in this section, plus a variety of other string-handling functions from the standard library.

Exercise 5-3. Write a pointer version of the function `strcat` that we showed in Chapter 2: `strcat(s,t)` copies the string `t` to the end of `s`. □

Exercise 5-4. Write the function `strend(s,t)`, which returns 1 if the string `t` occurs at the end of the string `s`, and zero otherwise. □

Exercise 5-5. Write versions of the library functions `strncpy`, `strncat`, and `strncmp`, which operate on at most the first `n` characters of their argument strings. For example, `strncpy(s,t,n)` copies at most `n` characters of `t` to `s`. Full descriptions are in Appendix B. □

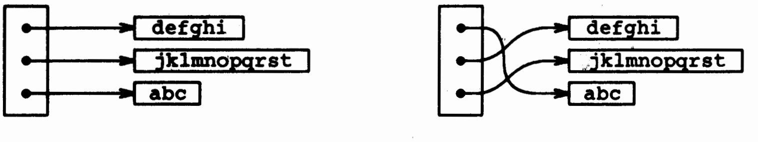
Exercise 5-6. Rewrite appropriate programs from earlier chapters and exercises with pointers instead of array indexing. Good possibilities include `getline` (Chapters 1 and 4), `atoi`, `itoa`, and their variants (Chapters 2, 3, and 4), `reverse` (Chapter 3), and `strindex` and `getop` (Chapter 4). □

5.6 Pointer Arrays; Pointers to Pointers

Since pointers are variables themselves, they can be stored in arrays just as other variables can. Let us illustrate by writing a program that will sort a set of text lines into alphabetic order, a stripped-down version of the UNIX program `sort`.

In Chapter 3 we presented a Shell sort function that would sort an array of integers, and in Chapter 4 we improved on it with a quicksort. The same algorithms will work, except that now we have to deal with lines of text, which are of different lengths, and which, unlike integers, can't be compared or moved in a single operation. We need a data representation that will cope efficiently and conveniently with variable-length text lines.

This is where the array of pointers enters. If the lines to be sorted are stored end-to-end in one long character array, then each line can be accessed by a pointer to its first character. The pointers themselves can be stored in an array. Two lines can be compared by passing their pointers to `strcmp`. When two out-of-order lines have to be exchanged, the pointers in the pointer array are exchanged, not the text lines themselves.



This eliminates the twin problems of complicated storage management and high overhead that would go with moving the lines themselves.

The sorting process has three steps:

read all the lines of input
sort them
print them in order

As usual, it's best to divide the program into functions that match this natural division, with the main routine controlling the other functions. Let us defer the sorting step for a moment, and concentrate on the data structure and the input and output.

The input routine has to collect and save the characters of each line, and build an array of pointers to the lines. It will also have to count the number of input lines, since that information is needed for sorting and printing. Since the input function can only cope with a finite number of input lines, it can return some illegal line count like -1 if too much input is presented.

The output routine only has to print the lines in the order in which they appear in the array of pointers.

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

#define MAXLINES 5000       /* max #lines to be sorted */

char *lineptr[MAXLINES];    /* pointers to text lines */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(char *lineptr[], int left, int right);

/* sort input lines */
main()
{
    int nlines;    /* number of input lines read */

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort(lineptr, 0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("error: input too big to sort\n");
        return 1;
    }
}
```

```
#define MAXLEN 1000   /* max length of any input line */
int getline(char *, int);
char *alloc(int);

/* readlines:  read input lines */
int readlines(char *lineptr[], int maxlines)
{
    int len, nlines;
    char *p, line[MAXLEN];

    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines || (p = alloc(len)) == NULL)
            return -1;
        else {
            line[len-1] = '\0'; /* delete newline */
            strcpy(p, line);
            lineptr[nlines++] = p;
        }
    return nlines;
}

/* writelines:  write output lines */
void writelines(char *lineptr[], int nlines)
{
    int i;

    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}
```

The function `getline` is from Section 1.9.

The main new thing is the declaration for `lineptr`:

```
char *lineptr[MAXLINES]
```

says that `lineptr` is an array of `MAXLINES` elements, each element of which is a pointer to a `char`. That is, `lineptr[i]` is a character pointer, and `*lineptr[i]` is the character it points to, the first character of the `i`-th saved text line.

Since `lineptr` is itself the name of an array, it can be treated as a pointer in the same manner as in our earlier examples, and `writelines` can be written instead as

```
/* writelines:  write output lines */
void writelines(char *lineptr[], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}
```

Initially `*lineptr` points to the first line; each increment advances it to the next line pointer while `nlines` is counted down.

With input and output under control, we can proceed to sorting. The quicksort from Chapter 4 needs minor changes: the declarations have to be modified, and the comparison operation must be done by calling `strcmp`. The algorithm remains the same, which gives us some confidence that it will still work.

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

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

Similarly, the swap routine needs only trivial changes:

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

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

Since any individual element of `v` (alias `lineptr`) is a character pointer, `temp` must be also, so one can be copied to the other.

Exercise 5-7. Rewrite `readlines` to store lines in an array supplied by `main`, rather than calling `alloc` to maintain storage. How much faster is the program? □

5.7 Multi-dimensional Arrays

C provides rectangular multi-dimensional arrays, although in practice they are much less used than arrays of pointers. In this section, we will show some of their properties.

Consider the problem of date conversion, from day of the month to day of the year and vice versa. For example, March 1 is the 60th day of a non-leap year, and the 61st day of a leap year. Let us define two functions to do the conversions: `day_of_year` converts the month and day into the day of the year, and `month_day` converts the day of the year into the month and day. Since this latter function computes two values, the month and day arguments will be pointers:

```
month_day(1988, 60, &m, &d)
```

sets `m` to 2 and `d` to 29 (February 29th).

These functions both need the same information, a table of the number of days in each month ("thirty days hath September ..."). Since the number of days per month differs for leap years and non-leap years, it's easier to separate them into two rows of a two-dimensional array than to keep track of what happens to February during computation. The array and the functions for performing the transformations are as follows:

```
static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year:  set day of year from month & day */
int day_of_year(int year, int month, int day)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; i < month; i++)
        day += daytab[leap][i];
    return day;
}

/* month_day:  set month, day from day of year */
void month_day(int year, int yearday, int *pmonth, int *pday)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; yearday > daytab[leap][i]; i++)
        yearday -= daytab[leap][i];
    *pmonth = i;
    *pday = yearday;
}
```

Recall that the arithmetic value of a logical expression, such as the one for `leap`, is either zero (false) or one (true), so it can be used as a subscript of the array `daytab`.

The array `daytab` has to be external to both `day_of_year` and `month_day`, so they can both use it. We made it `char` to illustrate a legitimate use of `char` for storing small non-character integers.

`daytab` is the first two-dimensional array we have dealt with. In C, a two-dimensional array is really a one-dimensional array, each of whose elements is an array. Hence subscripts are written as

