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

```
c = getchar()
```

can be used in an expression; its value is simply the value being assigned to the left 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; 2nd version */
{
	int c;
	                          only called once
	while ((c = getchar()) != EOF)
		putchar(c);
}
```

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

This version centralizes the input — there is now only one call to `getchar` — and shrinks the program. Nesting an assignment 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 within the conditional are really necessary. The *precedence* of `!=` is higher than that of =, which means that in the absence of parentheses the relational test `!=` would be done before the assignment =. So the statement

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

is equivalent to

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

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

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 statement

```
++nc;
```

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

The character counting program accumulates its count in a `long` variable instead of an `int`. 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 conversion specification `%ld` signals to `printf` that the corresponding argument is a `long` integer.

To cope with even bigger numbers, you can use a `double` (double length `float`). We will also use a `for` statement instead of a `while`, to illustrate an alternative way to write the loop.

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

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

`printf` uses `%f` for both `float` and `double`; `%.0f` suppresses printing of 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` statements help ensure that they do reasonable things with boundary conditions.

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.

```
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`. The `if` statement 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.). This 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.

Any single character can be written between single quotes, to produce a value equal to the numerical value of the character in the machine's character set; this is called a *character constant.* So, for example, `'A'` is a character constant; in the ASCII character set its value is 65, the internal representation of the character `A`. Of course `'A'` is to be preferred over `65`: its meaning is obvious, and it is independent of a particular character set.

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

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

Exercise 1-7. Write a program to copy its input to its output, replacing each string of one or more blanks by a single blank. □

Exercise 1-8. Write a program to replace each tab by the three-character sequence >, *backspace*, –, which prints as ⋗, and each backspace by the similar sequence ⋖. This makes tabs and backspaces visible. □

Word Counting

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

```
#define   YES  1
#define   NO   0

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

     inword = NO;
     nl = nw = nc = 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);
}
```

Every time the program encounters the first character of a word, it counts it. The variable inword records whether the program is currently in a word or not; 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

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

sets all three variables to zero. This is not a special case, but a consequence of the fact that an assignment has a value and assignments associate right to left. 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 escape sequence `\t` is a visible representation of the tab character.) There is a corresponding operator `&&` for AND. Expressions connected by `&&` or `||` are evaluated left to right, and it is guaranteed that evaluation will stop as soon as the truth or falsehood is known. 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. This isn't particularly important here, but is very significant in more complicated situations, as we will soon see.

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. The general form is

```
    if (expression)
        statement-1
    else
        statement-2
```

One and only one of the two statements associated with an `if-else` is done. If the *expression* is true, *statement-1* is executed; if not, *statement-2* is executed. Each *statement* can in fact be quite complicated. In the word count program, the one after the `else` is an `if` that controls two statements in braces.

Exercise 1-9. How would you test the word count program? What are some boundaries? ■

Exercise 1-10. Write a program which prints the words in its input, one per line. ■

Exercise 1-11. Revise the word count program to use a better definition of "word," for example, a sequence of letters, digits and apostrophes that begins with a letter. □

1.6 Arrays

Let us write a program to count the number of occurrences of each digit, of white space characters (blank, tab, newline), and 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, so it is convenient to use an array to hold the number of occurrences of each digit, rather than ten individual variables. Here is one version of the program:

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

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

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

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

The declaration

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

declares `ndigit` to be an array of 10 integers. Array subscripts always 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, which of course includes 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

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

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

```
c - '0'
```

This works only if `'0'`, `'1'`, etc., are positive and in increasing order, and if there is nothing but digits between `'0'` and `'9'`. Fortunately, this is true for all conventional character sets.

By definition, arithmetic involving `char`'s and `int`'s converts everything to `int` before proceeding, so `char` variables and constants are essentially identical to `int`'s in arithmetic contexts. This is quite natural and convenient; for example, `c - '0'` is an integer expression with a value between 0 and 9 corresponding to the character `'0'` to `'9'` stored in `c`, and is thus a valid subscript for the array `ndigit`.

