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.

CHAPTER 4: Functions and Program Structure

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

C has been designed to make functions efficient and easy to use; C programs generally consist of many small functions rather than a few big ones. A program may reside in one or more source files. Source files may be compiled separately and loaded together, along with previously compiled functions from libraries. We will not go into that process here, however, since the details vary from system to system.

Function declaration and definition is the area where the ANSI standard has made the most visible changes to C. As we saw first in Chapter 1, it is now possible to declare the types of arguments when a function is declared. The syntax of function definition also changes, so that declarations and definitions match. This makes it possible for a compiler to detect many more errors than it could before. Furthermore, when arguments are properly declared, appropriate type coercions are performed automatically.

The standard clarifies the rules on the scope of names; in particular, it requires that there be only one definition of each external object. Initialization is more general: automatic arrays and structures may now be initialized.

The C preprocessor has also been enhanced. New preprocessor facilities include a more complete set of conditional compilation directives, a way to create quoted strings from macro arguments, and better control over the macro expansion process.

4.1 Basics of Functions

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

letters "ould" in the set of lines

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

will produce the output

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

The job falls neatly into three pieces:

```
while (there's another line)
    if (the line contains the pattern)
        print it
```

Although it's certainly possible to put the code for all of this in `main`, a better way is to use the structure to advantage by making each part a separate function. Three small pieces are easier to deal with than one big one, because irrelevant details can be buried in the functions, and the chance of unwanted interactions is minimized. And the pieces may even be useful in other programs.

"While there's another line" is `getline`, a function that we wrote in Chapter 1, and "print it" is `printf`, which someone has already provided for us. This means we need only write a routine to decide whether the line contains an occurrence of the pattern.

We can solve that problem by writing a function `strindex(s,t)` that returns the position or index in the string `s` where the string `t` begins, or `-1` if `s` doesn't contain `t`. Because C arrays begin at position zero, indexes will be zero or positive, and so a negative value like `-1` is convenient for signaling failure. When we later need more sophisticated pattern matching, we only have to replace `strindex`; the rest of the code can remain the same. (The standard library provides a function `strstr` that is similar to `strindex`, except that it returns a pointer instead of an index.)

Given this much design, filling in the details of the program is straightforward. Here is the whole thing, so you can see how the pieces fit together. For now, the pattern to be searched for is a literal string, which is not the most general of mechanisms. We will return shortly to a discussion of how to initialize character arrays, and in Chapter 5 will show how to make the pattern a parameter that is set when the program is run. There is also a slightly different version of `getline`; you might find it instructive to compare it to the one in Chapter 1.

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

int getline(char line[], int max);
int strindex(char source[], char searchfor[]);

char pattern[] = "ould";    /* pattern to search for */

/* find all lines matching pattern */
main()
{
    char line[MAXLINE];
    int found = 0;

    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0) {
            printf("%s", line);
            found++;
        }
    return found;
}

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

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}

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

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

Each function definition has the form

```
return-type function-name(argument declarations)
{
    declarations and statements
}
```

Various parts may be absent; a minimal function is

```
dummy() {}
```

which does nothing and returns nothing. A do-nothing function like this is sometimes useful as a place holder during program development. If the return type is omitted, `int` is assumed.

A program is just a set of definitions of variables and functions. Communication between the functions is by arguments and values returned by the functions, and through external variables. The functions can occur in any order in the source file, and the source program can be split into multiple files, so long as no function is split.

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

```
return expression;
```

The *expression* will be converted to the return type of the function if necessary. Parentheses are often used around the *expression*, but they are optional.

The calling function is free to ignore the returned value. Furthermore, there need be no expression after `return`; in that case, no value is returned to the caller. Control also returns to the caller with no value when execution "falls off the end" of the function by reaching the closing right brace. It is not illegal, but probably a sign of trouble, if a function returns a value from one place and no value from another. In any case, if a function fails to return a value, its "value" is certain to be garbage.

The pattern-searching program returns a status from `main`, the number of matches found. This value is available for use by the environment that called the program.

The mechanics of how to compile and load a C program that resides on multiple source files vary from one system to the next. On the UNIX system, for example, the `cc` command mentioned in Chapter 1 does the job. Suppose that the three functions are stored in three files called `main.c`, `getline.c`, and `strindex.c`. Then the command

```
cc main.c getline.c strindex.c
```

compiles the three files, placing the resulting object code in files `main.o`, `getline.o`, and `strindex.o`, then loads them all into an executable file called `a.out`. If there is an error, say in `main.c`, that file can be recompiled by itself and the result loaded with the previous object files, with the command

```
cc main.c getline.o strindex.o
```

The `cc` command uses the "`.c`" versus "`.o`" naming convention to distinguish

source files from object files.

