

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 `get_line`, 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.

Recall that because the modern `stdio.h` library defines a `getline()` function, whenever the book writes this function to teach a feature of functions, we rename the function to `get_line()`.

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 `get_line`; you might find it instructive to compare it to the one in [Chapter 1](#).

```
#include <stdio.h>
#define MAXLINE 1000

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

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

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

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

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) ;
}
```

c_066_01

Copy

Edit

```
}  
}
```

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`, `get_line.c`, and `index.c`. Then the command

```
cc main.c get_line.c index.c
```

compiles the three files, places the resulting relocatable object code in files `main.o`, `get_line.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 get_line.o index.o
```

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

This `cc` example does not quite work as described above in modern C compilers. If you want the compile to leave the compiled object code around after the compile, you can add the `-c` option to the compiler call. Modern C compilers generally do accept multiple files with either `.c` or `.o` suffixes and combine them into a runnable application.

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 (get_line(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.

In the modern C language, you are required to provide a type for each function. If you leave off a type in a function declaration - at a minimum you will get a warning message.

But sometimes functions do not intend to return anything at all and so the `void` type was invented to indicate that a function returns nothing. We will talk more about the `void` type in the next chapter on pointers.

The rule of requiring a type on a function definition even if it is `void`, allows the compiler to check to make sure all of your `return` values in a function match the expected return type.

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);
}
```

c_069_01

Copy

Edit

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.

```
#include <stdio.h>
#define MAXLINE 100

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

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

c_070_01

Copy

Edit

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:

```
int 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.

Since including an array in an argument passes the *location* (or memory address) of the array into the function, the function can change the items in the array using array subscripts. In particular the array *contents* are not copied when an array is passed into a C function. When we get to `structs` in a future chapter - we will find that the content of `structs` also are passed using the address of the entire struct. So `structs` are passed by reference as well.

When thinking about pass by reference or pass by value, remember that a `char` variable is a single item similar to `int` and passed by value (i.e. copied). In C strings are arrays of characters so they are passed by reference.

Python follows this design decision for the same (efficiency) reason as C. Normal single variables like `int` or `float` are copied before being passed into a function and therefore are passed by value. Collections like `list` or `dict` are passed into functions by reference and so the contents can be changed within a function. Python strings are not copied when being passed into a function, but the way assignments happen in Python makes it seem like strings are passed by value (i.e. they cannot be modified). You can learn more with a bit of web research, but the easy way is to imagine that strings are passed by value in Python with a clever trick to avoid copying.

PHP follows the same pattern of passing numbers and strings by value and passing arrays as reference. PHP passes strings by value without requiring a copy using clever run-time

code.

Because in Java, JavaScript, and PHP strings are objects, those languages can make sure that strings act as if they were passed by value and not passed by reference the way they are passed in C.

C made decisions on run-time based on getting the maximum performance out of hardware in the 1970's at the expense of making it too easy to write code that overwrites memory and leads to corrupted programs that have dangerous and undefined behavior.

Languages like PHP, Java, and JavaScript add a small amount of run-time overhead to do things like store the length of an array and make sure we programmers don't over reference the array and overwrite random bits of our program's code or data.

The creators of C placed more priority on speed and efficient use of memory than safety. It is like driving an automobile in the rain without ABS (Automatic Braking System). It is fast but dangerous and should be reserved to be used by highly skilled and very careful programmers (and drivers).

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.

Interestingly modern languages like Python, PHP, and Java go to great lengths to make variable length argument lists work predictably and portably. The syntax for variable length argument lists in these languages can be a bit obtuse at times - but at least it is allowed, documented, reliable, and portable.

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](#).

```
#include <stdio.h>
#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)
```

c_074_01

Copy

Edit

```

        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;
}
}

```

```

#include <stdio.h>
#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;
}

```

c_075_01

Copy

Edit

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.

This example code above shows why it is important to remember the "K&R C Rearrangement License" as it applies to operators that are associative and commutative. If the code for the `'-'` operator above were written:

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

There is no guarantee the the left `pop()` will run before the right `pop()` - and since these functions access global variables and have side effects it is important to force the compiler not to rearrange the order of the function calls.

To force the evaluation order the code is broken into two statements:

```
op2 = pop();  
push(pop() - op2);
```

Now you might think that the lesson is that the "K&R C Rearrangement License" (which was done to allow optimization and performance) is a bad idea. But the more important lesson is that writing low-level utility functions like `push` and `pop` which use global variables and have side effects is a dangerous pattern in any language.

