```
	char *strsave(s)	/* save string s somewhere */
	char *s;
	{
		char *p, *alloc();

		if ((p = alloc(strlen(s)+1)) != NULL)
			strcpy(p, s);
		return(p);
	}
```

In practice, there would be a strong tendency to omit declarations:

```
	strsave(s)	/* save string s somewhere */
	{
		char *p;

		if ((p = alloc(strlen(s)+1)) != NULL)
			strcpy(p, s);
		return(p);
	}
```

This will work on many machines, since the default type for functions and arguments is `int`, and `int` and pointer can usually be safely assigned back and forth. Nevertheless this kind of code is inherently risky, for it depends on details of implementation and machine architecture which may not hold for the particular compiler you use. It's wiser to be complete in all declarations. (The program *lint* will warn of such constructions, in case they creep in inadvertently.)

5.7 Multi-Dimensional Arrays

C provides for rectangular multi-dimensional arrays, although in practice they tend to be 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 returns two values, the month and day arguments will be pointers:

```
	month_day(1977, 60, &m, &d)
```

sets `m` to 3 and `d` to 1 (March 1st).

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 try to keep track of what happens to February during computation. The array and the functions for performing the transformations are as follows:

```
static int day_tab[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(year, month, day) /* set day of year */
int year, month, day;         /* from month & day */
{
     int i, leap;

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

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

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

The array `day_tab` has to be external to both `day_of_year` and `month_day`, so they can both use it.

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

```
day_tab[i][j]
```

rather than

```
day_tab[i, j]
```

as in most languages. Other than this, a two-dimensional array can be treated in much the same way as in other languages. Elements are stored by rows, that is, the rightmost subscript 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 `day_tab` 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 easier than adjusting indices.

If a two-dimensional array is to be passed to a function, the argument declaration in the function *must* include the column dimension; the row dimension is irrelevant, since what is passed is, as before, a pointer. In this particular case, it is a pointer to objects which are arrays of 13 `int`'s. Thus if the array `day_tab` is to be passed to a function `f`, the declaration of `f` would be

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

The argument declaration in `f` could also be

```
int day_tab[][13];
```

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

```
int (*day_tab)[13];
```

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

```
int *day_tab[13];
```

is an array of 13 pointers to integers, as we shall see in the next section.

5.8 Pointer Arrays; Pointers to Pointers

Since pointers are variables themselves, you might expect that there would be uses for arrays of pointers. This is indeed the case. 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 utility *sort*.

In Chapter 3 we presented a Shell sort function that would sort an array of integers. The same algorithm 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 (maintained by `alloc`, perhaps), 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 actual lines.

The sorting process involves 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 things.

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.

```
#define NULL   0
#define LINES  100  /* max lines to be sorted */

main()    /* sort input lines */
{
     char *lineptr[LINES]; /* pointers to text lines */
     int  nlines;         /* number of input lines read */

     if ((nlines = readlines(lineptr, LINES)) >= 0) {
          sort(lineptr, nlines);
          writelines(lineptr, nlines);
     }
     else
          printf("input too big to sort\n");
}
```

```
#define MAXLEN  1000

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

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

The newline at the end of each line is deleted so it will not affect the order in which the lines are sorted.

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

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

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

```
char *lineptr[LINES];
```

says that `lineptr` is an array of `LINES` elements, each element of which is a pointer to a `char`. That is, `lineptr[i]` is a character pointer, and `*lineptr[i]` accesses a character.

Since `lineptr` is itself an array which is passed to `writelines`, it can be treated as a pointer in exactly the same manner as our earlier examples, and the function can be written instead as

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

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

With input and output under control, we can proceed to sorting. The Shell sort from Chapter 3 needs minor changes: the declarations have to be modified, and the comparison operation must be moved into a separate function. The basic algorithm remains the same, which gives us some confidence that it will still work.

```
sort(v, n)       /* sort strings v[0] ... v[n-1] */
char *v[];           /* into increasing order */
int n;
{
     int  gap, i, j;
     char *temp;

     for (gap = n/2; gap > 0; gap /= 2)
          for (i = gap; i < n; i++)
               for (j = i-gap; j >= 0; j -= gap) {
                    if (strcmp(v[j], v[j+gap]) <= 0)
                         break;
                    temp = v[j];
                    v[j] = v[j+gap];
                    v[j+gap] = temp;
               }
}
```

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

We wrote the program about as straightforwardly as possible, so as to get it working quickly. It might be faster, for instance, to copy the incoming lines directly into an array maintained by `readlines`, rather than copying them into `line` and then to a hidden place maintained by `alloc`. But it's wiser to make the first draft something easy to understand, and worry about "efficiency" later. The way to make this program significantly faster is probably not by avoiding an unnecessary copy of the input lines. Replacing the Shell sort by something better, like Quicksort, is more likely to make a difference.