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

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

The pattern

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

occurs frequently in programs as a way to express a multi-way decision. 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 is finished. (Of course *statement* can be several statements enclosed in braces.) If none of the conditions is satisfied, the *statement* after the final `else` is executed if it is present. If the final `else` and *statement* are omitted (as in the word count program), no action takes place. There can be 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. For contrast, we will present a `switch` version of this program in Chapter 3.

Exercise 1-12. Write a program to print a histogram of the lengths of words in its input. It is easiest to draw the histogram horizontally; a vertical orientation is more challenging. □

1.7 Functions

In C, a *function* is equivalent to a subroutine or function in Fortran, or a procedure in PL/I, Pascal, etc. A function provides a convenient way to encapsulate some computation in a black box, which can then be used without worrying about its innards. Functions are really the only way to cope with the potential complexity of large programs. With properly designed functions, it is possible to ignore *how* a job is done; knowing *what* is done is sufficient. C is designed to make the use of functions easy, convenient and efficient; you will often see a function only a few lines long called only once, just because it clarifies some piece of code.

So far we have used only functions like `printf`, `getchar` and `putchar` that have been provided for us; now it's time to write a few of our own. Since C has no exponentiation operator like the `**` of Fortran or PL/I, let us illustrate the mechanics of function definition by writing a function `power(m, n)` to raise an integer `m` to a positive integer power `n`. That is, the value of `power(2, 5)` is 32. This function certainly doesn't do the whole job of `**` since it handles only positive powers of small integers, but it's best to confuse only one issue at a time.

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

```
main()     /* test power function */
{
     int i;

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

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

Each function has the same form:

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

The functions can appear in either order, and in one source file or in two. Of course if the source appears in two files, you will have to say more to compile and load it than if it all appears in one, but that is an operating system matter, not a language attribute. For the moment, we will assume that both functions are in the same file, so whatever you have learned about running C programs will not change.

The function `power` is called twice in the line

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

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

In `power` the arguments have to be declared appropriately so their types are known. This is done by the line

```
int x, n;
```

that follows the function name. The argument declarations go between the argument list and the opening left brace; each declaration is terminated by a semicolon. The names used by `power` for its arguments are purely *local* to `power`, and not accessible to any other function: other routines can use the same names without conflict. This is also true of the variables `i` and `p`: the `i` in `power` is unrelated to the `i` in `main`.

The value that `power` computes is returned to `main` by the `return` statement, which is just as in PL/I. Any expression may occur within the parentheses. A function need not return a value; a `return` statement with no expression causes control, but no useful value, to be returned to the caller, as does "falling off the end" of a function by reaching the terminating right brace.

Exercise 1-13. Write a program to convert its input to lower case, using a function `lower(c)` which returns `c` if `c` is not a letter, and the lower case value of `c` if it is a letter. □

1.8 Arguments — Call by Value

One aspect of C functions may be unfamiliar to programmers who are 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 some different properties than are seen with "call by reference" languages like Fortran and PL/I, in which the called routine is handed the address of the argument, not its value.

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

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

```
power(x, n)  /* raise x to n-th power; n>0; version 2 */
int x, n;
{
	int p;

