed in Chapter 7.

The first function is **strcpy(s, t)**, which copies the string **t** to the string **s**. The arguments are written in this order by analogy to assignment, where one would say

```
s = t;
```

to assign **t** to **s**. The array version is reminiscent of **strlen**:

```
strcpy(s, t)   /* copy t to s */
char s[ ], t[ ];
{
        int i;

        i = 0;
        while (s[i] = t[i])
                i++;
}
```

For contrast, here is a first version of **strcpy** with pointers.

```
strcpy(s, t)   /* copy t to s */
char *s, *t;
{
        while (*s = *t) {
                s++;
                t++;
        }
}
```

Because arguments are passed by value, **strcpy** can use **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.

In practice, **strcpy** would not be written as we showed it above; it would almost certainly be

```
strcpy(s, t)   /* copy t to s */
char *s, *t;
{
        while (*s++ = *t++)
                ;
}
```

*t++ is a two step operation, with a side effect. Unary operators like * and ++ operate right to left. Thus in a combination like

```
*t++
```

the ++ operator is done first; this produces the *value* of t, and has the *side effect* of incrementing t. Then the * is applied to the old value of t to access the character t pointed to. In the same manner, this character is assigned to the old position that s pointed to and s is incremented. Finally, the value of the assignment statement, the original character that t pointed to, is used: the character is tested to determine whether the loop should go around again. When the \0 has been safely copied over, the loop will terminate.

Although this may seem unduly complicated, the notational convenience is considerable, and the idiom should be mastered.

The second function is strcat(s, t), which concatenates the string t to the end of s. strcat assumes that s is large enough to hold the combination; if this is not the case, it is possible to use alloc to assign new space for the created string. First, strcat with conventional array indexing, as in Chapter 2.

```
strcat(s, t)   /* concatenate t to end of s */
char s[ ], t[ ];
{
        int i, j;

        i = j = 0;
        while (s[i])   /* find end of s */
                i++;
        while (s[i++] = t[j++])  /* copy t */
                ;
}
```

Note the postfix ++ to increment i and j so they are always ready for the next character.

Now here is strcat with pointers.

```
strcat(s, t)    /* concatenate t to end of s */
char *s, *t;
{
        char *p;

        p = s;
        while (*p)      /* find end of s */
                p++;
        while (*p++ = *t++)     /* copy t */
                ;
}
```

p starts at the first character of s. Each time p is incremented (p++) it moves one character along the array, and eventually reaches the \0 that terminates s. From that point, characters of t are copied, including the \0 that terminates t. The construction

```
while (*p++ = *t++)
```

is identical to that explained with strcpy above.

The final routine is strcmp(s, t), which compares s to t, and returns negative, zero or positive according as s is less than, equal to, or greater than t. Both s and t are character strings.

```
strcmp(s, t)  /* return <0 if s<t, =0 if s==t, >0 if s>t */
char s[ ], t[ ];
{
        int i;

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

The pointer version of strcmp is not as close a parallel as our previous examples.

```
strcmp(s, t)  /* returns <0 if s<t, 0 if s==t, >0 if s>t */
char *s, *t;
{
        while (*s == *t++)
                if (*s++ == '\0')
                        return(0);
        return(*s - *(t-1));
}
```

By the time it is found that the strings differ at some point, t has been

incremented one position too far. To access the previous character, that is, the one before where t currently points, we need

```
*(t-1)
```

Accordingly, the **return** statement is

```
return(*s - *(t-1));
```

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

```
*++p
```

increments p *before* fetching the character that p points to;

```
*--p
```

decrements first. This means that we could have written the last **return** statement in **strcmp** as

```
return(*s - *--t);
```

Exercise 5-2: Write the function **strcat** with a call to **alloc** to allocate enough space for the created string. What value should **strcat** return, if any? □

5.6 Pointer Copying

You may notice in older C programs a rather cavalier attitude toward pointer copying. It has generally been true that in C a pointer may be assigned to an integer and back again without changing it. This has led to the taking of liberties with routines that return pointers which are then merely passed to other routines — the requisite pointer declarations are often left out. For example, in

```
if (strcmp(strcat(p, q), strcat(s, t)) == 0) ...
```

there would be a strong tendency not to bother declaring that **strcat** returns a character pointer. In fact, **strcat** itself would probably not be declared as returning a character pointer.

This kind of code is inherently risky, for it depends on details of implementation and machine architecture which may well not hold for the particular compiler you use. It's wiser to be complete in all declarations.

5.7 Two-Dimensional Arrays

Consider the problem of date conversion, from day of the month to day of the year and vice versa. These computations, which would presumably be done by two separate 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 easiest to separate them into two rows of a two-dimensional table, rather than try to keep track of what happens to February during computation. The table must be shared between two routines: **day_of_year**, which converts the month and day into the day of the year, and **month_day**, which converts the day of the year into the month and day. The table and the functions for performing the transformations are as follows:

```
static int day_tab[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(year, month, day)
int year, month, day;
{
        int i, leap;

        leap = (year%400 == 0) || (year%100 != 0 && year%4 == 0);
        for (i = 1; i < month; i++)
                day += day_tab[leap][i];
        return(day);
}

month_day(year, yearday, month, day)
int year, yearday, *month, *day;
{
        int i, leap;

        leap = (year%400 == 0) || (year%100 != 0 && year%4 == 0);
        for (i = 1; yearday > day_tab[leap][i]; i++)
                yearday -= day_tab[leap][i];
        *month = i;
        *day = yearday;
}
```

The table **day_tab** has to be external to both **day_of_year** and **month_day**. Since its definition precedes them both, no **extern** declarations are needed. And since **month_day** has to return a pair of values, it does so by assuming it has been called with two pointers.

The table **day_tab** is the first two-dimensional array we have dealt with. A two-dimensional array is really a one-dimensional array, each of whose

elements is an array. Hence subscripts are written as

```
day_tab[i][j]
```

rather than

```
day_tab[i, j]
```

as in most languages. Other than this, a two-dimensional array can be treated in much the same way as in other languages. Elements are stored by rows.

The array is initialized by a list of initializers in braces; each row is initialized by the corresponding list. We started the array with a column of zero so that month numbers can run from the intuitively natural 1 to 12 instead of 0 to 11. Since space is certainly not at a premium here, this is easier than mentally adjusting indices.

5.8 Pointer Arrays; Pointers to Pointers

Since pointers are variables themselves, you might reasonably expect that there would be a need for arrays of pointers. This is indeed the case. We have already seen a limited example, although we didn't stress it. A two-dimensional array like

```
int     day_tab[2][13]
```

is actually a one-dimensional array `day_tab[2]` with two elements, each of which is a one-dimensional array of 13 elements. And since we have been saying that pointers and arrays are really much the same thing, this means that `day_tab` can be considered an array of two pointers to integers, or a pointer to a pointer to integers.

Let us illustrate this further with a rather larger program than most we have written so far. The task is to write a program that will sort a set of text lines into alphabetic order, a stripped-down version of the Unix utility *sort*.

In Chapter 3 we presented a Shell sort function that would sort an array of integers. The same algorithm will work, except that now we have to deal with lines of text, which are of different sizes, and certainly can't be compared or moved in a single operation as can integers. What data representation will cope efficiently and conveniently with variable-length text lines?

This is where the array of pointers enters. If we store the lines to be sorted end to end in one long character array (maintained by `alloc`, perhaps) then we can refer to each line by a pointer to its first character. The pointers themselves can be stored in an array. Now when two out-of-order lines are 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 be part of moving the actual lines.

The sorting process involves 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 things.

Let us defer the sorting step for a moment, and concentrate on the data structure and the input and output. The main item of data structure needed is an array to hold the pointers to the lines as they are read in, and a line count for sorting. The input routine has to collect the lines, fill this pointer array, and count the input lines. Since the input function can only cope with a finite number of input lines, it can return some illegal line count like -1 to signal an input overflow. The output routine only has to print the lines in the order in which they appear in the array of pointers.

Here is the bulk of the code.

```
#define     NULL 0
#define     LINES 100   /* maximum lines to be sorted */
#define     MAXLINE     200   /* longest line to handled */

main( )     /* sort input lines */
{
        char *lineptr[LINES];       /* pointers to text lines */
        int nlines;                 /* number of input lines read */

        if ((nlines = readlines(lineptr)) >= 0) {
                sort(lineptr, nlines);
                writelines(lineptr, nlines);
        }
        else
                printf("input too big to sort\n");
}
```

```
readlines(lineptr)   /* read input lines for sorting */
char *lineptr[ ];
{
        int n, nlines;
        char *p, *alloc( ), line[MAXLINE];

        nlines = 0;
        while ((n = getline(line, MAXLINE)) >= 0)
                if (nlines >= LINES)
                        return(-1);
                else if ((p = alloc(n+1)) == NULL)
                        return(-1);
                else {
                        strcpy(p, line);
                        lineptr[nlines++] = p;
                }
        return(nlines);
}

writelines(lineptr, nlines)  /* write output lines */
char *lineptr[ ];
int nlines;
{
        int i;

        for (i = 0; i < nlines; i++)
                printf("%s\n", lineptr[i]);
}
```

The main new thing is the declaration for **lineptr**,

```
char *lineptr[LINES];       /* pointers to text lines */
```

which says that **lineptr** is an array of **LINES** elements, each a pointer to **char**. That is, **lineptr[i]** is a character pointer, and ***lineptr[i]** accesses a character.

main includes the call to **sort**, but we actually wrote the program originally without any **sort**, just **readlines** and **writelines**, to make sure that we could copy lines from input to output intact, and that the error-detecting parts worked properly. With that all checked out, we could then concentrate on the sorting. Writing a program in small steps — "incremental construction," if you like — is a good approach: when a program falls apart during development, it is almost certainly related to the most recent thing you added. Knowing that much makes it a lot easier to find the trouble.

Now we can proceed to sorting. The Shell sort from Chapter 3 needs only minor changes, mainly new declarations, and separation of the comparison operation into a separate function.

```
sort(v, n)      /* sort v[0] ... v[n-1] into increasing order */
char *v[ ];
int n;
{
        int gap, i, j;
        char *k;

        for (gap = n/2; gap > 0; gap /= 2)
                for (i = gap; i < n; i++)
                        for (j = i-gap; j >= 0; j -= gap) {
                                if (strcmp(v[j], v[j+gap]) <= 0)
                                        break;
                                k = v[j];
                                v[j] = v[j+gap];
                                v[j+gap] = k;
                        }
}
```

The declaration of k is

```
char *k
```

Since any individual element of v (alias lineptr) is a character pointer, k should be the same so one can be copied to the other.

We also wrote the program about as straightforwardly as possible, so as to get it working quickly. It is quite likely that the sort could be faster if, for instance, we copied the incoming lines directly into an array maintained by readlines, rather than copying them into line and then to a hidden place maintained by alloc. But it's wiser to make the first draft something easy to understand; worry about "efficiency" later. The way to make this program significantly faster is probably not by avoiding an unnecessary copy of the input lines. Replacing the Shell sort by something better, like Quicksort, is more likely to make a difference.

Initialization of an array of pointers is straighforward; again, all that is needed is a list of initializers enclosed in braces. One of the most common initializers is a set of character strings, as in this routine to print error messages:

```
char *msg[ ] ={
        "too bad",
        "tough luck",
        "it's all over",
        "so long"
};

print_msg(n)
int n;
{
        printf("error: %s\n", msg[n]);
}
```

The compiler supplies the proper count for the array by counting the initializers.

Exercise 5-3: Rewrite the routines **day_of_year** and **month_day** with pointers instead of indexing. □

5.9 Command Arguments

In most environments that support C, there is a way to specify arguments or parameters to be passed to the program when it begins executing. These arguments are available to the function **main** (if the program wishes to do anything with them) as an argument count **argc** and an array of character strings **argv** containing the arguments. Manipulating these character string arguments is one of the more common uses of multiple levels of pointers.

As the simplest illustration of the necessary declarations and use, here is a program that simply echoes its arguments. By convention, **argc** is greater than zero; the first argument (**argc**=1) in **argv[0]** is the command name itself.

