CHAPTER 4: Functions and Program Structure

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

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

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

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

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

4.1 Basics of Functions

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

letters "ould" in the set of lines

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

will produce the output

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

The job falls neatly into three pieces:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Each function definition has the form

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

Various parts may be absent; a minimal function is

```
dummy() {}
```

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

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

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

```
return expression;
```

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

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

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

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

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

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

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

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

source files from object files.

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

4.2 Functions Returning Non-integers

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

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

```
#include <ctype.h>

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

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

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

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

```
#include <stdio.h>

#define MAXLINE 100

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

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

The declaration

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

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

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

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

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

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

```
double atof();
```

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

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

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

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

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

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

```
return expression;
```

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

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

```
123.45e-6
```

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

4.3 External Variables

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

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

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

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

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

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

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

is entered as

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

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

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

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

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

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

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

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

```
#includes
#defines

function declarations for main

main() { ... }

external variables for push and pop

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

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

routines called by getop
```

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

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

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

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

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

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

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

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

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

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

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

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

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



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

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

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

```
#include <ctype.h>

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

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

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

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

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

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

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

```
#define BUFSIZE 100

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

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

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

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

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

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

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

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

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

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

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

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

4.4 Scope Rules

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

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

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

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

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

```
main() { ... }

int sp = 0;
double val[MAXVAL];

void push(double f) { ... }

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

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

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

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

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

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

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

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

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

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

Initialization of an external variable goes only with the definition.

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

In file1:

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

void push(double f) { ... }

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

In file2:

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

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

4.5 Header Files

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

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

`calc.h`:

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

`main.c`:

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

`getop.c`:

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

`stack.c`:

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

`getch.c`:

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

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

4.6 Static Variables

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

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

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

int getch(void) { ... }

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

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

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

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

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

4.7 Register Variables

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

The `register` declaration looks like

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

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

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

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

4.8 Block Structure

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

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

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

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

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

```
int x;
int y;

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

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

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

4.9 Initialization

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

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

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

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

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

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

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

instead of

```
    int low, high, mid;

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

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

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

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

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

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

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

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

is a shorthand for the longer but equivalent

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

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

4.10 Recursion

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

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

```
#include <stdio.h>

/* printd:  print n in decimal */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

When a function calls itself recursively, each invocation gets a fresh set of all the automatic variables, independent of the previous set. Thus in `printd(123)` the first `printd` receives the argument `n = 123`. It passes `12` to a second `printd`, which in turn passes `1` to a third. The third-level `printd` prints `1`, then returns to the second level. That `printd` prints `2`, then returns to the first level. That one prints `3` and terminates.

Another good example of recursion is quicksort, a sorting algorithm developed by C. A. R. Hoare in 1962. Given an array, one element is chosen and the others are partitioned into two subsets—those less than the partition element and those greater than or equal to it. The same process is then applied recursively to the two subsets. When a subset has fewer than two elements, it doesn't need any sorting; this stops the recursion.

Our version of quicksort is not the fastest possible, but it's one of the simplest. We use the middle element of each subarray for partitioning.

```
/* qsort:  sort v[left]...v[right] into increasing order */
void qsort(int v[], int left, int right)
{
    int i, last;
    void swap(int v[], int i, int j);

    if (left >= right)    /* do nothing if array contains */
        return;           /* fewer than two elements */
    swap(v, left, (left + right)/2); /* move partition elem */
    last = left;                     /* to v[0] */
    for (i = left+1; i <= right; i++)    /* partition */
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last);          /* restore partition elem */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

We moved the swapping operation into a separate function `swap` because it occurs three times in `qsort`.

```
/* swap:  interchange v[i] and v[j] */
void swap(int v[], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

The standard library includes a version of `qsort` that can sort objects of any type.

Recursion may provide no saving in storage, since somewhere a stack of the values being processed must be maintained. Nor will it be faster. But recursive code is more compact, and often much easier to write and understand than the non-recursive equivalent. Recursion is especially convenient for recursively defined data structures like trees; we will see a nice example in Section 6.5.