In Chapter 1 we pointed out that because `while` and `for` loops test the termination condition *before* executing the loop body even once, they help to ensure that programs will work at their boundaries, in particular with no input. It is illuminating to walk through the functions of the sorting program, checking what happens if there is no input text at all.

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

5.9 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. The topic of this section is how that array of names is initialized.

The syntax is quite similar to previous initializations:

```
char *month_name(n) /* return name of n-th month */
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 simply a list of character strings; each is assigned to the corresponding position in the array. More precisely, the characters of the `i`-th string are placed somewhere else, and a pointer to them is stored in `name[i]`. Since the size of the array `name` is not specified, the compiler itself counts the initializers and fills in the correct number.

5.10 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 declarations

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

the usage of `a` and `b` may be similar, in that `a[5][5]` and `b[5][5]` are both legal references to a single `int`. But `a` is a true array: all 100 storage cells have been allocated, and the conventional rectangular subscript calculation is done to find any given element. For `b`, however, the declaration only allocates 10 pointers; each must be set to point to an array of integers. Assuming that each does point to a ten-element array, then there will be 100 storage cells set aside, plus the ten cells for the pointers. Thus the array of pointers uses slightly more space, and may require an explicit initialization step. But it has two advantages: accessing an element is done by indirection through a pointer rather than by a multiplication and an addition, and the rows of the array may be of different lengths. That is, each element of `b` need not point to a ten-element vector; some may point to two elements, some to twenty, 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 like that shown in `month_name`: to store character strings of diverse lengths.

Exercise 5-6. Rewrite the routines `day_of_year` and `month_day` with pointers instead of indexing. □

5.11 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 to begin execution, it is called with two arguments. The first (conventionally called `argc`) is the number of command-line arguments the program was invoked with; the second (`argv`) is a pointer to an array of character strings that contain the arguments, one per string. Manipulating these character strings is a common use of multiple levels of pointers.

The simplest illustration of the necessary declarations and use is the program `echo`, which simply echoes its command-line arguments on a single line, separated by blanks. That is, if the command

```
echo hello, world
```

is given, the output is

```
hello, world
```

By convention, argv[0] is the name by which the program was invoked, so argc is at least 1. In the example above, argc is 3, and argv[0], argv[1] and argv[2] are "echo", "hello,", and "world" respectively. The first real argument is argv[1] and the last is argv[argc-1]. If argc is 1, there are no command-line arguments after the program name. This is shown in echo:

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

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

Since argv is a pointer to an array of pointers, there are several ways to write this program that involve manipulating the pointer rather than indexing an array. Let us show two variations.

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

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,

```
main(argc, argv)    /* echo arguments; 3rd version */
int argc;
char *argv[];
{
	while (--argc > 0)
		printf((argc > 1) ? "%s " : "%s\n", *++argv);
}
```

This version shows that the format argument of printf can be an expression just like any of the others. This usage is not very frequent, but worth remembering.

As a second example, let us make some enhancements to the pattern-finding program from Chapter 4. If you recall, we wired the search pattern deep into the program, an obviously unsatisfactory arrangement. Following the lead of the UNIX utility *grep*, let us change the program so the pattern to be matched is specified by the first argument on the command line.

```
#define  MAXLINE  1000

main(argc, argv)  /* find pattern from first argument */
int argc;
char *argv[];
{
     char line[MAXLINE];

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

The basic 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 with its line number."

A common convention for C programs is that an argument beginning 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  the
```

with the input

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

should produce the output

```
2: for all good men
```

? Optional arguments should be permitted in any order, and the rest of the program should be insensitive to the number of arguments which were actually present. In particular, the call to `index` should not refer to `argv[2]` when there was a single flag argument and to `argv[1]` when there wasn't. Furthermore, it is convenient for users if option arguments can be concatenated, as in

```
find -nx the
```

Here is the program.

```
#define   MAXLINE   1000

main(argc, argv)  /* find pattern from first argument */
int argc;
char *argv[];
{
        char line[MAXLINE], *s;
        long lineno = 0;
        int  except = 0, number = 0;

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

argv is incremented before each optional argument, and argc decremented. If there are no errors, at the end of the loop argc should be 1 and *argv should point at the pattern. Notice that *++argv is a pointer to an argument string; (*++argv)[0] is its first character. The parentheses are necessary, for without them the expression would be *++(argv[0]), which is quite different (and wrong). An alternate valid form would be

`**++argv`.