```
main(argc, argv)    /* echo arguments; 1st version */
int argc;
char *argv[ ];
{
        int i;

        for (i = 1; i < argc; i++)
                printf("%s%s", argv[i], (i < argc-1) ? " " : "\n");
}
```

Since the zeroth argument is the command name, we start with the first argument, **argv[1]**. When printed, each argument is a character string in exactly the same way as the lines of text were in the sort program. Each argument except the last is followed by a blank.

Since **argv** is an array of pointers, there are several different ways to write this program. Let us show two others.

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

Each time the pointer **argv** is incremented, one more argument is dropped from the "bottom" of the list. At the same time **argc** is decremented; when it becomes zero, there are no arguments left to print.

Alternatively

```
main(argc, argv)    /* echo arguments; 3rd version */
int argc;
char *argv[ ];
{
        while (--argc > 0)
                printf((argc > 1) ? "%s " : "%s\n", *++argv);
}
```

This version shows that the format argument of **printf** can be an expression just like any of the others. This usage is not very frequent, but worth remembering.

As a larger example, let us make some enhancements to the pattern-finding program we wrote in Chapter 4. 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 redesign so the pattern to be searched for is specified by the first argument on the command line.

```
#define     MAXLINE     1000

main(argc, argv)    /* search for pattern from first argument */
int argc;
char *argv[ ];
{
        char line[MAXLINE];

        while (getline(line, MAXLINE) >= 0)
                if (index(line, argv[1]) >= 0)
                        printf("%s\n", line);
}
```

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

A common convention for C programs is that an argument beginning with a minus sign '−' introduces an optional flag or parameter. If we choose −a (for "all but") to signal the inversion, and −n ("number") line numbering, then the command

```
find −a −n the
```

with the input

```
now is the time
for all good men
to come to the aid
of their party.
```

should produce the output

```
2:for all good men
```

Ideally the implementation of the optional arguments should be such that the rest of the program is relatively insensitive to the details of the arguments which were actually present. In particular, we don't want the call to **index** to refer to **argv[2]** when there was a flag argument and to **argv[1]** when there wasn't.

To achieve this requires us to treat **argv** itself as a pointer which can be incremented.

```
#define     MAXLINE    1000

main(argc, argv)    /* search for pattern */
int argc;
char *argv[ ];
{
        char line[MAXLINE];
        int allbut = 0, number = 0, n = 0;

        while (argc > 1 && argv[1][0] == '-') {
                switch (argv[1][1]) {
                        case 'a':       allbut = 1; break;
                        case 'n':       number = 1; break;
                }
                argc--;
                argv++;
        }
        while (getline(line, MAXLINE) >= 0) {
                n++;
                if ((index(line, argv[1]) >= 0) != allbut) {
                        if (number)
                                printf("%d:", n);
                        printf("%s\n", line);
                }
        }
}
```

argv is incremented if an optional argument is found. Since **argv** is a pointer to the beginning of the array of argument strings, incrementing it by 1 makes it point at the original **argv[1]** instead of **argv[0]** as it used to. At the same time we decrement **argc** in case anything further on in the program depends on that. (It doesn't in this case.) We also mentioned earlier that a function could treat arguments as either arrays or pointers, or even both; this example illustrates that.

Exercise 5-4: Write the program **tail** which prints the last *n* lines of its input. By default, *n* is 10, let us say, but it can be changed by an optional argument, so that

```
tail -n
```

prints the last *n* lines, for all reasonable *n*. Write the program so it makes the best use of available storage. □

5.10 Pointers to Functions

We said earlier that the operator & can be applied to functions, which implies that a *pointer to a function* is a legitimate object. In C, a function itself is not a variable, but it is quite possible to define a pointer to a function, which can be manipulated, passed to functions, and so on. This gives essentially the same capabilities as would be possible if functions were variables, but is much simpler to deal with.

Let us illustrate the use of pointers to functions, by modifying the sorting procedure that we wrote earlier in this chapter. A sort consists generally of three steps — a *comparison* which determines the ordering of any pair of objects, an *exchange* which reverses their order, and a sorting algorithm which makes comparisons and exchanges until the objects are in order.

If we use different comparison and exchange functions, we can sort objects any way we please. The sorting algorithm is independent of the comparison and exchange operations, so it need not change.

Let us modify the previous sorting program so the comparison and exchange functions are passed to **sort** as pointers to functions. **sort** in turn will call the functions via the pointers. First, the main routine needs to declare the functions **strcmp** and **swap** (a new routine), and pointers to them must be passed to **sort**:

```
#define     NULL 0
#define     LINES 100   /* maximum lines to be sorted */
#define     MAXLINE     200   /* longest line to handled */

main( )     /* sort input lines */
{
        char *lineptr[LINES];       /* pointers to text lines */
        int nlines;             /* number of input lines read */
        int strcmp( ), swap( );     /* comparison and exchange functions */

        if ((nlines = readlines(lineptr)) >= 0) {
                sort(lineptr, nlines, &strcmp, &swap);
                writelines(lineptr, nlines);
        }
        else
                printf("input too big to sort\n");
}
```

&strcmp and **&swap** are pointers to the functions; in fact, since they are known to be functions, the & is unnecessary.

The second step is to modify **sort**:

```
sort(v, n, comp, exch)      /* sort v[0] ... v[n-1] into increasing order */
char *v[ ];
int n;
int (*comp)( ), (*exch)( );
{
        int gap, i, j;

        for (gap = n/2; gap > 0; gap /= 2)
                for (i = gap; i < n; i++)
                        for (j = i-gap; j >= 0; j -= gap) {
                                if ((*comp)(v[j], v[j+gap]) <= 0)
                                        break;
                                (*exch)(&v[j], &v[j+gap]);
                        }
}
```

The declarations should be studied with some care.

```
int (*comp)( )
```

says that the object in question is a pointer to a function that returns an int. Without the first set of parentheses,

```
int *comp( )
```

would say that comp was a function returning a pointer to an int, which is quite a different thing.

The use of the comparison function in the line

```
if ((*comp)(v[j], v[j+gap]) <= 0)
```

is the consistent with the declaration: *comp is a pointer to a function; it is followed by its argument list.

The final step is to add the function swap which exchanges two pointers. This is of course adapted directly from a routine we presented early in the chapter.

```
swap(px, py) /* swap *px and *py */
char *px[ ], *py[ ];
{
        char *k;

        k = *px;
        *px = *py;
        *py = k;
}
```

Notice that the pointers involve two levels of indirection.

CHAPTER 6: STRUCTURES

A *structure* is a collection of one or more objects, probably of different types, grouped together under a single name for convenient handling. (Structures are called "records" in some languages, most notably Pascal.) The elements of a structure are called *members*.

The classical example of a structure is the payroll record: an "employee" is described by a set of attributes which would normally be separate variables, such as name, address, social security number, salary, etc. Some of these variables in turn could be structures: a name has several components, as does an address and even a salary.

If well used, structures contribute markedly to orderly data layouts. Large programs in particular benefit from the organizing force that structures provide. In this chapter we will try to illustrate how structures are used. The programs we will use are bigger than many of the others in the book, but still of modest size.

6.1 Basics

To illustrate the basic syntactic rules, let us revisit the date conversion routines of Chapter 5. A date clearly consists of several pieces — a year, a day of the year, a month, and a day of the month. So these could all be placed into a single structure like this:

```
struct date {
        int     year;
        char    leap;
        int     yearday;
        int     month;
        char    mon_name[4];
        int     day;
};
```

We include **leap** and **mon_name** to show a structure containing several types of variables.

The keyword **struct** introduces a structure declaration, which is a list of variable declarations enclosed in braces. The format of the declaration is free, but conventionally declarations are written one variable per line. An optional name called the *structure tag* may follow the word **struct** (as with **date** here). The tag names this kind of structure, and can be used subsequently as a shorthand for the whole declaration. The names used within the structure declaration are called "members."

Although a structure member and an ordinary variable can have the same name without conflict, this can be confusing unless used carefully.

The right brace may be followed by a list of variables, just as for any basic type. That is,

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

is syntactically equivalent to

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

in the sense that each statement declares x, y and z to be objects of the named type.

A structure declaration that is not followed by a list of variables allocates no storage; it merely defines a *template* or shape of structure. If the declaration is tagged, however, the tag can be used later to declare actual instances of the structure. For example, given the declaration of **date** above,

```
struct date   birthdate;
```

makes a variable **birthdate** which is a structure of type **date**.

An external or static structure can be initialized by following its declaration by a list of initializers for the components:

```
struct date birthdate = {1776, 1, 186, 7, "Jul", 4};
```

describes the date of independence of one of the larger American republics.

Now we can compute. A member of a structure is referenced in an expression by a construction of the form

structure . member

To set **leap** in the structure **d**, we write

```
d.leap = (d.year%400 == 0) || (d.year%100 != 0 && d.year%4 == 0);
```

Structures can be nested; a payroll record might actually look like

```
struct person {
        char   name[50];
        char   address[50];
        int    zipcode;
        long   ss_number;
        float  salary;
        struct date   birthdate;
        struct date   hiredate;
};
```

The **person** structure contains two dates. To refer to the month of birth for someone,

```
struct person p;

        m = p.birthdate.month
```

And to access the first character of the **name** field in this structure, write

```
p.name[0]
```

6.2 Structures and Functions

There are a number of restrictions on C structures. The essential rules are that the only things that you can do a structure are to take its address with &, and access one of its members. This implies that structures may not be assigned to or copied as a unit, and that they can not be passed to or returned from functions. Pointers to structures do not suffer these limitations, however, so structures and functions do work together comfortably. Finally, automatic structures, like automatic arrays, cannot be initialized; only external or static structures can.

Let us investigate some of these points by rewriting the date conversion functions of the last chapter to use structures. Since the rules prohibit passing a structure to a function directly, we must either pass the components separately, or pass a pointer to the whole thing. Since structure members are just ordinary variables, the first alternative is in effect what we've already done:

```
d.yearday = day_of_year(d.year, d.month, d.day);
```

So we might better pass the structure pointer. If we have

```
struct date   hiredate;
```

we can then write

```
yearday = day_or_year(&hiredate);
```

to pass the pointer to **day_of_year**. The function itself has to be modified too, since its argument is now a structure pointer rather than a list of variables.

```
day_of_year(pd)     /* return day of year */
struct date *pd;
{
        int i;

        pd->leap = (pd->year%400 == 0)
         || (pd->year%100 != 0 && pd->year%4 == 0);
        for (i = 1; i < pd->month; i++)
                pd->day += day_tab[pd->leap][i];
        return(pd->day);
}
```

The declaration

```
struct date *pd;
```

says that **pd** is a pointer to that kind of structure. The notation exemplified by

```
pd->leap
```

is new. If **p** is a pointer to a structure, then

p -> *member of structure*

refers to the particular member. So

--

```
pd->leap
```

is the leap year indicator,

```
pd->year
```

is the year, and so on. Structure pointers are so common that the -> notation is provided as a shorthand for the equivalent

```
(*pd).year
```

As you might expect, there is considerable potential for error in coupling a pointer to one structure with a member of another.

For completeness, here is the other function, **month_day**, rewritten with structures. The only change necessary is to convert references to arguments into structure offsets.