Exercise 4-12. Adapt the ideas of `printd` to write a recursive version of `itoa`; that is, convert an integer into a string by calling a recursive routine. □

Exercise 4-13. Write a recursive version of the function `reverse(s)`, which reverses the string `s` in place. □

4.11 The C Preprocessor

C provides certain language facilities by means of a preprocessor, which is conceptually a separate first step in compilation. The two most frequently used features are `#include`, to include the contents of a file during compilation, and `#define`, to replace a token by an arbitrary sequence of characters. Other features described in this section include conditional compilation and macros with arguments.

4.11.1 File Inclusion

File inclusion makes it easy to handle collections of `#define`s and declarations (among other things). Any source line of the form

```
#include "filename"
```

or

```
#include <filename>
```

is replaced by the contents of the file *filename*. If the *filename* is quoted, searching for the file typically begins where the source program was found; if it is not found there, or if the name is enclosed in < and >, searching follows an implementation-defined rule to find the file. An included file may itself contain

`#include` lines.

There are often several `#include` lines at the beginning of a source file, to include common `#define` statements and `extern` declarations, or to access the function prototype declarations for library functions from headers like `<stdio.h>`. (Strictly speaking, these need not be files; the details of how headers are accessed are implementation-dependent.)

`#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. Naturally, when an included file is changed, all files that depend on it must be recompiled.

4.11.2 Macro Substitution

A definition has the form

```
#define  name  replacement text
```

It calls for a macro substitution of the simplest kind—subsequent occurrences of the token *name* will be replaced by the *replacement text*. The name in a `#define` has the same form as a variable name; the replacement text is arbitrary. Normally the replacement text is the rest of the line, but a long definition may be continued onto several lines by placing a \ at the end of each 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 being compiled. A definition may use previous definitions. Substitutions are made only for tokens, and do not take place within quoted strings. For example, if `YES` is a defined name, there would be no substitution in `printf("YES")` or in `YESMAN`.

Any name may be defined with any replacement text. For example,

```
#define  forever  for (;;)   /* infinite loop */
```

defines a new word, `forever`, for an infinite loop.

It is also possible to define macros with arguments, so the replacement text can be different for different calls of the macro. As an example, define a macro called `max`:

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

Although it looks like a function call, a use of `max` expands into in-line code. Each occurrence of a formal parameter (here `A` or `B`) will be replaced by the corresponding actual argument. Thus the line

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

will be replaced by the line

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

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.

If you examine the expansion of `max`, you will notice some pitfalls. The expressions are evaluated twice; this is bad if they involve side effects like increment operators or input and output. For instance,

```
max(i++, j++)    /* WRONG */
```

will increment the larger value twice. Some care also has to be taken with parentheses to make sure the order of evaluation is preserved; consider what happens when the macro

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

is invoked as `square(z+1)`.

Nonetheless, macros are valuable. One practical example comes from `<stdio.h>`, in which `getchar` and `putchar` are often defined as macros to avoid the run-time overhead of a function call per character processed. The functions in `<ctype.h>` are also usually implemented as macros.

Names may be undefined with `#undef`, usually to ensure that a routine is really a function, not a macro:

```
#undef getchar

int getchar(void) { ... }
```

Formal parameters are not replaced within quoted strings. If, however, a parameter name is preceded by a # in the replacement text, the combination will be expanded into a quoted string with the parameter replaced by the actual argument. This can be combined with string concatenation to make, for example, a debugging print macro:

```
#define  dprint(expr)  printf(#expr " = %g\n", expr)
```

When this is invoked, as in

```
dprint(x/y);
```

the macro is expanded into

```
printf("x/y" " = %g\n", x/y);
```

and the strings are concatenated, so the effect is

```
printf("x/y = %g\n", x/y);
```

Within the actual argument, each " is replaced by \" and each \ by \\, so the result is a legal string constant.

The preprocessor operator ## provides a way to concatenate actual arguments during macro expansion. If a parameter in the replacement text is adjacent to a ##, the parameter is replaced by the actual argument, the ## and surrounding white space are removed, and the result is re-scanned. For example, the macro `paste` concatenates its two arguments:

```
#define  paste(front, back)  front ## back
```

so `paste(name, 1)` creates the token `name1`.

