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.

```
reverse(s)      /* reverse string s in place */
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.

Exercise 3-2. Write a function `expand(s1, s2)` which 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`. (A useful convention is that a leading or trailing - is taken literally.) □

3.6 Loops — Do-while

The `while` and `for` loops share the desirable attribute of testing the termination condition at the top, rather than at the bottom, as we discussed in Chapter 1. The third loop in C, the `do-while`, tests at the bottom *after* making each pass through the loop body; the body is always executed at least once. The syntax is

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

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

As might be expected, `do-while` is much less used than `while` and `for`, accounting for perhaps five percent of all loops. Nonetheless, it is from time to time 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(n, s)      /* convert n to characters in s */
char s[];
int n;
{
     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`, regardless of the value of `n`. We also used braces around the single statement that makes up the body of the `do-while`, even though they are unnecessary, so the hasty reader will not mistake the `while` part for the *beginning* of a `while` loop.

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

Exercise 3-4. Write the analogous function `itob(n, s)` which converts the unsigned integer `n` into a binary character representation in `s`. Write `itoh`, which converts an integer to hexadecimal representation. □

Exercise 3-5. Write a version of `itoa` which 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

It is sometimes convenient to be able to control loop exits 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` statement causes the innermost enclosing loop (or `switch`) to be exited immediately.

The following program removes trailing blanks and tabs from the end of each line of input, using a `break` to exit from a loop when the rightmost non-blank, non-tab is found.

```
#define   MAXLINE   1000

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

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

`getline` returns the length of the line. The inner `while` loop starts at the last character of `line` (recall that `--n` decrements `n` before using the value), and scans backwards looking for the first character that is not a blank, tab or newline. The loop is broken when one is found, or when `n` becomes negative (that is, when the entire line has been scanned). You should verify that this is correct behavior even when the line contains only white space characters.

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

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

This is inferior to the previous version, because the test is harder to understand. Tests which require a mixture of `&&`, `||`, `!`, or parentheses should generally be avoided.

3.8 Continue

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

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

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

Exercise 3-6. Write a program which copies its input to its output, except that it prints only one instance from each group of adjacent identical lines. (This is a simple version of the UNIX utility *uniq*.) □

3.9 Goto's and Labels

C provides the infinitely-abusable `goto` statement, and labels to branch to. Formally, the `goto` is never necessary, and in practice it is almost always easy to write code without it. We have not used `goto` in this book.

Nonetheless, we will suggest a few situations where `goto`'s may find a place. The most common use is to abandon processing in some deeply nested structure, such as breaking out of two loops at once. The `break` statement cannot be used directly since it leaves only 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`.

As another example, consider the problem of finding the first negative element in a two-dimensional array. (Multi-dimensional arrays are discussed in Chapter 5.) One possibility is

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

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

✓

Although we are not dogmatic about the matter, it does seem that `goto` statements should be used sparingly, 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 can often hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole, and easing the pain of making changes.

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

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

4.1 Basics

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

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

will produce the output

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

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

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

Although it's certainly possible to put the code for all of this in the main routine, a better way is to use the natural 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 minimized. And the pieces may even be useful in their own right.

"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 which decides if the line contains an occurrence of the pattern. We can solve that problem by stealing a design from PL/I: the function `index(s, t)` returns the position or index in the string `s` where the string `t` begins, or `-1` if `s` doesn't contain `t`. We use 0 rather than 1 as the starting position in `s` because C arrays begin at position zero. When we later need more sophisticated pattern matching we only have to replace `index`; the rest of the code can remain the same.

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 in the argument of `index`, 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. This is also a new version of `getline`; you might find it instructive to compare it to the one in Chapter 1.

```
#define   MAXLINE   1000

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

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

```
getline(s, lim)     /* get line into s, return length */
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);
}

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

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

Each function has the form

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

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

```
dummy() {}
```

which does nothing. (A do-nothing function is sometimes useful as a place holder during program development.) The function name may also be preceded by a type if the function returns something other than an integer value; this is the topic of the next section.

