CHAPTER 5 POINTERS AND ARRAYS 103

**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 declara­
  
tions. (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 argu­
  
ments 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

104 THE C PROGRAMMING LANGUAGE CHAPTER 5

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 II 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 II year%400 == 0;**

**for (i = 1; yearday > day\_tab[leap][i]; i++)**

**yearday -= day\_tab[leap][i];**

**\*pmonth =**

**\*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.