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 below, 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 `T00BIG` 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(T00BIG);
    }
}
```

c_078_01

Copy

Edit

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:

```
#include <stdio.h>
#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;
}
```

c_079_01

Copy

Edit

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 and data cannot conflict even inadvertently. For example, `getch` and `ungetch` form a "module" for character input and pushback; `buf` and `bufp` should be static so they are inaccessible from the outside. In the same way, `push`, `pop` and `clear` form a module for stack manipulation; `val` and `sp` should also be external static.

4.7 Register Variables

The fourth and final storage class is called `register`. A `register` declaration advises the compiler that the variable in question will be heavily used. When possible, `register` variables are placed in machine registers, which may result in smaller and faster programs.

The `register` declaration looks like

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


and so on; the `int` part may be omitted. `register` can only be applied to automatic variables and to the formal parameters of a function. In this latter case, the declaration looks like

```
f(c, n)
register int c, n;
{
    register int i;
    ...
}
```

In practice, there are some 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. 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](#)). The specific restrictions vary from machine to machine; as an example, on the PDP-11, only the first three register declarations in a function are effective, and the types must be `int`, `char`, or `pointer`.

This description of the details of the implementation of the `register` modifier on the PDP/11 is a delightful peek into how the C compiler generated run-time code on that system in the mid-1970's.

As compilers became more sophisticated, the compiler could decide which variables to keep in registers better than the programmer could. And since how variables would be allocated to registers might be different on different hardware architectures, the `register` indication is generally ignored by modern C compilers so you should never use it in your code.

As a matter of fact, I wrote the following sample C program and compiled it with the `-S` option so I could see the generated assembly language with and without the `register` declaration. There was no difference between the code generated between the two versions of the code.

The reason the generated assembly code was identical regardless of the use of the `register` keyword was that the C optimizer (on my ARM-based computer in 2022) realized that the best way to implement the code was to keep *both* `iter` and `compute` in registers because the loop code was so simple and the CPU in my computer had plenty of registers and optimized any loading and storing of the data memory right out of the program.

In 1978, the authors likely included `register` as a feature to convince experienced assembly language programmers that they should write all but the lowest level code in C.

```
#include <stdio.h>
```

c_081_01

Copy

Edit

```
/* Exploring the register keyword in modern compilers
 *
 * to compile on a Unix style system use the command:
 *
 * gcc -S c_081_01.c
 *
 * and compare the contents of the c_081_01.s files with and without
 * the register keyword.
 */
```

```

int main() {
    int compute;
    register int iter;

    scanf("%d", &compute);
    printf("compute %d\n", compute);
    for(iter=0; iter<1000; iter++) {
        compute = (compute * 22) / 7;
        if ( compute > 1000 ) compute = compute % 1000;
    }
    printf("compute %d\n", compute);
}

```

4.8 Block Structure

C is not a block-structured language in the sense of PL/I or Algol, in that functions may not be defined within other functions.

On the other hand, variables can be defined in a block-structured fashion. 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 supersede 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 other `i` in the program. Block structure also applies to external variables. Given the declarations

```

int x;

f() {
    double x;
    ...
}

```

then within the function `f`, occurrences of `x` refer to the internal double variable; outside of `f`, they refer to the external integer. The same is true of the names of formal parameters:

```

int z;

f(z)
double z;
{
    ...
}

```

Within the function `f`, `z` refers to the formal parameter, not the external.

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) values. Simple variables (not arrays or structures) may be initialized when they are declared, by following the name with an equals sign and a constant expression:

```
int x = 1;
char squote = '\'';
long day = 60 * 24; /* minutes in a day */
```

For external and static variables, the initialization is done once, conceptually at compile time. 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 in fact be any valid expression involving previously defined values, even function calls. For example, the initializations of the binary search program in [Chapter 3](#) could be written as

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

instead of

```
binary(x, v, n)
int x, v[], n;
{
    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.