A program is just a set of individual function definitions. Communication between the functions is (in this case) by arguments and values returned by the functions; it can also be via external variables. The functions can occur in any order on 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 calling function is free to ignore the returned value if it wishes. 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, the "value" of a function which does not return one is certain to be garbage. The C verifier *lint* checks for such errors.

The mechanics of how to compile and load a C program which 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 on three files called *main.c*, *getline.c*, and *index.c*. Then the command

cc main.c getline.c index.c

compiles the three files, places the resulting relocatable object code in files *main.o*, *getline.o*, and *index.o*, and 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 index.o

The *cc* command uses the "*.c*" versus "*.o*" naming convention to distinguish source files from object files.

Exercise 4-1. Write the function `rindex(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, none of our programs has contained any declaration of the type of a function. This is because by default a function is implicitly declared by its appearance in an expression or statement, such as

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

If a name which 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. Furthermore, by default the function is assumed to return an `int`. Since `char` promotes to `int` in expressions, there is no need to declare functions that return `char`. These assumptions cover the majority of cases, including all of our examples so far.

But what happens if a function must return some other type? Many numerical functions like `sqrt`, `sin`, and `cos` return `double`; other specialized functions return other types. To illustrate how 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 wrote 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. (This is *not* a high-quality input conversion routine; that would take more space than we care to use.)

First, `atof` itself must declare the type of value it returns, since it is not `int`. Because `float` is converted to `double` in expressions, there is no point to saying that `atof` returns `float`; we might as well make use of the extra precision and thus we declare it to return `double`. The type name precedes the function name, like this:

```
double atof(s) /* convert string s to double */
char s[];
{
	double val, power;
	int  i, 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 (val = 0; s[i] >= '0' && s[i] <= '9'; i++)
		val = 10 * val + s[i] - '0';
	if (s[i] == '.')
		i++;
	for (power = 1; s[i] >= '0' && s[i] <= '9'; i++) {
		val = 10 * val + s[i] - '0';
		power *= 10;
	}
	return(sign * val / power);
}
```

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

```
#define   MAXLINE   100

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

      sum = 0;
      while (getline(line, MAXLINE) > 0)
            printf("\t%.2f\n", sum += atof(line));
}
```

The declaration

```
double sum, atof();
```

says that `sum` is a `double` variable, and that `atof` is a function that returns a `double` value. As a mnemonic, it suggests that `sum` and `atof(...)` are both double-precision floating point values.

Unless `atof` is explicitly declared in both places, C assumes that it returns an integer, and you'll get nonsense answers. If `atof` itself and the call to it in `main` are typed inconsistently in the same source file, it 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` which `main` would treat as an `int`, and meaningless answers would result. (*lint* catches this error.)

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

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

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

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

`return(`*expression*`)`

is always 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 a `return`, since the function `atoi` returns an `int`. (The conversion of a floating point value to `int` truncates any fractional part, as discussed in Chapter 2.)

Exercise 4-2. Extend `atof` so it handles 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 More on Function Arguments

In Chapter 1 we discussed the fact that function arguments are passed by value, that is, the called function receives a private, temporary copy of each argument, not its address. This means that the function cannot affect the original argument in the calling function. Within a function, each argument is in effect a local variable initialized to the value with which the function was called.

When an array name appears as an argument to a function, the location of the beginning of the array is passed; elements are not copied. The function can alter elements of the array by subscripting from this location. The effect is that arrays are passed by reference. In Chapter 5 we will discuss the use of pointers to permit functions to affect non-arrays in calling functions.

By the way, there is no entirely satisfactory way to write a portable function that accepts a variable number of arguments, because there is no portable way for the called function to determine how many arguments were actually passed to it in a given call. Thus, you can't write a truly portable function that will compute the maximum of an arbitrary number of arguments, as will the MAX built-in functions of Fortran and PL/I.

It is generally safe to deal with a variable number of arguments if the called function doesn't use an argument which was not actually supplied, and if the types are consistent. `printf`, the most common C function with a variable number of arguments, uses information from the first argument to determine how many other arguments are present and what their types are. It fails badly if the caller does not supply enough arguments or if the types are not what the first argument says. It is also non-portable and must be modified for different environments.

Alternatively, if the arguments are of known types it is possible to mark the end of the argument list in some agreed-upon way, such as a special argument value (often zero) that stands for the end of the arguments.