```
daytab[i][j]    /* [row][col] */
```

rather than

```
daytab[i,j]     /* WRONG */
```

Other than this notational distinction, a two-dimensional array can be treated in much the same way as in other languages. Elements are stored by rows, so the rightmost subscript, or column, varies fastest as elements are accessed in storage order.

An array is initialized by a list of initializers in braces; each row of a two-dimensional array is initialized by a corresponding sub-list. We started the array `daytab` with a column of zero so that month numbers can run from the natural 1 to 12 instead of 0 to 11. Since space is not at a premium here, this is clearer than adjusting the indices.

If a two-dimensional array is to be passed to a function, the parameter declaration in the function must include the number of columns; the number of rows is irrelevant, since what is passed is, as before, a pointer to an array of rows, where each row is an array of 13 `int`s. In this particular case, it is a pointer to objects that are arrays of 13 `int`s. Thus if the array `daytab` is to be passed to a function `f`, the declaration of `f` would be

```
f(int daytab[2][13]) { ... }
```

It could also be

```
f(int daytab[][13]) { ... }
```

since the number of rows is irrelevant, or it could be

```
f(int (*daytab)[13]) { ... }
```

which says that the parameter is a pointer to an array of 13 integers. The parentheses are necessary since brackets `[]` have higher precedence than `*`. Without parentheses, the declaration

```
int *daytab[13]
```

is an array of 13 pointers to integers. More generally, only the first dimension (subscript) of an array is free; all the others have to be specified.

Section 5.12 has a further discussion of complicated declarations.

Exercise 5-8. There is no error checking in `day_of_year` or `month_day`. Remedy this defect. □

5.8 Initialization of Pointer Arrays

Consider the problem of writing a function `month_name(n)`, which returns a pointer to a character string containing the name of the n-th month. This is an ideal application for an internal `static` array. `month_name` contains a private array of character strings, and returns a pointer to the proper one when called. This section shows how that array of names is initialized.

The syntax is similar to previous initializations:

```
/* month_name:  return name of n-th month */
char *month_name(int n)
{
    static char *name[] = {
        "Illegal month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };

    return (n < 1 || n > 12) ? name[0] : name[n];
}
```

The declaration of `name`, which is an array of character pointers, is the same as `lineptr` in the sorting example. The initializer is a list of character strings; each is assigned to the corresponding position in the array. The characters of the `i`-th string are placed somewhere, and a pointer to them is stored in `name[i]`. Since the size of the array `name` is not specified, the compiler counts the initializers and fills in the correct number.

5.9 Pointers vs. Multi-dimensional Arrays

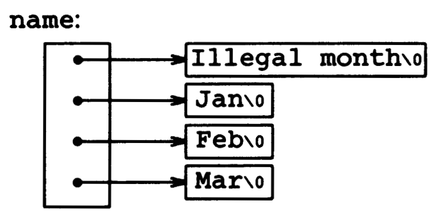
Newcomers to C are sometimes confused about the difference between a two-dimensional array and an array of pointers, such as `name` in the example above. Given the definitions

```
int a[10][20];
int *b[10];
```

then `a[3][4]` and `b[3][4]` are both syntactically legal references to a single `int`. But `a` is a true two-dimensional array: 200 `int`-sized locations have been set aside, and the conventional rectangular subscript calculation 20×*row*+*col* is used to find the element `a`[*row*][*col*]. For `b`, however, the definition only allocates 10 pointers and does not initialize them; initialization must be done explicitly, either statically or with code. Assuming that each element of `b` does point to a twenty-element array, then there will be 200 `int`s set aside, plus ten cells for the pointers. The important advantage of the pointer array is that the rows of the array may be of different lengths. That is, each element of `b` need not point to a twenty-element vector; some may point to two elements, some to fifty, and some to none at all.

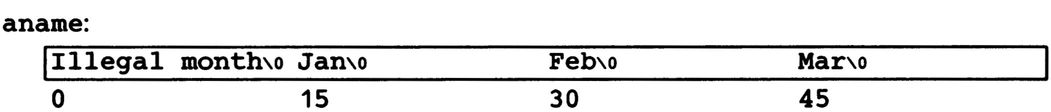
Although we have phrased this discussion in terms of integers, by far the most frequent use of arrays of pointers is to store character strings of diverse lengths, as in the function `month_name`. Compare the declaration and picture for an array of pointers:

```
char *name[] = { "Illegal month", "Jan, "Feb", "Mar" };
```



with those for a two-dimensional array:

```
char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```



Exercise 5-9. Rewrite the routines `day_of_year` and `month_day` with pointers instead of indexing. □

5.10 Command-line Arguments

In environments that support C, there is a way to pass command-line arguments or parameters to a program when it begins executing. When `main` is called, it is called with two arguments. The first (conventionally called `argc`, for argument count) is the number of command-line arguments the program was invoked with; the second (`argv`, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string. We customarily use multiple levels of pointers to manipulate these character strings.

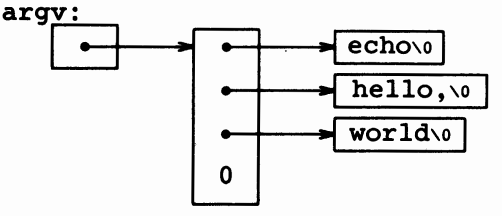
The simplest illustration is the program `echo`, which echoes its command-line arguments on a single line, separated by blanks. That is, the command

```
echo hello, world
```

prints the output

```
hello, world
```

By convention, `argv[0]` is the name by which the program was invoked, so `argc` is at least 1. If `argc` is 1, there are no command-line arguments after the program name. In the example above, `argc` is 3, and `argv[0]`, `argv[1]`, and `argv[2]` are `"echo"`, `"hello,"`, and `"world"` respectively. The first optional argument is `argv[1]` and the last is `argv[argc-1]`; additionally, the standard requires that `argv[argc]` be a null pointer.



The first version of `echo` treats `argv` as an array of character pointers:

```
#include <stdio.h>

/* echo command-line arguments; 1st version */
main(int argc, char *argv[])
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    return 0;
}
```

Since `argv` is a pointer to an array of pointers, we can manipulate the pointer rather than index the array. This next variation is based on incrementing `argv`, which is a pointer to pointer to `char`, while `argc` is counted down:

```
#include <stdio.h>

/* echo command-line arguments; 2nd version */
main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}
```

Since `argv` is a pointer to the beginning of the array of argument strings, incrementing it by 1 (`++argv`) makes it point at the original `argv[1]` instead of `argv[0]`. Each successive increment moves it along to the next argument; `*argv` is then the pointer to that argument. At the same time, `argc` is decremented; when it becomes zero, there are no arguments left to print.

Alternatively, we could write the `printf` statement as

```
        printf((argc > 1) ? "%s " : "%s", *++argv);
```

This shows that the format argument of `printf` can be an expression too.

As a second example, let us make some enhancements to the pattern-finding program from Section 4.1. If you recall, we wired the search pattern deep into the program, an obviously unsatisfactory arrangement. Following the lead of the UNIX program `grep`, let us change the program so the pattern to be matched is specified by the first argument on the command line.