Automatic arrays may not be initialized. External and static arrays may be initialized by following the declaration with a list of initializers enclosed in braces and separated by commas. For example, the character counting program of [Chapter 1](#), which began

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

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

can be written instead as

```
int nwhite = 0;
int nother = 0;
int ndigit[10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

main() /* count digits, white space, others */
{
    int c, i;
    ...
}
```

These initializations are actually unnecessary since all are zero, but it's good form to make them explicit anyway. If there are fewer initializers than the specified size, the others will be zero. It is an error to have too many initializers. Regrettably, 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 intervening 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[] = "the";
```

This is a shorthand for the longer but equivalent

```
char pattern[] = { 't', 'h', 'e', '\0' };
```

When the size of an array of any type is omitted, the compiler will compute the length by counting the initializers. In this specific case, the size is 4 (three characters plus the terminating '\0').

The primary difference between C and C-influenced languages like Java, PHP, and JavaScript, is that in C strings are character arrays while in the other languages, strings are "objects".

These string objects do have an array of the actual bytes / characters - but they also keep track of the length of a string and provide functionality like "extract a substring" in methods in those objects.

In C, there is a set of library functions that perform string operations like "compare two strings" while string comparison is built into the string objects in each language.

Strings as character arrays allows programmers to build very fast, low level code in libraries and operating systems. But to write the code well, you need to understand what is really going on at the low level.

4.10 Recursion

C functions may be used recursively; that is, a function may call *itself* either directly or indirectly. One traditional example involves 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 in [Chapter 3](#) with `itoa`. The first version of `printd` follows this pattern.

```
#include <stdio.h>

printd(n) /* print n in decimal */
int n;
{
    char s[10];
    int i;

    if (n < 0) {
        putchar('-');
        n = -n;
    }
    i = 0;
    do {
        s[i++] = n % 10 + '0'; /* get next char */
    } while ((n /= 10) > 0); /* discard it */
    while (--i >= 0)
        putchar(s[i]);
}
```

c_085_01

Copy

Edit

The alternative is a recursive solution, in which each call of `printd` first calls itself to cope with any leading digits, then prints the trailing digit.

```
#include <stdio.h>

printd(n) /* print n in decimal (recursive) */
int n;
{
    int i;

    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if ((i = n/10) != 0)
        printd(i);
    putchar(n % 10 + '0');
}
```

c_085_02

Copy

Edit

When a function calls itself recursively, each invocation gets a fresh set of all the automatic variables, quite independent of the previous set. Thus in `printd(123)` the first `printd` has `n = 123`. It passes 12 to a second `printd`, then prints 3 when that one returns. In the same way, the second `printd` passes 1 to a third (which prints it), then prints 2.

Recursion generally provides no saving in storage, since somewhere a stack of the values being processed has to be maintained. Nor will it be faster. But recursive code is more compact, and often much easier to write and understand. Recursion is especially convenient for recursively defined data structures like trees; we will see a nice example in [Chapter 6](#).

Ah recursion. Recursion is a beloved concept in Computer Science. It is often taught early in most programming courses because it is "so cool". Most examples like computing factorial or the example above of converting an integer to a string are not good uses of recursion.

But when you need to traverse a tree-based structure like an XML document, or parse a mathematical expression, recursion is the ideal solution. So the problem is not recursion, but when it is taught and what examples are used.

Interestingly, Kernighan and Ritchie include the correct warning about using recursion when it is not the best solution in the above text - it bears another read.

Exercise 4-7. Adapt the ideas of `printd` to write a recursive version of `itoa`; that is, convert an integer into a string with a recursive routine.

Exercise 4-8. Write a recursive version of the function `reverse(s)`, which reverses the string `s`.

4.11 The C Preprocessor

C provides certain language extensions by means of a simple macro preprocessor. The `#define` capability which we have used is the most common of these extensions; another is the ability to include the contents of other files during compilation.

File Inclusion

To facilitate handling collections of `#define`'s and declarations (among other things) C provides a file inclusion feature. Any line that looks like

```
#include "filename"
```

is replaced by the contents of the file *filename*. (The quotes are mandatory.) Often a line or two of this form appears at the beginning of each source file, to include common `#define` statements and `extern` declarations for global variables. `#include`'s may be nested. `#include` is the preferred way to tie the declarations together for a large program. It guarantees that all the source files will be supplied with the same definitions and variable declarations, and thus eliminates a particularly nasty kind of bug. Of course, when an included file is changed, all files that depend on it must be recompiled.

Macro Substitution

A definition of the form

```
#define YES 1
```

calls for a macro substitution of the simplest kind - replacing a name by a string of characters. Names in `#define` have the same form as C identifiers; the replacement text is arbitrary. Normally the replacement text is the rest of the line; a long definition may be continued by placing a `\` at the end of the line to be continued. The "scope" of a name defined with `#define` is from its point of definition to the end of the source file. Names may be redefined, and a definition may use previous definitions. Substitutions do not take place within quoted strings, so, for example, if `YES` is a defined name, there would be no substitution in `printf ("YES")`.

Since implementation of `#define` is a macro prepass, not part of the compiler proper, there are very few grammatical restrictions on what can be defined. For example, Algol fans can say