4.4 External Variables

A C program consists of a set of external objects, which are either variables or functions. The adjective "external" is used primarily in contrast to "internal," which describes the arguments and automatic variables defined inside functions. External variables are defined outside 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 are also "global," so that all references to such a variable by the same name (even from functions compiled separately) are references to the same thing. In this sense, external variables are analogous to Fortran COMMON or PL/I EXTERNAL. We will see later how to define external variables and functions that are not globally available, but are instead visible only within a single source file.

Because external variables are globally accessible, they provide an alternative to function arguments and returned 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 many data connections between functions.

A second reason for using external variables concerns initialization. In particular, external arrays may be initialized, but automatic arrays may not. We will treat initialization near the end of this chapter.

The third reason for using external variables is their scope and lifetime. Automatic variables are internal to a function; they come into existence when the routine is entered, and disappear when it is left. External variables, on the other hand, are permanent. They do not come and go, so they retain values from one function invocation to the next. Thus if two functions must share some data, yet neither calls the other, 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 another calculator program, better than the previous one. This one permits +, −, *, /, and = (to print the answer). Because it is somewhat easier to implement, the calculator will use reverse Polish notation instead of infix. (Reverse Polish is the scheme used by, for example, Hewlett-Packard pocket calculators.) 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 implementation is quite simple. Each operand is pushed onto a stack; when an operator arrives, the proper number of operands (two for binary operators) are popped, the operator applied to them, and the result 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 = operator prints the top element without removing it (so intermediate steps in a calculation can be checked).

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. Thus the structure of the program is

```
while (next operator or operand is not end of file)
    if (number)
        push it
    else if (operator)
        pop operands
        do operation
        push result
    else
        error
```

The main design decision that has not yet been discussed is where the stack is, that is, what 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 should think only in terms of pushing and popping. So we have decided to make the stack and its associated information external variables accessible to the `push` and `pop` functions but not to `main`.

Translating this outline into code is easy enough. The main program is primarily a big `switch` on the type of operator or operand; this is perhaps a more typical use of `switch` than the one shown in Chapter 3.

```
#define MAXOP   20   /* max size of operand, operator */
#define NUMBER  '0'  /* signal that number found */
#define TOOBIG  '9'  /* signal that string is too big */

main()    /* reverse Polish desk calculator */
{
     int type;
     char s[MAXOP];
     double op2, atof(), pop(), push();

     while ((type = getop(s, MAXOP)) != 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("zero divisor popped\n");
               break;
          case '=':
               printf("\t%f\n", push(pop()));
               break;
          case 'c':
               clear();
               break;
          case TOOBIG:
               printf("%.20s ... is too long\n", s);
               break;
          default:
               printf("unknown command %c\n", type);
               break;
          }
}
```

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

int sp = 0;     /* stack pointer */
double val[MAXVAL]; /* value stack */

double push(f) /* push f onto value stack */
double f;
{
	if (sp < MAXVAL)
		return(val[sp++] = f);
	else {
		printf("error: stack full\n");
		clear();
		return(0);
	}
}

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

clear()    /* clear stack */
{
	sp = 0;
}
```

The command c clears the stack, with a function `clear` which is also used by `push` and `pop` in case of error. We'll return to `getop` in a moment.

As discussed in Chapter 1, a variable is external if it is defined outside the body of any function. Thus the stack and stack pointer which must be shared by `push`, `pop`, and `clear` are defined outside of these three functions. But `main` itself does *not* refer to the stack or stack pointer — the representation is carefully hidden. Thus the code for the = operator must use

```
push(pop());
```

to examine the top of the stack without disturbing it.

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

Exercise 4-3. Given the basic framework, it's straightforward to extend the calculator. Add the modulus (`%`) and unary minus operators. Add an "erase" command which erases the top entry on the stack. Add commands for handling variables. (Twenty-six single-letter variable names is easy.) □

4.5 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. The two questions of interest are

> How are declarations written so that variables are properly declared during compilation?
> How are declarations set up so that all the pieces will be properly connected when the program is loaded?

The *scope* of a name is the part of the program over which the name is defined. For an automatic variable declared at the beginning of a function, the scope is the function in which the name is declared, and variables of the same name in different functions are unrelated. The same is true of the arguments of the function.

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

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

double push(f) { ... }

double pop() { ... }

clear() { ... }
```

then the variables `val` and `sp` may be used in `push`, `pop` and `clear` simply by naming them; no further declarations are needed.

