CHAPTER 5 POINTERS AND ARRAYS 105

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 it'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] [1 3] ;**

* • •

The argument declaration in f could also be

**int day\_tab [1 [1 3] ;**

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

**int (\*day\_tab) [1 3] ;**

-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 [1 3] ;**

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 illus­
  
trate 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.

106 THE C PROGRAMMING LANGUAGE CHAPTER 5

The pointers themselves can be stored in an array. Two lines can be com­
  
pared 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");**

**CHAPTER 5 POINTERS AND ARRAYS 107**

**#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 (**

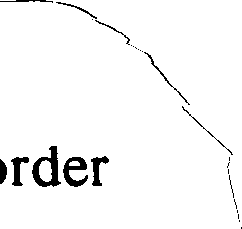
**line[len-1] = '\0'; /\* zap newline \*/**

**strcpy(p, line);**

**lineptr[nlines++] = p;**

**}**

**return (nlines);**



**)**

**The newline at the end of each line is deleted so it will not affect the c** 
  
**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 NJ 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 exam­**
  
**ples, and the function can be written instead as**

**108** THE C PROGRAMMING LANGUAGE CHAPTER 5

**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 a 1 loc. 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

CHAPTER 5 POINTERS AND ARRAYS 109

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 pro­
  
gram, 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 **anoc** to maintain storage. How much faster is
  
the program? o

**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 \*namell =(
  
"illegal month",
  
"January",**

**"February",**

**"March",**

**"April",**

**"June",**

**"July",**

**"August",**

**"September",
  
"October",
  
"November",
  
"December"**

) ;

**return((n < 1 II n > 12) ? name [01 : 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 char­
  
acter 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 [ii.** Since the size of the array
  
**name** is not specified, the compiler itself counts the initializers and fills in
  
the correct number.

110 THE C PROGRAMMING LANGUAGE CHAPTER 5

**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 exam­
  
ple above. Given the declarations

**int a [1 0] [1 0] ;
  
int \*b [1 0] ;**

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 calcula­
  
tion 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 initializa­
  
tion 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 ele­
  
ments, 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. D

**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 pro­
  
gram 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**

CHAPTER 5 POINTERS AND ARRAYS 111

**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 [01,** 
  
**argv [1] and argv [2] are "echo", "he llo , ", and "world" respec­**
  
**tively. 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 index­**
  
**ing 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 \*argil[];**

**while (--argc > 0)**

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

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

112 THE C PROGRAMMING LANGUAGE CHAPTER 5

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("%su, line);**

The basic model can now be elaborated to illustrate further pointer con­
  
structions. Suppose we want to allow two optional arguments. One says
  
"print all lines *except* those that match the pattern;" the second says "pre­
  
cede 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 [21** when there was a single flag argument and to **argv [11** when
  
there wasn't. Furthermore, it is convenient for users if option arguments

CHAPTER 5 POINTERS AND ARRAYS **113**

**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);**

1

)

)

**argv is incremented before each optional argument, and argc decre-**
  
**mented. If there are no errors, at the end of the loop argc should be 1 and** 
  
**\*argv should point at the pattern. Notice that \*-1-+argy 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**

114 THE C PROGRAMMING LANGUAGE CHAPTER 5

**\*\*++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 x (3**+4). o**

**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. o

**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 con­
  
venient (for the user) default behavior. 0

**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 pro­
  
cedure 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 com­
  
parison 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 swap­
  
ping by **swap** as before; we will also need a routine **numcmp** which com­
  
pares 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

**CHAPTER 5 POINTERS AND ARRAYS 115**

**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 argu­**
  
**ments, 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:**

116 THE C PROGRAMMING LANGUAGE CHAPTER 5

**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 1= 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+gapp;**

)

)

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:

**CHAPTER 5 POINTERS AND ARRAYS** 117

**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 pro­
  
gram; 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. 0

**Exercise 5-13. Add the —d** ("dictionary order") option, which makes com­
  
parisons only on letters, numbers and blanks. Make sure it works in con­
  
junction with **—f. 0**

**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.)**