```
int power();
```

No parameter list was permitted, so the compiler could not readily check that `power` was being called correctly. Indeed, since by default `power` would have been assumed to return an `int`, the entire declaration might well have been omitted.

The new syntax of function prototypes makes it much easier for a compiler to detect errors in the number of arguments or their types. The old style of declaration and definition still works in ANSI C, at least for a transition period, but we strongly recommend that you use the new form when you have a compiler that supports it.

Exercise 1-15. Rewrite the temperature conversion program of Section 1.2 to use a function for conversion. □

1.8 Arguments—Call by Value

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

The main distinction is that in C the called function cannot directly 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 parameters can be treated as conveniently initialized local variables in the called routine. For example, here is a version of `power` that makes use of this property.

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

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

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

When necessary, it is possible to arrange for a function to modify a variable

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

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

1.9 Character Arrays

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

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

This outline makes it clear that the program divides naturally into pieces. One piece gets a new line, another 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. We will try to make the function useful in other contexts. At the minimum, `getline` has to return a signal about possible end of file; a more useful design would be to return the length of the line, or zero if end of file is encountered. Zero is an acceptable end-of-file return because it is never a valid line length. Every text line has at least one character; even a line containing only a newline has length 1.

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

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

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

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

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

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

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

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

/* copy:  copy 'from' into 'to'; assume to is big enough */
void copy(char to[], char from[])
{
    int i;

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

The functions `getline` and `copy` are declared at the beginning of the program, which we assume is contained in one file.

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

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

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

Some functions return a useful value; others, like `copy`, are used only for their effect and return no value. The return type of `copy` is `void`, which states explicitly that no value is returned.

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

```
"hello\n"
```

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

h	e	l	l	o	\n	\0

The `%s` format specification in `printf` expects the corresponding argument to be a string represented in this form. `copy` also relies on the fact that its input argument is terminated by `'\0'`, and it copies this character into the output argument. (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 safely, in that it stops collecting when the array is full, even if no newline has been seen. By testing the length and the last character returned, `main` can determine whether the line was too long, and then cope as it wishes. In the interests of brevity, we have ignored 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 1-16. 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 1-17. Write a program to print all input lines that are longer than 80 characters. □

Exercise 1-18. Write a program to remove trailing blanks and tabs from each line of input, and to delete entirely blank lines. □

Exercise 1-19. Write a function `reverse(s)` that reverses the character string `s`. Use it to write a program that reverses its input a line at a time. □

1.10 External Variables and Scope

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

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

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

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

```
#include <stdio.h>

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

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

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



/* print longest input line; specialized version */
main()
{
    int len;
    extern int max;
    extern char longest[];

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



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

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

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

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

The external variables in `main`, `getline`, and `copy` are defined by the first lines of the example above, which state their type and cause storage to be allocated for them. Syntactically, external definitions are just like definitions of local variables, but since they occur outside of functions, the variables are external. Before a function can use an external variable, the name of the variable must be made known to the function. 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 definition of an external variable occurs in the source file before its use in a particular function, then there is no need for an `extern` declaration in the function. The `extern` declarations in `main`, `getline` and `copy` are thus redundant. In fact, common practice is to place definitions of all external variables at the beginning of the source file, and then omit all `extern` declarations.

If the program is in several source files, and a variable is defined in *file1* and used in *file2* and *file3*, then `extern` declarations are needed in *file2* and *file3* to connect the occurrences of the variable. The usual practice is to collect `extern` declarations of variables and functions in a separate file, historically called a *header*, that is included by `#include` at the front of each source file. The suffix `.h` is conventional for header names. The functions of the standard library, for example, are declared in headers like `<stdio.h>`. This topic is discussed at length in Chapter 4, and the library itself in Chapter 7 and Appendix B.

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

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

By the way, there is a tendency to make everything in sight an `extern` variable because it appears to simplify communications—argument lists are short

and variables are always there when you want them. But external variables are always there even when you don't want them. Relying too heavily on external variables is fraught with peril since it leads to programs whose data connections are not at all obvious—variables can be changed in unexpected and even inadvertent ways, and the program is hard to modify. The second version of the longest-line program is inferior to the first, partly for these reasons, and partly because it destroys the generality of two useful functions by wiring into them the names of the variables they manipulate.

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

Exercise 1-20. Write a program `detab` that replaces tabs in the input with the proper number of blanks to space to the next tab stop. Assume a fixed set of tab stops, say every *n* columns. Should *n* be a variable or a symbolic parameter? □

Exercise 1-21. Write a program `entab` that replaces strings of blanks by the minimum number of tabs and blanks to achieve the same spacing. Use the same tab stops as for `detab`. When either a tab or a single blank would suffice to reach a tab stop, which should be given preference? □

Exercise 1-22. Write a program to "fold" long input lines into two or more shorter lines after the last non-blank character that occurs before the *n*-th column of input. Make sure your program does something intelligent with very long lines, and if there are no blanks or tabs before the specified column. □

Exercise 1-23. Write a program to remove all comments from a C program. Don't forget to handle quoted strings and character constants properly. C comments do not nest. □

Exercise 1-24. 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, escape sequences, 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. The type of an object determines the set of values it can have and what operations can be performed on it. These building blocks are the topics of this chapter.

The ANSI standard has made many small changes and additions to basic types and expressions. There are now `signed` and `unsigned` forms of all integer types, and notations for unsigned constants and hexadecimal character constants. Floating-point operations may be done in single precision; there is also a `long double` type for extended precision. String constants may be concatenated at compile time. Enumerations have become part of the language, formalizing a feature of long standing. Objects may be declared `const`, which prevents them from being changed. The rules for automatic coercions among arithmetic types have been augmented to handle the richer set of types.

2.1 Variable Names

Although we didn't say so in Chapter 1, there are some restrictions on the names of variables and symbolic constants. Names are made up of letters and digits; the first character must be a letter. The underscore "_" counts as a letter; it is sometimes useful for improving the readability of long variable names. Don't begin variable names with underscore, however, since library routines often use such names. Upper case and lower case letters are distinct, so `x` and `X` are two different names. Traditional C practice is to use lower case for variable names, and all upper case for symbolic constants.

At least the first 31 characters of an internal name are significant. For function names and external variables, the number may be less than 31, because external names may be used by assemblers and loaders over which the language has no control. For external names, the standard guarantees uniqueness only for 6 characters and a single case. Keywords like `if`, `else`, `int`, `float`, etc.,

are reserved: you can't use them as variable names. They must be in lower case.

It's wise to choose variable names that are related to the purpose of the variable, and that are unlikely to get mixed up typographically. We tend to use short names for local variables, especially loop indices, and longer names for external variables.

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 that can be applied to these basic types. `short` and `long` apply to integers:

```
short int sh;
long int counter;
```

The word `int` can be omitted in such declarations, and typically is.

The intent is that `short` and `long` should provide different lengths of integers where practical; `int` will normally be the natural size for a particular machine. `short` is often 16 bits, `long` 32 bits, and `int` either 16 or 32 bits. Each compiler is free to choose appropriate sizes for its own hardware, subject only to the restriction that `shorts` and `ints` are at least 16 bits, `longs` are at least 32 bits, and `short` is no longer than `int`, which is no longer than `long`.

The qualifier `signed` or `unsigned` may be applied to `char` or any integer. `unsigned` numbers are always positive or zero, and obey the laws of arithmetic modulo 2^n, where n is the number of bits in the type. So, for instance, if `chars` are 8 bits, `unsigned char` variables have values between 0 and 255, while `signed chars` have values between −128 and 127 (in a two's complement machine). Whether plain `chars` are signed or unsigned is machine-dependent, but printable characters are always positive.

The type `long double` specifies extended-precision floating point. As with integers, the sizes of floating-point objects are implementation-defined; `float`, `double` and `long double` could represent one, two or three distinct sizes.

The standard headers `<limits.h>` and `<float.h>` contain symbolic constants for all of these sizes, along with other properties of the machine and compiler. These are discussed in Appendix B.

Exercise 2-1. Write a program to determine the ranges of `char`, `short`, `int`,

and `long` variables, both `signed` and `unsigned`, by printing appropriate values from standard headers and by direct computation. Harder if you compute them: determine the ranges of the various floating-point types. □

2.3 Constants

An integer constant like `1234` is an `int`. A `long` constant is written with a terminal `l` (ell) or `L`, as in `123456789L`; an integer too big to fit into an `int` will also be taken as a `long`. Unsigned constants are written with a terminal `u` or `U`, and the suffix `ul` or `UL` indicates `unsigned long`.

Floating-point constants contain a decimal point (`123.4`) or an exponent (`1e-2`) or both; their type is `double`, unless suffixed. The suffixes `f` or `F` indicate a `float` constant; `l` or `L` indicate a `long double`.

The value of an integer can be specified in octal or hexadecimal instead of decimal. A leading `0` (zero) on an integer constant means octal; a leading `0x` or `0X` means hexadecimal. For example, decimal 31 can be written as `037` in octal and `0x1f` or `0X1F` in hex. Octal and hexadecimal constants may also be followed by `L` to make them `long` and `U` to make them `unsigned`: `0XFUL` is an `unsigned long` constant with value 15 decimal.

A *character constant* is an integer, written as one character within single quotes, such as `'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 constant `'0'` has the value 48, which is unrelated to the numeric value 0. If we write `'0'` instead of a numeric value like 48 that depends on character set, the program is independent of the particular value and easier to read. Character constants participate in numeric operations just as any other integers, although they are most often used in comparisons with other characters.

Certain characters can be represented in character and string constants by escape sequences like `\n` (newline); these sequences look like two characters, but represent only one. In addition, an arbitrary byte-sized bit pattern can be specified by

```
'\ooo'
```

where *ooo* is one to three octal digits (0...7) or by

```
'\xhh'
```

where *hh* is one or more hexadecimal digits (`0...9`, `a...f`, `A...F`). So we might write

```
#define VTAB '\013'     /* ASCII vertical tab */
#define BELL '\007'     /* ASCII bell character */
```

or, in hexadecimal,

```
#define VTAB '\xb'      /* ASCII vertical tab */
#define BELL '\x7'      /* ASCII bell character */
```

The complete set of escape sequences is

`\a`	alert (bell) character	`\\`	backslash
`\b`	backspace	`\?`	question mark
`\f`	formfeed	`\'`	single quote
`\n`	newline	`\"`	double quote
`\r`	carriage return	`\`*ooo*	octal number
`\t`	horizontal tab	`\x`*hh*	hexadecimal number
`\v`	vertical tab		

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

A *constant expression* is an expression that involves only constants. Such expressions may be evaluated during compilation rather than run-time, and accordingly may be used in any place that a constant can occur, as in

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

or

```
#define LEAP 1    /* in leap years */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

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

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

or

```
""  /* the empty string */
```

The quotes are not part of the string, but serve only to delimit it. The same escape sequences used in character constants apply in strings; `\"` represents the double-quote character. String constants can be concatenated at compile time:

```
"hello," " world"
```

is equivalent to

```
"hello, world"
```

This is useful for splitting long strings across several source lines.