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 (its type, size, etc.); a definition also causes storage to be allocated. 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 allocated, 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 allocate 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 an `extern` declaration in the file containing the definition.) Any initialization of an external variable goes only with the definition. Array sizes must be specified with the definition, but are optional with an `extern` declaration.

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

In file 1:

```
    int sp = 0;    /* stack pointer */
    double val[MAXVAL]; /* value stack */
```

In file 2:

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

    double push(f) { ... }

    double pop() { ... }

    clear() { ... }
```

Because the `extern` declarations in *file 2* lie ahead of and outside the three functions, they apply to all; one set of declarations suffices for all of *file 2*.

For larger programs, the `#include` file inclusion facility discussed later in this chapter allows one to keep only a single copy of the `extern` declarations for the program and have that inserted in each source file as it is being compiled.

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

The routine is substantially complicated by an attempt to handle the situation properly when an input number is too long. `getop` reads digits (perhaps with an intervening decimal point) until it doesn't see any more, but only stores the ones that fit. If there was no overflow, it returns `NUMBER` and the string of digits. If the number was too long, however, `getop` discards the rest of the input line so the user can simply retype the line from the point of error; it returns `TOOBIG` as the overflow signal.

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

     while ((c = getch()) == ' ' || c == '\t' || c == '\n')
          ;
     if (c != '.' && (c < '0' || c > '9'))
          return(c);
     s[0] = c;
     for (i = 1; (c = getchar()) >= '0' && c <= '9'; i++)
          if (i < lim)
               s[i] = c;
     if (c == '.') {      /* collect fraction */
          if (i < lim)
               s[i] = c;
          for (i++; (c=getchar()) >= '0' && c <= '9'; i++)
               if (i < lim)
                    s[i] = c;
     }
     if (i < lim) { /* number is ok */
          ungetch(c);
          s[i] = '\0';
          return(NUMBER);
     } else {  /* it's too big; skip rest of line */
          while (c != '\n' && c != EOF)
               c = getchar();
          s[lim-1] = '\0';
          return(TOOBIG);
     }
}
```

What are `getch` and `ungetch`? It is often the case that a program reading input cannot determine that it has read enough until it has read too much. One instance is collecting the characters that make up a 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` puts a character back on the input, so that the next call to `getch` will return it again.

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; it calls `getchar` if the buffer is empty. There must also be an index variable which records the position of the current character in the buffer.

Since the buffer and the index are shared by `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 */

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

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

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

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

Exercise 4-5. Suppose that there will never be more than one character of pushback. Modify `getch` and `ungetch` accordingly. □

Exercise 4-6. Our `getch` and `ungetch` do not handle a pushed-back `EOF` in a portable way. Decide what their properties ought to be if an `EOF` is pushed back, then implement your design. □

4.6 Static Variables

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

`static` variables may be either internal or external. Internal `static` variables are local to a particular function just as 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 in a function. Character strings that appear within a function, such as the arguments of `printf`, are internal static.

An external `static` variable is known within the remainder of the *source file* in which it is declared, but not in any other file. 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`, so there is no possibility of conflict. If the two routines and the two variables are compiled in one file, as

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

getch() { ... }

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

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

Static storage, whether internal or external, is specified by prefixing the normal declaration with the word `static`. The variable is external if it is defined outside of any function, and internal if defined inside a function.

Normally, functions are external objects; their names are known globally. It is possible, however, for a function to be declared `static`; this makes its name unknown outside of the file in which it is declared.

In C, "`static`" connotes not only permanence but also a degree of what might be called "privacy." Internal `static` objects are known only inside one function; external `static` objects (variables or functions) are known only within the source file in which they appear, and their names do not interfere with variables or functions of the same name in other files.

External `static` variables and functions provide a way to conceal data objects and any internal routines that manipulate them so that other routines