Lecture 6
Pointers-Part 3

Dr. Hacer Yalım Keleş

Ref: Programming in ANSI C, Kumar

MULTI-DIMENSIONAL ARRAYS AND POINTERS

A multi-dimensional array in C is really a one-dimensional array, whose elements are themselves arrays, and is stored such that the last subscript varies most rapidly. The name of the multi-dimensional array is, therefore, a pointer to the first array. Thus, the declaration

```
    int matrix [3] [5] ;
```

specifies that the array `matrix` consists of three elements, each of which is an array of five integer elements, and that the name `matrix` is a pointer to the first row of the matrix.

Instead of using subscripts, an element in a multi-dimensional array can be referenced using an equivalent pointer expression. For example, the element `matrix[ i ][ j ]` can be referenced using the pointer expression

```
*(*(matrix+i)+j)
```

since

`matrix` is a pointer to the first row;

`matrix+i` is a pointer to the ith row;

`*(matrix+i)` is the ith row which is converted into a pointer to the first element in the ith row;

`*(matrix+i)+j` is a pointer to the jth element in the ith row; and

`*(*(matrix+i)+j)` is `matrix[i][j]`, the jth element in the ith row.

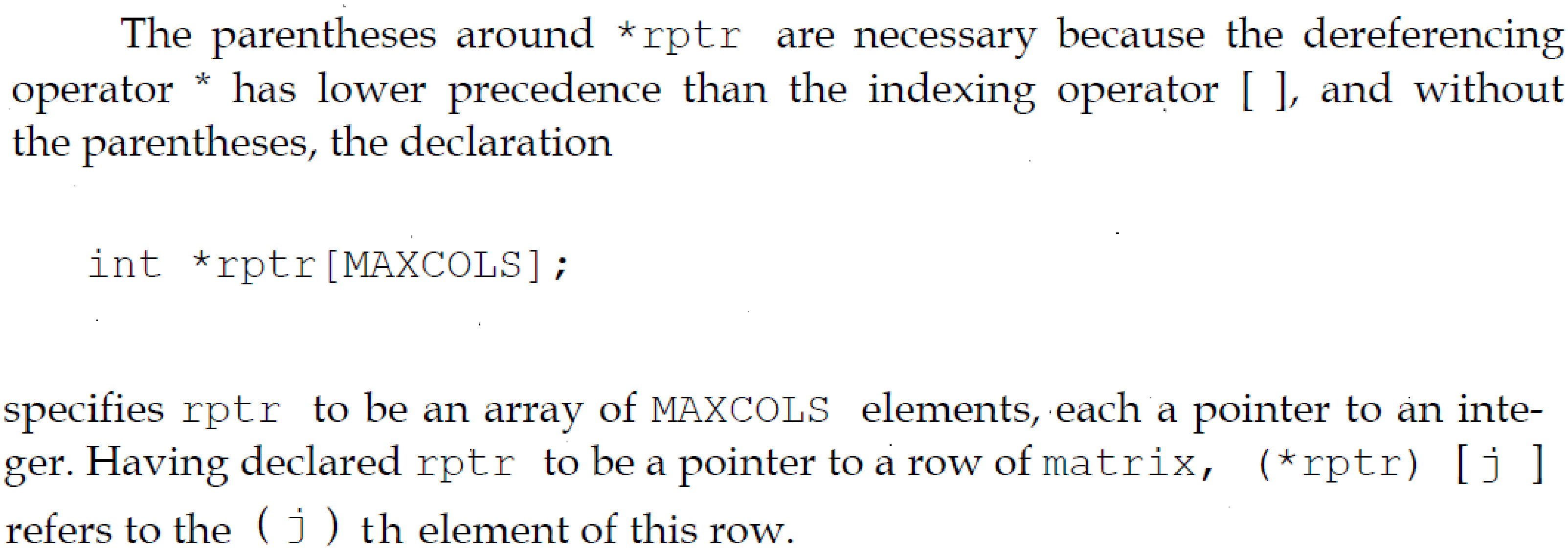
```
int matrix[MAXROWS][MAXCOLS];
```

Let `rptr` be the pointer to the rows of `matrix`.