Technically, a string constant is an array of characters. The internal representation of a string has a null character `'\0'` at the end, so the physical storage required is one more than the number of characters written between the quotes. This representation means that there is no limit to how long a string can be, but programs must scan a string completely to determine its length. The standard library function `strlen(s)` returns the length of its character

string argument s, excluding the terminal '\0'. Here is our version:

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

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

strlen and other string functions are declared in the standard header <string.h>.

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 an integer, used to produce the numeric value of the letter *x* in the machine's character set. The latter is an array of characters that contains one character (the letter *x*) and a '\0'.

There is one other kind of constant, the *enumeration constant*. An enumeration is a list of constant integer values, as in

```
enum boolean { NO, YES };
```

The first name in an enum has value 0, the next 1, and so on, unless explicit values are specified. If not all values are specified, unspecified values continue the progression from the last specified value, as in the second of these examples:

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };
                    /* FEB is 2, MAR is 3, etc. */
```

Names in different enumerations must be distinct. Values need not be distinct in the same enumeration.

Enumerations provide a convenient way to associate constant values with names, an alternative to #define with the advantage that the values can be generated for you. Although variables of enum types may be declared, compilers need not check that what you store in such a variable is a valid value for the enumeration. Nevertheless, enumeration variables offer the chance of checking and so are often better than #defines. In addition, a debugger may be able to print values of enumeration variables in their symbolic form.

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 contains 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 space, but is convenient for adding a comment to each declaration or for subsequent modifications.

A variable may also be initialized in its declaration. If the name is followed by an equals sign and an expression, the expression serves as an initializer, as in

```
char  esc = '\\';
int   i = 0;
int   limit = MAXLINE+1;
float eps = 1.0e-5;
```

If the variable in question is not automatic, the initialization is done once only, conceptually before the program starts executing, and the initializer must be a constant expression. An explicitly initialized automatic variable is initialized each time the function or block it is in is entered; the initializer may be any expression. External and static variables are initialized to zero by default. Automatic variables for which there is no explicit initializer have undefined (i.e., garbage) values.

The qualifier `const` can be applied to the declaration of any variable to specify that its value will not be changed. For an array, the `const` qualifier says that the elements will not be altered.

```
const double e = 2.71828182845905;
const char msg[] = "warning: ";
```

The `const` declaration can also be used with array arguments, to indicate that the function does not change that array:

```
int strlen(const char[]);
```

The result is implementation-defined if an attempt is made to change a `const`.

2.5 Arithmetic Operators

The binary arithmetic operators are +, -, *, /, and the modulus operator %. Integer division truncates any fractional part. The expression

```
x % y
```

produces the remainder when x is divided by y, and thus is zero when y divides x 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 % 4 == 0 && year % 100 != 0) || year % 400 == 0)
    printf("%d is a leap year\n", year);
else
    printf("%d is not a leap year\n", year);
```

The % operator cannot be applied to `float` or `double`. The direction of truncation for / and the sign of the result for % are machine-dependent for negative operands, as is the action taken on overflow or underflow.

The binary + and - operators have the same precedence, which is lower than the precedence of *, /, and %, which is in turn lower than unary + and -. Arithmetic operators associate left to right.

Table 2-1 at the end of this chapter summarizes precedence and associativity for all operators.

2.6 Relational and Logical Operators

The relational operators are

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

They all have the same precedence. Just below them in precedence are the equality operators:

```
==   !=
```

Relational operators have lower precedence than arithmetic operators, so an expression like `i < lim-1` is 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. Most C programs rely on these properties. For example, here is a loop from the input function `getline` that we wrote in Chapter 1:

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

Before reading a new character it is necessary to check that there is room to store it in the array s, so the test `i < lim-1` *must* be made first. Moreover, 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` is called; therefore the call and assignment must occur before the character in `c` is tested.

The precedence of `&&` is higher than that of `||`, and both are lower than relational and equality operators, so expressions like

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

need no extra parentheses. But since the precedence of `!=` is higher than assignment, parentheses are needed in

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

to achieve the desired result of assignment to `c` and then comparison with `'\n'`.

By definition, the numeric value of a relational or logical expression is 1 if the relation is true, and 0 if the relation is false.

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

```
if (!valid)
```

rather than

```
if (valid == 0)
```

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

Exercise 2-2. Write a loop equivalent to the `for` loop above without using `&&` or `||`. □

2.7 Type Conversions

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a "narrower" operand into a "wider" one without losing information, such as converting an integer to floating point in an expression like `f + i`. Expressions that don't make sense, like using a `float` as a subscript, are disallowed. Expressions that might lose information, like assigning a longer integer type to a shorter, or a floating-point type to an integer, may draw a warning, but they are not illegal.

A `char` is just a small integer, so `char`s may be freely used in arithmetic expressions. This permits considerable flexibility in certain kinds of character transformations. One is exemplified by this naive implementation of the function `atoi`, which converts a string of digits into its numeric equivalent.

```
/* atoi:  convert s to integer */
int atoi(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;
}
```

As we discussed in Chapter 1, the expression

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

gives the numeric value of the character stored in `s[i]`, because the values of `'0'`, `'1'`, etc., form a contiguous increasing sequence.