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

int getline(char *line, int max);

/* find:  print lines that match pattern from 1st arg */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    int found = 0;

    if (argc != 2)
        printf("Usage: find pattern\n");
    else
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1]) != NULL) {
                printf("%s", line);
                found++;
            }
    return found;
}
```

The standard library function `strstr(s,t)` returns a pointer to the first occurrence of the string `t` in the string `s`, or `NULL` if there is none. It is declared in `<string.h>`.

The model can now be elaborated to illustrate further pointer constructions. Suppose we want to allow two optional arguments. One says "print all lines *except* those that match the pattern;" the second says "precede each printed line by its line number."

A common convention for C programs on UNIX systems is that an argument that begins with a minus sign introduces an optional flag or parameter. If we choose `-x` (for "except") to signal the inversion, and `-n` ("number") to request line numbering, then the command

`find -x -n` *pattern*

will print each line that doesn't match the pattern, preceded by its line number.

Optional arguments should be permitted in any order, and the rest of the program should be independent of the number of arguments that were present. Furthermore, it is convenient for users if option arguments can be combined, as in

```
find -nx pattern
```

Here is the program:

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

int getline(char *line, int max);

/* find:  print lines that match pattern from 1st arg */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    long lineno = 0;
    int c, except = 0, number = 0, found = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        while (c = *++argv[0])
            switch (c) {
            case 'x':
                except = 1;
                break;
            case 'n':
                number = 1;
                break;
            default:
                printf("find: illegal option %c\n", c);
                argc = 0;
                found = -1;
                break;
            }
    if (argc != 1)
        printf("Usage: find -x -n pattern\n");
    else
        while (getline(line, MAXLINE) > 0) {
            lineno++;
            if ((strstr(line, *argv) != NULL) != except) {
                if (number)
                    printf("%ld:", lineno);
                printf("%s", line);
                found++;
            }
        }
    return found;
}
```

`argc` is decremented and `argv` is incremented before each optional argument. At the end of the loop, if there are no errors, `argc` tells how many arguments remain unprocessed and `argv` points to the first of these. Thus `argc` should be 1 and `*argv` should point at the pattern. Notice that `*++argv` is a pointer to an argument string, so `(*++argv)[0]` is its first character. (An alternate valid form would be `**++argv`.) Because `[]` binds tighter than `*` and `++`, the parentheses are necessary; without them the expression would be taken as `*++(argv[0])`. In fact, that is what we used in the inner loop, where the task is to walk along a specific argument string. In the inner loop, the expression `*++argv[0]` increments the pointer `argv[0]`!

It is rare that one uses pointer expressions more complicated than these; in such cases, breaking them into two or three steps will be more intuitive.

Exercise 5-10. Write the program `expr`, which evaluates a reverse Polish expression from the command line, where each operator or operand is a separate argument. For example,

```
expr  2  3  4  +  *
```

evaluates 2 × (3+4). □

Exercise 5-11. Modify the programs `entab` and `detab` (written as exercises in Chapter 1) to accept a list of tab stops as arguments. Use the default tab settings if there are no arguments. □

Exercise 5-12. Extend `entab` and `detab` to accept the shorthand

`entab -`*m* `+`*n*

to mean tab stops every *n* columns, starting at column *m*. Choose convenient (for the user) default behavior. □

Exercise 5-13. Write the program `tail`, which prints the last *n* lines of its input. By default, *n* is 10, let us say, but it can be changed by an optional argument, so that

`tail -`*n*

prints the last *n* lines. The program should behave rationally no matter how unreasonable the input or the value of *n*. Write the program so it makes the best use of available storage; lines should be stored as in the sorting program of Section 5.6, not in a two-dimensional array of fixed size. □

5.11 Pointers to Functions

In C, a function itself is not a variable, but it is possible to define pointers to functions, which can be assigned, placed in arrays, passed to functions, returned by functions, and so on. We will illustrate this by modifying the sorting procedure written earlier in this chapter so that if the optional argument `-n` is given, it will sort the input lines numerically instead of lexicographically.

A sort often consists of three parts—a comparison that determines the ordering of any pair of objects, an exchange that reverses their order, and a sorting algorithm that makes comparisons and exchanges until the objects are in order. The sorting algorithm is independent of the comparison and exchange operations, so by passing different comparison and exchange functions to it, we can arrange to sort by different criteria. This is the approach taken in our new sort.

Lexicographic comparison of two lines is done by `strcmp`, as before; we will also need a routine `numcmp` that compares two lines on the basis of numeric value and returns the same kind of condition indication as `strcmp` does. These functions are declared ahead of `main` and a pointer to the appropriate one is passed to `qsort`. We have skimped on error processing for arguments, so as to concentrate on the main issues.

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

#define MAXLINES 5000      /* max #lines to be sorted */
char *lineptr[MAXLINES];   /* pointers to text lines */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(void *lineptr[], int left, int right,
           int (*comp)(void *, void *));
int numcmp(char *, char *);

/* sort input lines */
main(int argc, char *argv[])
{
    int nlines;            /* number of input lines read */
    int numeric = 0;       /* 1 if numeric sort */

    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort((void **) lineptr, 0, nlines-1,
          (int (*)(void*,void*))(numeric ? numcmp : strcmp));
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("input too big to sort\n");
        return 1;
    }
}
```

In the call to `qsort`, `strcmp` and `numcmp` are addresses of functions. Since they are known to be functions, the `&` operator is not necessary, in the same way that it is not needed before an array name.

We have written `qsort` so it can process any data type, not just character strings. As indicated by the function prototype, qsort expects an array of pointers, two integers, and a function with two pointer arguments. The generic pointer type void * is used for the pointer arguments. Any pointer can be cast to void * and back again without loss of information, so we can call qsort by casting arguments to void *. The elaborate cast of the function argument casts the arguments of the comparison function. These will generally have no effect on actual representation, but assure the compiler that all is well.

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

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

The declarations should be studied with some care. The fourth parameter of qsort is

```
int (*comp)(void *, void *)
```

which says that comp is a pointer to a function that has two void * arguments and returns an int.

The use of comp in the line

```
if ((*comp)(v[i], v[left]) < 0)
```

is consistent with the declaration: comp is a pointer to a function, *comp is the function, and

```
(*comp)(v[i], v[left])
```

is the call to it. The parentheses are needed so the components are correctly associated; without them,

```
int *comp(void *, void *)    /* WRONG */
```

says that comp is a function returning a pointer to an int, which is very different.

We have already shown strcmp, which compares two strings. Here is numcmp, which compares two strings on a leading numeric value, computed by calling `atof`:

```
#include <stdlib.h>

/* numcmp:  compare s1 and s2 numerically */
int numcmp(char *s1, char *s2)
{
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}
```

The `swap` function, which exchanges two pointers, is identical to what we presented earlier in the chapter, except that the declarations are changed to `void *`.

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

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

A variety of other options can be added to the sorting program; some make challenging exercises.

Exercise 5-14. Modify the sort program to handle a `-r` flag, which indicates sorting in reverse (decreasing) order. Be sure that `-r` works with `-n`. □

Exercise 5-15. Add the option `-f` to fold upper and lower case together, so that case distinctions are not made during sorting; for example, `a` and `A` compare equal. □

Exercise 5-16. Add the `-d` ("directory order") option, which makes comparisons only on letters, numbers and blanks. Make sure it works in conjunction with `-f`. □

Exercise 5-17. Add a field-handling capability, so sorting may be done on fields within lines, each field sorted according to an independent set of options. (The index for this book was sorted with `-df` for the index category and `-n` for the page numbers.) □

5.12 Complicated Declarations

C is sometimes castigated for the syntax of its declarations, particularly ones that involve pointers to functions. The syntax is an attempt to make the declaration and the use agree; it works well for simple cases, but it can be confusing for the harder ones, because declarations cannot be read left to right, and because parentheses are over-used. The difference between

