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

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

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

```
char backslash = '\\';
int  i = 0;
float eps = 1.0e-5;
```

If the variable in question is external or static, the initialization is done once only, conceptually before the program starts executing. Explicitly initialized automatic variables are initialized each time the function they are in is called. Automatic variables for which there is no explicit initializer have undefined (i.e., garbage) values. External and static variables are initialized to zero by default, but it is good style to state the initialization anyway.

We will discuss initialization further as new data types are introduced.

2.5 Arithmetic Operators

The binary arithmetic operators are +, −, *, /, and the modulus operator %. There is a unary −, but no unary +.

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)
    it's a leap year
else
    it's not
```

The % operator cannot be applied to `float` or `double`.

The + and − operators have the same precedence, which is lower than the (identical) precedence of *, /, and %, which are in turn lower than unary minus. Arithmetic operators group left to right. (A table at the end of this chapter summarizes precedence and associativity for all operators.) The order of evaluation is not specified for associative and commutative operators like * and +; the compiler may rearrange a parenthesized computation involving one of these. Thus `a+(b+c)` can be evaluated as

`(a+b)+c`. This rarely makes any difference, but if a particular order is required, explicit temporary variables must be used.

The action taken on overflow or underflow depends on the machine at hand.

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:

```
==   !=
```

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

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

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

Clearly, 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. Not only that, but if this test fails, we must not go on and read another character.

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

The precedence of `&&` is greater 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.

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

```
if (!inword)
```

rather than

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

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

Exercise 2-1. Write a loop equivalent to the `for` loop above without using `&&`. □

2.7 Type Conversions

When operands of different types appear in expressions, they are converted to a common type according to a small number of rules. In general, the only conversions that happen automatically are those that make sense, 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.

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

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

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

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 positive 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 only.* If the character is not an upper case letter, `lower` returns it unchanged.

```
lower(c)  /* convert c to lower case; ASCII only */
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 (IBM 360/370), so this code fails on such systems — it converts more than letters.

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? Unfortunately, this varies from machine to machine, reflecting differences in architecture. On some machines (PDP-11, for instance), 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 character set will never be negative, so these characters may be used freely in expressions as positive quantities. But arbitrary bit patterns stored in character variables may appear to be negative on some machines, yet positive on others.

The most common occurrence of this situation is when the value −1 is used for `EOF`. Consider the code

```
char c;

c = getchar();
if (c == EOF)
     ...
```

On a machine which does not do sign extension, `c` is always positive because it is a `char`, yet `EOF` is negative. As a result, the test always fails. To avoid this, we have been careful to use `int` instead of `char` for any variable which holds a value returned by `getchar`.

The real reason for using `int` instead of `char` is not related to any questions of possible sign extension. It is simply that `getchar` must return all possible characters (so that it can be used to read arbitrary input) and, in addition, a distinct `EOF` value. Thus its value *cannot* be represented as a `char`, but must instead be stored as an `int`.

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

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

sets `isdigit` to 1 if `c` is a digit, and to 0 if not. (In the test part of `if`, `while`, `for`, etc., "true" just means "non-zero.")

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

`char` and `short` are converted to `int`, and `float` is converted to `double`.

Then if either operand is `double`, the other is converted to `double`, and the result is `double`.

Otherwise if either operand is `long`, the other is converted to `long`, and the result is `long`.

Otherwise if either operand is `unsigned`, the other is converted to `unsigned`, and the result is `unsigned`.

Otherwise the operands must be `int`, and the result is `int`.

Notice that all `float`'s in an expression are converted to `double`; all floating point arithmetic in C is done in double precision.

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. The reverse operation, `int` to `char`, is well-behaved — excess high-order bits are simply discarded. 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.

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. `double` is converted to `float` by rounding. Longer `int`'s are converted to shorter ones or to `char`'s by dropping the excess high-order bits.