We can declare and initialize this pointer to the first row of `matrix` as

```
int (*rptr)[MAXCOLS] = matrix;
```

This declaration specifies that `rptr` is a pointer to an array of `MAXCOLS` integers.



The parentheses around `*rptr` are necessary because the dereferencing operator * has lower precedence than the indexing operator [ ], and without the parentheses, the declaration

```
int *rptr[MAXCOLS];
```

specifies `rptr` to be an array of `MAXCOLS` elements, each a pointer to an integer. Having declared `rptr` to be a pointer to a row of `matrix`, `(*rptr) [j ]` refers to the `( j )` th element of this row.

```
int colsum(int (*matrix)[MAXCOLS],
           int rows, int column)
 {
   int (*rptr)[MAXCOLS] = matrix;
   int i, sum;

   for (i = 0, sum = 0; i < rows; i++)
       sum += (*rptr++)[column];
   return sum;
 }
```

Since, pointer arithmetic works correctly with pointers of any type, a two-dimensional array can be traversed by initializing a pointer to the first row of the array and then incrementing the pointer each time we need to get to the next row.

```
int (*rptr)[MAXCOLS] = matrix;
```

This declaration specifies that `rptr` is a pointer to an array of `MAXCOLS` integers.

`(*rptr++)[column];` incrementing

the pointer each time we get to the next row.

Note that the parameter declaration

```
    int (*matrix)[MAXCOLS]
```

specifies that `matrix` is a pointer to an array of `MAXCOLS` integer elements. This declaration is equivalent to

```
    int matrix[][MAXCOLS]
```

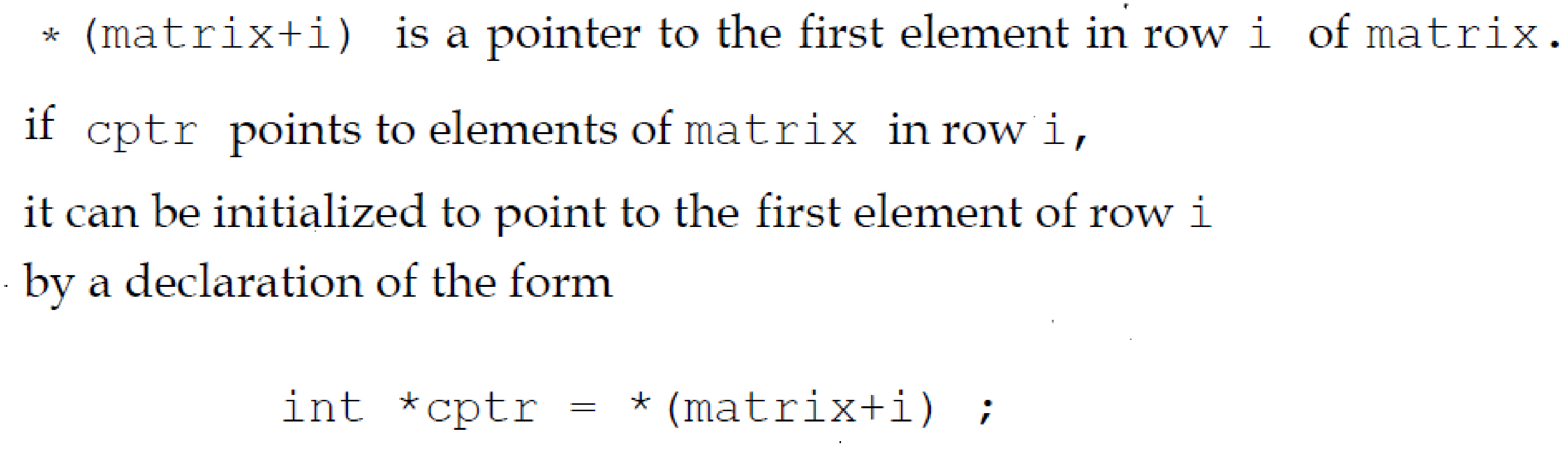
Given that

```
int m[][MAXCOLS] =
  {
     {1, 2, 3},
     {4, 5, 6}
  }
```

the function call

```
colsumf(m, 2, 0)
```

produces 5 as the sum of the first column.