Exercise 5-7. Write the program `add` which evaluates a reverse Polish expression from the command line. For example,

```
add  2  3  4  +  *
```

evaluates 2 × (3+4). □

Exercise 5-8. Modify the programs `entab` and `detab` (written as exercises in Chapter 1) to accept a list of tab stops as arguments. Use the normal tab settings if there are no arguments. □

Exercise 5-9. Extend `entab` and `detab` to accept the shorthand

```
entab m +n
```

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

Exercise 5-10. 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 `sort`, not in a two-dimensional array of fixed size. □

5.12 Pointers to Functions

In C, a function itself is not a variable, but it is possible to define a *pointer to a function*, which can be manipulated, passed to functions, placed in arrays, 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* which determines the ordering of any pair of objects, an *exchange* which reverses their order, and a *sorting algorithm* which 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.

The lexicographic comparison of two lines is done by `strcmp` and swapping by `swap` as before; we will also need a routine `numcmp` which compares two lines on the basis of numeric value and returns the same kind of condition indication as `strcmp` does. These three functions are declared in main and pointers to them are passed to sort. sort in turn calls the functions via the pointers. We have skimped on error processing for arguments, so as to concentrate on the main issues.

```
#define LINES 100  /* max number of lines to be sorted */

main(argc, argv)    /* sort input lines */
int argc;
char *argv[];
{
	char *lineptr[LINES];  /* pointers to text lines */
	int  nlines;         /* number of input lines read */
	int  strcmp(), numcmp(); /* comparison functions */
	int  swap();   /* exchange function */
	int  numeric = 0;   /* 1 if numeric sort */

	if (argc>1 && argv[1][0] == '-' && argv[1][1] == 'n')
		numeric = 1;
	if ((nlines = readlines(lineptr, LINES)) >= 0) {
		if (numeric)
			sort(lineptr, nlines, numcmp, swap);
		else
			sort(lineptr, nlines, strcmp, swap);
		writelines(lineptr, nlines);
	} else
		printf("input too big to sort\n");
}
```

strcmp, numcmp and swap 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. The compiler arranges for the address of the function to be passed.

The second step is to modify sort:

```
sort(v, n, comp, exch)   /* sort strings v[0]...v[n-1] */
char *v[];               /* into increasing order */
int n;
int (*comp)(), (*exch)();
{
	int gap, i, j;

	for (gap = n/2; gap > 0; gap /= 2)
		for (i = gap; i < n; i++)
			for (j = i-gap; j >= 0; j -= gap) {
				if ((*comp)(v[j], v[j+gap]) <= 0)
					break;
				(*exch)(&v[j], &v[j+gap]);
			}
}
```

The declarations should be studied with some care.

```
int (*comp)()
```

says that comp is a pointer to a function that returns an int. The first set of parentheses are necessary; without them,

```
int *comp()
```

would say that comp is a function returning a pointer to an int, which is quite a different thing.

The use of comp in the line

```
if ((*comp)(v[j], v[j+gap]) <= 0)
```

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

```
(*comp)(v[j], v[j+gap])
```

is the call to it. The parentheses are needed so the components are correctly associated.

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

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

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

The final step is to add the function `swap` which exchanges two pointers. This is adapted directly from what we presented early in the chapter.

```
swap(px, py)	/* interchange *px and *py */
char *px[], *py[];
{
	char *temp;

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

There are a variety of other options that can be added to the sorting program; some make challenging exercises.

Exercise 5-11. Modify `sort` to handle a `-r` flag, which indicates sorting in reverse (decreasing) order. Of course `-r` must work with `-n`. □

Exercise 5-12. Add the option `-f` to fold upper and lower case together, so that case distinctions are not made during sorting: upper and lower case data are sorted together, so that `a` and `A` appear adjacent, not separated by an entire case of the alphabet. □

Exercise 5-13. Add the `-d` ("dictionary order") option, which makes comparisons only on letters, numbers and blanks. Make sure it works in conjunction with `-f`. □

Exercise 5-14. Add a field-handling capability, so sorting may be done on fields within lines, each field 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.) □

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, most notably Pascal.)

The 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.

Structures help to organize complicated data, particularly in large programs, because in many situations they permit a group of related variables to be treated as a unit instead of as separate entities. In this chapter we will try to illustrate how structures are used. The programs we will use are bigger than many of the others in the book, but still of modest size.

6.1 Basics

Let us revisit the date conversion routines of Chapter 5. A date consists of several parts, such as the day, month, and year, and perhaps the day of the year and the month name. These five variables can all be placed into a single structure like this:

```
struct date {
     int  day;
     int  month;
     int  year;
     int  yearday;
     char mon_name[4];
};
```

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 `date` here). The tag names this kind of structure, and can be used subsequently as a shorthand for the detailed declaration.

The elements or variables mentioned 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. Of course as a matter of style one would normally use the same names only for closely related objects.

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 allocated for them.

A structure declaration that is not followed by a list of variables allocates 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 actual instances of the structure. For example, given the declaration of `date` above,