```
month_day(pd)       /* set month and day from day of year */
struct date *pd;
{
        int i;

        pd->leap = (pd->year%400 == 0)
          || (pd->year%100 != 0 && pd->year%4 == 0);
        pd->day = pd->yearday;
        for (i = 1; pd->day > day_tab[pd->leap][i]; i++)
                pd->day -= day_tab[pd->leap][i];
        pd->month = i;
}
```

For complicated pointer expressions, it's wise to use parentheses to make it clear who goes with what. For example, given the declaration

```
struct {
        int x;
        int *y;
} *p;
```

then

```
p->x++          increments x
++p->x          so does this
(++p)->x          increments p before getting x
*p->y++         uses y as a pointer, then increments it
*(p->y)++         so does this
*(p++)->y         uses y as a pointer, then increments p
```

The way to remember these is that ->, . (dot), ( ) and [ ] bind very tightly. An expression involving one of these is treated as a unit: `p->x`, `a[i]`, `y.x` and `f(b)` are names exactly as `x` is.

6.3 Arrays of Structures

Suppose we want a program that counts occurrences of each C keyword. One possibility is to use two parallel arrays **keyword** and **keycount** to store a pointer to the name and the count. But the very fact that the arrays are parallel indicates that a different organization is possible. Each keyword entry is really a pair:

```
char  *keyword;
int   keycount;
```

and there is a whole set of pairs.

The structure declaration

```
struct key {
        char *keyword;
        char keycount;
} keytab[NKEYS];
```

defines an array **keytab** of structures of this type, and allocates storage to them. Since this structure actually contains a constant set of names, however, it is easy to initialize it once and for all when it is declared. The structure initialization is quite analogous to earlier ones — the declaration is followed by a list of initializers enclosed in braces:

```
struct key {
        char  *keyword;
        int   keycount;
} keytab[ ] ={
        "break",        0,
        "case",         0,
        "char",0,
        "continue",     0,
        "default",      0,
        "do",   0,
        /* ... */
        "while",        0,
        NULL, 0
};
```

The declarations are listed in pairs corresponding to the structure members. It is more precise to enclose initializers for each "row" or structure in braces, as in

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

but the braces may be omitted when the initializers are simple and there are no omissions. We terminated the list with a null pointer to make it easy for programs to find the end of the array. As usual, the compiler computes the size of the array **keytab** itself when initializers are present and the [ ] is left empty.

The counting program can now be finished. To illustrate communication with external structure definitions, we have chosen to modify the binary search program so that the array it searches is an external structure, rather than an argument. This

degeneralizes it, of course, but in this case it is actually the easiest way to do the job. We must also alter the binary search routine to compare strings of characters instead of integers.

```
struct key {
        char   *keyword;
        int    keycount;
} keytab[ ] ={
        "break",        0,
        "case",         0,
        /* ... */
        "while",        0,
        NULL, 0
};

#define     MAXWORD  20

main( )     /* count occurrences of C keywords */
{
        int n, t;
        char word[MAXWORD];

        while ((t = getword(word, MAXWORD)) != EOF)
                if (t == LETTER)
                        if ((n = binary(word)) >= 0)
                                keytab[n].keycount++;
        for (n = 0; n < NKEYS; n++)
                if (keytab[n].keycount > 0)
                        printf("%4d %s\n",
                          keytab[n].keycount, keytab[n].keyword);
}
```

--

```
binary(word) /* find word in keytab[0] ... keytab[NKEYS-1] */
char *word;
{
        int low = 0, high = NKEYS-1, mid, cond;

        while (low <= high) {
                mid = (low+high) / 2;
                if ((cond = strcmp(word, keytab[mid].keyword)) == 0)
                        return(mid);
                else if (cond < 0)
                        high = mid - 1;
                else
                        low = mid + 1;
        }
        return(-1);
}
```

We will describe the function **getword** in a moment; for now it suffices to say that it returns **LETTER** each time it finds a word, and copies the word into 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 to do it by machine. One possibility would be to write a loop which counts along **keytab** until it finds the null pointer, then assign it to a global variable.

But this is more than is needed, since the size of the array is completely determined at compile time. C provides a compile-time unary operator called **sizeof** which can be used to compute the size of any object. The expression

sizeof (*object*)

yields an integer equal to the size of the specified object. (The size is given in unspecified units called "bytes," which are generally the same size as a **char**.) The object can be an actual variable or array or structure, or it can be a type like **int** or **double**. In our case, the number of keywords is

sizeof (keytab) / sizeof (individual entry)

Thus by including in the program the definition

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

NKEYS is defined properly.

Now the function **getword**. We have actually written a more general **getword** than is necessary for this program, but it is not really more complicated. **getword** returns the next "token" from the input, where a token is either an identifier in the C sense (that is, a maximal string of letters and digits beginning with a letter), or a single character. The type of the object is returned as a function value; it is **LETTER** if the token is a word, **DIGIT** for a digit, or the character itself if it is non-alphanumeric.

```
getword(w, lim)      /* get next word from input */
char *w;
int lim;
{
        int c;

        if (type(c = *w++ = getc( )) != LETTER) {
                *w = '\0';
                return(c);
        }
        while (--lim > 0)
                if ((c = type(*w++ = getc( ))) != LETTER && c != DIGIT)
                        break;
        ungetc(c);
        *(w-1) = '\0';
        return(LETTER);
}
```

Notice that **getword** uses the routines **getc** and **ungetc** which we wrote in Chapter 4: when the collection of an alphabetic token stops, **getword** has gone one character too far. The call to **ungetc** pushes that character back on the input for the next call.

getword calls **type** to determine the type of each individual character of input.

```
type(c)      /* return type of character */
int c;
{
        if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
                return(LETTER);
        else if (c >= '0' && c <= '9')
                return(DIGIT);
        else
                return(c);
}
```

The symbolic constants **LETTER** and **DIGIT** can have any values that do not conflict with non-alphanumeric characters and **EOF**; The obvious choice is

```
#define     LETTER     'a'
#define     DIGIT '0'
```

Exercise 6-1: **getword** can be faster if calls to the function **type** are replced by references to an appropriate array **type[ ]**. Make this modification, and measure the change in speed of the program. □

6.4 Pointers to Structures

To illustrate some of the considerations involved with pointers and arrays of structures, let us write the keyword-counting program over again using pointers instead of array indices.

The external declaration of **key** need not change, but **main** and **binary** do need modification.

```
main( )       /* count occurrences of C keywords; pointer version */
{
        int n, t;
        char word[MAXWORD];
        struct key *binary( ), *p;

        while ((t = getword(word, MAXWORD)) != EOF)
                if (t == LETTER)
                        if ((p = binary(word)) != NULL)
                                p->keycount++;
        for (p = keytab; p < keytab + NKEYS; p++)
                if (p->keycount > 0)
                        printf("%4d %s\n", p->keycount, p->keyword);
}

struct key *binary(word)   /* find word in keytab[0] ... keytab[NKEYS-1] */
char *word;
{
        int cond;
        struct key *low = keytab, *high = keytab+NKEYS-1, *mid;

        while (low <= high) {
                mid = low + (high-low) / 2;
                if ((cond = strcmp(word, mid->keyword)) == 0)
                        return(mid);
                else if (cond < 0)
                        high = mid - 1;
                else
                        low = mid + 1;
        }
        return(NULL);
}
```

There are several things worthy of note here. First, the declaration of **binary** must indicate that it returns a pointer to the structure type **key**, instead of an integer; this is declared both in **main** and in **binary**.

Second, all the accessing of elements of **keytab** is done by pointers. This causes one significant change in **binary**: the computation of the middle element can no longer be simply

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

because the *addition* of two pointers will not in general produce any kind of a useful answer (even when divided by 2). This must be rearranged into

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

which increments low by the right number of elements.

You should also study the initializers for **low** and **high**. It is possible to initialize a structure pointer to the address of a previously defined object; that is precisely what we have done here.

Finally, in **main** we wrote

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

We could equally well have used the fact that **keytab** is terminated by a null pointer, and written

```
for (p = keytab; p->keyword != NULL; p++)
```

If **p** is a pointer to a structure, any arithmetic on **p** takes into account the actual size of the structure. For instance, **p++** increments **p** by the correct amount to get the next element of the array of structures. But don't assume that the size of a structure is the sum of the sizes of its members — because of alignments of different sized objects, there may be "holes" in a structure.

6.5 Nested 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 sort it and use a binary search. Yet we can't do a linear search for each word as it arrives; the program would take forever. How can we organize the data to cope efficiently with a list of arbitrary words?

One solution is to keep the set of words sorted at all times, by placing each word into its proper position in the order as it arrives. This can'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 actual word
the count
a pointer to the left child
a pointer to the right child

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

The nodes are maintained so that at any node the left subtree contains only words which are less than the word at the node, and the right subtree contains only words that are greater. The tree is inherently recursive, of course, so recursive routines for insertion and printing will be most natural.

Going back to the description of a node, it is clearly a structure with four components:

```
struct node {        /* the basic node */
        char   *word;/* points to the text */
        int    count;
        struct node  *left;  /* left child */
        struct node  *right; /* right child */
};
```

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

```
struct node  *left;
```

is a *pointer* to a node, not a node itself.

The code for the whole program is surprisingly small, given a handful of supporting routines that we have already written. These are **getword**, to fetch each input word, and **alloc**, to provide space for squirreling the words away.

```
#define     MAXWORD  20

main( )     /* word frequency count */
{
        struct node *top, *tree( );
        char word[MAXWORD];
        int t;

        top = NULL;
        while ((t = getword(word, MAXWORD)) != EOF)
                if (t == LETTER)
                        top = tree(top, word);
        treeprint(top);
}
```

```
struct node *tree(p, w)     /* install w at or below p */
struct node *p;
char *w;
{
        struct node *talloc( );
        char *alloc( );
        int cond;

        if (p == NULL) {    /* a new word */
                p = talloc( );/* make a new node */
                p->word = alloc(strlen(w)+1);
                strcpy(p->word, w);
                p->count = 1;
                p->left = p->right = NULL;
        } else if ((cond = strcmp(w, p->word)) == 0)      /* repeated word */
                p->count++;
        else if (cond < 0)  /* lower goes into left */
                p->left = tree(p->left, w);
        else            /* greater into right */
                p->right = tree(p->right, w);
        return(p);
}

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

The main routine simply reads words with **getword** and installs them in the tree with **tree**.

tree itself is straightforward. 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. Eventually the word either matches something already in the tree (in which case the count is incremented), or it has to be added at the edge. If a null pointer is encountered, a node must be created and added to the tree.

Storage for the new node is fetched by a routine **talloc**, which is an obvious adaptation of the **alloc** we wrote earlier. The new word is copied to a hidden place provided by **alloc**, the count initialized, and the two children made null. This part of the code is executed only at the edge of the tree, when a new node is being added. We have (unwisely for a production program) omitted error checking on the values returned by **alloc** and **talloc**.

If the word has been seen before, it will be found in the interior of the tree. In that case its count is incremented. Otherwise, **tree** is called recursively to place

the word in either the left or right subtree as appropriate.

treeprint prints the tree in left subtree 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 recursion, draw yourself a tree and print it with **treeprint**; it's one of the cleanest recursive routines you can find.

6.6 Table Lookup

In this section we will write the innards of a table-lookup package as an illustration of more aspects of structures. This code is typical of what might be found at the heart of a macro processor or compiler for symbol table management.

There are two major routines and a table. **install(s, t)** installs the string **s** into the table with the string value **t**. **lookup(s)** searches for **s** in the table; if found, it returns a pointer to the place in the table where it was found.

The algorithm used is a hash table search — the incoming name is converted into a small integer, which is then used to index into a table. each table entry is the beginning of a list of values that have that hash value.

Here is the code.

```
#define     HASHSIZE   199   /* must be prime */

struct nlist { /* basic table entry */
        char  *name;
        char  *def;
        struct nlist *next;   /* next entry in the chain */
};

struct nlist   *hshtab[HASHSIZE];       /* the table of pointers */

struct nlist *lookup(str)    /* look for str in hshtab */
char *str;
{
        register char *s1;
        register struct nlist *np;
        static struct nlist nodef;   /* return this if not found */

        s1 = str;
        for (hshval = 0; *s1; )
                hshval += *s1++;
        hshval %= HASHSIZE;
        for (np = hshtab[hshval]; np != NULL; np = np->next)
                if (strcmp(str, np->name) == 0)
                        return(np);
        return(&nodef);
}
```

lookup performs the hashing operation on the string, in this case adding up the character values and forming the remainder modulo the table size. This produces a starting index in the table **hshtab**; if the argument is to be found, it will be in the chain of entries beginning there. If **lookup** finds the netry already present, it

returns a pointer to it; if not, it returns a pointer to a structure known not to be part of the list.

install uses **lookup** to determine whether the name being installed is already present; if so, the new definition must supersede the old one. Otherwise, a completely new entry is created.

```
char *install(nam, val)      /* install (nam, val) in hshtab */
char *nam, *val;
{
        register struct nlist *np, *lookup( );
        char *strsave( );

        if ((np = lookup(nam))->name == NULL) { /* not found */
                if ((np = alloc(sizeof(*np))) == NULL)
                        return(np);
                np->name = strsave(nam);
                np->def = strsave(val);
                np->next = hshtab[hshval];
                hshtab[hshval] = np;
        } else {                /* already there */
                free(np->def);
                np->def = strsave(val);
        }
        return(np->def);
}
```

strsave merely copies the string given by its argument into a safe place, obtained by a call on **alloc**.

```
char *strsave(s)     /* save string s somewhere */
char *s;
{
        char *p, *alloc( );

        if ((p = alloc(strlen(s)+1)) != NULL)
                strcpy(p, s);
        return(p);
}
```

With this must code in hand, it's a very small step to write, for example, a small macro processor, capable of handling the **#define** statement so long as there are no arguments.