Exercise 4-1. Write the function `strrindex(s,t)`, which returns the position of the *rightmost* occurrence of `t` in `s`, or `-1` if there is none. □

4.2 Functions Returning Non-integers

So far our examples of functions have returned either no value (`void`) or an `int`. What if a function must return some other type? Many numerical functions like `sqrt`, `sin`, and `cos` return `double`; other specialized functions return other types. To illustrate how to deal with this, let us write and use the function `atof(s)`, which converts the string `s` to its double-precision floating-point equivalent. `atof` is an extension of `atoi`, which we showed versions of in Chapters 2 and 3. It handles an optional sign and decimal point, and the presence or absence of either integer part or fractional part. Our version is *not* a high-quality input conversion routine; that would take more space than we care to use. The standard library includes an `atof`; the header `<stdlib.h>` declares it.

First, `atof` itself must declare the type of value it returns, since it is not `int`. The type name precedes the function name:

```
#include <ctype.h>

/* atof:  convert string s to double */
double atof(char s[])
{
    double val, power;
    int i, sign;

    for (i = 0; isspace(s[i]); i++)  /* skip white space */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10.0;
    }
    return sign * val / power;
}
```

Second, and just as important, the calling routine must know that `atof` returns a non-`int` value. One way to ensure this is to declare `atof` explicitly

in the calling routine. The declaration is shown in this primitive calculator (barely adequate for check-book balancing), which reads one number per line, optionally preceded by a sign, and adds them up, printing the running sum after each input:

```
#include <stdio.h>

#define MAXLINE 100

/* rudimentary calculator */
main()
{
    double sum, atof(char []);
    char line[MAXLINE];
    int getline(char line[], int max);

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

The declaration

```
double sum, atof(char []);
```

says that `sum` is a `double` variable, and that `atof` is a function that takes one `char[]` argument and returns a `double`.

The function `atof` must be declared and defined consistently. If `atof` itself and the call to it in `main` have inconsistent types in the same source file, the error will be detected by the compiler. But if (as is more likely) `atof` were compiled separately, the mismatch would not be detected, `atof` would return a `double` that `main` would treat as an `int`, and meaningless answers would result.

In the light of what we have said about how declarations must match definitions, this might seem surprising. The reason a mismatch can happen is that if there is no function prototype, a function is implicitly declared by its first appearance in an expression, such as

```
sum += atof(line)
```

If a name that has not been previously declared occurs in an expression and is followed by a left parenthesis, it is declared by context to be a function name, the function is assumed to return an `int`, and nothing is assumed about its arguments. Furthermore, if a function declaration does not include arguments, as in

```
double atof();
```

that too is taken to mean that nothing is to be assumed about the arguments of `atof`; all parameter checking is turned off. This special meaning of the empty

argument list is intended to permit older C programs to compile with new compilers. But it's a bad idea to use it with new programs. If the function takes arguments, declare them; if it takes no arguments, use `void`.

Given `atof`, properly declared, we could write `atoi` (convert a string to `int`) in terms of it:

```
/* atoi:  convert string s to integer using atof */
int atoi(char s[])
{
    double atof(char s[]);

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

Notice the structure of the declarations and the `return` statement. The value of the expression in

```
return expression;
```

is converted to the type of the function before the return is taken. Therefore, the value of `atof`, a `double`, is converted automatically to `int` when it appears in this `return`, since the function `atoi` returns an `int`. This operation does potentially discard information, however, so some compilers warn of it. The cast states explicitly that the operation is intended, and suppresses any warning.

Exercise 4-2. Extend `atof` to handle scientific notation of the form

```
123.45e-6
```

where a floating-point number may be followed by `e` or `E` and an optionally signed exponent. □

4.3 External Variables

A C program consists of a set of external objects, which are either variables or functions. The adjective "external" is used in contrast to "internal," which describes the arguments and variables defined inside functions. External variables are defined outside of any function, and are thus potentially available to many functions. Functions themselves are always external, because C does not allow functions to be defined inside other functions. By default, external variables and functions have the property that all references to them by the same name, even from functions compiled separately, are references to the same thing. (The standard calls this property *external linkage*.) In this sense, external variables are analogous to Fortran COMMON blocks or variables in the outermost block in Pascal. We will see later how to define external variables and functions that are visible only within a single source file.