Since a function argument is an expression, type conversions also take place when arguments are passed to functions: in particular, `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 construct 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 in fact as if *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. So if `n` is an integer,

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

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

Exercise 2-2. Write the function `htoi(s)`, which converts a string of hexadecimal digits 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; 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* using its value, 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 x=(i+j)++ is illegal.

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

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

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

```
squeeze(s, c)  /* delete all c from s */
char s[];
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 which 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, the function `strcat(s, t)` concatenates the string `t` to the end of the string `s`. `strcat` assumes that there is enough space in `s` to hold the combination.

```
strcat(s, t)    /* concatenate t to end of s */
char s[], t[]; /* s must be big enough */
{
    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 2-3. Write an alternate version of `squeeze(s1, s2)` which deletes each character in `s1` which matches any character in the *string* `s2`. □

Exercise 2-4. 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`. □

2.9 Bitwise Logical Operators

C provides a number of operators for bit manipulation; these may not be applied to `float` or `double`.

&	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,

```
c = 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 | MASK;
```

sets to one in x the bits that are set to one in MASK.

You should carefully distinguish the bitwise operators & and | from the logical connectives && 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. (Why?)

The shift operators << and >> perform left and right shifts of their left operand by the number of bit positions given by the right operand. Thus x << 2 shifts x left by two positions, filling vacated bits with 0; this is equivalent to multiplication by 4. Right shifting an unsigned quantity fills vacated bits with 0. Right shifting a signed quantity will fill with sign bits ("arithmetic shift") on some machines such as the PDP-11, 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. This operator typically finds use in expressions like

```
x & ~077
```

which masks 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 and thus evaluated at compile time.

To illustrate the use of some of the bit operators, consider the function getbits(x, p, n) which returns (right adjusted) the 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(x, p, n)      /* get n bits from position p */
unsigned x, p, n;
{
        return((x >> (p+1-n)) & ~(~0 << n));
}
```

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

Exercise 2-5. Modify `getbits` to number bits from left to right. □

Exercise 2-6. Write a function `wordlength()` which computes the word length of the host machine, that is, the number of bits in an `int`. The function should be portable, in the sense that the same source code works on all machines. □

Exercise 2-7. Write the function `rightrot(n, b)` which rotates the integer `n` to the right by `b` bit positions. □

Exercise 2-8. Write the function `invert(x, p, n)` which inverts (i.e., changes 1 into 0 and vice versa) the `n` bits of `x` that begin at position `p`, leaving the others unchanged. □

2.10 Assignment Operators and Expressions

Expressions such as

```
i = i + 2
```

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

```
i += 2
```

using an *assignment operator* like +=.

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

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

If *e1* and *e2* are expressions, then

e1 op= e2

is equivalent to

e1 = (*e1*) *op* (*e2*)

except that *e1* is computed only once. Notice the parentheses around *e2*:

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

is actually

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

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

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 `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 the compiler to produce more efficient code.

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

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

Assignments using the other assignment operators (`+=`, `-=`, etc.) can also occur in expressions, although it is a less frequent occurrence.

The type of an assignment expression is the type of its left operand.

Exercise 2-9. In a 2's complement number system, `x & (x-1)` deletes the rightmost 1-bit in `x`. (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;
```

of course 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

```
e1 ? e2 : e3
```

the expression *e1* is evaluated first. If it is non-zero (true), then the expression *e2* is evaluated, and that is the value of the conditional expression. Otherwise *e3* is evaluated, and that is the value. Only one of *e2* and *e3* 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 just as any other expression. If *e2* and *e3* 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 `double` regardless of whether `n` is positive or not.

Parentheses are not necessary around the first expression of a conditional expression, since the precedence of `?:` is very low, 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 exactly one 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. Although this might look tricky, it's instructive to try to write it without the conditional expression.

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

The table below summarizes the rules for precedence and associativity of all operators, including those which 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 + and −.

Operator	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
`=  +=  -=` etc.	right to left
`,` (Chapter 3)	left to right

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) and `&` (address of).

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

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

must be fully parenthesized to give proper results.