```
int *f();        /* f: function returning pointer to int */
```

and

```
int (*pf)();     /* pf: pointer to function returning int */
```

illustrates the problem: `*` is a prefix operator and it has lower precedence than `()`, so parentheses are necessary to force the proper association.

Although truly complicated declarations rarely arise in practice, it is important to know how to understand them, and, if necessary, how to create them. One good way to synthesize declarations is in small steps with `typedef`, which is discussed in Section 6.7. As an alternative, in this section we will present a pair of programs that convert from valid C to a word description and back again. The word description reads left to right.

The first, `dcl`, is the more complex. It converts a C declaration into a word description, as in these examples:

```
char **argv
    argv:  pointer to pointer to char
int (*daytab)[13]
    daytab:  pointer to array[13] of int
int *daytab[13]
    daytab:  array[13] of pointer to int
void *comp()
    comp:  function returning pointer to void
void (*comp)()
    comp:  pointer to function returning void
char (*(*x())[])()
    x: function returning pointer to array[] of
    pointer to function returning char
char (*(*x[3])())[5]
    x: array[3] of pointer to function returning
    pointer to array[5] of char
```

`dcl` is based on the grammar that specifies a declarator, which is spelled out precisely in Appendix A, Section 8.5; this is a simplified form:

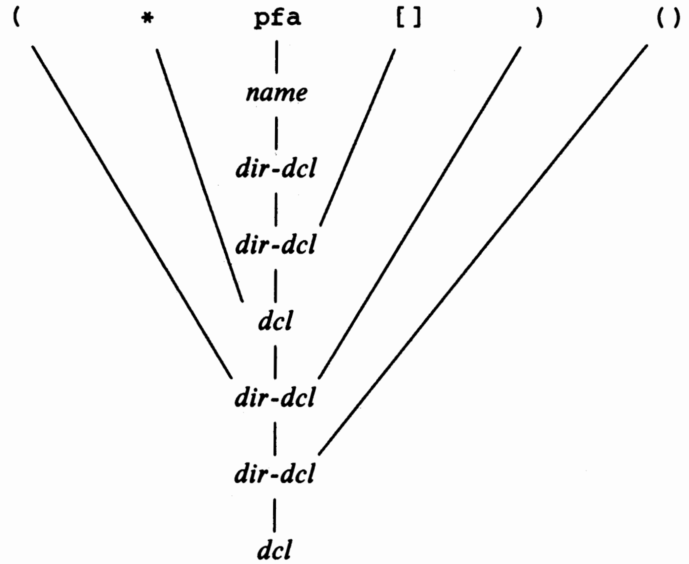
dcl:	*optional *'s direct-dcl*
direct-dcl:	*name*
	(dcl)
	direct-dcl()
	direct-dcl[optional size]

In words, a *dcl* is a *direct-dcl*, perhaps preceded by *'s. A *direct-dcl* is a name, or a parenthesized *dcl*, or a *direct-dcl* followed by parentheses, or a *direct-dcl* followed by brackets with an optional size.

This grammar can be used to parse declarations. For instance, consider this declarator:

```
(*pfa[])()
```

`pfa` will be identified as a *name* and thus as a *direct-dcl*. Then `pfa[]` is also a *direct-dcl*. Then `*pfa[]` is a recognized as a *dcl*, so `(*pfa[])` is a *direct-dcl*. Then `(*pfa[])()` is a *direct-dcl* and thus a *dcl*. We can also illustrate the parse with a parse tree like this (where *direct-dcl* has been abbreviated to *dir-dcl*):



The heart of the `dcl` program is a pair of functions, `dcl` and `dirdcl`, that parse a declaration according to this grammar. Because the grammar is recursively defined, the functions call each other recursively as they recognize pieces of a declaration; the program is called a recursive-descent parser.

```
/* dcl:  parse a declarator */
void dcl(void)
{
    int ns;

    for (ns = 0; gettoken() == '*'; )   /* count *'s */
        ns++;
    dirdcl();
    while (ns-- > 0)
        strcat(out, " pointer to");
}
```

```
/* dirdcl:  parse a direct declarator */
void dirdcl(void)
{
    int type;

    if (tokentype == '(') {          /* ( dcl ) */
        dcl();
        if (tokentype != ')')
            printf("error: missing )\n");
    } else if (tokentype == NAME)    /* variable name */
        strcpy(name, token);
    else
        printf("error: expected name or (dcl)\n");
    while ((type=gettoken()) == PARENS || type == BRACKETS)
        if (type == PARENS)
            strcat(out, " function returning");
        else {
            strcat(out, " array");
            strcat(out, token);
            strcat(out, " of");
        }
}
```

Since the programs are intended to be illustrative, not bullet-proof, there are significant restrictions on `dcl`. It can only handle a simple data type like `char` or `int`. It does not handle argument types in functions, or qualifiers like `const`. Spurious blanks confuse it. It doesn't do much error recovery, so invalid declarations will also confuse it. These improvements are left as exercises.

Here are the global variables and the main routine:

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

#define  MAXTOKEN  100

enum { NAME, PARENS, BRACKETS };

void dcl(void);
void dirdcl(void);

int  gettoken(void);
int  tokentype;            /* type of last token */
char token[MAXTOKEN];      /* last token string */
char name[MAXTOKEN];       /* identifier name */
char datatype[MAXTOKEN];   /* data type = char, int, etc. */
char out[1000];            /* output string */
```

```
main()  /* convert declaration to words */
{
    while (gettoken() != EOF) {   /* 1st token on line */
        strcpy(datatype, token);  /* is the datatype */
        out[0] = '\0';
        dcl();       /* parse rest of line */
        if (tokentype != '\n')
            printf("syntax error\n");
        printf("%s: %s %s\n", name, out, datatype);
    }
    return 0;
}
```

The function `gettoken` skips blanks and tabs, then finds the next token in the input; a "token" is a name, a pair of parentheses, a pair of brackets perhaps including a number, or any other single character.

```
int gettoken(void)  /* return next token */
{
    int c, getch(void);
    void ungetch(int);
    char *p = token;

    while ((c = getch()) == ' ' || c == '\t')
        ;
    if (c == '(') {
        if ((c = getch()) == ')') {
            strcpy(token, "()");
            return tokentype = PARENS;
        } else {
            ungetch(c);
            return tokentype = '(';
        }
    } else if (c == '[') {
        for (*p++ = c; (*p++ = getch()) != ']'; )
            ;
        *p = '\0';
        return tokentype = BRACKETS;
    } else if (isalpha(c)) {
        for (*p++ = c; isalnum(c = getch()); )
            *p++ = c;
        *p = '\0';
        ungetch(c);
        return tokentype = NAME;
    } else
        return tokentype = c;
}
```

`getch` and `ungetch` were discussed in Chapter 4.