```
    struct date d;
```

defines a variable `d` which is a structure of type `date`. An external or static structure can be initialized by following its definition with a list of initializers for the components:

```
    struct date d ={ 4, 7, 1776, 186, "Jul" };
```

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 set `leap` from the date in structure `d`, for example,

```
    leap = d.year % 4 == 0 && d.year % 100 != 0
                || d.year % 400 == 0;
```

or to check the month name,

```
    if (strcmp(d.mon_name, "Aug") == 0) ...
```

or to convert the first character of the month name to lower case,

```
    d.mon_name[0] = lower(d.mon_name[0]);
```

Structures can be nested; a payroll record might actually look like

```
struct person {
     char name[NAMESIZE];
     char address[ADRSIZE];
     long zipcode;
     long ss_number;
     double salary;
     struct date birthdate;
     struct date hiredate;
};
```

The person structure contains two dates. If we declare emp as

```
struct person emp;
```

then

```
emp.birthdate.month
```

refers to the month of birth. The structure member operator . associates left to right.

6.2 Structures and Functions

There are a number of restrictions on C structures. The essential rules are that the only operations that you can perform on a structure are take its address with &, and access one of its members. This implies that structures may not be assigned to or copied as a unit, and that they can not be passed to or returned from functions. (These restrictions will be removed in forthcoming versions.) Pointers to structures do not suffer these limitations, however, so structures and functions do work together comfortably. Finally, automatic structures, like automatic arrays, cannot be initialized; only external or static structures can.

Let us investigate some of these points by rewriting the date conversion functions of the last chapter to use structures. Since the rules prohibit passing a structure to a function directly, we must either pass the components separately, or pass a pointer to the whole thing. The first alternative uses day_of_year as we wrote it in Chapter 5:

```
d.yearday = day_of_year(d.year, d.month, d.day);
```

The other way is to pass a pointer. If we have declared hiredate as

```
struct date hiredate;
```

and re-written day_of_year, we can then say

```
hiredate.yearday = day_of_year(&hiredate);
```

to pass a pointer to hiredate to day_of_year. The function has to be modified because its argument is now a pointer rather than a list of variables.

```
day_of_year(pd)  /* set day of year from month, day */
struct date *pd;
{
	int i, day, leap;

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

The declaration

```
struct date *pd;
```

says that `pd` is a pointer to a structure of type `date`. The notation exemplified by

```
pd->year
```

is new. If `p` is a pointer to a structure, then

p->*member-of-structure*

refers to the particular member. (The operator `->` is a minus sign followed by `>`.)

Since `pd` points to the structure, the `year` member could also be referred to as

```
(*pd).year
```

but pointers to structures are so frequently used that the `->` notation is provided as a convenient shorthand. The parentheses are necessary in `(*pd).year` because the precedence of the structure member operator `.` is higher than `*`. Both `->` and `.` associate from left to right, so

```
p->q->memb
emp.birthdate.month
```

are

```
(p->q)->memb
(emp.birthdate).month
```

For completeness here is the other function, `month_day`, rewritten to use the structure.

```
month_day(pd)  /* set month and day from day of year */
struct date *pd;
{
     int i, leap;

     leap = pd->year % 4 == 0 && pd->year % 100 != 0
               || pd->year % 400 == 0;
     pd->day = pd->yearday;
     for (i = 1; pd->day > day_tab[leap][i]; i++)
          pd->day -= day_tab[leap][i];
     pd->month = i;
}
```

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

```
struct {
     int  x;
     int  *y;
} *p;
```

then

```
++p->x
```

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

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

6.3 Arrays of Structures

Structures are especially suitable for managing arrays of related variables. For instance, consider 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 indicates that a different organization is possible. Each keyword entry is really a pair:

```
	char *keyword;
	int  keycount;
```

and there is an array of pairs. The structure declaration

```
	struct key {
		char *keyword;
		int  keycount;
	} keytab[NKEYS];
```

defines an array `keytab` of structures of this type, and allocates storage to them. Each element of the array is a structure. This could also be written

```
	struct key {
		char *keyword;
		int  keycount;
	};

	struct key keytab[NKEYS];
```

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

```
	struct key {
		char *keyword;
		int  keycount;
	} keytab[] ={
		"break", 0,
		"case", 0,
		"char", 0,
		"continue", 0,
		"default", 0,
		/* ... */
		"unsigned", 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

```
		{ "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 compiler will compute the number of entries in the array `keytab` if initializers are present and the `[]` is left empty.