`*(matrix+i)` is a pointer to the first element in row `i` of `matrix`.

if `cptr` points to elements of `matrix` in row `i`,

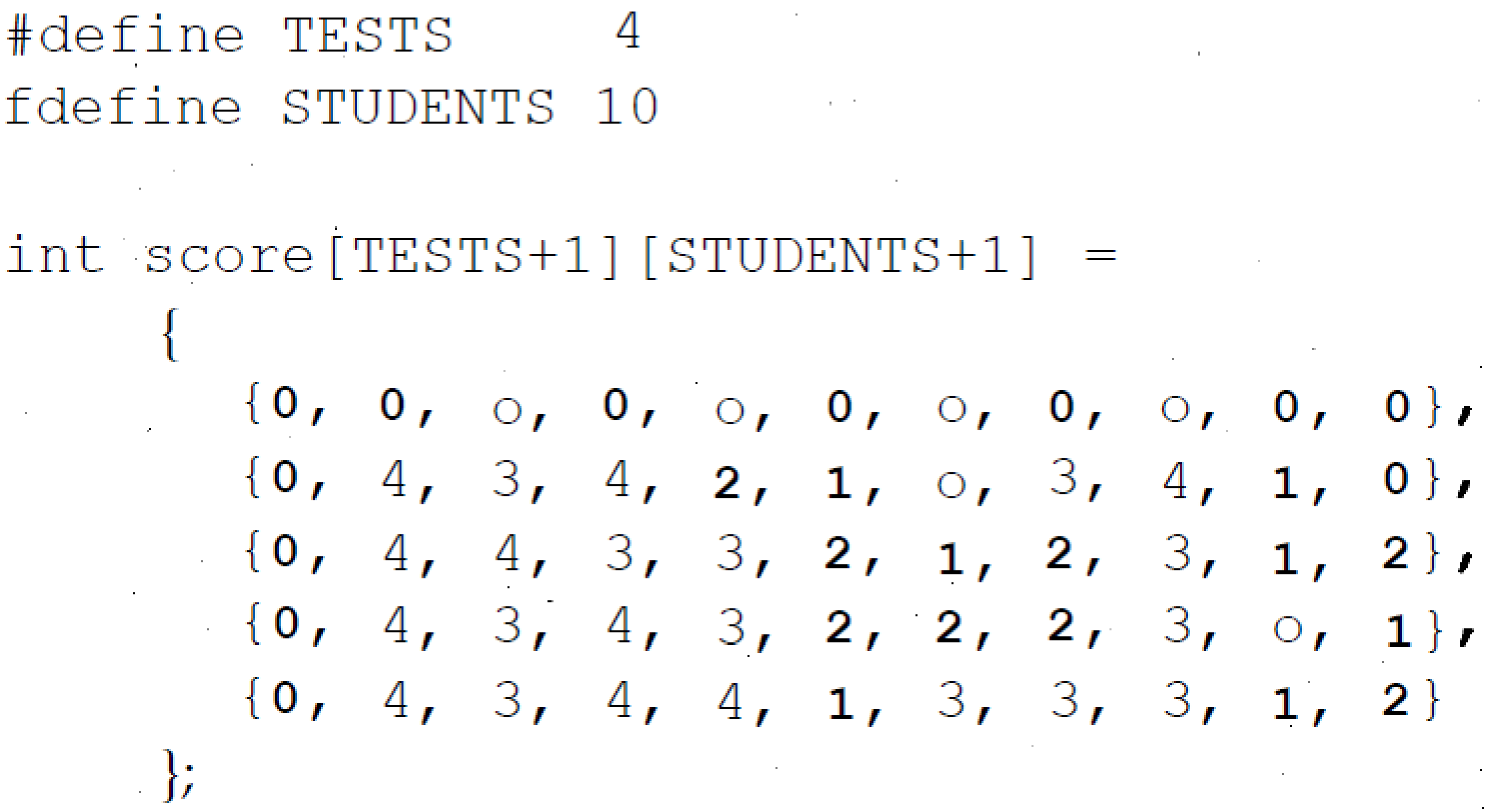
it can be initialized to point to the first element of row `i`