As mentioned before, expressions involving one of the associative and commutative operators (`*`, `+`, `&`, `^`, `|`) can be rearranged even when parenthesized. In most cases this makes no difference whatsoever; in situations where it might, explicit temporary variables can be used to force a particular order of evaluation.

C, like most languages, does not specify in what order the operands of an operator are evaluated. For example, in a statement like

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

`f` may be evaluated before `g` or vice versa; thus if either `f` or `g` alters an external variable that the other depends on, `x` can depend on the order of evaluation. Again, 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 (and does) produce different results on different machines, depending on whether or not `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 stored. 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. The compiler can do this in different ways, and generate different answers depending on its interpretation. When side effects (assignment to actual variables) takes place is left to the discretion of the compiler, since the best order strongly depends on machine architecture.

The moral of this discussion is that writing code which 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, that innocence may help to protect you. (The C verifier *lint* will detect most dependencies on order of evaluation.)

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

3.1 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 Algol-like languages.

The 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 actually be declared inside *any* block; we will talk about this in Chapter 4.) There is never a semicolon after the right brace that ends a block.

3.2 If-Else

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

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

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 done. 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 is 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 in the usual way — the `else` is associated 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:

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

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

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

This is because grammatically, a *statement* follows the `if`, and 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
        statement
```

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

The last `else` part handles the "none of the above" or default case where none of the other conditions was satisfied. Sometimes there is no explicit action for the default; in that case 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(x, v, n)      /* find x in v[0] ... v[n-1] */
int x, v[], n;
{
     int low, high, mid;

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

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`.

3.4 Switch

The `switch` statement is a special multi-way decision maker that tests whether an expression matches one of a number of *constant* values, and branches accordingly. In Chapter 1 we wrote a program to count the 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`.

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

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

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

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

The `switch` evaluates the integer expression in parentheses (in this program the character `c`) and compares its value to all the cases. Each case must be labeled by an integer or character constant or constant expression. If a case matches the expression value, execution starts at that case. The case labeled `default` is executed if none of the other cases is satisfied. A `default` is optional; if it isn't there and if none of the cases matches, no action at all takes place. Cases and default can occur in any order. Cases must all be different.

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 well, as will be discussed later in this chapter.

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

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

Exercise 3-1. Write a function `expand(s, t)` which converts characters like newline and tab into visible escape sequences like `\n` and `\t` as it copies the string `s` to `t`. Use a `switch`. □

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;
}
```

Grammatically, the three components of a `for` are expressions. Most commonly, *expr1* and *expr3* are assignments or function calls and *expr2* is a relational expression. Any of the three parts can be omitted, although the

semicolons must remain. If *expr1* or *expr3* is left out, it is simply dropped from the expansion. If the test, *expr2*, is not present, it is taken as permanently true, so

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

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

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

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

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

The `for` is clearly superior when there is a simple initialization and re-initialization, since it keeps the loop control statements close together and visible at the top of the loop. This is 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 or PL/I DO loop. The analogy is not perfect, however, since the limits of a `for` loop can be altered from within the loop, and the controlling 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 a `for`; it is 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 more general; it copes with optional leading white space and an optional + or – sign. (Chapter 4 shows `atof`, which does the same conversion for floating point numbers.)

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

skip white space, if any
get sign, if any
get integer part, 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.

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

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

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 the Shell sort is that in early stages, far-apart elements are compared, rather than adjacent ones, as in simple interchange sorts. This tends to eliminate large amounts of disorder quickly, so later stages have less work to do. The interval between compared elements is gradually decreased to one, at which point the sort effectively becomes an adjacent interchange method.

```
shell(v, n)  /* sort v[0]...v[n-1] into increasing order */
int v[], n;
{
    int gap, i, j, 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 loop controls the gap between compared elements, shrinking it from `n/2` by a factor of two each pass until it becomes zero. The middle loop compares each pair of elements that is separated by `gap`; the innermost loop reverses any that are out of order. Since `gap` is eventually reduced to one, all elements are eventually ordered correctly. Notice that 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