Exercise 6-2: Write a routine which will remove a name and definition from the table maintained by **lookup** and **install**. □

CHAPTER 7: INPUT AND OUTPUT

Input and output facilities are not part of the C language, so we have de-emphasized them in our presentation thus far. Nonetheless, real programs do interact with their environment in much more complicated ways than those we have shown before. In this chapter we will describe "the standard I/O library," a set of functions designed to provide a standard I/O system for C programs. The functions are intended to present a convenient programming interface, yet reflect only operations that can be provided on most modern operating systems. The routines are efficient enough that users will not feel the need to circumvent them "for efficiency" regardless of how critical the application. Finally, the routines are meant to be "portable," in the sense that they will exist in compatible form on any system where C exists, and that programs which confine their system interactions to facilities provided by the standard library can be moved from one system to another essentially without change.

We will not try to describe the entire I/O library here; we are more interested in showing the essentials of writing C programs that interact with their operating system environment.

7.1 Access to the Standard Library

Each source file that refers to a standard library function must contain the line

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

near the beginning. The file **stdio.h** defines certain macros and variables used by the I/O library. (The use of < and > instead of the double quotes implies a search for the file in a special place.) Furthermore, it may be necessary when loading the program to specify the library explicitly; for example, on Unix, the command to compile a program would be

```
cc [source files, etc.] -lS
```

where **-lS** indicates loading from the standard library.

7.2 "Standard Input" and "Standard Output" — getchar and putchar

The simplest input mechanism is to read a character at a time from the "standard input," which is generally the user's terminal, with **getchar**. **getchar( )** returns the next input character each time it is called. In most environments that support C, a file may be substituted for the terminal by using the '<' convention: if **prog** uses **getchar**, then the command line

```
prog <infile
```

causes **prog** to read **infile** instead of the terminal. **prog** itself knows nothing about where its input is coming from. This is also true if the input comes from another program via the pipe mechanism:

```
otherprog | prog
```

will provide the input for **prog** from the output of **otherprog**.

getchar returns the value **EOF** when it encounters end of file (or an error) on whatever input is being read. The standard library defines the symbolic constant **EOF** to be -1, but tests should be written in terms of **EOF**, not -1. so as to be independent of the specific value.

For output, **putchar(c)** puts the character **c** on the "standard output," which is also by default the terminal. The output can be captured on a file by using '>': if **prog** uses **putchar**,

```
prog >outfile
```

will write the output onto **outfile** instead of the terminal. On Unix or GCOS, a pipe can also be used:

```
prog | otherprog
```

puts the output of **prog** into the input of **otherprog**.

The function **printf**, which formats output in various ways, uses the same mechanism as **putchar** so output produced by **printf** also finds its way to the standard output, and calls to **putchar** and **printf** may be interleaved.

A surprising number of programs read only one input and write one output; for such programs I/O with **getchar**, **putchar**, and **printf** may be entirely adequate, and it is almost always enough to get started. This is particularly true given a pipe facility for connecting the output of one program to the input of the next. For example, here is a complete program that acts as a "filter" to strip out all ASCII control characters except newline and tab from its input. The program relies on the numerical properties of the ASCII character set — control characters are less than blank, or greater than or equal to octal 177.

```
main( )        /* strip out control characters */
{
        int c;

        while ((c = getchar( )) != EOF)
                if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
                        putchar(c);
}
```

If it is necessary to treat multiple files, you can use a program like the utility *cat* to collect the files for you:

```
cat file1 file2 ... | ccstrip >output
```

and thus avoid learning how to access files from a program. (cat is presented later in this chapter.)

As an aside, in the standard I/O library the "functions" **getchar** and **putchar** are actually macros, and thus avoid the overhead of a function call per character.

7.3 Formatted Output — printf

The two routines **printf** for output and **scanf** for input permit translation to and from character representations of numerical quantities. They also allow generation or interpretation of formatted lines. We have used **printf** informally throughout the previous chapters; here is a complete description.

```
printf(control, arg1, arg2, ...)
```

printf converts, formats, and prints its arguments on the standard output under control of the control string. The control string contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to **printf**.

Each conversion specification is introduced by the character %. Following the % there may be:

- an optional minus sign '–' which specifies left adjustment of the converted argument in it field;
- an optional digit string specifying a minimum field width; if the converted argument has fewer characters than the field width it will be padded on the left (or right, if the left adjustment indicator has been given) to make up the field width; the padding character is blank normally and zero if the field width was specified with a leading zero (note that this does not imply an octal field width);
- an optional period '.' which serves to separate the field width from the next digit string;
- an optional digit string (the precision) which specifies the maximum number of characters to be printed from a string, or the number of digits to be printed to the right of the decimal point of a floating or double number.

- an optional length modifier **l** (letter ell) which indicates that the corresponding data item is a **long** rather than an **int**.
- a character which indicates the type of conversion to be applied.

The conversion characters and their meanings are:

d The argument is converted to decimal notation.

o The argument is converted to octal notation (including a leading zero).

x The argument is converted to hexadecimal notation.

u The argument is converted to unsigned decimal notation.

c The argument is taken to be a single character.

s The argument is taken to be a string and characters from the string are printed until a null character is reached or until the number of characters indicated by the precision specification is exhausted.

e The argument is taken to be a **float** or **double** and converted to decimal notation of the form [−]m.nnnnnnE[−]xx where the length of the string of *n*'s is specified by the precision. The default precision is 6 and the maximum is 22.

f The argument is taken to be a **float** or **double** and converted to decimal notation of the form [−]mmm.nnnnn where the length of the string of *n*'s is specified by the precision. The default precision is 6 and the maximum is 22. Note that the precision does not determine the number of significant digits printed in f format.

g Use **%e** or **%f**, whichever is shorter; print no unnecessary zeros.

If no recognizable conversion character appears after the % that character is printed; thus % may be printed by %%.

7.4 Formatted Input - scanf

The function **scanf** is the input analog of **printf**, and provides many of the same conversion facilities.

```
scanf(control, arg1, arg2, ...)
```

scanf reads characters from the standard input, interprets them according to a format, and stores the results in its arguments. It expects a control argument, described below, and a set of arguments, *each of which must be a pointer,* indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

(1) Blanks, tabs or newlines, which are ignored.

(2) Ordinary characters (not %) which are expected to match the next non-space character of the input stream (space characters are blank, tab or newline).

(3) Conversion specifications, consisting of the character %, an optional assignment suppression character *, an optional numerical maximum field width, and a conversion character.

A conversion specification is used to direct the conversion of the next input field; the result is placed in the variable pointed to by the corresponding

argument, unless assignment suppression was indicated by the * character. An input field is defined as a string of non-space characters; it extends either to the next space character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. Pointers, rather than variable names, are required by the "call by value" semantics of the C language. The following conversion characters are legal:

% indicates that a single % character is expected in the input stream at this point; no assignment is done.

d indicates that a decimal integer is expected in the input stream; the corresponding argument should be an integer pointer.

o indicates that an octal integer is expected in the input stream; the corresponding argument should be a integer pointer.

x indicates that a hexadecimal integer is expected in the input stream; the corresponding argument should be an integer pointer.

s indicates that a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \0 which will be added. The input field is terminated by a space character or a newline.

c indicates that a single character is expected; the corresponding argument should be a character pointer; the next input character is placed at the indicated spot. The normal skip over space characters is suppressed in this case; to read the next non-space character, try %1s.

e or f indicates that a floating point number is expected in the input stream; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a float. The input format for float's is an optional sign, a string of numbers possibly containing a decimal point, followed by an optional exponent field containing an E or e followed by a possibly signed integer.

[indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not circumflex (^), the input field is all characters until the first character not in the set between the brackets; if the first character after the left bracket is ^, the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters d, o and x may be preceded by l (letter ell) to indicate that a pointer to long rather than int is expected. Similarly, the conversion characters e or f may be preceded by l to indicate that a pointer to double rather than float is in the argument list. The character h will function similarly in the future to indicate short data items.

For example, the call

```
int i;
float x;
char name[50];
scanf("%d %f %s", &i, &x, name);
```

with the input line

```
25  54.32E-1  thompson
```

will assign to i the value 25, x the value 5.432, and name will contain "thompson\0". Or,

```
int i;
float x;
char name[50];
scanf("%2d %f %*d %[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to i, 789.0 to x, skip "0123", and place the string "56\0" in name. The next call to any input routine will return a.

scanf returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, −1 is returned; note that this is different from 0, which means that the next input character does not match what you called for in the control string.

7.5 In-memory Format Conversion

The functions scanf and printf have siblings called sscanf and sprintf which perform identical conversions, but operate on a string instead of a file. The general format is

```
sprintf(string, control, arg1, arg2, ...)
sscanf(string, control, arg1, arg2, ...)
```

sprintf formats the arguments in arg1, arg2, etc., according to control as before, but places the result in string instead of on the standard output. string of course had better be big enough to receive the result.

sscanf does the reverse conversions — it scans the string according to the format in control, and places the resulting values in arg1, arg2, etc. These arguments must be pointers.

7.6 File Access

The programs we have written so far have all read the standard input and written the standard output, which we have assumed are magically predefined for a program by the local operating system.

The next step in I/O is to write a program which accesses a file which is *not* already connected to the program. One simple example which clearly illustrates the need for such operations is **cat**, which concatenates a set of named files onto the standard output. **cat** is used as a general-purpose input collector for programs which do not have the capability of accessing files by name, and merely for listing things. For example, the command

```
cat foo gorp
```

will concatenate the contents of the files **foo** and **gorp** onto the standard output.

The question is how to arrange for the named files to be read — that is, how to connect the external names that a user thinks of to the statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function **fopen**. **fopen** takes the external name (like **foo**), does some housekeeping and negotiation with the operating system (details of which needn't concern us), and returns an internal name which must be used in subsequent reads or write of the file.

This internal name is actually a pointer to a structure which contains information about the file. We will call the pointer the *file pointer*. Users don't need to know the details of the structure that the file pointer refers to, for part of the standard I/O definitions obtained by **#include <stdio.h>** is a definition called **FILE**. The only declaration needed for a file pointer is exemplified by

```
FILE  *fopen( ), *fp;
```

This says that **fp** is a pointer to a **FILE**, and **fopen** returns a pointer to a **FILE**.

The actual call to **fopen** is

```
fp = fopen("foo", "r");
```

The first argument of **fopen** is the name of the file as a character string. The second argument is the *mode*, which indicates whether one intends to read (**"r"**), write (**"w"**), or append (**"a"**) to the file.

If you open a file which does not exist for writing or appending, it is created (if possible). Opening a file that does not exist for reading is an error, and of course there may be other causes of error (like trying to read a file when you don't have permission). Any error causes **fopen** to return the null pointer value **NULL**.

The next thing needed is a way to read or write the file once it is open. There are several possibilities. The function **getc** returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in c the next character from the file referred to by fp. Like getchar, getc is a macro, not a function.

The macro putc is the inverse of getc:

```
putc(c, fp)
```

puts the character c on the file fp and returns c.

For formatted input or output on files, the functions fprintf and fscanf may be used. These are identical to printf and scanf, save that the first argument is a file pointer that specifies the file to be read or written.

With all of these preliminaries out of the way, we can actually write the program to concatenate files. The basic design that we have selected for cat is one that has been found convenient for many programs: if there are arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv)    /* cat: concatenate files */
int argc;
char *argv[ ];
{
        FILE *fp, *fopen( );

        if (argc == 1)      /* no arguments; copy standard input */
                filecopy(stdin);
        else
                while (--argc > 0)
                        if ((fp = fopen(*++argv, "r")) == NULL) {
                                printf("cat: can't open %s\n", *argv);
                                break;
                        } else {
                                filecopy(fp);
                                fclose(fp);
                        }
}
```

```
filecopy(fp)  /* copy fp to standard output */
FILE *fp;
{
        int c;

        while ((c = getc(fp)) != EOF)
                putc(c, stdout);
}
```

The file pointers **stdin** and **stdout** are pre-defined in the I/O library as the standard input and standard output; they may be used anywhere an object of type **FILE** can be. They are *not* variables, however, so don't try to assign to them.

getchar and **putchar** are defined in terms of **getc** and **putc** and **stdin** and **stdout** as follows:

```
#define     getchar     getc(stdin)
#define     putchar(c)  putc(c, stdout)
```

If you call any of these four "functions" with the wrong number of arguments, you will get some rather mysterious syntax error messages from the compiler.