The rules for nested uses of `##` are arcane; further details may be found in Appendix A.

Exercise 4-14. Define a macro `swap(t,x,y)` that interchanges two arguments of type `t`. (Block structure will help.) □

4.11.3 Conditional Inclusion

It is possible to control preprocessing itself with conditional statements that are evaluated during preprocessing. This provides a way to include code selectively, depending on the value of conditions evaluated during compilation.

The `#if` line evaluates a constant integer expression (which may not include `sizeof`, casts, or `enum` constants). If the expression is non-zero, subsequent lines until an `#endif` or `#elif` or `#else` are included. (The preprocessor statement `#elif` is like `else if`.) The expression `defined(`*name*`)` in a `#if` is 1 if the *name* has been defined, and 0 otherwise.

For example, to make sure that the contents of a file `hdr.h` are included only once, the contents of the file are surrounded with a conditional like this:

```
#if !defined(HDR)
#define HDR

/* contents of hdr.h go here */

#endif
```

The first inclusion of `hdr.h` defines the name `HDR`; subsequent inclusions will find the name defined and skip down to the `#endif`. A similar style can be used to avoid including files multiple times. If this style is used consistently, then each header can itself include any other headers on which it depends, without the user of the header having to deal with the interdependence.

This sequence tests the name `SYSTEM` to decide which version of a header to include:

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```

The `#ifdef` and `#ifndef` lines are specialized forms that test whether a

name is defined. The first example of `#if` above could have been written

```
#ifndef HDR
#define HDR

/* contents of hdr.h go here */

#endif
```

CHAPTER 5: Pointers and Arrays

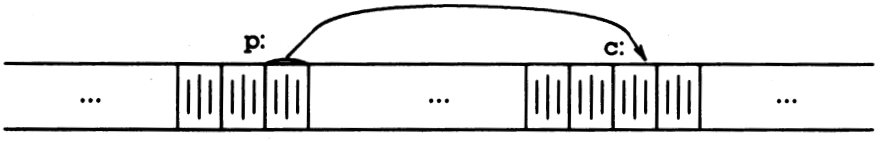
A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways. Pointers and arrays are closely related; this chapter also explores this relationship and shows how to exploit it.

Pointers have been lumped with the `goto` statement as a marvelous way to create impossible-to-understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can also be used to achieve clarity and simplicity. This is the aspect that we will try to illustrate.

The main change in ANSI C is to make explicit the rules about how pointers can be manipulated, in effect mandating what good programmers already practice and good compilers already enforce. In addition, the type `void *` (pointer to `void`) replaces `char *` as the proper type for a generic pointer.

5.1 Pointers and Addresses

Let us begin with a simplified picture of how memory is organized. A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups. One common situation is that any byte can be a `char`, a pair of one-byte cells can be treated as a `short` integer, and four adjacent bytes form a `long`. A pointer is a group of cells (often two or four) that can hold an address. So if `c` is a `char` and `p` is a pointer that points to it, we could represent the situation this way:



The unary operator `&` gives the address of an object, so the statement

```
p = &c;
```

assigns the address of c to the variable p, and p is said to "point to" c. The & operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or `register` variables.

The unary operator * is the *indirection* or *dereferencing* operator; when applied to a pointer, it accesses the object the pointer points to. Suppose that x and y are integers and ip is a pointer to `int`. This artificial sequence shows how to declare a pointer and how to use & and *:

```
int x = 1, y = 2, z[10];
int *ip;          /* ip is a pointer to int */

ip = &x;          /* ip now points to x */
y = *ip;          /* y is now 1 */
*ip = 0;          /* x is now 0 */
ip = &z[0];       /* ip now points to z[0] */
```

The declarations of x, y, and z are what we've seen all along. The declaration of the pointer ip,

```
int *ip;
```

is intended as a mnemonic; it says that the expression `*ip` is an `int`. The syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear. This reasoning applies to function declarations as well. For example,

```
double *dp, atof(char *);
```

says that in an expression `*dp` and `atof(s)` have values of type `double`, and that the argument of `atof` is a pointer to `char`.