Going in the other direction is easier, especially if we do not worry about generating redundant parentheses. The program `undcl` converts a word description like "x is a function returning a pointer to an array of pointers to functions returning char," which we will express as

```
x () * [] * () char
```

to

```
char (*(*x())[])()
```

The abbreviated input syntax lets us reuse the gettoken function. undcl also uses the same external variables as dcl does.

```
/* undcl:  convert word description to declaration */
main()
{
    int type;
    char temp[MAXTOKEN];

    while (gettoken() != EOF) {
        strcpy(out, token);
        while ((type = gettoken()) != '\n')
            if (type == PARENS || type == BRACKETS)
                strcat(out, token);
            else if (type == '*') {
                sprintf(temp, "(*%s)", out);
                strcpy(out, temp);
            } else if (type == NAME) {
                sprintf(temp, "%s %s", token, out);
                strcpy(out, temp);
            } else
                printf("invalid input at %s\n", token);
        printf("%s\n", out);
    }
    return 0;
}
```

Exercise 5-18. Make dcl recover from input errors. □

Exercise 5-19. Modify undcl so that it does not add redundant parentheses to declarations. □

Exercise 5-20. Expand dcl to handle declarations with function argument types, qualifiers like const, and so on. □

CHAPTER 6: Structures

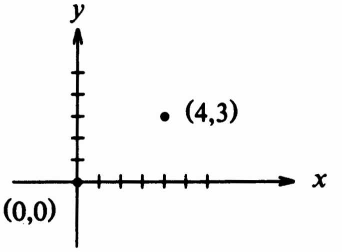
A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called "records" in some languages, notably Pascal.) Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

One traditional example of a structure is the payroll record: an employee is described by a set of attributes such as name, address, social security number, salary, etc. Some of these in turn could be structures: a name has several components, as does an address and even a salary. Another example, more typical for C, comes from graphics: a point is a pair of coordinates, a rectangle is a pair of points, and so on.

The main change made by the ANSI standard is to define structure assignment—structures may be copied and assigned to, passed to functions, and returned by functions. This has been supported by most compilers for many years, but the properties are now precisely defined. Automatic structures and arrays may now also be initialized.

6.1 Basics of Structures

Let us create a few structures suitable for graphics. The basic object is a point, which we will assume has an *x* coordinate and a *y* coordinate, both integers.



The two components can be placed in a structure declared like this:

```
struct point {
    int x;
    int y;
};
```

The keyword `struct` introduces a structure declaration, which is a list of declarations enclosed in braces. An optional name called a *structure tag* may follow the word `struct` (as with `point` here). The tag names this kind of structure, and can be used subsequently as a shorthand for the part of the declaration in braces.

The variables named in a structure are called *members*. A structure member or tag and an ordinary (i.e., non-member) variable can have the same name without conflict, since they can always be distinguished by context. Furthermore, the same member names may occur in different structures, although as a matter of style one would normally use the same names only for closely related objects.

A `struct` declaration defines a type. The right brace that terminates the list of members may be followed by a list of variables, just as for any basic type. That is,

```
struct { ... } x, y, z;
```

is syntactically analogous to

```
int x, y, z;
```

in the sense that each statement declares `x`, `y` and `z` to be variables of the named type and causes space to be set aside for them.

A structure declaration that is not followed by a list of variables reserves no storage; it merely describes a template or the shape of a structure. If the declaration is tagged, however, the tag can be used later in definitions of instances of the structure. For example, given the declaration of `point` above,

```
struct point pt;
```

defines a variable `pt` which is a structure of type `struct point`. A structure can be initialized by following its definition with a list of initializers, each a constant expression, for the members:

```
struct point maxpt = { 320, 200 };
```

An automatic structure may also be initialized by assignment or by calling a function that returns a structure of the right type.

A member of a particular structure is referred to in an expression by a construction of the form

structure-name . *member*

The structure member operator "`.`" connects the structure name and the member name. To print the coordinates of the point `pt`, for instance,

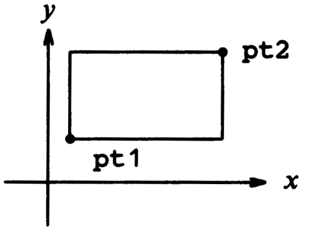
```
printf("%d,%d", pt.x, pt.y);
```

or to compute the distance from the origin (0,0) to `pt`,

```
double dist, sqrt(double);

dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Structures can be nested. One representation of a rectangle is a pair of points that denote the diagonally opposite corners:




```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

The `rect` structure contains two `point` structures. If we declare `screen` as

```
struct rect screen;
```

then

```
screen.pt1.x
```

refers to the *x* coordinate of the `pt1` member of `screen`.

6.2 Structures and Functions

The only legal operations on a structure are copying it or assigning to it as a unit, taking its address with `&`, and accessing its members. Copy and assignment include passing arguments to functions and returning values from functions as well. Structures may not be compared. A structure may be initialized by a list of constant member values; an automatic structure may also be initialized by an assignment.

Let us investigate structures by writing some functions to manipulate points and rectangles. There are at least three possible approaches: pass components separately, pass an entire structure, or pass a pointer to it. Each has its good points and bad points.

The first function, `makepoint`, will take two integers and return a `point` structure:

```
/* makepoint:  make a point from x and y components */
struct point makepoint(int x, int y)
{
    struct point temp;

    temp.x = x;
    temp.y = y;
    return temp;
}
```

Notice that there is no conflict between the argument name and the member with the same name; indeed the re-use of the names stresses the relationship.

makepoint can now be used to initialize any structure dynamically, or to provide structure arguments to a function:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);

screen.pt1 = makepoint(0, 0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                   (screen.pt1.y + screen.pt2.y)/2);
```

The next step is a set of functions to do arithmetic on points. For instance,

```
/* addpoint:  add two points */
struct point addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

Here both the arguments and the return value are structures. We incremented the components in p1 rather than using an explicit temporary variable to emphasize that structure parameters are passed by value like any others.

As another example, the function ptinrect tests whether a point is inside a rectangle, where we have adopted the convention that a rectangle includes its left and bottom sides but not its top and right sides:

```
/* ptinrect:  return 1 if p in r, 0 if not */
int ptinrect(struct point p, struct rect r)
{
    return p.x >= r.pt1.x && p.x < r.pt2.x
        && p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

This assumes that the rectangle is represented in a standard form where the pt1 coordinates are less than the pt2 coordinates. The following function returns a rectangle guaranteed to be in canonical form:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))

/* canonrect:  canonicalize coordinates of rectangle */
struct rect canonrect(struct rect r)
{
    struct rect temp;

    temp.pt1.x = min(r.pt1.x, r.pt2.x);
    temp.pt1.y = min(r.pt1.y, r.pt2.y);
    temp.pt2.x = max(r.pt1.x, r.pt2.x);
    temp.pt2.y = max(r.pt1.y, r.pt2.y);
    return temp;
}
```

If a large structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure. Structure pointers are just like pointers to ordinary variables. The declaration

```
struct point *pp;
```

says that `pp` is a pointer to a structure of type `struct point`. If `pp` points to a `point` structure, `*pp` is the structure, and `(*pp).x` and `(*pp).y` are the members. To use `pp`, we might write, for example,

```
struct point origin, *pp;

pp = &origin;
printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
```

The parentheses are necessary in `(*pp).x` because the precedence of the structure member operator `.` is higher than `*`. The expression `*pp.x` means `*(pp.x)`, which is illegal here because `x` is not a pointer.

Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If `p` is a pointer to a structure, then

`p->`*member-of-structure*

refers to the particular member. (The operator `->` is a minus sign immediately followed by `>`.) So we could write instead

```
printf("origin is (%d,%d)\n", pp->x, pp->y);
```

Both `.` and `->` associate from left to right, so if we have

```
struct rect r, *rp = &r;
```

then these four expressions are equivalent:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

The structure operators . and ->, together with ( ) for function calls and [ ] for subscripts, are at the top of the precedence hierarchy and thus bind very tightly. For example, given the declaration

```
struct {
    int len;
    char *str;
} *p;
```

then

```
++p->len
```

increments `len`, not `p`, because the implied parenthesization is `++(p->len)`. Parentheses can be used to alter the binding: `(++p)->len` increments `p` before accessing `len`, and `(p++)->len` increments `p` afterward. (This last set of parentheses is unnecessary.)

In the same way, `*p->str` fetches whatever `str` points to; `*p->str++` increments `str` after accessing whatever it points to (just like `*s++`); `(*p->str)++` increments whatever `str` points to; and `*p++->str` increments `p` after accessing whatever `str` points to.

6.3 Arrays of Structures

Consider writing a program to count the occurrences of each C keyword. We need an array of character strings to hold the names, and an array of integers for the counts. One possibility is to use two parallel arrays, `keyword` and `keycount`, as in

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

But the very fact that the arrays are parallel suggests a different organization, an array of structures. Each keyword entry is a pair:

```
char *word;
int count;
```

and there is an array of pairs. The structure declaration

```
struct key {
    char *word;
    int count;
} keytab[NKEYS];
```

declares a structure type `key`, defines an array `keytab` of structures of this type, and sets aside storage for them. Each element of the array is a structure. This could also be written

```
struct key {
    char *word;
    int count;
};

struct key keytab[NKEYS];
```

Since the structure keytab contains a constant set of names, it is easiest to make it an external variable and initialize it once and for all when it is defined. The structure initialization is analogous to earlier ones—the definition is followed by a list of initializers enclosed in braces:

```
struct key {
    char *word;
    int count;
} keytab[] = {
    "auto", 0,
    "break", 0,
    "case", 0,
    "char", 0,
    "const", 0,
    "continue", 0,
    "default", 0,
    /* ... */
    "unsigned", 0,
    "void", 0,
    "volatile", 0,
    "while", 0
};
```

The initializers are listed in pairs corresponding to the structure members. It would be more precise to enclose initializers for each "row" or structure in braces, as in

```
    { "auto", 0 },
    { "break", 0 },
    { "case", 0 },
    ...
```

but the inner braces are not necessary when the initializers are simple variables or character strings, and when all are present. As usual, the number of entries in the array keytab will be computed if initializers are present and the [] is left empty.

The keyword-counting program begins with the definition of keytab. The main routine reads the input by repeatedly calling a function getword that fetches one word at a time. Each word is looked up in keytab with a version of the binary search function that we wrote in Chapter 3. The list of keywords must be sorted in increasing order in the table.

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

#define MAXWORD 100

int getword(char *, int);
int binsearch(char *, struct key *, int);

/* count C keywords */
main()
{
    int n;
    char word[MAXWORD];

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((n = binsearch(word, keytab, NKEYS)) >= 0)
                keytab[n].count++;
    for (n = 0; n < NKEYS; n++)
        if (keytab[n].count > 0)
            printf("%4d %s\n",
                keytab[n].count, keytab[n].word);
    return 0;
}

/* binsearch:  find word in tab[0]...tab[n-1] */
int binsearch(char *word, struct key tab[], int n)
{
    int cond;
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
        if ((cond = strcmp(word, tab[mid].word)) < 0)
            high = mid - 1;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return -1;
}
```

We will show the function `getword` in a moment; for now it suffices to say that each call to `getword` finds a word, which is copied into the array named as its first argument.

The quantity `NKEYS` is the number of keywords in `keytab`. Although we could count this by hand, it's a lot easier and safer to do it by machine, especially if the list is subject to change. One possibility would be to terminate the list of initializers with a null pointer, then loop along `keytab` until the end is found.

But this is more than is needed, since the size of the array is completely determined at compile time. The size of the array is the size of one entry times the number of entries, so the number of entries is just

size of `keytab` / *size of* `struct key`

C provides a compile-time unary operator called `sizeof` that can be used to compute the size of any object. The expressions

`sizeof` *object*

and

`sizeof(`*type name*`)`

yield an integer equal to the size of the specified object or type in bytes. (Strictly, `sizeof` produces an unsigned integer value whose type, `size_t`, is defined in the header `<stddef.h>`.) An object can be a variable or array or structure. A type name can be the name of a basic type like `int` or `double`, or a derived type like a structure or a pointer.

In our case, the number of keywords is the size of the array divided by the size of one element. This computation is used in a `#define` statement to set the value of `NKEYS`:

```
#define  NKEYS  (sizeof keytab / sizeof(struct key))
```

Another way to write this is to divide the array size by the size of a specific element:

```
#define  NKEYS  (sizeof keytab / sizeof keytab[0])
```

This has the advantage that it does not need to be changed if the type changes.

A `sizeof` can not be used in a `#if` line, because the preprocessor does not parse type names. But the expression in the `#define` is not evaluated by the preprocessor, so the code here is legal.

Now for the function `getword`. We have written a more general `getword` than is necessary for this program, but it is not complicated. `getword` fetches the next "word" from the input, where a word is either a string of letters and digits beginning with a letter, or a single non-white space character. The function value is the first character of the word, or `EOF` for end of file, or the character itself if it is not alphabetic.

```
/* getword:  get next word or character from input */
int getword(char *word, int lim)
{
    int c, getch(void);
    void ungetch(int);
    char *w = word;

    while (isspace(c = getch()))
        ;
    if (c != EOF)
        *w++ = c;
    if (!isalpha(c)) {
        *w = '\0';
        return c;
    }
    for ( ; --lim > 0; w++)
        if (!isalnum(*w = getch())) {
            ungetch(*w);
            break;
        }
    *w = '\0';
    return word[0];
}
```

`getword` uses the `getch` and `ungetch` that we wrote in Chapter 4. When the collection of an alphanumeric token stops, `getword` has gone one character too far. The call to `ungetch` pushes that character back on the input for the next call. `getword` also uses `isspace` to skip white space, `isalpha` to identify letters, and `isalnum` to identify letters and digits; all are from the standard header `<ctype.h>`.

Exercise 6-1. Our version of `getword` does not properly handle underscores, string constants, comments, or preprocessor control lines. Write a better version. □

6.4 Pointers to Structures

To illustrate some of the considerations involved with pointers to and arrays of structures, let us write the keyword-counting program again, this time using pointers instead of array indices.

The external declaration of `keytab` need not change, but `main` and `binsearch` do need modification.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

int getword(char *, int);
struct key *binsearch(char *, struct key *, int);

/* count C keywords; pointer version */
main()
{
    char word[MAXWORD];
    struct key *p;

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((p=binsearch(word, keytab, NKEYS)) != NULL)
                p->count++;
    for (p = keytab; p < keytab + NKEYS; p++)
        if (p->count > 0)
            printf("%4d %s\n", p->count, p->word);
    return 0;
}

/* binsearch:  find word in tab[0]...tab[n-1] */
struct key *binsearch(char *word, struct key *tab, int n)
{
    int cond;
    struct key *low = &tab[0];
    struct key *high = &tab[n];
    struct key *mid;

    while (low < high) {
        mid = low + (high-low) / 2;
        if ((cond = strcmp(word, mid->word)) < 0)
            high = mid;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return NULL;
}
```

There are several things worthy of note here. First, the declaration of `binsearch` must indicate that it returns a pointer to `struct key` instead of an integer; this is declared both in the function prototype and in `binsearch`. If `binsearch` finds the word, it returns a pointer to it; if it fails, it returns `NULL`.