The function **fclose** is the inverse of **fopen**; it breaks the connection between the file pointer and the external name that was established by **fopen**, freeing the file pointer for another file. Since most operating systems have some limit on the number of simultaneously open files that a program may have, it's a good idea to free things when they are no longer needed, as we did here in **cat**.

7.7 Error Handling — stderr and exit

The treatment of errors in **cat** is adequate, but not ideal. The trouble is that if one of the files can't be accessed for some reason, the diagnostic is at the end of the concatenated output. That is acceptable if that output is going to a terminal, but bad if it's going into another file.

To handle this situation in a better way, a second output file, called **stderr**, is assigned to a program in the same way that **stdin** and **stdout** are. If at all possible, output written on **stderr** appears on the user's terminal even when output for **stdout** is sent to a file.

Let us revise **cat** to handle errors better.

```
#include <stdio.h>

main(argc, argv)    /* concatenate files */
int argc;
char *argv[ ];
{
        FILE *fp, *fopen( );

        if (argc == 1)      /* no arguments; copy standard input */
                filecopy(stdin);
        else
                while (--argc > 0)
                        if ((fp = fopen(*++argv, "r")) == NULL) {
                                fprintf(stderr, "cat: can't open %s\n", *argv);
                                exit(1);
                        } else {
                                filecopy(fp);
                                fclose(fp);
                        }
        exit(0);
}
```

The program signals errors two ways. The diagnostic output goes onto **stderr**, so it finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program also uses the standard library function **exit**, which terminates program execution when it is called. The argument of **exit** is available to whatever process called this one, so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well, and various non-zero values signal abnormal situations.

7.8 The #include Processor

The next major example is a program which processes lines which contain **#include** statements. It is very easy for this program to handles nested **#include**'s by recursion, with a fresh level of recursion for each level of nesting.

Our version uses two new routines from the standard library. **fgets** is reminiscent of the **getline** function that we have used throughout the book. The call

```
fgets(line, MAXLINE, fp)
```

reads the next input line from file **fp** into the character array **line**; at most **MAXLINE** characters will be read. The resulting line is terminated with \0. Normally **fgets** returns **line**; on end of file it returns **NULL**.

fputs writes a line to a file:

```
fputs(line, fp)
```

and returns line.

We have also used sscanf to break out the first two strings of each input line rather than write a separate routine.

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

main( )     /* process #include */
{
        include(stdin);
        exit(0);
}

include(fp)   /* include file fp */
FILE *fp;
{
        char s1[MAXLINE], s2[MAXLINE], line[MAXLINE];
        FILE *f, *fopen( );
        int n;

        while (fgets(line, MAXLINE, fp) != NULL) {
                n = sscanf(line, "%s %s", s1, s2);
                if (n != 2 || strcmp(s1, "#include") != 0)
                        fputs(line, stdout);
                else if ((f = fopen(s2, "r")) == NULL) {
                        fprintf(stderr, "include: can't open %s\n", s2);
                        exit(1);
                } else {
                        include(f);
                        fclose(f);
                }
        }
}
```

By the way, there is nothing magic about the function fgets; it is just a C function. Here it is, copied directly from the I/O library:

```
#include    <stdio.h>

fgets(s, n, iop)
char *s;
register FILE *iop;
{
        register c;
        register char *cs;