```
#define then
#define begin {
#define end ; }
```

and then write

```
if (i > 0) then
  begin
    a = 1;
    b = 2
  end
```

It is also possible to define macros with arguments, so the replacement text depends on the way the macro is called. As an example, define a macro called `max` like this:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Now the line

```
x = max(p+q, r+s);
```

will be replaced by the line

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

This provides a "maximum function" that expands into in-line code rather than a function call. So long as the arguments are treated consistently, this macro will serve for any data type; there is no need for different kinds of `max` for different data types, as there would be with functions.

Of course, if you examine the expansion of `max` above, you will notice some pitfalls. The expressions are evaluated twice; this is bad if they involve side effects like function calls and increment operators. Some care has to be taken with parentheses to make sure the order of evaluation is preserved. (Consider the macro

```
#define square(x) x * x
```

when invoked as `square(z+1)` .) There are even some purely lexical problems: there can be no space between the macro name and the left parenthesis that introduces its argument list.

Nonetheless, macros are quite valuable. One practical example is the standard I/O library to be described in [Chapter 7](#), in which `getchar` and `putchar` are defined as macros (obviously `putchar` needs an argument), thus avoiding the overhead of a function call per character processed.

Other capabilities of the macro processor are described in Appendix A.

In this section we are talking about the "pre-processor". It is probably a good time to talk a bit about why we use this terminology. For those of you with a Computer Science degree from "back in the day", many of you wrote a compiler as a senior project (just like I did). Building a compiler was a great project because part of the goal of Computer Science is to understand the technologies that make programming possible from the language syntax

down to the hardware. The compiler that translates our source code into machine code is an essential part of that technology stack.

Early compilers for languages like the early FORTRAN variants tended to be "translators" - they translated code one line at a time from the high level language to the assembly language. You could think of early FORTRAN programs in the 1950's and 1960's as just more convenient ways to write assembly language for programmers that knew assembly language. You needed to be aware of assembly language and the translation that was going on to write fast FORTRAN. Programs were small and optimization was done at the FORTRAN level, often leading to some hard-to-understand code.

By the mid 1970's programming languages were based on parsing theory and we used what is called a "grammar" to define a language. Kernighan and Ritchie kept I/O statements out of the C language to keep its formal description (i.e. its grammar) as simple as possible. As these new languages emerged, they allowed for a more theoretical and powerful approach to converting source code to machine language.

The theoretical advances in compiler and language design meant that parts of a compiler might be reusable across multiple programming languages - each language would have its own syntax and "grammar rules" and they would be plugged into a compiler and "poof" you had a new programming language.

It got to the point where UNIX systems had a tool called 'yacc' which stood for "Yet Another Compiler Compiler" - you would give it a grammar for your "new" language and it would make a compiler for you. As a matter of fact, the JavaScript language was created in 10 days back in 1995 because Brendan Eich had a lot of experience with "compiler generators". He defined a grammar for JavaScript and generated his first compiler.

Part of what made a "compiler generator" possible was the idea of a multi-step compiler - where the tasks of a compiler were broken down into a series of simpler and more well defined steps. Here are the steps of a typical C compiler in the 1970's:

- A preprocessor step that takes C code with syntax like `#define` and `#include` as its input and produces raw C code output with those instructions processed. The preprocessor was a C to C transformation.
- A parser step that took the raw C code, applied the grammar to the language and created what is called a "parse tree". Think of the tree as a hierarchy of statements, grouped into blocks, grouped into functions etc. - Things like loops are just nodes in the parse tree.
- A code generation step that turns the parse tree into some kind of simplistic internal code that expanded things like loops and if-then-else statements into code
- A code optimization step that looked at the internal code, and moved things around and eliminated any redundant computations (i.e. don't compute the same thing twice). This step is why the authors make such a big fuss about how there are times where C

might do things in a slightly different order in an expression even in the presence of parenthesis. Remember the "K&R C Rearrangement License" back in Chapter 2? That rule removes constraints on the compiler's optimization step so it can generate the most efficient code.

- I would note that all of the steps up to this point do not depend in any way on the actual machine language of the system they were running on. This meant that a preprocessor, parser, code generator, and code optimizer could be written in C and used on any architecture.