Because external variables are globally accessible, they provide an alternative to function arguments and return values for communicating data between functions. Any function may access an external variable by referring to it by name, if the name has been declared somehow.

If a large number of variables must be shared among functions, external variables are more convenient and efficient than long argument lists. As pointed out in Chapter 1, however, this reasoning should be applied with some caution, for it can have a bad effect on program structure, and lead to programs with too many data connections between functions.

External variables are also useful because of their greater scope and lifetime. Automatic variables are internal to a function; they come into existence when the function is entered, and disappear when it is left. External variables, on the other hand, are permanent, so they retain values from one function invocation to the next. Thus if two functions must share some data, yet neither calls the other, it is often most convenient if the shared data is kept in external variables rather than passed in and out via arguments.

Let us examine this issue further with a larger example. The problem is to write a calculator program that provides the operators +, -, *, and /. Because it is easier to implement, the calculator will use reverse Polish notation instead of infix. (Reverse Polish is used by some pocket calculators, and in languages like Forth and Postscript.)

In reverse Polish notation, each operator follows its operands; an infix expression like

```
(1 - 2) * (4 + 5)
```

is entered as

```
1 2 - 4 5 + *
```

Parentheses are not needed; the notation is unambiguous as long as we know how many operands each operator expects.

The implementation is simple. Each operand is pushed onto a stack; when an operator arrives, the proper number of operands (two for binary operators) is popped, the operator is applied to them, and the result is pushed back onto the stack. In the example above, for instance, 1 and 2 are pushed, then replaced by their difference, −1. Next, 4 and 5 are pushed and then replaced by their sum, 9. The product of −1 and 9, which is −9, replaces them on the stack. The value on the top of the stack is popped and printed when the end of the input line is encountered.

The structure of the program is thus a loop that performs the proper operation on each operator and operand as it appears:

```
while (next operator or operand is not end-of-file indicator)
    if (number)
        push it
    else if (operator)
        pop operands
        do operation
        push result
    else if (newline)
        pop and print top of stack
    else
        error
```

The operations of pushing and popping a stack are trivial, but by the time error detection and recovery are added, they are long enough that it is better to put each in a separate function than to repeat the code throughout the whole program. And there should be a separate function for fetching the next input operator or operand.

The main design decision that has not yet been discussed is where the stack is, that is, which routines access it directly. One possibility is to keep it in `main`, and pass the stack and the current stack position to the routines that push and pop it. But `main` doesn't need to know about the variables that control the stack; it only does push and pop operations. So we have decided to store the stack and its associated information in external variables accessible to the `push` and `pop` functions but not to `main`.

Translating this outline into code is easy enough. If for now we think of the program as existing in one source file, it will look like this:

```
#includes
#defines

function declarations for main

main() { ... }

external variables for push and pop

void push(double f) { ... }
double pop(void) { ... }

int getop(char s[]) { ... }

routines called by getop
```

Later we will discuss how this might be split into two or more source files.

The function `main` is a loop containing a big `switch` on the type of operator or operand; this is a more typical use of `switch` than the one shown in Section 3.4.

```
#include <stdio.h>
#include <stdlib.h>    /* for atof() */

#define MAXOP   100    /* max size of operand or operator */
#define NUMBER  '0'    /* signal that a number was found */

int getop(char []);
void push(double);
double pop(void);

/* reverse Polish calculator */
main()
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
        case NUMBER:
            push(atof(s));
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break;
        case '/':
            op2 = pop();
            if (op2 != 0.0)
                push(pop() / op2);
            else
                printf("error: zero divisor\n");
            break;
        case '\n':
            printf("\t%.8g\n", pop());
            break;
        default:
            printf("error: unknown command %s\n", s);
            break;
        }
    }
    return 0;
}
```

Because + and * are commutative operators, the order in which the popped operands are combined is irrelevant, but for - and / the left and right operands must be distinguished. In

```
push(pop() - pop());    /* WRONG */
```

the order in which the two calls of pop are evaluated is not defined. To guarantee the right order, it is necessary to pop the first value into a temporary variable as we did in main.

```
#define MAXVAL  100    /* maximum depth of val stack */

int sp = 0;            /* next free stack position */
double val[MAXVAL];    /* value stack */

/* push:  push f onto value stack */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n", f);
}



/* pop:  pop and return top value from stack */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("error: stack empty\n");
        return 0.0;
    }
}
```

A variable is external if it is defined outside of any function. Thus the stack and stack index that must be shared by push and pop are defined outside of these functions. But main itself does not refer to the stack or stack position—the representation can be hidden.