        cs = s;
        while (--n>0 && (c = getc(iop))>=0) {
                *cs++ = c;
                if (c=='\n')
                        break;
        }
        if (c<0 && cs==s)
                return(NULL);
        *cs++ = '\0';
        return(s);
}
```

Exercise 7-1: Write a program to compare two files, printing the first line and character position where they differ. □

Exercise 7-2: Modify the pattern finding program of Chapter 5 to take its input from a set of named files or, if no files are named as arguments, from the standard input. Should the file name be printed when a matching line is found? □

Exercise 7-3: Write a program to print a set of files, starting each new one on a new page, with a title and a running page count for each file. □

C Reference Manual

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

May 1, 1977

1. Introduction

C is a computer language which offers a rich selection of operators and data types and the ability to impose useful structure on both control flow and data. All the basic operations and data objects are close to those actually implemented by most real computers, so that a very efficient implementation is possible, but the design is not tied to any particular machine and with a little care it is possible to write easily portable programs.

This manual describes the current version of the C language as it exists on the PDP-11, the Honeywell 6000, the IBM System/370, and the Interdata 8/32. Where differences exist, it concentrates on the PDP-11, but tries to point out implementation-dependent details. With few exceptions, these dependencies follow directly from the underlying properties of the hardware; the various compilers are generally quite compatible.

2. Lexical conventions

Blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. Some space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1 Comments

The characters /* introduce a comment, which terminates with the characters */. Comments do not nest.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore '_' counts as alphabetic. Upper and lower case letters are considered different. On the PDP-11, no more than the first eight characters are significant, and only the first seven for external identifiers.

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

int	extern	else
char	register	for
float	typedef	do
double	static	while
struct	goto	switch
union	return	case
long	sizeof	default
short	break	entry
unsigned	continue	
auto	if	

The **entry** keyword is not currently implemented by any compiler but is reserved for future use. Some implementations also reserve the word **fortran**.

2.4 Constants

There are several kinds of constants, as follows:

2.4.1 Integer constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with **0** (digit zero), decimal otherwise. The digits **8** and **9** have octal value 10 and 11 respectively. A sequence of digits preceded by **0x** or **0X** (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include **a** or **A** through **f** or **F** with values 10 through 15. A decimal constant whose value exceeds the largest signed machine integer (32767 on the PDP-11) is taken to be **long**; an octal or hex constant which exceeds the largest unsigned machine integer (0177777 or 0xFFFF on the PDP-11) is likewise taken to be **long**.

2.4.2 Explicit long constants

A decimal, octal, or hexadecimal integer constant immediately followed by **l** (letter ell) or **L** is a long constant, which, on the PDP-11, has 32 significant bits. As discussed below, on other machines integer and long values may be considered identical.

2.4.3 Character constants

A character constant is a sequence of characters enclosed in single quotes ' '. Within a character constant a single quote must be preceded by a backslash '\'. Certain non-graphic characters, and '\' itself, may be escaped according to the following table:

BS	\b
NL (LF)	\n
CR	\r
HT	\t
FF	\f
ddd	*ddd*
\	\\

The escape '*ddd*' consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is '\0' (not followed by a digit) which indicates the character NUL. If the character following a backslash is not one of those specified, the backslash vanishes.

The value of a single-character constant is the numerical value of the character in the machine's character set (ASCII for the PDP-11). On the PDP-11 at most two characters are permitted in a character constant and the second character of a pair is stored in the high-order byte of the integer value. Character constants with more than one character are inherently machine-dependent and should be avoided.

2.4.4 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an **e** or **E**, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the **e** and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.

2.5 Strings

A string is a sequence of characters surrounded by double quotes '"'. A string has type 'array of characters' and storage class 'static' (see below) and is initialized with the given characters. The compiler places a null byte '\0' at the end of each string so that programs which scan the string can find its end. In a string, the character '"' must be preceded by a '\'; in addition, the same escapes as described for character constants may be used. Finally, a '\' and an immediately following new-line are ignored.

All strings, even when written identically, are distinct.

3. Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in **sans-serif** type. Alternatives are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript 'opt,' so that

$$\{\ expression_{opt}\ \}$$

would indicate an optional expression in braces. The complete syntax is given in §16, in the notation of YACC.

4. What's in a Name?

C bases the interpretation of an identifier upon two attributes of the identifier: its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

There are four declarable storage classes: automatic, static, external, and register. Automatic variables are local to each invocation of a block, and are discarded upon exit from the block; static variables are local to a block, but retain their values upon reentry to a block even after control has left the block; external variables exist and retain their values throughout the execution of the entire program, and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables they are local to each block and disappear on exit from the block.

C supports several fundamental types of objects:

Objects declared as characters **(char)** are large enough to store any member of the implementation's character set, and if a genuine character is stored in a character variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine-dependent. On the PDP-11, characters are stored as signed 8-bit integers, and the character set is ASCII.

Up to three sizes of integer, declared **short int**, **int**, and **long int** are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers, or long integers, or both equivalent to plain integers. 'Plain' integers have the natural size suggested by the host machine architecture; the other sizes are provided to meet special needs. On the PDP-11, short and plain integers are both represented in 16-bit 2's complement notation. Long integers are 32-bit 2's complement.

Unsigned integers, declared **unsigned**, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. (16 on the PDP-11; long and short unsigned quantities

are not supported.)

Single precision floating point (**float**) quantities, on the PDP-11, have magnitude in the range approximately $10^{\pm 38}$ or 0; their precision is 24 bits or about seven decimal digits.

Double-precision floating-point (**double**) quantities on the PDP-11 have the same range as **float**s and a precision of 56 bits or about 17 decimal digits. Some implementations may make **float** and **double** synonymous.

Because objects of these types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. Types **char** and **int** of all sizes will collectively be called *integral* types. **Float** and **double** will collectively be called *floating* types.

Besides the fundamental arithmetic types there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

arrays of objects of most types;

functions which return objects of a given type;

pointers to objects of a given type;

structures containing a sequence of objects of various types;

unions capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

5. Objects and lvalues

An *object* is a manipulatable region of storage; an *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if E is an expression of pointer type, then *E is an lvalue expression referring to the object to which E points. The name 'lvalue' comes from the assignment expression '*E*1 = *E*2' in which the left operand E1 must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

6. Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions. §6.6 summarizes the conversions demanded by most ordinary operators; it will be supplemented as required by the discussion of each operator.

6.1 Characters and integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a short integer always involves sign extension; short integers are signed quantities. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. On the PDP-11, character variables range in value from −128 to 127; a character constant specified using an octal escape also suffers sign extension and may appear negative, for example ''\214''.

When a longer integer is converted to a shorter or to a **char**, it is truncated on the left.

6.2 Float and double

All floating arithmetic in C is carried out in double-precision; whenever a **float** appears in an expression it is lengthened to **double** by zero-padding its fraction. When a **double** must be converted to **float**, for example by an assignment, the **double** is rounded before truncation to **float** length.

6.3 Floating and integral

Conversions of floating values to integral type tend to be rather machine-dependent. On the PDP-11, truncation is towards 0. The result is undefined if the value will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of precision occurs if the destination lacks sufficient bits.

6.4 Pointers and integers

An integer or long integer may be added to or subtracted from a pointer; in such a case the first is converted as specified in the discussion of the addition operator.

Two pointers to objects of the same type may be subtracted; in this case the result is converted to an integer as specified in the discussion of the subtraction operator.

6.5 Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value (on the PDP-11) is the least unsigned integer congruent to the signed integer (modulo 2^{16}). Because of the 2's complement notation, this conversion is conceptual and there is no actual change in the bit pattern.

When an unsigned integer is converted to long, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

6.6 Arithmetic conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the 'usual arithmetic conversions.'

First, any operands of type **char** or **short** are converted to **int**, and any of type **float** are converted to **double**.

Then, if either operand is **double**, the other is converted to **double** and that is the type of the result.

Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result.

Otherwise, if either operand is **unsigned**, the other is converted to **unsigned** and that is the type of the result.

Otherwise, both operands must be **int**, and that is the type of the result.

7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Thus the expressions referred to as the operands of + (§7.4) are those expressions defined in §§7.1-7.3. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in the collected grammar.

Otherwise the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects. Expressions involving a commutative and associative operator may be rearranged arbitrarily, even in the presence of parentheses; to force a particular order of evaluation an explicit temporary must be used.

7.1 Primary expressions

Primary expressions involving ., −>, subscripting, and function calls group left to right.

primary-expression:
identifier
constant
string
(*expression*)
primary-expression [*expression*]
primary-expression (*expression-list*$_{opt}$)
primary-lvalue . *identifier*
primary-expression −> *identifier*

expression-list:
expression
expression-list , *expression*

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. However, if the type of the identifier is 'array of ...', then the value of the identifier-expression is a pointer to the first object in the array, and the type of the expression is 'pointer to ...'. Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared 'function returning ...', when used except in the function-name position of a call, is converted to 'pointer to function returning ...'.

A constant is a primary expression. Its type may be **int**, **long**, or **double** depending on its form.

A string is a primary expression. Its type is originally 'array of **char**'; but following the same rule given above for identifiers, this is modified to 'pointer to **char**' and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see §8.6.)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type 'pointer to ...', the subscript expression is **int**, and the type of the result is '...'. The expression 'E1[E2]' is identical (by definition) to '*((E1) + (E2))'. All the clues needed to understand this notation are contained in this section together with the discussions in §§ 7.1, 7.2, and 7.4 on identifiers, *, and + respectively; §14.3 below summarizes the implications.

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type 'function returning ...', and the result of the function call is of type '...'. As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type **float** are converted to **double** before the call; any of type **char** or **short** are converted to **int**.

In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. On the other hand, it is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ.

Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be an lvalue naming a structure or union, and the identifier must name a member of the structure or union. The result is an lvalue referring to the named member of the structure or union.

A primary expression followed by an arrow (built from a '−' and a '>') followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points.

Thus the expression 'E1−>MOS' is the same as '(*E1).MOS'. Structures and unions are discussed in §8.5. The rules given here for the use of structures and unions are not enforced strictly, in order to allow an escape from the typing mechanism. See §14.1.

7.2 Unary operators

Expressions with unary operators group right-to-left.

unary-expression:
- * *expression*
- & *lvalue*
- − *expression*
- ! *expression*
- ~ *expression*
- ++ *lvalue*
- −− *lvalue*
- *lvalue* ++
- *lvalue* −−
- (*type-name*) *expression*
- **sizeof** *expression*
- **sizeof** (*type-name*)

The unary * operator means *indirection:* the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is 'pointer to ...', the type of the result is '...'.

The result of the unary & operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is '...', the type of the result is 'pointer to ...'.

The result of the unary − operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n, where n is 16 on the PDP-11.

The result of the logical negation operator ! is 1 if the value of its operand is 0, 0 if the value of its operand is non-zero. The type of the result is **int**. It is applicable to any arithmetic type or to pointers.

The ~ operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix '++' is incremented. The value is the new value of the operand, but is not an lvalue. The expression '++a' is equivalent to '(a += 1)'. See the discussions of addition (§7.4) and assignment operators (§7.14) for information on conversions.

The lvalue operand of prefix '−−' is decremented analogously to the ++ operator.

When postfix '++' is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix '−−' is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the

prefix −− operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. The construction of type names is described in §8.7.

The **sizeof** operator yields the size, in bytes, of its operand. (A *byte* is undefined by the language except in terms of the value of **sizeof**. However in all existing implementations a byte is the space required to hold a **char**.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an integer constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The **sizeof** operator may also be applied to a parenthesized type name. In that case it yields the size, in bytes, of an object of the indicated type.

The construction 'sizeof(type)' is taken to be a unit, so the expression 'sizeof(type)−2' is the same as '(sizeof(type))−2'.

7.3 Multiplicative operators

The multiplicative operators *, /, and % group left-to-right. The usual arithmetic conversions are performed.

multiplicative-expression:
expression * *expression*
expression / *expression*
expression % *expression*

The binary * operator indicates multiplication. The * operator is associative and expressions with several multiplications at the same level may be rearranged.

The binary / operator indicates division. When positive integers are divided truncation is toward 0, but the form of truncation is machine-dependent if either operand is negative. In all cases it is true that $(a/b)*b + a\%b = a$. On the PDP-11, the remainder has the same sign as the dividend.

The binary % operator yields the remainder from the division of the first expression by the second. The usual arithmetic conversions are performed. On the PDP-11, the remainder has the same sign as the dividend. The operands must not be floating.

7.4 Additive operators

The additive operators + and − group left-to-right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
expression + *expression*
expression − *expression*

The result of the '+' operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer, and which points to another object in the same array, appropriately offset from the original object. Thus if P is a pointer to an object in an array, the expression 'P+1' is a pointer to the next object in the array.

No further type combinations are allowed.

The + operator is associative and expressions with several additions at the same level may be rearranged.

The result of the '−' operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a

pointer, and then the same conversions as for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an **int** representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object-length.

7.5 Shift operators

The shift operators << and >> group left-to-right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to **int**; the type of the result is that of the left operand. The result is undefined if the right operand is negative or larger than the number of bits in the object.

shift-expression:
expression << *expression*
expression >> *expression*

The value of 'E1<<E2' is E1 (interpreted as a bit pattern) left-shifted E2 bits; vacated bits are 0-filled. The value of 'E1>>E2' is E1 right-shifted E2 bit positions. The shift is guaranteed to be logical (0-fill) if E1 is **unsigned**; otherwise it may be (and is, on the PDP-11) arithmetic (fill by a copy of the sign bit).

7.6 Relational operators

The relational operators group left-to-right, but this fact is not very useful; 'a<b<c' does not mean what it seems to.

relational-expression:
expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is **int**. The usual arithmetic conversions are performed. Two pointers may be compared, and the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

7.7 Equality operators

equality-expression:
expression == *expression*
expression != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus 'a<b == c<d' is 1 whenever a<b and c<d have the same truth-value).

A pointer may be compared to an integer, but the result is machine dependent unless the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object, and will appear to be equal to 0; in conventional usage, such a pointer is considered to be null.

7.8 Bitwise *and* operator

and-expression:
expression & *expression*

The & operator is associative and expressions involving & may be rearranged. The usual arithmetic conversions are performed; the result is the bit-wise 'and' function of the operands. The operator applies only to integral operands.

7.9 Bitwise *exclusive or* operator

exclusive-or-expression:
expression ^ *expression*

The ^ operator is associative and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is is the bit-wise 'exclusive or' function of the operands. The operator applies only to integral operands.

7.10 Bitwise *inclusive or* operator

inclusive-or-expression:
expression | *expression*

The | operator is associative and expressions with | may be rearranged. The usual arithmetic conversions are performed; the result is the bit-wise 'inclusive or' function of its operands. The operator applies only to integral operands.

7.11 Logical *and* operator

logical-and-expression:
expression && *expression*

The && operator groups left-to-right. It returns 1 if both its operands are non-zero, 0 otherwise. Unlike &, && guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always int.

7.12 Logical *or* operator

logical-or-expression:
expression || *expression*

The || operator groups left-to-right. It returns 1 if either of its operands is non-zero, and 0 otherwise. Unlike |, || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always int.

7.13 Conditional operator

conditional-expression:
expression ? *expression* : *expression*

Conditional expressions group right-to-left. The first expression is evaluated and if it is non-zero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type; otherwise, if both are pointers of the same type, the result has the common type; otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

7.14 Assignment operators

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:
lvalue = *expression*
lvalue + = *expression*
lvalue − = *expression*
lvalue * = *expression*
lvalue / = *expression*
lvalue % = *expression*
lvalue >> = *expression*
lvalue << = *expression*
lvalue & = *expression*
lvalue ^ = *expression*
lvalue | = *expression*

Notice that the representation of the compound assignment operators has changed; formerly the '=' came first and the other operator came second (without any space). The compiler continues to accept the previous notation.

In the simple assignment with '=', the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment.

The behavior of an expression of the form 'E1 op = E2' may be inferred by taking it as equivalent to 'E1 = E1 op (E2)'; however, E1 is evaluated only once. In += and −=, the left operand may be a pointer, in which case the (integral) right operand is converted as explained in §7.4; all right operands and all non-pointer left operands must have arithmetic type.

The compiler currently allows a pointer to be assigned to an integer, an integer to a pointer, and a pointer to a pointer of another type. The assignment is a pure copy operation, with no conversion. This usage is nonportable, and may produce pointers which cause addressing exceptions when used. However, it is guaranteed that assignment of the constant 0 to a pointer will produce a null pointer distinguishable from a pointer to any object.

7.15 Comma operator

comma-expression:
expression , *expression*

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right. In contexts where comma is given a special meaning, for example in a list of actual arguments to functions (§7.1) and lists of initializers (§8.6), the comma operator as described in this section can only appear in parentheses; for example, 'f(a, (t = 3, t+2), c)' has three arguments, the second of which has the value 5.

8. Declarations

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:
decl-specifiers declarator-list$_{opt}$;

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

decl-specifiers:
type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$

The list must be self-consistent in a way described below.

8.1 Storage class specifiers

The sc-specifiers are:

sc-specifier:
auto
static
extern
register
typedef

The **typedef** specifier does not reserve storage and is called a 'storage class specifier' only for syntactic convenience; it is discussed in §8.8.

The meanings of the various storage classes were discussed in §4.

The **auto**, **static**, and **register** declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the **extern** case there must be an external definition (§10) for the given identifiers somewhere outside the function in which they are declared.

A **register** declaration is best thought of as an **auto** declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few (three, for the PDP-11) such declarations are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are **int**, **char**, or pointer. One restriction applies to register variables: the address-of operator **&** cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future developments may render them unnecessary.

At most one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be **auto** inside a function, **extern** outside. Exception: functions are always **extern**.

8.2 Type specifiers

The type-specifiers are

type-specifier:
char
short
int
long
unsigned
float
double
struct-or-union-specifier
typedef-name

The words **long**, **short**, and **unsigned** may be thought of as adjectives; the following combinations are acceptable (in any order).

short int
long int
unsigned int
long float

The meaning of the last is the same as **double**. Otherwise, at most one type-specifier may be given in a declaration. If the type-specifier is missing from a declaration, it is taken to be **int**.

Specifiers for structures and unions are discussed in §8.5; declarations with **typedef** names are discussed in §8.8.

8.3 Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

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

init-declarator:
declarator initializer$_{opt}$

Initializers are discussed in §8.6. The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
identifier
(*declarator*)
* *declarator*
declarator ( )
declarator [*constant-expression*$_{opt}$]

The grouping is the same as in expressions.

8.4 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class. Each declarator contains exactly one identifier; it is this identifier that is declared.

If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

If a declarator has the form

*** D**

for D a declarator, then the contained identifier has the type 'pointer to ...', where '...' is the type which the identifier would have had if the declarator had been simply D.

If a declarator has the form

D ( )

then the contained identifier has the type 'function returning ...', where '...' is the type which the identifier would have had if the declarator had been simply D.

A declarator may have the form

D[constant-expression]

or

D[ ]

Such declarators make the contained identifier have type 'array.' If the unadorned declarator D would specify a non-array of type '...', then the declarator 'D[i]' yields a 1-dimensional array with rank i of objects of type '...'. If the unadorned declarator D would specify an n-dimensional array with rank $i_1 \times i_2 \times \cdots \times i_n$, then the declarator D[$i_{n+1}$] yields an $(n+1)$-dimensional array with rank $i_1 \times i_2 \times \cdots \times i_n \times i_{n+1}$.

In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is **int**. (Constant expressions are defined precisely in §15.) The constant expression of an array declarator may be missing only for the first dimension. This notation is useful when the array is external and the actual declaration, which allocates storage, is given elsewhere. The constant-expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays, structures or functions, although they may return pointers to such things; there are no arrays of functions, although there may be arrays of pointers to functions. Likewise a structure may not contain a function, but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f( ), *fip( ), (*pfi) ( );
```

declares an integer *i*, a pointer *ip* to an integer, a function *f* returning an integer, a function *fip* returning a pointer to an integer, and a pointer *pfi* to a function which returns an integer. It is especially useful to compare the last two. The binding of '*fip()' is '*(fip())', so that the declaration suggests, and the same construction in an expression requires, the calling of a function *fip*, and then using indirection through the (pointer) result to yield an integer. In the declarator '(*pfi) ()', the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called.

