

5.10 Pointers vs. Multi-dimensional Arrays (110)

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.

Read this
here.

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

5.11 Command-line Arguments (110) - 112

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