Let us now turn to the implementation of getop, the function that fetches the next operator or operand. The task is easy. Skip blanks and tabs. If the next character is not a digit or a decimal point, return it. Otherwise, collect a string of digits (which might include a decimal point), and return NUMBER, the signal that a number has been collected.

```
#include <ctype.h>

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

/* getop:  get next operator or numeric operand */
int getop(char s[])
{
    int i, c;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c;      /* not a number */
    i = 0;
    if (isdigit(c))    /* collect integer part */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.')      /* collect fraction part */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}
```

What are `getch` and `ungetch`? It is often the case that a program cannot determine that it has read enough input until it has read too much. One instance is collecting the characters that make up a number: until the first non-digit is seen, the number is not complete. But then the program has read one character too far, a character that it is not prepared for.

The problem would be solved if it were possible to "un-read" the unwanted character. Then, every time the program reads one character too many, it could push it back on the input, so the rest of the code could behave as if it had never been read. Fortunately, it's easy to simulate un-getting a character, by writing a pair of cooperating functions. `getch` delivers the next input character to be considered; `ungetch` remembers the characters put back on the input, so that subsequent calls to `getch` will return them before reading new input.

How they work together is simple. `ungetch` puts the pushed-back characters into a shared buffer—a character array. `getch` reads from the buffer if there is anything there, and calls `getchar` if the buffer is empty. There must also be an index variable that records the position of the current character in the buffer.

Since the buffer and the index are shared by `getch` and `ungetch` and must retain their values between calls, they must be external to both routines. Thus we can write `getch`, `ungetch`, and their shared variables as:

```
#define BUFSIZE 100

char buf[BUFSIZE];  /* buffer for ungetch */
int  bufp = 0;      /* next free position in buf */

int getch(void) /* get a (possibly pushed back) character */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c) /* push character back on input */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}
```

The standard library includes a function `ungetc` that provides one character of pushback; we will discuss it in Chapter 7. We have used an array for the pushback, rather than a single character, to illustrate a more general approach.

Exercise 4-3. Given the basic framework, it's straightforward to extend the calculator. Add the modulus (`%`) operator and provisions for negative numbers. □

Exercise 4-4. Add commands to print the top element of the stack without popping, to duplicate it, and to swap the top two elements. Add a command to clear the stack. □

Exercise 4-5. Add access to library functions like `sin`, `exp`, and `pow`. See `<math.h>` in Appendix B, Section 4. □

Exercise 4-6. Add commands for handling variables. (It's easy to provide twenty-six variables with single-letter names.) Add a variable for the most recently printed value. □

Exercise 4-7. Write a routine `ungets(s)` that will push back an entire string onto the input. Should `ungets` know about `buf` and `bufp`, or should it just use `ungetch`? □

Exercise 4-8. Suppose that there will never be more than one character of pushback. Modify `getch` and `ungetch` accordingly. □

Exercise 4-9. Our `getch` and `ungetch` do not handle a pushed-back `EOF` correctly. Decide what their properties ought to be if an `EOF` is pushed back, then implement your design. □

Exercise 4-10. An alternate organization uses `getline` to read an entire input line; this makes `getch` and `ungetch` unnecessary. Revise the calculator to use this approach. □

4.4 Scope Rules

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

- How are declarations written so that variables are properly declared during compilation?
- How are declarations arranged so that all the pieces will be properly connected when the program is loaded?
- How are declarations organized so there is only one copy?
- How are external variables initialized?

Let us discuss these topics by reorganizing the calculator program into several files. As a practical matter, the calculator is too small to be worth splitting, but it is a fine illustration of the issues that arise in larger programs.

The *scope* of a name is the part of the program within which the name can be used. For an automatic variable declared at the beginning of a function, the scope is the function in which the name is declared. Local variables of the same name in different functions are unrelated. The same is true of the parameters of the function, which are in effect local variables.