You should also note the implication that a pointer is constrained to point to a particular kind of object: every pointer points to a specific data type. (There is one exception: a "pointer to `void`" is used to hold any type of pointer but cannot be dereferenced itself. We'll come back to it in Section 5.11.)

If ip points to the integer x, then `*ip` can occur in any context where x could, so

```
*ip = *ip + 10;
```

increments `*ip` by 10.

The unary operators * and & bind more tightly than arithmetic operators, so the assignment

```
y = *ip + 1
```

takes whatever ip points at, adds 1, and assigns the result to y, while

```
*ip += 1
```

increments what ip points to, as do

```
++*ip
```

and

```
(*ip)++
```

The parentheses are necessary in this last example; without them, the expression would increment `ip` instead of what it points to, because unary operators like `*` and `++` associate right to left.

Finally, since pointers are variables, they can be used without dereferencing. For example, if `iq` is another pointer to `int`,

```
iq = ip
```

copies the contents of `ip` into `iq`, thus making `iq` point to whatever `ip` pointed to.

5.2 Pointers and Function Arguments

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two out-of-order elements with a function called `swap`. It is not enough to write

```
swap(a, b);
```

where the `swap` function is defined as

```
void swap(int x, int y)  /* WRONG */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Because of call by value, `swap` can't affect the arguments `a` and `b` in the routine that called it. The function above only swaps *copies* of `a` and `b`.

The way to obtain the desired effect is for the calling program to pass *pointers* to the values to be changed:

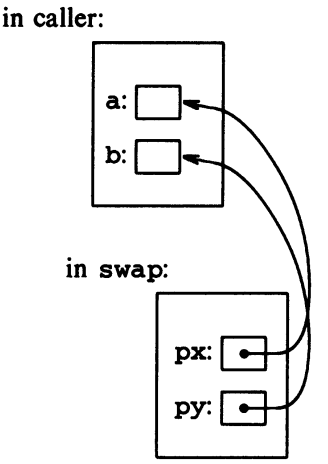
```
swap(&a, &b);
```

Since the operator `&` produces the address of a variable, `&a` is a pointer to `a`. In `swap` itself, the parameters are declared to be pointers, and the operands are accessed indirectly through them.

```
void swap(int *px, int *py)  /* interchange *px and *py */
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

Pictorially:



Pointer arguments enable a function to access and change objects in the function that called it. As an example, consider a function `getint` that performs free-format input conversion by breaking a stream of characters into integer values, one integer per call. `getint` has to return the value it found and also signal end of file when there is no more input. These values have to be passed back by separate paths, for no matter what value is used for `EOF`, that could also be the value of an input integer.

One solution is to have `getint` return the end of file status as its function value, while using a pointer argument to store the converted integer back in the calling function. This is the scheme used by `scanf` as well; see Section 7.4.

The following loop fills an array with integers by calls to `getint`:

```
int n, array[SIZE], getint(int *);

for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++)
    ;
```

Each call sets `array[n]` to the next integer found in the input and increments `n`. Notice that it is essential to pass the address of `array[n]` to `getint`. Otherwise there is no way for `getint` to communicate the converted integer back to the caller.

Our version of `getint` returns `EOF` for end of file, zero if the next input is not a number, and a positive value if the input contains a valid number.

```
#include <ctype.h>

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

/* getint:  get next integer from input into *pn */
int getint(int *pn)
{
    int c, sign;

    while (isspace(c = getch()))    /* skip white space */
        ;
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c);     /* it's not a number */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c); c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}
```

Throughout `getint`, `*pn` is used as an ordinary `int` variable. We have also used `getch` and `ungetch` (described in Section 4.3) so the one extra character that must be read can be pushed back onto the input.