Second, the elements of `keytab` are now accessed by pointers. This requires significant changes in `binsearch`.

The initializers for `low` and `high` are now pointers to the beginning and just past the end of the table.

The computation of the middle element can no longer be simply

```
mid = (low+high) / 2    /* WRONG */
```

because the addition of two pointers is illegal. Subtraction is legal, however, so `high-low` is the number of elements, and thus

```
mid = low + (high-low) / 2
```

sets `mid` to point to the element halfway between `low` and `high`.

The most important change is to adjust the algorithm to make sure that it does not generate an illegal pointer or attempt to access an element outside the array. The problem is that `&tab[-1]` and `&tab[n]` are both outside the limits of the array `tab`. The former is strictly illegal, and it is illegal to dereference the latter. The language definition does guarantee, however, that pointer arithmetic that involves the first element beyond the end of an array (that is, `&tab[n]`) will work correctly.

In `main` we wrote

```
for (p = keytab; p < keytab + NKEYS; p++)
```

If `p` is a pointer to a structure, arithmetic on `p` takes into account the size of the structure, so `p++` increments `p` by the correct amount to get the next element of the array of structures, and the test stops the loop at the right time.

Don't assume, however, that the size of a structure is the sum of the sizes of its members. Because of alignment requirements for different objects, there may be unnamed "holes" in a structure. Thus, for instance, if a `char` is one byte and an `int` four bytes, the structure

```
struct {
    char c;
    int i;
};
```

might well require eight bytes, not five. The `sizeof` operator returns the proper value.

Finally, an aside on program format: when a function returns a complicated type like a structure pointer, as in

```
struct key *binsearch(char *word, struct key *tab, int n)
```

the function name can be hard to see, and to find with a text editor. Accordingly an alternate style is sometimes used:

```
struct key *
binsearch(char *word, struct key *tab, int n)
```

This is a matter of personal taste; pick the form you like and hold to it.

6.5 Self-referential Structures

Suppose we want to handle the more general problem of counting the occurrences of *all* the words in some input. Since the list of words isn't known in advance, we can't conveniently sort it and use a binary search. Yet we can't do a linear search for each word as it arrives, to see if it's already been seen; the program would take too long. (More precisely, its running time is likely to grow quadratically with the number of input words.) How can we organize the data to cope efficiently with a list of arbitrary words?

One solution is to keep the set of words seen so far sorted at all times, by placing each word into its proper position in the order as it arrives. This shouldn't be done by shifting words in a linear array, though—that also takes too long. Instead we will use a data structure called a *binary tree.*

The tree contains one "node" per distinct word; each node contains

a pointer to the text of the word
a count of the number of occurrences
a pointer to the left child node
a pointer to the right child node

No node may have more than two children; it might have only zero or one.

The nodes are maintained so that at any node the left subtree contains only words that are lexicographically less than the word at the node, and the right subtree contains only words that are greater. This is the tree for the sentence "now is the time for all good men to come to the aid of their party", as built by inserting each word as it is encountered:

```
                 now
                /   \
              is     the
             /  \   /    \
          for   men of     time
          /  \        \    /  \
       all   good   party their to
       / \
    aid  come
```

To find out whether a new word is already in the tree, start at the root and compare the new word to the word stored at that node. If they match, the question is answered affirmatively. If the new word is less than the tree word, continue searching at the left child, otherwise at the right child. If there is no child in the required direction, the new word is not in the tree, and in fact the empty slot is the proper place to add the new word. This process is recursive, since the search from any node uses a search from one of its children. Accordingly, recursive routines for insertion and printing will be most natural.

Going back to the description of a node, it is conveniently represented as a structure with four components:

```
struct tnode {           /* the tree node: */
    char *word;              /* points to the text */
    int count;               /* number of occurrences */
    struct tnode *left;      /* left child */
    struct tnode *right;     /* right child */
};
```

This recursive declaration of a node might look chancy, but it's correct. It is illegal for a structure to contain an instance of itself, but

```
struct tnode *left;
```

declares `left` to be a pointer to a `tnode`, not a `tnode` itself.

Occasionally, one needs a variation of self-referential structures: two structures that refer to each other. The way to handle this is:

```
struct t {
    ...
    struct s *p;     /* p points to an s */
};
struct s {
    ...
    struct t *q;     /* q points to a t */
};
```

The code for the whole program is surprisingly small, given a handful of supporting routines like `getword` that we have already written. The main routine reads words with `getword` and installs them in the tree with `addtree`.

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

#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);

/* word frequency count */
main()
{
    struct tnode *root;
    char word[MAXWORD];

    root = NULL;
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            root = addtree(root, word);
    treeprint(root);
    return 0;
}
```

The function addtree is recursive. A word is presented by main to the top level (the root) of the tree. At each stage, that word is compared to the word already stored at the node, and is percolated down to either the left or right subtree by a recursive call to addtree. Eventually the word either matches something already in the tree (in which case the count is incremented), or a null pointer is encountered, indicating that a node must be created and added to the tree. If a new node is created, addtree returns a pointer to it, which is installed in the parent node.

```
struct tnode *talloc(void);
char *strdup(char *);

/* addtree:  add a node with w, at or below p */
struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if (p == NULL) {     /* a new word has arrived */
        p = talloc();    /* make a new node */
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;      /* repeated word */
    else if (cond < 0)   /* less than into left subtree */
        p->left = addtree(p->left, w);
    else                 /* greater than into right subtree */
        p->right = addtree(p->right, w);
    return p;
}
```

Storage for the new node is fetched by a routine talloc, which returns a pointer to a free space suitable for holding a tree node, and the new word is copied to a hidden place by strdup. (We will discuss these routines in a moment.) The count is initialized, and the two children are made null. This part of the code is executed only at the leaves of the tree, when a new node is being added. We have (unwisely) omitted error checking on the values returned by strdup and talloc.

treeprint prints the tree in sorted order; at each node, it prints the left subtree (all the words less than this word), then the word itself, then the right subtree (all the words greater). If you feel shaky about how recursion works, simulate treeprint as it operates on the tree shown above.