	for (p = 1; n > 0; --n)
		p = p * x;
	return(p);
}
```

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

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

When the name of an array is used as an argument, the value passed to the function is actually the location or address of the beginning of the array. (There is *no* copying of array elements.) By subscripting this value, the function can access and alter any element of the array. This is the topic of the next section.

1.9 Character Arrays

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

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

This outline 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 controls the process.

Since things divide so nicely, it would be well to write them that way too. Accordingly, let us first write a separate function `getline` to fetch the *next line* of input; this is a generalization of `getchar`. To make the function useful in other contexts, we'll try to make it as flexible as possible. 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 zero if end of file is encountered. Zero is never a valid line length since every line has at least one character; even a line containing only a newline has length 1.

When we find a line that is longer than the previous longest, it must be saved somewhere. This suggests a second function, `copy`, to copy the new line to a safe place.

Finally, we need a main program to control `getline` and `copy`. Here is the result.

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

main()    /* find longest line */
{
    int  len; /* current line length */
    int  max; /* maximum length seen so far */
    char line[MAXLINE]; /* current input line */
    char save[MAXLINE]; /* longest line, saved */

    max = 0;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
            max = len;
            copy(line, save);
        }
    if (max > 0)   /* there was a line */
        printf("%s", save);
}
                  string

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

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

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

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

main and getline communicate both through a pair of arguments and a returned value. In getline, the arguments are declared by the lines

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

which specify that the first argument is an array, and the second is an integer. The length of the array `s` is not specified in `getline` since it is determined in `main`. `getline` uses `return` to send a value back to the caller, just as the function `power` did. Some functions return a useful value; others, like `copy`, are only used for their effect and return no value.

`getline` puts the character `\0` (the *null character*, whose value is zero) at the end of the array it is creating, to mark the end of the string of characters. This convention is also used by the C compiler: when a string constant like

```
    "hello\n"
```

is written in a C program, the compiler creates an array of characters containing the characters of the string, and terminates it with a `\0` so that functions such as `printf` can detect the end:

`h`	`e`	`l`	`l`	`o`	`\n`	`\0`

The `%s` format specification in `printf` expects a string represented in this form. If you examine `copy`, you will discover that it too relies on the fact that its input argument `s1` is terminated by `\0`, and it copies this character onto the output argument `s2`. (All of this implies that `\0` is not a part of normal text.)

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

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

Exercise 1-14. Revise the main routine of the longest-line program so it will correctly print the length of arbitrarily long input lines, and as much as possible of the text. □

Exercise 1-15. Write a program to print all lines that are longer than 80 characters. □

Exercise 1-16. Write a program to remove trailing blanks and tabs from each line of input, and to delete entirely blank lines. □

Exercise 1-17. Write a function `reverse(s)` which reverses the character string `s`. Use it to write a program which reverses its input a line at a time. □

1.10 Scope; External Variables

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 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. It is for this reason that such variables are usually known as *automatic* variables, following terminology in other languages. We will use the term automatic henceforth to refer to these dynamic local variables. (Chapter 4 discusses the `static` storage class, in which local variables do retain their values between function invocations.)

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

As an alternative to automatic variables, it is possible to define variables 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.) Because external variables are globally accessible, they can be used instead of argument lists to communicate data between functions. Furthermore, because external variables remain in existence permanently, rather than appearing and disappearing as functions are called and exited, they retain their values even after the functions that set them are done.

An external variable has to be *defined* outside of any function; this allocates actual storage for it. The variable must also be *declared* in each function that wants to access it; this may be done either by an explicit `extern` declaration or implicitly by context. 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 all three functions.

external variables

```
#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; specialized version */
{
        int len;
        extern int max;
        extern char save[];

        max = 0;
        while ((len = getline()) > 0)
                if (len > max) {
                        max = len;
                        copy();
                }
        if (max > 0)    /* there was a line */
                printf("%s", save);
}

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

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

function can use external variable, However, cannot to be done so if declared at main.

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

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

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

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

If the program is on several source files, and a variable is defined in, say, *file1* and used in *file2*, then an `extern` declaration is needed in *file2* to connect the two occurrences of the variable. This topic is discussed at length in Chapter 4.

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

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

Exercise 1-18. The test in the `for` statement of `getline` above is rather ungainly. Rewrite the program to make it clearer, but retain the same behavior at end of file or buffer overflow. Is this behavior the most reasonable? □

1.11 Summary

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

After you have this much of C under control, it will be well worth your effort to read on, for the features covered in the next few chapters are where the power and expressiveness of the language begin to become apparent.