Another example of `char` to `int` conversion is the function `lower`, which maps a single character to lower case *for the ASCII character set.* If the character is not an upper case letter, `lower` returns it unchanged.

```
/* lower:  convert c to lower case; ASCII only */
int lower(int 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 each alphabet is contiguous—there is nothing but letters between `A` and `Z`. This latter observation is not true of the EBCDIC character set, however, so this code would convert more than just letters in EBCDIC.

The standard header `<ctype.h>`, described in Appendix B, defines a family of functions that provide tests and conversions that are independent of character set. For example, the function `tolower(c)` returns the lower case value of `c` if `c` is upper case, so `tolower` is a portable replacement for the function `lower` shown above. Similarly, the test

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

can be replaced by

```
isdigit(c)
```

We will use the `<ctype.h>` functions from now on.

There is one subtle point about the conversion of characters to integers. The language does not specify whether variables of type `char` are signed or unsigned quantities. When a `char` is converted to an `int`, can it ever produce a negative integer? The answer varies from machine to machine, reflecting

differences in architecture. On some machines a `char` whose leftmost bit is 1 will be converted to a negative integer ("sign extension"). On others, a `char` is promoted to an `int` by adding zeros at the left end, and thus is always positive.

The definition of C guarantees that any character in the machine's standard printing character set will never be negative, so these characters will always be positive quantities in expressions. But arbitrary bit patterns stored in character variables may appear to be negative on some machines, yet positive on others. For portability, specify `signed` or `unsigned` if non-character data is to be stored in `char` variables.

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

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

sets `d` to 1 if `c` is a digit, and 0 if not. However, functions like `isdigit` may return any non-zero value for true. In the test part of `if`, `while`, `for`, etc., "true" just means "non-zero," so this makes no difference.

Implicit arithmetic conversions work much as expected. In general, if an operator like + or * that 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. Section 6 of Appendix A states the conversion rules precisely. If there are no `unsigned` operands, however, the following informal set of rules will suffice:

If either operand is `long double`, convert the other to `long double`.

Otherwise, if either operand is `double`, convert the other to `double`.

Otherwise, if either operand is `float`, convert the other to `float`.

Otherwise, convert `char` and `short` to `int`.

Then, if either operand is `long`, convert the other to `long`.

Notice that `float`s in an expression are not automatically converted to `double`; this is a change from the original definition. In general, mathematical functions like those in `<math.h>` will use double precision. The main reason for using `float` is to save storage in large arrays, or, less often, to save time on machines where double-precision arithmetic is particularly expensive.

Conversion rules are more complicated when `unsigned` operands are involved. The problem is that comparisons between signed and unsigned values are machine-dependent, because they depend on the sizes of the various integer types. For example, suppose that `int` is 16 bits and `long` is 32 bits. Then `-1L < 1U`, because `1U`, which is an `int`, is promoted to a `signed long`. But `-1L > 1UL`, because `-1L` is promoted to `unsigned long` and thus appears to be a large positive number.

Conversions take place across assignments; the value of the right side is converted to the type of the left, which is the type of the result.

A character is converted to an integer, either by sign extension or not, as described above.

Longer integers are converted to shorter ones or to `char`s by dropping the excess high-order bits. Thus in

```
int  i;
char c;

i = c;
c = i;
```

the value of `c` is unchanged. This is true whether or not sign extension is involved. Reversing the order of assignments might lose information, however.

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. When `double` is converted to `float`, whether the value is rounded or truncated is implementation-dependent.

Since an argument of a function call is an expression, type conversions also take place when arguments are passed to functions. In the absence of a function prototype, `char` and `short` become `int`, and `float` becomes `double`. This is why we have declared function arguments to be `int` and `double` even when the function is called with `char` and `float`.

Finally, explicit type conversions can be forced ("coerced") in any expression, with a unary operator called a *cast*. In the construction

(type-name) expression

the *expression* is converted to the named type by the conversion rules above. The precise meaning of a cast is as if the *expression* were assigned to a variable of the specified type, which is then used in place of the whole construction. For example, the library routine `sqrt` expects a `double` argument, and will produce nonsense if inadvertently handed something else. (`sqrt` is declared in `<math.h>`.) So if `n` is an integer, we can use

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

to convert the value of `n` to `double` before passing it to `sqrt`. Note that the cast produces the *value* of `n` in the proper type; `n` itself is not altered. The cast operator has the same high precedence as other unary operators, as summarized in the table at the end of this chapter.

If arguments are declared by a function prototype, as they normally should be, the declaration causes automatic coercion of any arguments when the function is called. Thus, given a function prototype for `sqrt`:

```
double sqrt(double);
```

the call

```
root2 = sqrt(2);
```

coerces the integer 2 into the `double` value `2.0` without any need for a cast.

The standard library includes a portable implementation of a pseudo-random number generator and a function for initializing the seed; the former illustrates a cast:

```
unsigned long int next = 1;

/* rand:  return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand:  set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Exercise 2-3. Write the function `htoi(s)`, which converts a string of hexadecimal digits (including an optional `0x` or `0X`) into its equivalent integer value. The allowable digits are `0` through `9`, `a` through `f`, and `A` through `F`. □

2.8 Increment and Decrement Operators

C provides two unusual operators for incrementing and decrementing variables. The increment operator `++` adds 1 to its operand, while 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 used either as prefix operators (before the variable, as in `++n`), or postfix (after the variable: `n++`). In both cases, the effect is to increment `n`. But the expression `++n` increments `n` *before* its value is used, while `n++` increments `n` *after* its value has been used. This means that in a context where the value is being used, not just the effect, `++n` and `n++` are different. If `n` is 5, then

```
x = n++;
```

sets `x` to 5, but

```
x = ++n;
```

sets `x` to 6. In both cases, `n` becomes 6. The increment and decrement operators can only be applied to variables; an expression like `(i+j)++` is illegal.