Exercise 5-1. As written, `getint` treats a + or - not followed by a digit as a valid representation of zero. Fix it to push such a character back on the input. □

Exercise 5-2. Write `getfloat`, the floating-point analog of `getint`. What type does `getfloat` return as its function value? □

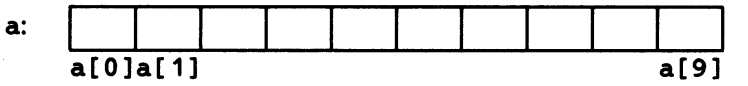
5.3 Pointers and Arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.

The declaration

```
int a[10];
```

defines an array a of size 10, that is, a block of 10 consecutive objects named a[0], a[1], ..., a[9].



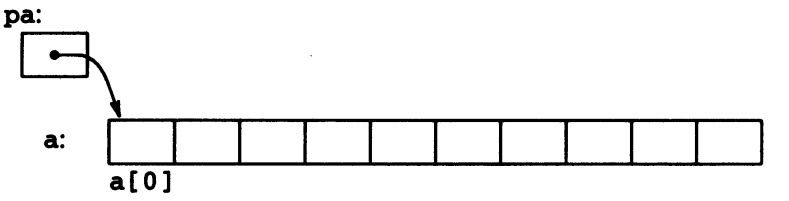
The notation a[i] refers to the i-th element of the array. If pa is a pointer to an integer, declared as

```
int *pa;
```

then the assignment

```
pa = &a[0];
```

sets pa to point to element zero of a; that is, pa contains the address of a[0].



Now the assignment

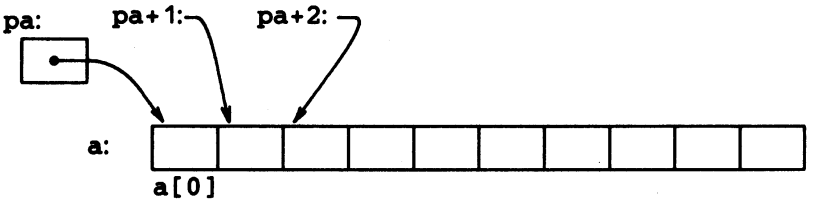
```
x = *pa;
```

will copy the contents of a[0] into x.

If pa points to a particular element of an array, then by definition pa+1 points to the next element, pa+i points i elements after pa, and pa-i points i elements before. Thus, if pa points to a[0],

```
*(pa+1)
```

refers to the contents of a[1], pa+i is the address of a[i], and *(pa+i) is the contents of a[i].



These remarks are true regardless of the type or size of the variables in the array a. The meaning of "adding 1 to a pointer," and by extension, all pointer arithmetic, is that pa+1 points to the next object, and pa+i points to the i-th

object beyond pa.

The correspondence between indexing and pointer arithmetic is very close. By definition, the value of a variable or expression of type array is the address of element zero of the array. Thus after the assignment

```
pa = &a[0];
```

pa and a have identical values. Since the name of an array is a synonym for the location of the initial element, the assignment pa=&a[0] can also be written as

```
pa = a;
```

Rather more surprising, at least at first sight, is the fact that a reference to a[i] can also be written as *(a+i). In evaluating a[i], C converts it to *(a+i) immediately; the two forms are equivalent. Applying the operator & to both parts of this equivalence, it follows that &a[i] and a+i are also identical: a+i is the address of the i-th element beyond a. As the other side of this coin, if pa is a pointer, expressions may use it with a subscript; pa[i] is identical to *(pa+i). In short, an array-and-index expression is equivalent to one written as a pointer and offset.

There is one difference between an array name and a pointer that must be kept in mind. A pointer is a variable, so pa=a and pa++ are legal. But an array name is not a variable; constructions like a=pa and a++ are illegal.