- A "code generator" step takes the optimized intermediate code and generates the actual assembly and then machine language for the processor. For fun, you can add the `-s` parameter to your C compiler and see the resulting assembly language output for your system.

If you look at the machine language generated on an Intel or AMD processor and compare it to the machine language on an ARM processor, it will look very different.

Because all but the final compiler steps above do not depend on the computer where the program is being run, you could create a C compiler on a new computer architecture by writing a code generator on the new computer, then running all but the last step of the compiler on one computer, then copying the internal code generated by the compiler to the new computer and running the code generation step. Then you have a working C compiler on the new computer and can re-compile the C compiler itself from source code and produce a fully-native C compiler on the new computer that can compile all of the rest of the C code (including possibly the portable elements of UNIX) on the new computer.

Yes - describing how to cross-compile and bootstrap a C compiler onto a new computer hardware architecture can give you a headache if you think about it too much. But this notion of bootstrapping a C compiler onto a new architecture was an important technique to move C and then UNIX to a wide range of very different computer architectures. We see this in action as the UNIX-like MacOS operating system over the past 20 years was delivered initially on Motorola processors, then on PowerPC processors, then on Intel processors and most recently on ARM-based processors built by Apple. Using the software portability patterns that come from C and UNIX and described by Kernighan and Ritchie in this book, Apple now makes their own hardware that can be tuned and evolved over time as their operating system and application requirements dictate.

The use of a grammar to define a programming language is one of the reasons that syntax errors are so obtuse. The compiler is not looking at your code like a human - it is just following a set of rules to parse your code and it is stuck with something illogical and gives you a message like "Unexpected statement, block or constant on line 17" and the error is nowhere near line 17.

Modern compilers are more sophisticated than the steps above - but these steps give you a sense that a compiler does many things to make it so your code can actually run efficiently.

And given that Kernighan and Ritchie were building a programming language C, a mostly

portable operating system written in C (UNIX), and a mostly portable C compiler written in C - some of their innovative work and research into compiler design finds its way into this book and so we have a section in this chapter called "The C Pre Processor".

Here at the end of Chapter 4, it is a good time to talk about the word "address". Up to this point in the book, the word "address" has been used ten times without a precise definition beyond the notion that data is stored in memory and the "address" of the data is *where* the data is stored in the memory.

In the next chapter, this notion of "the address of where the data is stored" becomes very real and tangible as we explore "pointers" as well the `&` and `*` operators.

Up to now, an experienced JavaScript, PHP or Java programmer can view C as "just another set of syntax rules" with a few quirky run-time bits. But in the next chapter we will deeply explore the concept of data allocation and location.

It turns out that every programming language pays a lot of attention to data allocation and location, but the run-time environments of modern languages work very hard not to expose you to those details. Just because modern languages hide the "difficult bits" from us, it does not mean that those languages solve the problem using "magic" - eventually the problem needs to be solved.

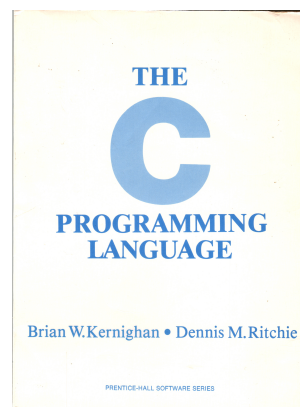
And that is why the compilers and low-level run-time elements of languages like PHP, JavaScript, and Java are usually written in C. So the builders of those languages can solve difficult data storage and allocation problems for you.

Exercise 4-9. Define a macro `swap(x, y)` which interchanges its two `int` arguments. (Block structure will help.)

This work (www.cc4e.com) is based on the 1978 "[The C Programming Language](#)" book written by Brian W. Kernighan and Dennis M. Ritchie. Their book is copyright all rights reserved by AT&T but is being used in this work under "fair use" because of the book's historical and scholarly significance, its lack of availability, and lack of an accessible version of the book.

The book is augmented in places to help understand its rightful place in a historical context amidst the major changes of the 1970's and 1980's as Computer Science evolved from a hardware-first / vendor-centered approach to a software-centered approach where portable operating systems and applications written in C could run on any hardware.

This is not the ideal book to learn C programming because the 1978 edition does not reflect the modern C language. Using an obsolete book gives us an opportunity to take students back in time and understand how the C language was evolving as it laid the ground work for a future with portable applications.



If you want to learn modern C programming from a book that reflects the modern C language, I suggest you use the [C Programming Language, 2nd Edition](#), also written by Brian W. Kernighan and Dennis M. Ritchie, initially published in 1988 and based on the [ANSI C](#) (C89) version of the language.

The source code to this book is at <https://github.com/csev/cc4e>. You are welcome to help in the creation and editing of the book.