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

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

prefix and postfix are the same. 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:  delete all c from s */
void squeeze(char s[], int 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. This is exactly equivalent to

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

Another example of a similar construction comes from the `getline` function that we wrote in Chapter 1, where we can replace

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

by the more compact

```
if (c == '\n')
    s[i++] = c;
```

As a third example, consider the standard function `strcat(s,t)`, which concatenates the string `t` to the end of the string `s`. `strcat` assumes that there is enough space in `s` to hold the combination. As we have written it, `strcat` returns no value; the standard library version returns a pointer to the resulting string.

```
/* strcat:  concatenate t to end of s; s must be big enough */
void strcat(char s[], char t[])
{
    int i, j;

    i = j = 0;
    while (s[i] != '\0')    /* find end of s */
        i++;
    while ((s[i++] = t[j++]) != '\0')   /* copy t */
        ;
}
```

As each character is copied from `t` to `s`, the postfix `++` is applied to both `i` and `j` to make sure that they are in position for the next pass through the loop.

Exercise 2-4. Write an alternate version of `squeeze(s1,s2)` that deletes each character in `s1` that matches any character in the *string* `s2`. □

Exercise 2-5. Write the function `any(s1,s2)`, which returns the first location in the string `s1` where any character from the string `s2` occurs, or `-1` if `s1` contains no characters from `s2`. (The standard library function `strpbrk` does the same job but returns a pointer to the location.) □

2.9 Bitwise Operators

C provides six operators for bit manipulation; these may only be applied to integral operands, that is, `char`, `short`, `int`, and `long`, whether signed or unsigned.

`&`	bitwise AND
`|`	bitwise inclusive OR
`^`	bitwise exclusive OR
`<<`	left shift
`>>`	right shift
`~`	one's complement (unary)

The bitwise AND operator `&` is often used to mask off some set of bits; for example,

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

sets to zero all but the low-order 7 bits of `n`.

The bitwise OR operator `|` is used to turn bits on:

```
x = x | SET_ON;
```

sets to one in `x` the bits that are set to one in `SET_ON`.

The bitwise exclusive OR operator `^` sets a one in each bit position where its operands have different bits, and zero where they are the same.

One must distinguish the bitwise operators & and | from the logical operators && 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, which must be positive. Thus x << 2 shifts the value of x left by two positions, filling vacated bits with zero; this is equivalent to multiplication by 4. Right shifting an unsigned quantity always fills vacated bits with zero. Right shifting a signed quantity will fill with sign bits ("arithmetic shift") on some machines and with 0-bits ("logical shift") on others.

The unary operator ~ yields the one's complement of an integer; that is, it converts each 1-bit into a 0-bit and vice versa. For example,

```
x = x & ~077
```

sets the last six bits of x to zero. Note that x & ~077 is independent of word length, and is thus preferable to, for example, x & 0177700, which assumes that x is a 16-bit quantity. The portable form involves no extra cost, since ~077 is a constant expression that can be evaluated at compile time.

As an illustration of some of the bit operators, consider the function getbits(x,p,n) that returns the (right adjusted) n-bit field of x that begins at position p. We assume that bit position 0 is at the right end and that n and p are sensible positive values. For example, getbits(x,4,3) returns the three bits in bit positions 4, 3 and 2, right adjusted.

```
/* getbits:  get n bits from position p */
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

The expression x >> (p+1-n) moves the desired field to the right end of the word. ~0 is all 1-bits; shifting it left n bit positions with ~0<<n places zeros in the rightmost n bits; complementing that with ~ makes a mask with ones in the rightmost n bits.

Exercise 2-6. Write a function setbits(x,p,n,y) that returns x with the n bits that begin at position p set to the rightmost n bits of y, leaving the other bits unchanged. □

Exercise 2-7. Write a function invert(x,p,n) that returns x with the n bits that begin at position p inverted (i.e., 1 changed into 0 and vice versa), leaving the others unchanged. □

Exercise 2-8. Write a function rightrot(x,n) that returns the value of the integer x rotated to the right by n bit positions. □

2.10 Assignment Operators and Expressions

Expressions such as

```
i = i + 2
```

in which the variable on the left hand side is repeated immediately on the right, can be written in the compressed form