The scope of an external variable or a function lasts from the point at which it is declared to the end of the file being compiled. For example, if `main`, `sp`, `val`, `push`, and `pop` are defined in one file, in the order shown above, that is,

```
main() { ... }

int sp = 0;
double val[MAXVAL];

void push(double f) { ... }

double pop(void) { ... }
```

then the variables `sp` and `val` may be used in `push` and `pop` simply by naming them; no further declarations are needed. But these names are not visible in `main`, nor are `push` and `pop` themselves.

On the other hand, if an external variable is to be referred to before it is defined, or if it is defined in a different source file from the one where it is being used, then an `extern` declaration is mandatory.

It is important to distinguish between the *declaration* of an external variable and its *definition*. A declaration announces the properties of a variable (primarily its type); a definition also causes storage to be set aside. If the lines

```
int sp;
double val[MAXVAL];
```

appear outside of any function, they *define* the external variables `sp` and `val`,

cause storage to be set aside, and also serve as the declaration for the rest of that source file. On the other hand, the lines

```
extern int sp;
extern double val[];
```

declare for the rest of the source file that `sp` is an `int` and that `val` is a `double` array (whose size is determined elsewhere), but they do not create the variables or reserve storage for them.

There must be only one *definition* of an external variable among all the files that make up the source program; other files may contain `extern` declarations to access it. (There may also be `extern` declarations in the file containing the definition.) Array sizes must be specified with the definition, but are optional with an `extern` declaration.

Initialization of an external variable goes only with the definition.

Although it is not a likely organization for this program, the functions `push` and `pop` could be defined in one file, and the variables `val` and `sp` defined and initialized in another. Then these definitions and declarations would be necessary to tie them together:

In file1:

```
extern int sp;
extern double val[];

void push(double f) { ... }

double pop(void) { ... }
```

In file2:

```
int sp = 0;
double val[MAXVAL];
```

Because the `extern` declarations in *file1* lie ahead of and outside the function definitions, they apply to all functions; one set of declarations suffices for all of *file1*. This same organization would also be needed if the definitions of `sp` and `val` followed their use in one file.

4.5 Header Files

Let us now consider dividing the calculator program into several source files, as it might be if each of the components were substantially bigger. The `main` function would go in one file, which we will call `main.c`; `push`, `pop`, and their variables go into a second file, `stack.c`; `getop` goes into a third, `getop.c`. Finally, `getch` and `ungetch` go into a fourth file, `getch.c`; we separate them from the others because they would come from a separately-compiled library in a realistic program.

There is one more thing to worry about—the definitions and declarations shared among the files. As much as possible, we want to centralize this, so that there is only one copy to get right and keep right as the program evolves. Accordingly, we will place this common material in a *header file*, `calc.h`, which will be included as necessary. (The `#include` line is described in Section 4.11.) The resulting program then looks like this:

`calc.h`:

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

`main.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}
```

`getop.c`:

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop() {
    ...
}
```

`stack.c`:

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

`getch.c`:

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```

There is a tradeoff between the desire that each file have access only to the information it needs for its job and the practical reality that it is harder to maintain more header files. Up to some moderate program size, it is probably best to have one header file that contains everything that is to be shared between any two parts of the program; that is the decision we made here. For a much larger program, more organization and more headers would be needed.

4.6 Static Variables

The variables `sp` and `val` in `stack.c`, and `buf` and `bufp` in `getch.c`, are for the private use of the functions in their respective source files, and are not meant to be accessed by anything else. The `static` declaration, applied to an external variable or function, limits the scope of that object to the rest of the source file being compiled. External `static` thus provides a way to hide names like `buf` and `bufp` in the `getch-ungetch` combination, which must be external so they can be shared, yet which should not be visible to users of `getch` and `ungetch`.

Static storage is specified by prefixing the normal declaration with the word `static`. If the two routines and the two variables are compiled in one file, as in

```
static char buf[BUFSIZE];  /* buffer for ungetch */
static int  bufp = 0;      /* next free position in buf */

int getch(void) { ... }

void ungetch(int c) { ... }
```

then no other routine will be able to access `buf` and `bufp`, and those names will not conflict with the same names in other files of the same program. In the same way, the variables that `push` and `pop` use for stack manipulation can be hidden, by declaring `sp` and `val` to be `static`.

The external `static` declaration is most often used for variables, but it can be applied to functions as well. Normally, function names are global, visible to any part of the entire program. If a function is declared `static`, however, its name is invisible outside of the file in which it is declared.