```
/* treeprint:  in-order print of tree p */
void treeprint(struct tnode *p)
{
    if (p != NULL) {
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}
```

A practical note: if the tree becomes "unbalanced" because the words don't arrive in random order, the running time of the program can grow too much. As a worst case, if the words are already in order, this program does an expensive simulation of linear search. There are generalizations of the binary tree that do not suffer from this worst-case behavior, but we will not describe them here.

Before we leave this example, it is also worth a brief digression on a problem related to storage allocators. Clearly it's desirable that there be only one storage allocator in a program, even though it allocates different kinds of objects. But if one allocator is to process requests for, say, pointers to `char`s and pointers to `struct tnode`s, two questions arise. First, how does it meet the requirement of most real machines that objects of certain types must satisfy alignment restrictions (for example, integers often must be located at even addresses)? Second, what declarations can cope with the fact that an allocator must necessarily return different kinds of pointers?

Alignment requirements can generally be satisfied easily, at the cost of some wasted space, by ensuring that the allocator always returns a pointer that meets *all* alignment restrictions. The `alloc` of Chapter 5 does not guarantee any particular alignment, so we will use the standard library function `malloc`, which does. In Chapter 8 we will show one way to implement `malloc`.

The question of the type declaration for a function like `malloc` is a vexing one for any language that takes its type-checking seriously. In C, the proper method is to declare that `malloc` returns a pointer to `void`, then explicitly coerce the pointer into the desired type with a cast. `malloc` and related routines are declared in the standard header `<stdlib.h>`. Thus `talloc` can be written as

```
#include <stdlib.h>

/* talloc:  make a tnode */
struct tnode *talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}
```

`strdup` merely copies the string given by its argument into a safe place, obtained by a call on `malloc`:

```
char *strdup(char *s)    /* make a duplicate of s */
{
    char *p;

    p = (char *) malloc(strlen(s)+1);  /* +1 for '\0' */
    if (p != NULL)
        strcpy(p, s);
    return p;
}
```

`malloc` returns `NULL` if no space is available; `strdup` passes that value on, leaving error-handling to its caller.

Storage obtained by calling `malloc` may be freed for re-use by calling `free`; see Chapters 7 and 8.

Exercise 6-2. Write a program that reads a C program and prints in alphabetical order each group of variable names that are identical in the first 6 characters, but different somewhere thereafter. Don't count words within strings and comments. Make 6 a parameter that can be set from the command line. □

Exercise 6-3. Write a cross-referencer that prints a list of all words in a document, and, for each word, a list of the line numbers on which it occurs. Remove noise words like "the," "and," and so on. □

Exercise 6-4. Write a program that prints the distinct words in its input sorted into decreasing order of frequency of occurrence. Precede each word by its count. □

6.6 Table Lookup

In this section we will write the innards of a table-lookup package, to illustrate more aspects of structures. This code is typical of what might be found in the symbol table management routines of a macro processor or a compiler. For example, consider the `#define` statement. When a line like

```
#define  IN  1
```

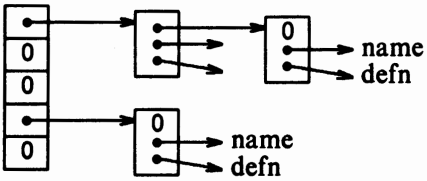
is encountered, the name `IN` and the replacement text `1` are stored in a table. Later, when the name `IN` appears in a statement like

```
state = IN;
```

it must be replaced by `1`.

There are two routines that manipulate the names and replacement texts. `install(s,t)` records the name `s` and the replacement text `t` in a table; `s` and `t` are just character strings. `lookup(s)` searches for `s` in the table, and returns a pointer to the place where it was found, or `NULL` if it wasn't there.

The algorithm is a hash search—the incoming name is converted into a small non-negative integer, which is then used to index into an array of pointers. An array element points to the beginning of a linked list of blocks describing names that have that hash value. It is NULL if no names have hashed to that value.



A block in the list is a structure containing pointers to the name, the replacement text, and the next block in the list. A null next-pointer marks the end of the list.

```
struct nlist {          /* table entry: */
    struct nlist *next;   /* next entry in chain */
    char *name;           /* defined name */
    char *defn;           /* replacement text */
};
```

The pointer array is just

```
#define HASHSIZE 101

static struct nlist *hashtab[HASHSIZE]; /* pointer table */
```

The hashing function, which is used by both lookup and install, adds each character value in the string to a scrambled combination of the previous ones and returns the remainder modulo the array size. This is not the best possible hash function, but it is short and effective.

```
/* hash:  form hash value for string s */
unsigned hash(char *s)
{
    unsigned hashval;

    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + 31 * hashval;
    return hashval % HASHSIZE;
}
```

Unsigned arithmetic ensures that the hash value is non-negative.

The hashing process produces a starting index in the array hashtab; if the string is to be found anywhere, it will be in the list of blocks beginning there. The search is performed by lookup. If lookup finds the entry already present, it returns a pointer to it; if not, it returns NULL.

```
/* lookup:  look for s in hashtab */
struct nlist *lookup(char *s)
{
    struct nlist *np;

    for (np = hashtab[hash(s)]; np != NULL; np = np->next)
        if (strcmp(s, np->name) == 0)
            return np;  /* found */
    return NULL;        /* not found */
}
```

The `for` loop in `lookup` is the standard idiom for walking along a linked list:

```
for (ptr = head; ptr != NULL; ptr = ptr->next)
    ...
```

`install` uses `lookup` to determine whether the name being installed is already present; if so, the new definition will supersede the old one. Otherwise, a new entry is created. `install` returns `NULL` if for any reason there is no room for a new entry.

```
struct nlist *lookup(char *);
char *strdup(char *);

/* install:  put (name, defn) in hashtab */
struct nlist *install(char *name, char *defn)
{
    struct nlist *np;
    unsigned hashval;

    if ((np = lookup(name)) == NULL) {  /* not found */
        np = (struct nlist *) malloc(sizeof(*np));
        if (np == NULL || (np->name = strdup(name)) == NULL)
            return NULL;
        hashval = hash(name);
        np->next = hashtab[hashval];
        hashtab[hashval] = np;
    } else      /* already there */
        free((void *) np->defn);  /* free previous defn */
    if ((np->defn = strdup(defn)) == NULL)
        return NULL;
    return np;
}
```

Exercise 6-5. Write a function `undef` that will remove a name and definition from the table maintained by `lookup` and `install`. □

Exercise 6-6. Implement a simple version of the `#define` processor (i.e., no arguments) suitable for use with C programs, based on the routines of this section. You may also find `getch` and `ungetch` helpful. □

6.7 Typedef

C provides a facility called `typedef` for creating new data type names. For example, the declaration

```
typedef int Length;
```

makes the name `Length` a synonym for `int`. The type `Length` can be used in declarations, casts, etc., in exactly the same ways that the type `int` can be:

```
Length  len, maxlen;
Length  *lengths[];
```

Similarly, the declaration

```
typedef char *String;
```

makes `String` a synonym for `char *` or character pointer, which may then be used in declarations and casts:

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

Notice that the type being declared in a `typedef` appears in the position of a variable name, not right after the word `typedef`. Syntactically, `typedef` is like the storage classes `extern`, `static`, etc. We have used capitalized names for `typedef`s, to make them stand out.

As a more complicated example, we could make `typedef`s for the tree nodes shown earlier in this chapter:

```
typedef struct tnode *Treeptr;

typedef struct tnode {   /* the tree node: */
    char *word;              /* points to the text */
    int count;               /* number of occurrences */
    Treeptr left;            /* left child */
    Treeptr right;           /* right child */
} Treenode;
```

This creates two new type keywords called `Treenode` (a structure) and `Treeptr` (a pointer to the structure). Then the routine `talloc` could become

```
Treeptr talloc(void)
{
    return (Treeptr) malloc(sizeof(Treenode));
}
```

It must be emphasized that a `typedef` declaration does not create a new type in any sense; it merely adds a new name for some existing type. Nor are there any new semantics: variables declared this way have exactly the same properties as variables whose declarations are spelled out explicitly. In effect, `typedef` is like `#define`, except that since it is interpreted by the compiler, it