```
i += 2
```

The operator += is called an *assignment operator.*

Most binary operators (operators like + that have a left and right operand) have a corresponding assignment operator *op*=, where *op* is one of

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

If $expr_1$ and $expr_2$ are expressions, then

$expr_1$ *op*= $expr_2$

is equivalent to

$expr_1$ = ($expr_1$) *op* ($expr_2$)

except that $expr_1$ is computed only once. Notice the parentheses around $expr_2$:

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

means

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

rather than

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

As an example, the function `bitcount` counts the number of 1-bits in its integer argument.

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

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

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

Quite apart from conciseness, assignment operators have the advantage that they correspond better to the way people think. We say "add 2 to `i`" or

"increment `i` by 2," not "take `i`, add 2, then put the result back in `i`." Thus the expression `i += 2` is preferable to `i = i+2`. In addition, for a complicated expression like

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

the assignment operator makes the code easier to understand, since the reader doesn't have to check painstakingly that two long expressions are indeed the same, or to wonder why they're not. And an assignment operator may even help a compiler to produce efficient code.

We have already seen 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 occur in expressions, although this is less frequent.

In all such expressions, the type of an assignment expression is the type of its left operand, and the value is the value after the assignment.

Exercise 2-9. In a two's complement number system, `x &= (x-1)` deletes the rightmost 1-bit in `x`. Explain why. Use this observation to write a faster version of `bitcount`. □

2.11 Conditional Expressions

The statements

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

compute in `z` the maximum of `a` and `b`. The *conditional expression*, written with the ternary operator "`?:`", provides an alternate way to write this and similar constructions. In the expression

$$expr_1 \ ? \ expr_2 \ : \ expr_3$$

the expression $expr_1$ is evaluated first. If it is non-zero (true), then the expression $expr_2$ is evaluated, and that is the value of the conditional expression. Otherwise $expr_3$ is evaluated, and that is the value. Only one of $expr_2$ and $expr_3$ is evaluated. Thus to set `z` to the maximum of `a` and `b`,

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

It should be noted that the conditional expression is indeed an expression, and it can be used wherever any other expression can be. If $expr_2$ and $expr_3$

are of different types, the type of the result is determined by the conversion rules discussed earlier in this 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.

Parentheses are not necessary around the first expression of a conditional expression, since the precedence of `?:` is very low, just above assignment. 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, with each column separated by one blank, and with each line (including the last) terminated by a newline.

```
for (i = 0; i < n; i++)
   printf("%6d%c", a[i], (i%10==9 || i==n-1) ? '\n' : ' ');
```

A newline is printed after every tenth element, and after the `n`-th. All other elements are followed by one blank. This might look tricky, but it's more compact than the equivalent `if-else`. Another good example is

```
printf("You have %d item%s.\n", n, n==1 ? "" : "s");
```

Exercise 2-10. Rewrite the function `lower`, which converts upper case letters to lower case, with a conditional expression instead of `if-else`. □

2.12 Precedence and Order of Evaluation

Table 2-1 summarizes the rules for precedence and associativity of all operators, including those that we have not yet discussed. Operators on the same line have the same precedence; rows are in order of decreasing precedence, so, for example, `*`, `/`, and `%` all have the same precedence, which is higher than that of binary `+` and `-`. The "operator" `( )` refers to function call. The operators `->` and `.` are used to access members of structures; they will be covered in Chapter 6, along with `sizeof` (size of an object). Chapter 5 discusses `*` (indirection through a pointer) and `&` (address of an object), and Chapter 3 discusses the comma operator.

Note that the precedence of the bitwise operators `&`, `^`, and `|` falls below `==` and `!=`. This implies that bit-testing expressions like

```
if ((x & MASK) == 0) ...
```

must be fully parenthesized to give proper results.

C, like most languages, does not specify the order in which the operands of an operator are evaluated. (The exceptions are `&&`, `||`, `?:`, and `','`.) For example, in a statement like

TABLE 2-1. PRECEDENCE AND ASSOCIATIVITY OF OPERATORS

OPERATORS	ASSOCIATIVITY
`() [] -> .`	left to right
`! ~ ++ -- + - * & (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

Unary +, -, and * have higher precedence than the binary forms.

```
x = f() + g();
```

f may be evaluated before g or vice versa; thus if either f or g alters a variable on which the other depends, x can depend on the order of evaluation. Intermediate results can be stored in temporary variables to ensure a particular sequence.

Similarly, the order in which function arguments are evaluated is not specified, so the statement

```
printf("%d %d\n", ++n, power(2, n));     /* WRONG */
```

can produce different results with different compilers, depending on whether n is incremented before power is called. The solution, of course, is to write

```
++n;
printf("%d %d\n", n, power(2, n));
```

Function calls, nested assignment statements, and increment and decrement operators 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 variables taking part in the expression are updated. One unhappy situation is typified by the statement

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

The question is whether the subscript is the old value of i or the new.

Compilers can interpret this in different ways, and generate different answers depending on their interpretation. The standard intentionally leaves most such matters unspecified. When side effects (assignment to variables) take place within an expression is left to the discretion of the compiler, since the best order depends strongly on machine architecture. (The standard does specify that all side effects on arguments take effect before a function is called, but that would not help in the call to `printf` above.)

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

CHAPTER 3: Control Flow

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

3.1 Statements and Blocks

An expression such as `x = 0` or `i++` or `printf(...)` becomes a *statement* when it is followed by a semicolon, as in

```
x = 0;
i++;
printf(...);
```

In C, the semicolon is a statement terminator, rather than a separator as it is in languages like Pascal.

Braces { and } are used to group declarations and statements together into a *compound statement*, or *block*, 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`, `else`, `while`, or `for` are another. (Variables can be declared inside *any* block; we will talk about this in Chapter 4.) There is no semicolon after the right brace that ends a block.

3.2 If-Else

The `if-else` statement is used to express decisions. Formally, the syntax is

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

where the `else` part is optional. The *expression* is evaluated; if it is true (that is, if *expression* has a non-zero value), *statement*$_1$ is executed. If it is false (*expression* is zero) and if there is an `else` part, *statement*$_2$ is executed instead.

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 can be cryptic.

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 by associating the `else` with the closest previous `else`-less `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 this:

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

The indentation shows unequivocally what you want, but the compiler doesn't get the message, and associates the `else` with the inner `if`. This kind of bug can be hard to find; it's a good idea to use braces when there are nested `if`s.

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 an expression statement like "`z = a;`" is always terminated by a semicolon.

3.3 Else-If

The construction

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

occurs so often that it is worth a brief separate discussion. This sequence of `if` statements is the most general way of writing a multi-way decision. The *expressions* are evaluated in order; if any *expression* is true, the *statement* associated with it is executed, and this terminates the whole chain. As always, 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 is satisfied. Sometimes there is no explicit action for the default; in that case the trailing