When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address. We can use this fact to write another version of strlen, which computes the length of a string.

```
/* strlen:  return length of string s */
int strlen(char *s)
{
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

Since s is a pointer, incrementing it is perfectly legal; s++ has no effect on the character string in the function that called strlen, but merely increments strlen's private copy of the pointer. That means that calls like

```
strlen("hello, world");  /* string constant */
strlen(array);           /* char array[100]; */
strlen(ptr);             /* char *ptr; */
```

all work.

As formal parameters in a function definition,

```
char s[];
```

and

```
char *s;
```

are equivalent; we prefer the latter because it says more explicitly that the parameter is a pointer. When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or a pointer, and manipulate it accordingly. It can even use both notations if it seems appropriate and clear.

It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray. For example, if `a` is an array,

```
f(&a[2])
```

and

```
f(a+2)
```

both pass to the function `f` the address of the subarray that starts at `a[2]`. Within `f`, the parameter declaration can read

```
f(int arr[]) { ... }
```

or

```
f(int *arr) { ... }
```

So as far as `f` is concerned, the fact that the parameter refers to part of a larger array is of no consequence.

If one is sure that the elements exist, it is also possible to index backwards in an array; `p[-1]`, `p[-2]`, and so on are syntactically legal, and refer to the elements that immediately precede `p[0]`. Of course, it is illegal to refer to objects that are not within the array bounds.

5.4 Address Arithmetic

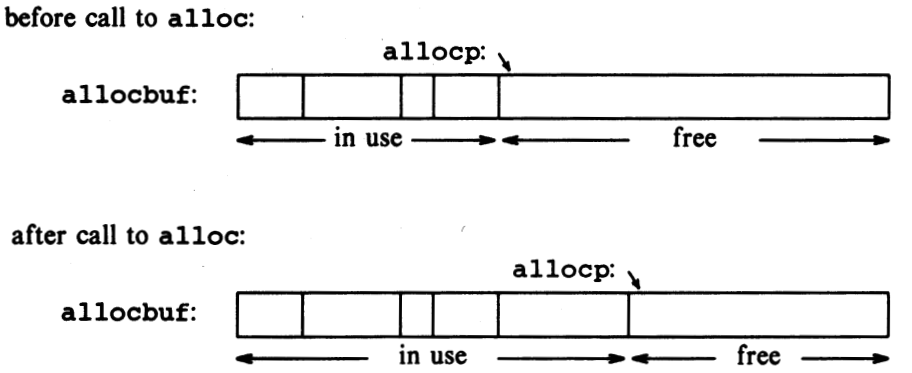
If `p` is a pointer to some element of an array, then `p++` increments `p` to point to the next element, and `p+=i` increments it to point `i` elements beyond where it currently does. These and similar constructions are the simplest forms of pointer or address arithmetic.

C is consistent and regular in its approach to address arithmetic; its integration of pointers, arrays, and address arithmetic is one of the strengths of the language. Let us illustrate by writing a rudimentary storage allocator. There are two routines. The first, `alloc(n)`, returns a pointer `p` to `n` consecutive character positions, which can be used by the caller of `alloc` for storing characters. The second, `afree(p)`, releases the storage thus acquired so it can be re-used later. The routines are "rudimentary" because the calls to `afree` must be made in the opposite order to the calls made on `alloc`. That is, the storage

managed by `alloc` and `afree` is a stack, or last-in, first-out list. The standard library provides analogous functions called `malloc` and `free` that have no such restrictions; in Section 8.7 we will show how they can be implemented.

The easiest implementation is to have `alloc` hand out pieces of a large character array that we will call `allocbuf`. This array is private to `alloc` and `afree`. Since they deal in pointers, not array indices, no other routine need know the name of the array, which can be declared `static` in the source file containing `alloc` and `afree`, and thus be invisible outside it. In practical implementations, the array may well not even have a name; it might instead be obtained by calling `malloc` or by asking the operating system for a pointer to some unnamed block of storage.

The other information needed is how much of `allocbuf` has been used. We use a pointer, called `allocp`, that points to the next free element. When `alloc` is asked for `n` characters, it checks to see if there is enough room left in `allocbuf`. If so, `alloc` returns the current value of `allocp` (i.e., the beginning of the free block), then increments it by `n` to point to the next free area. If there is no room, `alloc` returns zero. `afree(p)` merely sets `allocp` to `p` if `p` is inside `allocbuf`.