The `static` declaration can also be applied to internal variables. Internal `static` variables are local to a particular function just as automatic variables are, but unlike automatics, they remain in existence rather than coming and going each time the function is activated. This means that internal `static` variables provide private, permanent storage within a single function.

Exercise 4-11. Modify `getop` so that it doesn't need to use `ungetch`. Hint: use an internal `static` variable. □

4.7 Register Variables

A `register` declaration advises the compiler that the variable in question will be heavily used. The idea is that `register` variables are to be placed in machine registers, which may result in smaller and faster programs. But compilers are free to ignore the advice.

The `register` declaration looks like

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

and so on. The `register` declaration can only be applied to automatic variables and to the formal parameters of a function. In this latter case, it looks like

```
f(register unsigned m, register long n)
{
    register int i;
    ...
}
```

In practice, there are restrictions on register variables, reflecting the realities of underlying hardware. Only a few variables in each function may be kept in registers, and only certain types are allowed. Excess register declarations are harmless, however, since the word `register` is ignored for excess or disallowed declarations. And it is not possible to take the address of a register variable (a topic to be covered in Chapter 5), regardless of whether the variable is actually placed in a register. The specific restrictions on number and types of register variables vary from machine to machine.

4.8 Block Structure

C is not a block-structured language in the sense of Pascal or similar languages, because functions may not be defined within other functions. On the other hand, variables can be defined in a block-structured fashion within a function. Declarations of variables (including initializations) may follow the left brace that introduces *any* compound statement, not just the one that begins a function. Variables declared in this way hide any identically named variables in outer blocks, and remain in existence until the matching right brace. For example, in

```
if (n > 0) {
    int i;  /* declare a new i */

    for (i = 0; i < n; i++)
        ...
}
```

the scope of the variable `i` is the "true" branch of the `if`; this `i` is unrelated to any `i` outside the block. An automatic variable declared and initialized in a block is initialized each time the block is entered. A `static` variable is initialized only the first time the block is entered.

Automatic variables, including formal parameters, also hide external variables and functions of the same name. Given the declarations

```
int x;
int y;

f(double x)
{
    double y;
    ...
}
```

then within the function `f`, occurrences of `x` refer to the parameter, which is a `double`; outside of `f`, they refer to the external `int`. The same is true of the variable `y`.

As a matter of style, it's best to avoid variable names that conceal names in an outer scope; the potential for confusion and error is too great.

4.9 Initialization

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

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

Scalar variables may be initialized when they are defined, by following the name with an equals sign and an expression:

```
int  x = 1;
char squote = '\'';
long day = 1000L * 60L * 60L * 24L;  /* milliseconds/day */
```

For external and static variables, the initializer must be a constant expression; the initialization is done once, conceptually before the program begins execution. For automatic and register variables, it is done each time the function or block is entered.

For automatic and register variables, the initializer is not restricted to being a constant: it may be any expression involving previously defined values, even function calls. For example, the initializations of the binary search program in Section 3.3 could be written as

```
int binsearch(int x, int v[], int n)
{
    int low = 0;
    int high = n - 1;
    int mid;
    ...
}
```

instead of

```
    int low, high, mid;

    low = 0;
    high = n - 1;
```

In effect, initializations of automatic variables are just shorthand for assignment statements. Which form to prefer is largely a matter of taste. We have generally used explicit assignments, because initializers in declarations are harder to see and further away from the point of use.

An array may be initialized by following its declaration with a list of initializers enclosed in braces and separated by commas. For example, to initialize an array `days` with the number of days in each month:

```
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

When the size of the array is omitted, the compiler will compute the length by counting the initializers, of which there are 12 in this case.

If there are fewer initializers for an array than the number specified, the missing elements will be zero for external, static, and automatic variables. It is an error to have too many initializers. There is no way to specify repetition of an initializer, nor to initialize an element in the middle of an array without supplying all the preceding values as well.

Character arrays are a special case of initialization; a string may be used instead of the braces and commas notation:

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

is a shorthand for the longer but equivalent

```
    char pattern[] = { 'o', 'u', 'l', 'd', '\0' };
```

In this case, the array size is five (four characters plus the terminating `'\0'`).

4.10 Recursion

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

There are two solutions to this problem. One is to store the digits in an array as they are generated, then print them in the reverse order, as we did with `itoa` in Section 3.6. The alternative is a recursive solution, in which `printd` first calls itself to cope with any leading digits, then prints the trailing digit. Again, this version can fail on the largest negative number.