```
else
    statement
```

can be omitted, or it may be used for error checking to catch an "impossible" condition.

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`. The elements of `v` must be in increasing order. The function returns the position (a number between 0 and `n-1`) if `x` occurs in `v`, and `-1` if not.

Binary search first compares the input value `x` to the middle element of the array `v`. If `x` is less than the middle value, searching focuses on the lower half of the table, otherwise on the upper half. In either case, the next step is to compare `x` to the middle element of the selected half. This process of dividing the range in two continues until the value is found or the range is empty.

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

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
        if (x < v[mid])
            high = mid - 1;
        else if (x > v[mid])
            low = mid + 1;
        else    /* found match */
            return mid;
    }
    return -1;  /* no match */
}
```

The fundamental decision is whether `x` is less than, greater than, or equal to the middle element `v[mid]` at each step; this is a natural for `else-if`.

Exercise 3-1. Our binary search makes two tests inside the loop, when one would suffice (at the price of more tests outside). Write a version with only one test inside the loop and measure the difference in run-time. □

3.4 Switch

The `switch` statement is a multi-way decision that tests whether an expression matches one of a number of *constant* integer values, and branches accordingly.

```
switch (expression) {
    case const-expr:  statements
    case const-expr:  statements
    default:  statements
}
```

Each case is labeled by one or more integer-valued constants or constant expressions. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labeled `default` is executed if none of the other cases are satisfied. A `default` is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order.

In Chapter 1 we wrote a program to count the occurrences of each digit, white space, and all other characters, using a sequence of `if ... else if ... else`. Here is the same program with a `switch`:

```
#include <stdio.h>

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(", white space = %d, other = %d\n",
        nwhite, nother);
    return 0;
}
```

The `break` statement causes an immediate exit from the `switch`. Because cases serve just as 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`. A `break` statement can also be used to force an immediate exit from `while`, `for`, and `do` loops, as will be discussed later in this chapter.

Falling through cases is a mixed blessing. On the positive side, it allows several cases to be attached to a single action, as with the digits 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, and commented.

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 3-2. Write a function `escape(s,t)` that converts characters like newline and tab into visible escape sequences like `\n` and `\t` as it copies the string `t` to `s`. Use a `switch`. Write a function for the other direction as well, converting escape sequences into the real characters. □

3.5 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 zero, at which point execution resumes after *statement*.

The `for` statement

```
for (expr1; expr2; expr3)
    statement
```

is equivalent to

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

except for the behavior of `continue`, which is described in Section 3.7.

Grammatically, the three components of a `for` loop are expressions. Most commonly, $expr_1$ and $expr_3$ are assignments or function calls and $expr_2$ is a relational expression. Any of the three parts can be omitted, although the semicolons must remain. If $expr_1$ or $expr_3$ is omitted, it is simply dropped from the expansion. If the test, $expr_2$, 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 personal preference. For example, in

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

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

The `for` is preferable when there is a simple initialization and increment, since it keeps the loop control statements close together and visible at the top of

the loop. This is most 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 DO loop or the Pascal `for`. The analogy is not perfect, however, since the index and limit of a C `for` loop can be altered from within the loop, and the index variable `i` retains its value when the loop terminates for any reason. Because the components of the `for` are arbitrary expressions, `for` loops are not restricted to arithmetic progressions. Nonetheless, it is bad style to force unrelated computations into the initialization and increment of a `for`, which are better reserved for loop control operations.

As a larger example, here is another version of `atoi` for converting a string to its numeric equivalent. This one is slightly more general than the one in Chapter 2; 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 structure of the program reflects the form of the input:

skip white space, if any
get sign, if any
get integer part and convert it

Each step does its part, and leaves things in a clean state for the next. The whole process terminates on the first character that could not be part of a number.

```
#include <ctype.h>

/* atoi:  convert s to integer; version 2 */
int atoi(char s[])
{
    int i, n, sign;

    for (i = 0; isspace(s[i]); i++)  /* skip white space */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')  /* skip sign */
        i++;
    for (n = 0; isdigit(s[i]); i++)
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

The standard library provides a more elaborate function `strtol` for conversion of strings to long integers; see Section 5 of Appendix B.

The advantages of keeping loop control centralized are even more obvious when there are several nested loops. The following function is a Shell sort for sorting an array of integers. The basic idea of this sorting algorithm, which was

invented in 1959 by D. L. Shell, is that in early stages, far-apart elements are compared, rather than adjacent ones as in simpler 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.

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

    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) {
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] = temp;
            }
}
```

There are three nested loops. The outermost 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 steps along the elements. The innermost loop compares each pair of elements that is separated by `gap` and reverses any that are out of order. Since `gap` is eventually reduced to one, all elements are eventually ordered correctly. Notice how the generality of the `for` makes the outer loop fit the same form as the others, even though it is not an arithmetic progression.