```
#define ALLOCSIZE 10000 /* size of available space */

static char allocbuf[ALLOCSIZE];  /* storage for alloc */
static char *allocp = allocbuf;   /* next free position */

char *alloc(int n)  /* return pointer to n characters */
{
    if (allocbuf + ALLOCSIZE - allocp >= n) { /* it fits */
        allocp += n;
        return allocp - n; /* old p */
    } else        /* not enough room */
        return 0;
}
```

```
void afree(char *p)  /* free storage pointed to by p */
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}
```

In general a pointer can be initialized just as any other variable can, though normally the only meaningful values are zero or an expression involving the addresses of previously defined data of appropriate type. The declaration

```
static char *allocp = allocbuf;
```

defines `allocp` to be a character pointer and initializes it to point to the beginning of `allocbuf`, which is the next free position when the program starts. This could have also been written

```
static char *allocp = &allocbuf[0];
```

since the array name *is* the address of the zeroth element.

The test

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* it fits */
```

checks if there's enough room to satisfy a request for `n` characters. If there is, the new value of `allocp` would be at most one beyond the end of `allocbuf`. If the request can be satisfied, `alloc` returns a pointer to the beginning of a block of characters (notice the declaration of the function itself). If not, `alloc` must return some signal that no space is left. C guarantees that zero is never a valid address for data, so a return value of zero can be used to signal an abnormal event, in this case, no space.

Pointers and integers are not interchangeable. Zero is the sole exception: the constant zero may be assigned to a pointer, and a pointer may be compared with the constant zero. The symbolic constant `NULL` is often used in place of zero, as a mnemonic to indicate more clearly that this is a special value for a pointer. `NULL` is defined in `<stdio.h>`. We will use `NULL` henceforth.

Tests like

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* it fits */
```

and

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

show several important facets of pointer arithmetic. First, pointers may be compared under certain circumstances. If `p` and `q` point to members of the same array, then relations like `==`, `!=`, `<`, `>=`, etc., work properly. For example,

```
p < q
```

is true if `p` points to an earlier member of the array than `q` does. Any pointer can be meaningfully compared for equality or inequality with zero. But the behavior is undefined for arithmetic or comparisons with pointers that do not

point to members of the same array. (There is one exception: the address of the first element past the end of an array can be used in pointer arithmetic.)

Second, we have already observed that a pointer and an integer may be added or subtracted. The construction

```
p + n
```

means the address of the n-th object beyond the one p currently points to. This is true regardless of the kind of object p points to; n is scaled according to the size of the objects p points to, which is determined by the declaration of p. If an `int` is four bytes, for example, the `int` will be scaled by four.

Pointer subtraction is also valid: if p and q point to elements of the same array, and p<q, then q-p+1 is the number of elements from p to q inclusive. This fact can be used to write yet another version of `strlen`:

```
/* strlen:  return length of string s */
int strlen(char *s)
{
    char *p = s;

    while (*p != '\0')
        p++;
    return p - s;
}
```

In its declaration, p is initialized to s, that is, to point to the first character of the string. In the `while` loop, each character in turn is examined until the `'\0'` at the end is seen. Because p points to characters, p++ advances p to the next character each time, and p-s gives the number of characters advanced over, that is, the string length. (The number of characters in the string could be too large to store in an `int`. The header <stddef.h> defines a type `ptrdiff_t` that is large enough to hold the signed difference of two pointer values. If we were being very cautious, however, we would use `size_t` for the return type of `strlen`, to match the standard library version. `size_t` is the unsigned integer type returned by the `sizeof` operator.)

Pointer arithmetic is consistent: if we had been dealing with `floats`, which occupy more storage than `chars`, and if p were a pointer to `float`, p++ would advance to the next `float`. Thus we could write another version of `alloc` that maintains `floats` instead of `chars`, merely by changing `char` to `float` throughout `alloc` and `afree`. All the pointer manipulations automatically take into account the size of the object pointed to.

The valid pointer operations are assignment of pointers of the same type, adding or subtracting a pointer and an integer, subtracting or comparing two pointers to members of the same array, and assigning or comparing to zero. All other pointer arithmetic is illegal. It is not legal to add two pointers, or to multiply or divide or shift or mask them, or to add `float` or `double` to them, or even, except for `void *`, to assign a pointer of one type to a pointer of another type without a cast.

5.5 Character Pointers and Functions

A *string constant*, written as

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

is an array of characters. In the internal representation, the array is terminated with the null character `'\0'` so that programs can find the end. The length in storage is thus one more than the number of characters between the double quotes.

Perhaps the most common occurrence of string constants is as arguments to functions, as in