Exercise 1-19. Write a program `detab` which replaces tabs in the input with the proper number of blanks to space to the next tab stop. Assume a fixed set of tab stops, say every *n* positions. □

Exercise 1-20. Write the program `entab` which replaces strings of blanks by the minimum number of tabs and blanks to achieve the same spacing. Use the same tab stops as for `detab`. □

Exercise 1-21. Write a program to "fold" long input lines after the last non-blank character that occurs before the *n*-th column of input, where *n* is a parameter. Make sure your program does something intelligent with very long lines, and if there are no blanks or tabs before the specified column. □

Exercise 1-22. Write a program to remove all comments from a C program. Don't forget to handle quoted strings and character constants properly. □

Exercise 1-23. Write a program to check a C program for rudimentary syntax errors like unbalanced parentheses, brackets and braces. Don't forget about quotes, both single and double, and comments. (This program is hard if you do it in full generality.) □

CHAPTER 2: TYPES, OPERATORS AND EXPRESSIONS

Variables and constants are the basic data objects manipulated in a program. Declarations list the variables to be used, and state 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.

2.1 Variable Names

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

Only the first eight characters of an internal name are significant, although more may be used. For external names such as function names and external variables, the number may be less than eight, because external names are used by various assemblers and loaders. Appendix A lists details. Furthermore, keywords like `if`, `else`, `int`, `float`, etc., are *reserved*: 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 purpose of the variable, and that are unlikely to get mixed up typographically.

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` numbers obey the laws of arithmetic modulo 2^n, where n is the number of bits in an `int`; `unsigned` numbers are always positive. 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
		ASCII	ASCII	EBCDIC	ASCII
1	`char`	8 bits	9 bits	8 bits	8 bits
2	`int`	16	36	32	32
2	`short`	16	36	16	16
4	`long`	32	36	32	32
4	`float`	32	36	32	32
4	`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

`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`. 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) on an `int` constant implies octal; a leading `0x` or `0X` indicates hexadecimal. For example, decimal 31 can be written as `037` in octal and `0x1f` or `0X1F` in hex. Hexadecimal and octal constants may also be followed by `L` to make them `long`.

A *character constant* is a single character written within single quotes, as in `'x'`. The value of a character constant is the numeric value of the character in the machine's character set. For example, in the ASCII character set the character zero, or `'0'`, is 48, and in EBCDIC `'0'` is 240, both quite different from the numeric value `0`. Writing `'0'` instead of a numeric value like 48 or 240 makes the program independent of the particular value. Character constants participate in numeric operations just as any other numbers, although they are most often used in comparisons with other characters. A later section treats conversion rules.

Certain non-graphic characters can be represented in character constants by escape sequences like `\n` (newline), `\t` (tab), `\0` (null), `\\` (backslash), `\'` (single quote), etc., which look like two characters, but are actually only one. In addition, an arbitrary byte-sized bit pattern can be generated by writing

```
'\ddd'
```

where *ddd* is one to three octal digits, as in

```
#define  FORMFEED  '\014'   /* ASCII form feed */
```

The character constant `'\0'` represents the character with value zero. `'\0'` is often written instead of `0` to emphasize the character nature of some expression.

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

```
#define  MAXLINE  1000
char line[MAXLINE+1];
```

or

```
seconds = 60 * 60 * hours;
```

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; `\"` represents the double quote character.

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 the 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 one more location than the number of characters written between the quotes. The following function `strlen(s)` returns the length of a character string `s`, 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: `'x'` is not the same as `"x"`. The former is a single character, used to produce the numeric value of the letter *x* in the machine's character set. The latter is a character string that contains one character (the letter *x*) and a `\0`.

2.4 Declarations

All variables must be declared before use, although certain declarations can be made implicitly by context. A declaration specifies a type, 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