One final C operator is the comma ",", which most often finds use in the `for` statement. A pair of expressions separated by a comma is evaluated left to right, and the type and value of the result are the type and value of the right operand. Thus in a `for` statement, it is possible to place multiple expressions in the various parts, for example to process two indices in parallel. This is illustrated in the function `reverse(s)`, which reverses the string `s` in place.

```
#include <string.h>

/* reverse:  reverse string s in place */
void reverse(char s[])
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

The commas that separate function arguments, variables in declarations, etc., are *not* comma operators, and do not guarantee left to right evaluation.

Comma operators should be used sparingly. The most suitable uses are for constructs strongly related to each other, as in the `for` loop in `reverse`, and in macros where a multistep computation has to be a single expression. A comma expression might also be appropriate for the exchange of elements in `reverse`, where the exchange can be thought of as a single operation:

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
    c = s[i], s[i] = s[j], s[j] = c;
```

Exercise 3-3. Write a function `expand(s1,s2)` that expands shorthand notations like `a-z` in the string `s1` into the equivalent complete list `abc...xyz` in `s2`. Allow for letters of either case and digits, and be prepared to handle cases like `a-b-c` and `a-z0-9` and `-a-z`. Arrange that a leading or trailing `-` is taken literally. □

3.6 Loops—Do-while

As we discussed in Chapter 1, the `while` and `for` loops test the termination condition at the top. By contrast, the third loop in C, the `do-while`, tests at the bottom *after* making each pass through the loop body; the body is always executed at least once.

The syntax of the `do` is

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

The *statement* is executed, then *expression* is evaluated. If it is true, *statement* is evaluated again, and so on. When the expression becomes false, the loop terminates. Except for the sense of the test, `do-while` is equivalent to the Pascal `repeat-until` statement.

Experience shows that `do-while` is much less used than `while` and `for`. Nonetheless, from time to time it is valuable, as in the following function `itoa`, which converts 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 to generate the string backwards, then reverse it.

```
/* itoa:  convert n to characters in s */
void itoa(int n, char s[])
{
    int i, sign;

    if ((sign = n) < 0)  /* record sign */
        n = -n;          /* make n positive */
    i = 0;
    do {       /* generate digits in reverse order */
        s[i++] = n % 10 + '0';   /* get next digit */
    } while ((n /= 10) > 0);     /* delete it */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

The `do-while` is necessary, or at least convenient, since at least one character must be installed in the array `s`, even if `n` is zero. We also used braces around the single statement that makes up the body of the `do-while`, even though they are unnecessary, so the hasty reader will not mistake the `while` part for the *beginning* of a `while` loop.

Exercise 3-4. In a two's complement number representation, our version of `itoa` does not handle the largest negative number, that is, the value of `n` equal to $-(2^{wordsize-1})$. Explain why not. Modify it to print that value correctly, regardless of the machine on which it runs. □

Exercise 3-5. Write the function `itob(n,s,b)` that converts the integer `n` into a base `b` character representation in the string `s`. In particular, `itob(n,s,16)` formats `n` as a hexadecimal integer in `s`. □

Exercise 3-6. Write a version of `itoa` that accepts three arguments instead of two. The third argument is a minimum field width; the converted number must be padded with blanks on the left if necessary to make it wide enough. □

3.7 Break and Continue

It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom. The `break` statement provides an early exit from `for`, `while`, and `do`, just as from `switch`. A `break` causes the innermost enclosing loop or `switch` to be exited immediately.

The following function, `trim`, removes trailing blanks, tabs, and newlines from the end of a string, using a `break` to exit from a loop when the rightmost non-blank, non-tab, non-newline is found.

```
/* trim:  remove trailing blanks, tabs, newlines */
int trim(char s[])
{
    int n;

    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}
```

`strlen` returns the length of the string. The `for` loop starts at the end and scans backwards looking for the first character that is not a blank or tab or newline. The loop is broken when one is found, or when `n` becomes negative (that is, when the entire string has been scanned). You should verify that this is correct behavior even when the string is empty or contains only white space characters.

The `continue` statement is related to `break`, but less often used; it causes the next iteration of the enclosing `for`, `while`, or `do` loop to begin. In the `while` and `do`, this means that the test part is executed immediately; in the `for`, control passes to the increment step. The `continue` statement applies only to loops, not to `switch`. A `continue` inside a `switch` inside a loop causes the next loop iteration.

As an example, this fragment processes only the non-negative elements in the array `a`; negative values are skipped.

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

The `continue` statement is often used when the part of the loop that follows is complicated, so that reversing a test and indenting another level would nest the program too deeply.

3.8 Goto 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. We have not used `goto` in this book.

Nevertheless, there are a few situations where `goto`s may find a place. The most common is to abandon processing in some deeply nested structure, such as breaking out of two or more loops at once. The `break` statement cannot be used directly since it only exits from the innermost loop. Thus:

```
        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 places.

A label has the same form as a variable name, and is followed by a colon. It can be attached to any statement in the same function as the `goto`. The scope of a label is the entire function.

As another example, consider the problem of determining whether two arrays `a` and `b` have an element in common. One possibility is

```
        for (i = 0; i < n; i++)
            for (j = 0; j < m; j++)
                if (a[i] == b[j])
                    goto found;
        /* didn't find any common element */
        ...
    found:
        /* got one:  a[i] == b[j] */
        ...
```

Code involving a `goto` can always be written without one, though perhaps at the price of some repeated tests or an extra variable. For example, the array search becomes

```
        found = 0;
        for (i = 0; i < n && !found; i++)
            for (j = 0; j < m && !found; j++)
                if (a[i] == b[j])
                    found = 1;
        if (found)
            /* got one:  a[i-1] == b[j-1] */
            ...
        else
            /* didn't find any common element */
            ...
```

With a few exceptions like those cited here, code that relies on `goto` statements is generally harder to understand and to maintain than code without `goto`s. Although we are not dogmatic about the matter, it does seem that `goto` statements should be used rarely, if at all.