```
printf("hello, world\n");
```

When a character string like this appears in a program, access to it is through a character pointer; `printf` receives a pointer to the beginning of the character array. That is, a string constant is accessed by a pointer to its first element.

String constants need not be function arguments. If `pmessage` is declared as

```
char *pmessage;
```

then the statement

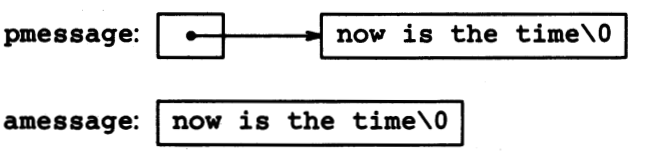
```
pmessage = "now is the time";
```

assigns to `pmessage` a pointer to the character array. This is *not* a string copy; only pointers are involved. C does not provide any operators for processing an entire string of characters as a unit.

There is an important difference between these definitions:

```
char amessage[] = "now is the time";    /* an array */
char *pmessage = "now is the time";     /* a pointer */
```

`amessage` is an array, just big enough to hold the sequence of characters and `'\0'` that initializes it. Individual characters within the array may be changed but `amessage` will always refer to the same storage. On the other hand, `pmessage` is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.



We will illustrate more aspects of pointers and arrays by studying versions of two useful functions adapted from the standard library. The first function is `strcpy(s,t)`, which copies the string `t` to the string `s`. It would be nice just to say `s=t` but this copies the pointer, not the characters. To copy the

characters, we need a loop. The array version is first:

```
/* strcpy:  copy t to s; array subscript version */
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

For contrast, here is a version of `strcpy` with pointers:

```
/* strcpy:  copy t to s; pointer version 1 */
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Because arguments are passed by value, `strcpy` can use the parameters `s` and `t` in any way it pleases. Here they are conveniently initialized pointers, which are marched along the arrays a character at a time, until the `'\0'` that terminates `t` has been copied to `s`.

In practice, `strcpy` would not be written as we showed it above. Experienced C programmers would prefer

```
/* strcpy:  copy t to s; pointer version 2 */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

This moves the increment of `s` and `t` into the test part of the loop. The value of `*t++` is the character that `t` pointed to before `t` was incremented; the postfix `++` doesn't change `t` until after this character has been fetched. In the same way, the character is stored into the old `s` position before `s` is incremented. This character is also the value that is compared against `'\0'` to control the loop. The net effect is that characters are copied from `t` to `s`, up to and including the terminating `'\0'`.

As the final abbreviation, observe that a comparison against `'\0'` is redundant, since the question is merely whether the expression is zero. So the function would likely be written as

```
/* strcpy:  copy t to s; pointer version 3 */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

Although this may seem cryptic at first sight, the notational convenience is considerable, and the idiom should be mastered, because you will see it frequently in C programs.

The `strcpy` in the standard library (`<string.h>`) returns the target string as its function value.

The second routine that we will examine is `strcmp(s,t)`, which compares the character strings `s` and `t`, and returns negative, zero or positive if `s` is lexicographically less than, equal to, or greater than `t`. The value is obtained by subtracting the characters at the first position where `s` and `t` disagree.

```
/* strcmp:  return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
    int i;

    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

The pointer version of `strcmp`:

```
/* strcmp:  return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

Since `++` and `--` are either prefix or postfix operators, other combinations of `*` and `++` and `--` occur, although less frequently. For example,

```
*--p
```

decrements `p` before fetching the character that `p` points to. In fact, the pair of expressions

```
*p++ = val;     /* push val onto stack */
val = *--p;     /* pop top of stack into val */
```

are the standard idioms for pushing and popping a stack; see Section 4.3.

The header `<string.h>` contains declarations for the functions mentioned