As another example,

```
float fa[17], *afp[17];
```

declares an array of **float** numbers and an array of pointers to **float** numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank 3×5×7. In complete detail, *x3d* is an array of three items: each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions 'x3d', 'x3d[i]', 'x3d[i][j]', 'x3d[i][j][k]' may reasonably appear in an expression. The first three have type 'array', the last has type **int**.

8.5 Structure and union declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

structure-or-union-specifier:
    struct-or-union { *struct-decl-list* }
    struct-or-union identifier { *struct-decl-list* }
    struct-or-union identifier

struct-or-union:
    struct
    union

The struct-decl-list is a sequence of declarations for the members of the structure or union:

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

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

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

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field;* its length is set off from the field name by a colon.

struct-declarator:
declarator
declarator : *constant-expression*
: *constant-expression*

Within a structure, the objects declared have addresses which increase as their declarations are read left-to-right. Each non-field member of a structure begins on an addressing boundary appropriate to its type. On the PDP-11 the only requirement is that non-characters begin on a word boundary; therefore, there may be 1-byte, unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word. On the PDP-11, fields are assigned right-to-left.

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, an unnamed field with a width of 0 specifies alignment of the next field at a word boundary. The 'next field' presumably is a field, not an ordinary structure member, because in the latter case the alignment would have been automatic.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even **int** fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

struct *identifier* { *struct-decl-list* }
union *identifier* { *struct-decl-list* }

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

struct *identifier*
union *identifier*

Structure tags allow definition of self-referential structures; they also permit the long part of the declaration to be given once and used several times. It is however absurd to declare a structure or union which contains an instance of itself, as distinct from a pointer to an instance of itself.

The names of members and tags may be the same as ordinary variables. However, names of tags and members must be mutually distinct.

Two structures may share a common initial sequence of members; that is, the same member may appear in two different structures if it has the same type in both and if all previous members are the same in both. (Actually, the compiler checks only that a name in two different structures has the same type and offset in both, but if preceding members differ the construction is nonportable.)

A simple example of a structure declaration is

```
struct tnode {
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the following declaration makes sense:

```
struct tnode s, *sp;
```

which declares *s* to be a structure of the given sort and *sp* to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the *count* field of the structure to which *sp* points;

```
s.left
```

refers to the left subtree pointer of the structure *s*. Finally,

```
s.right->tword[0]
```

refers to the first character of the *tword* member of the right subtree of *s*.

8.6 Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by '=', and consists of an expression or a list of values nested in braces.

initializer:
= *expression*
= { *initializer-list* }
= { *initializer-list* , }

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

The '=' is a new addition to the syntax, intended to alleviate potential ambiguities. The current compiler allows it to be omitted when the rest of the initializer is a very simple expression (just a name, string, or constant) or when the rest of the initializer is enclosed in braces.

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in §15, or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving previously declared variables.

Static and external variables which are not initialized are guaranteed to start off as 0; automatic and register variables which are not initialized are guaranteed to start off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array) then the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate, written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with 0's. It is not permitted to initialize unions or automatic aggregates. Currently, the PDP-11 compiler also

forbids initializing fields in structures.

Braces may be elided as follows. If the initializer begins with a left brace, then the succeding comma-separated list of initializers initialize the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a **char** array to be initialized by a string. In this case successive members of the string initialize the members of the array.

For example,

```
int x[ ] = { 1, 3, 5 };
```

declares and initializes *x* as a 1-dimensional array which has three members, since no size was specified and there are three initializers.

```
float  y[4][3] = {
       { 1, 3, 5 },
       { 2, 4, 6 },
       { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array *y*[0], namely *y*[0][0], *y*[0][1], and *y*[0][2]. Likewise the next two lines initialize *y*[1] and *y*[2]. The initializer ends early and therefore *y*[3] is initialized with 0. Precisely the same effect could have been achieved by

```
float  y[4][3] = {
       1, 3, 5, 2, 4, 6, 3, 5, 7,
};
```

The initializer for *y* begins with a left brace, but that for *y*[0] does not, therefore 3 elements from the list are used. Likewise the next three are taken successively for *y*[1] and *y*[2]. Also,

```
float  y[4][3] = {
       { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of *y* (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char   msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

8.7 Type names

In two contexts (to specify type conversions explicitly, and as an argument of **sizeof**) it is desired to supply the name of a data type. This is accomplished using a 'type name,' which in essence is a declaration for an object of that type which omits the name of the object.

type-name:
    type-specifier abstract-declarator

abstract-declarator:
empty
(*abstract-declarator*)
* *abstract-declarator*
abstract-declarator ( )
abstract-declarator [*constant-expression*$_{opt}$]

To avoid ambiguity, in the construction

(*abstract-declarator*)

the abstract-declarator is required to be nonempty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
```

name respectively the types 'integer,' 'pointer to integer,' 'array of 3 pointers to integers,' and 'pointer to an array of 3 integers.' As another example,

```
int i;
...
sin( (double) i);
```

calls the *sin* routine (which accepts a **double** argument) with an argument appropriately converted.

8.8 Typedef

Declarations whose 'storage class' is **typedef** do not define storage, but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types. Within the scope of a declaration involving **typedef**, each of the identifiers appearing as part of any declarators therein become syntactically equivalent to type keywords naming the type associated with the identifiers in the way described in §8.4.

typedef-name:
identifier

For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im;} complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of *distance* is 'int', that of *metricp* is 'pointer to int,' and that of *z* is the specified structure. *Zp* is a pointer to such a structure.

Typedef does not introduce brand new types, only synonyms for types which could be specified in another way. Thus in the example above *distance* is considered to have exactly the same type as any other int variable.

9. Statements

Except as indicated, statements are executed in sequence.

9.1 Expression statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

9.2 Compound statement, or block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called 'block') is provided:

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, at which time it resumes its force.

Any initializations of **auto** or **register** variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of **static** variables are performed only once when the program begins execution. Inside a block, **external** declarations do not reserve storage so initialization is not permitted.

9.3 Conditional statement

The two forms of the conditional statement are

if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the 'else' ambiguity is resolved by connecting an **else** with the last encountered elseless if.

9.4 While statement

The **while** statement has the form

while (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

9.5 Do statement

The **do** statement has the form

do *statement* **while** (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes zero. The test takes place after each execution of the statement.

9.6 For statement

The **for** statement has the form

for (*expression-1*$_{opt}$; *expression-2*$_{opt}$; *expression-3*$_{opt}$) *statement*

This statement is equivalent to

expression-1;
while (*expression-2*) {
    statement
    expression-3;
}

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression typically specifies an incrementation which is performed after each iteration.

Any or all of the expressions may be dropped. A missing expression-2 makes the implied **while** clause equivalent to 'while(1)'; other missing expressions are simply dropped from the expansion above.

9.7 Switch statement

The **switch** statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

switch (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be **int**. The statement is typically compound. Any statement within the statement may be labelled with one or more case prefixes as follows:

case *constant-expression* :

where the constant expression must be **int**. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in §15.

There may also be at most one statement prefix of the form

default :

When the **switch** statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression, and if there is a **default** prefix, control passes to the prefixed statement. If no case matches and if there is no **default** then none of the statements in the switch is executed.

Case and **default** prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see **break**, §9.8.

Usually the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, initializations of automatic or register variables are ineffective.

9.8 Break statement

The statement

break ;

causes termination of the smallest enclosing **while**, **do**, **for**, or **switch** statement; control passes to the statement following the terminated statement.

9.9 Continue statement

The statement

```
continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement; that is to the end of the loop. More precisely, in each of the statements

```
while ( ... ) {          do {                for ( ... ) {
  ...                      ...                 ...
contin: ;                contin: ;           contin: ;
}                        } while ( ... ) ;   }
```

a **continue** is equivalent to 'goto contin'. (Following the 'contin:' is a null statement, §9.13.)

9.10 Return statement

A function returns to its caller by means of the **return** statement, which has one of the forms

```
return ;
return expression ;
```

In the first case the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

9.11 Goto statement

Control may be transferred unconditionally by means of the statement

```
goto identifier ;
```

The identifier must be a label (§9.12) located in the current function. Previous versions of C had an incompletely implemented notion of label variable, which has been withdrawn.

9.12 Labelled statement

Any statement may be preceded by label prefixes of the form

```
identifier :
```

which serve to declare the identifier as a label. The only use of a label is as a target of a **goto**. The scope of a label is the current function, excluding any sub-blocks in which the same identifier has been redeclared. See §11.

9.13 Null statement

The null statement has the form

```
;
```

A null statement is useful to carry a label just before the '}' of a compound statement or to supply a null body to a looping statement such as **while**.

10. External definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class **extern** (by default) or perhaps **static**, and a specified type. The type-specifier (§8.2) may also be empty, in which case the type is taken to be **int**. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations, except that only at this level may the code for functions be given.

10.1 External function definitions

Function definitions have the form

function-definition:
decl-specifiers$_{opt}$ *function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are **extern** or **static**; See §11.2 for the distinction between them. A function declarator is similar to a declarator for a 'function returning ...' except that it lists the formal parameters of the function being defined.

function-declarator:
declarator (*parameter-list*$_{opt}$)

parameter-list:
identifier
identifier , *parameter-list*

The function-body has the form

function-body:
declaration-list compound-statement

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be **int**. The only storage class which may be specified is **register**; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
int a, b, c;
{
        int m;
        m = (a > b)? a : b;
        return(m > c? m : c);
}
```

Here 'int' is the type-specifier; 'max(a, b, c)' is the function-declarator; 'int a, b, c;' is the declaration-list for the formal parameters; '{ ... }' is the block giving the code for the statement. The parentheses in the **return** are not required.

C converts all **float** actual parameters to **double**, so formal parameters declared **float** have their declaration adjusted to read **double**. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared 'array of ...' are adjusted to read 'pointer to ...'. Finally, because neither structures nor functions can be passed to a function, it is useless to declare a formal parameter to be a structure or function (pointers to structures or functions are of course permitted).

A free **return** statement is supplied at the end of each function definition, so running off the end causes control, but no value, to be returned to the caller.

10.2 External data definitions

An external data definition has the form

data-definition:
declaration

The storage class of such data may be **extern** (which is the default) or **static**, but not **auto** or **register**.

11. Scope rules

A C program need not all be compiled at the same time: the source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, what may be called the *lexical scope* of an identifier, which is essentially the region of a program during which it may be used without drawing 'undefined identifier' diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

11.1 Lexical scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of blocks persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

Because all references to the same external identifier refer to the same object (see §11.2) the compiler checks all declarations of the same external identifier for compatibility; in effect their scope is increased to the whole file in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (§8.5) that identifiers associated with ordinary variables on the one hand and those associated with structure and union members and tags on the other form two disjoint classes which do not conflict. **Typedef** names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{   auto int distance;
    ...
```

The **int** must be present in the second declaration, or it would be taken to be a declaration with no declarators and type **distance.***

11.2 Scope of externals

If a function declares an identifier to be **extern**, then somewhere among the files or libraries constituting the complete program there must be an external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and extent specified in the definition are compatible with those specified by each function which references the data.

In PDP-11 C, compatible external definitions of the same identifier may be present in several of the separately-compiled pieces of a complete program, or even twice within the same program file, with the limitation that the identifier may be initialized in at most one of the definitions. In other operating systems, however, the compiler must know in just which file the storage for the identifier is allocated, and in which file the identifier is merely being referred to. The appearance of the **extern** keyword in an external definition indicates that storage for the identifiers being declared will be allocated in another file. Thus in a multi-file program, an

*It is agreed that the ice is thin here.

external data definition without the **extern** specifier must appear in exactly one of the files. Any other files which wish to give an external definition for the identifier must include the **extern** in the definition. The identifier can be initialized only in the declaration where storage is allocated.

Identifiers declared **static** at the top level in external definitions are not visible in other files.

12. Compiler control lines

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with '#' communicate with this preprocessor. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

12.1 Token replacement

A compiler-control line of the form

 # define *identifier token-string*

(note: no trailing semicolon) causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. A line of the form

 # define *identifier*(*identifier* , ... , *identifier*) *token-string*

where there is no space between the first identifier and the '(', is a macro definition with arguments. Subsequent instances of the first identifier followed by a '(', a sequence of tokens delimited by commas, and a ')' are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Text inside a string or a character constant is not subject to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing '\' at the end of the line to be continued.

This facility is most valuable for definition of 'manifest constants', as in

```
# define TABSIZE 100
...
int table[TABSIZE];
```

A control line of the form

 # undef *identifier*

causes the identifier's preprocessor definition to be forgotten.

12.2 File inclusion

A compiler control line of the form

 # include "*filename*"

causes the replacement of that line by the entire contents of the file *filename.*

The named file is searched for first in the directory of the original source file, and then in a sequence of standard places. Alternatively, a control line of the form

 # include <filename>

searches only the standard places, and not the directory of the source file.

Includes may be nested.

12.3 Conditional compilation

A compiler control line of the form

 # if *constant-expression*

checks whether the constant expression (see §15) evaluates to non-zero. A control line of the form

 # ifdef *identifier*

checks whether the identifier is currently defined in the preprocessor; that is, whether it has been the subject of a **#define** control line. A control line of the form

 # ifndef *identifier*

checks whether the identifier is currently undefined in the preprocessor.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

 # else

and then by a control line

 # endif

If the checked condition is true then any lines between **#else** and **#endif** are ignored. If the checked condition is false then any lines between the test and an **#else** or, lacking an **#else**, the **#endif**, are ignored.

These constructions may be nested.

12.4 Line control

For the benefit of other preprocessors which generate C programs, a line of the form

 # line *constant identifier*

causes the compiler to believe, for purposes of error diagnostics, that the next line number is given by the constant and the current input file is named by the identifier. If the identifier is absent the remembered file name does not change.

13. Implicit declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. Sometimes the storage class is supplied by the context: in external definitions, and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be **int**; if a type but no storage class is indicated, the identifier is assumed to be **auto**. An exception to the latter rule is made for functions, since **auto** functions are meaningless (C being incapable of compiling code into the stack). If the type of an identifier is 'function returning ...', it is implicitly declared to be **extern**.

In an expression, an identifier followed by (and not currently declared is contextually declared to be 'function returning **int**'.

14. Types revisited

This section summarizes the operations which can be performed on objects of certain types.

14.1 Structures and unions

There are only two things that can be done with a structure or union: name one of its members (by means of the . operator); or take its address (by unary &). Other operations, such as assigning from or to it or passing it as a parameter, draw an error message. In the future, it is expected that these operations, but not necessarily others, will be allowed.

§7.1 says that in a direct or indirect structure reference (with . or −>) the name on the

right must be a member of the structure named or pointed to by the expression on the left. To allow an escape from the typing rules, this restriction is not firmly enforced by the compiler. In fact, any lvalue is allowed before '.', and that lvalue is then assumed to have the form of the structure of which the name on the right is a member. Also, the expression before a '−>' is required only to be a pointer or an integer. If a pointer, it is assumed to point to a structure of which the name on the right is a member. If an integer, it is taken to be the absolute address, in machine storage units, of the appropriate structure.

Such constructions are non-portable.

14.2 Functions

There are only two things that can be done with a function: call it, or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f( );
...
g(f);
```

Then the definition of *g* might read

```
g(funcp)
int (*funcp) ( );
{
        ...
        (*funcp) ( );
        ...
}
```

Notice that *f* was declared explicitly in the calling routine since its first appearance was not followed by (.

14.3 Arrays, pointers, and subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator [] is interpreted in such a way that 'E1[E2]' is identical to '*((E1) + (E2))'. Because of the conversion rules which apply to +, if E1 is an array and E2 an integer, then E1[E2] refers to the E2-th member of E1. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multi-dimensional arrays. If E is an n-dimensional array of rank $i \times j \times \cdots \times k$, then E appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \cdots \times k$. If the * operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to $(n-1)$-dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here *x* is a 3×5 array of integers. When *x* appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression 'x[i]', which is equivalent to '*(x+i)', *x* is first converted to a pointer as described; then *i* is converted to the type of *x*, which involves multiplying *i* by the length the object to which the pointer points, namely 5 integer objects. The results are added and indirection applied to yield an array (of 5 integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer.

It follows from all this that arrays in C are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed

by an array but plays no other part in subscript calculations.

15. Constant expressions

In several places C requires expressions which evaluate to a constant: after **case**, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, and **sizeof** expressions, possibly connected by the binary operators

```
+  -  *  /  %  &  |  ^  <<  >>  ==  !=  <  >  <=  >=
```

or by the unary operators

```
-  ~
```

or by the ternary operator

```
?  :
```

Parentheses can be used for grouping, but not for function calls.

A bit more latitude is permitted for initializers; besides constant expressions as discussed above, one can also apply the unary & operator to external or static objects, and to external or static arrays subscripted with a constant expression. The unary & can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

16. Grammar revisited.

This section repeats the grammar of C in notation somewhat different than given before. The description below is adapted directly from a YACC grammar actually used by several compilers; thus it may (aside from possible editing errors) be regarded as authentic. The notation is pure YACC with the exception that the '|' separating alternatives for a production is omitted, since alternatives are always on separate lines; the ';' separating productions is omitted since a blank line is left between productions.

The lines with '%term' name the terminal symbols, which are either commented upon or should be self-evident. The lines with '%left,' '%right,' and '%binary' indicate whether the listed terminals are left-associative, right-associative, or non-associative, and describe a precedence structure. The precedence (binding strength) increases as one reads down the page. When the construction '%prec *x*' appears the precedence of the rule is that of the terminal *x;* otherwise the precedence of the rule is that of its leftmost terminal.

```
%term NAME
%term STRING
%term ICON
%term FCON
%term PLUS
%term MINUS
%term MUL
%term AND
%term QUEST
%term COLON
%term ANDAND
%term OROR
%term ASOP      /* old-style =+ etc. */
%term RELOP     /* <= >= < > */
%term EQUOP     /* == != */
%term DIVOP     /* / % */
%term OR        /* | */
%term EXOR      /* ^ */
%term SHIFTOP   /* << >> */
%term INCOP     /* ++ -- */
%term UNOP      /* ! ~ */
%term STROP     /* . -> */


%term TYPE      /* int, char, long, float, double, unsigned, short */
%term CLASS     /* extern, register, auto, static, typedef */
%term STRUCT    /* struct or union */
%term RETURN
%term GOTO
%term IF
%term ELSE
%term SWITCH
%term BREAK
%term CONTINUE
%term WHILE
%term DO
%term FOR
%term DEFAULT
%term CASE
%term SIZEOF
```

```
%term LP          /* ( */
%term RP          /* ) */
%term LC          /* { */
%term RC          /* } */
%term LB          /* [ */
%term RB          /* ] */
%term CM          /* , */
%term SM          /* ; */
%term ASSIGN      /* = */


%left    CM
%right   ASOP     ASSIGN
%right   QUEST    COLON
%left    OROR
%left    ANDAND
%left    OROP
%left    AND
%binary  EQUOP
%binary  RELOP
%left    SHIFTOP
%left    PLUS     MINUS
%left    MUL      DIVOP
%right   UNOP
%right   INCOP    SIZEOF
%left    LB       LP       STROP


program:          ext_def_list

ext_def_list:     ext_def_list external_def
                  /* empty */

external_def:     optattrib SM
                  optattrib init_dcl_list SM
                  optattrib fdeclarator function_body

function_body:    dcl_list compoundstmt

dcl_list:         dcl_list declaration
                  /* empty */

declaration:      specifiers declarator_list SM
                  specifiers SM

optattrib:        specifiers
                  /* empty */

specifiers:       CLASS type
                  type CLASS
                  CLASS
                  type
```

```
type:           TYPE
                TYPE TYPE
                struct_dcl

struct_dcl:     STRUCT NAME LC type_dcl_list RC
                STRUCT LC type_dcl_list RC
                STRUCT NAME

type_dcl_list:  type_declaration
                type_dcl_list type_declaration

type_declaration: type declarator_list SM
                struct_dcl SM
                type SM

declarator_list: declarator
                declarator_list CM declarator

declarator:     fdeclarator
                nfdeclarator
                nfdeclarator COLON con_e %prec CM
                COLON con_e %prec CM

nfdeclarator:   MUL nfdeclarator
                nfdeclarator LP RP
                nfdeclarator LB RB
                nfdeclarator LB con_e RB
                NAME
                LP nfdeclarator RP

fdeclarator:    MUL fdeclarator
                fdeclarator LP RP
                fdeclarator LB RB
                fdeclarator LB con_e RB
                LP fdeclarator RP
                NAME LP name_list RP
                NAME LP RP

name_list:      NAME
                name_list CM NAME

init_dcl_list:  init_declarator %prec CM
                init_dcl_list CM init_declarator

init_declarator: nfdeclarator
                nfdeclarator ASSIGN initializer
                nfdeclarator initializer
                fdeclarator

init_list:      initializer %prec CM
                init_list CM initializer

initializer:    e %prec CM
                LC init_list RC
```

```
                LC init_list CM RC

compoundstmt:   LC dcl_list stmt_list RC

stmt_list:      stmt_list statement
                /* empty */

statement:      e SM
                compoundstmt
                IF LP e RP statement
                IF LP e RP statement ELSE statement
                WHILE LP e RP statement
                DO statement WHILE LP e RP SM
                FOR LP opt_e SM opt_e SM opt_e RP statement
                SWITCH LP e RP statement
                BREAK SM
                CONTINUE SM
                RETURN SM
                RETURN e SM
                GOTO NAME SM
                SM
                label statement

label:          NAME COLON
                CASE con_e COLON
                DEFAULT COLON

con_e:          e %prec CM

opt_e:          e
                /* empty */

elist:          e %prec CM
                elist CM e

e:              e MUL e
                e CM e
                e DIVOP e
                e PLUS e
                e MINUS e
                e SHIFTOP e
                e RELOP e
                e EQUOP e
                e AND e
                e OROP e
                e ANDAND e
                e OROR e
                e MUL ASSIGN e
                e DIVOP ASSIGN e
                e PLUS ASSIGN e
                e MINUS ASSIGN e
                e SHIFTOP ASSIGN e
                e AND ASSIGN e
                e OROP ASSIGN e
```

```
                e QUEST e COLON e
                e ASOP e
                e ASSIGN e
                term

term:           term INCOP
                MUL term
                AND term
                MINUS term
                UNOP term
                INCOP term
                SIZEOF term
                LP type_name RP term %prec STROP
                SIZEOF LP type_name RP %prec SIZEOF
                term LB e RB
                term LP RP
                term LP elist RP
                term STROP NAME
                NAME
                ICON
                FCON
                STRING
                LP e RP

type_name:      type abst_decl

abst_decl:      /* empty */
                LP RP
                LP abst_decl RP LP RP
                MUL abst_decl
                abst_decl LB RB
                abst_decl LB con_e RB
                LP abst_decl RP
```