by a declaration of the form

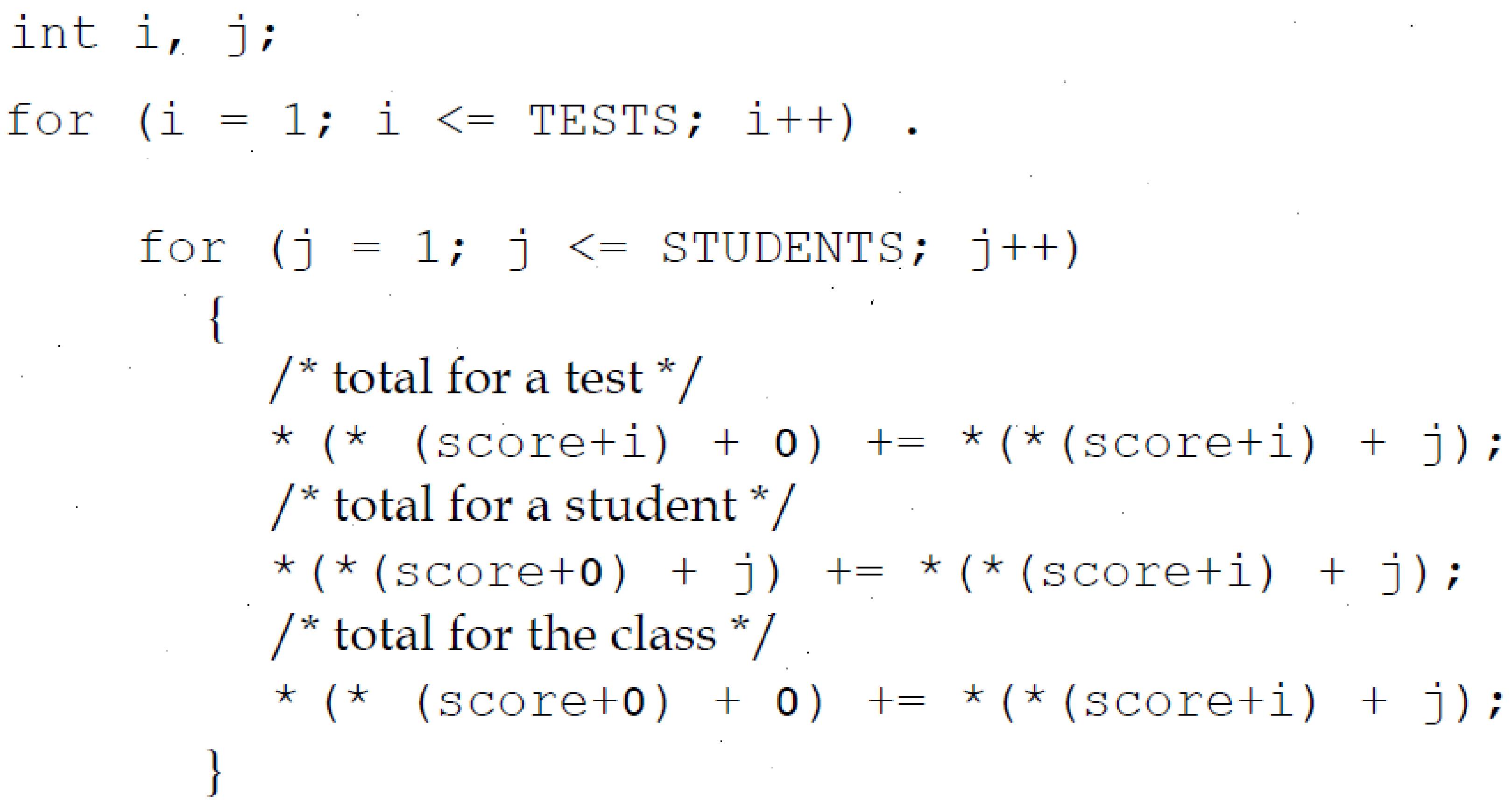
```
int *cptr = *(matrix+i) ;
```

The function `rowsum` is as follows:

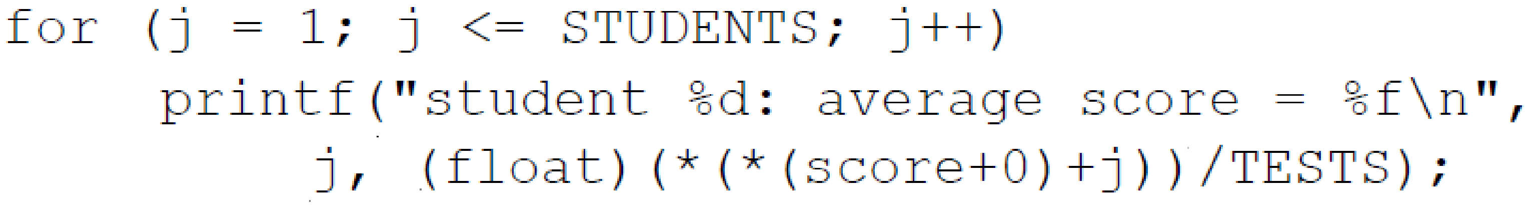
```
int rowsum(int (*matrix)[MAXCOLS],
             int columns, int row)
 {
   int *cptr = *(matrix+row) ;
   int j, sum;

   for (j = 0, sum =0; j < columns; j++)
        sum += *cptr++;
   return sum;
 }
```



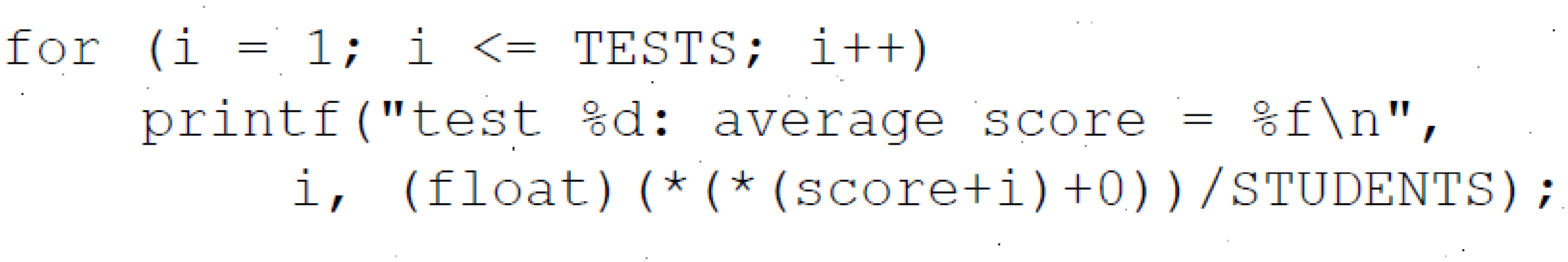
```
#define TESTS     4
fdefine STUDENTS 10

int score[TESTS+1][STUDENTS+1] =
    {
       {0, 0, o, 0, o, 0, o, 0, o, 0, 0},
       {0, 4, 3, 4, 2, 1, o, 3, 4, 1, 0},
       {0, 4, 4, 3, 3, 2, 1, 2, 3, 1, 2},
       {0, 4, 3, 4, 3, 2, 2, 2, 3, o, 1},
       {0, 4, 3, 4, 4, 1, 3, 3, 3, 1, 2}
    };
```



```
int i, j;
for (i = 1; i <= TESTS; i++) .

    for (j = 1; j <= STUDENTS; j++)
      {
        /* total for a test */
        * (* (score+i) + 0) += *(*(score+i) + j);
        /* total for a student */
        *(*(score+0) + j) += *(*(score+i) + j);
        /* total for the class */
        * (* (score+0) + 0) += *(*(score+i) + j);
      }
```

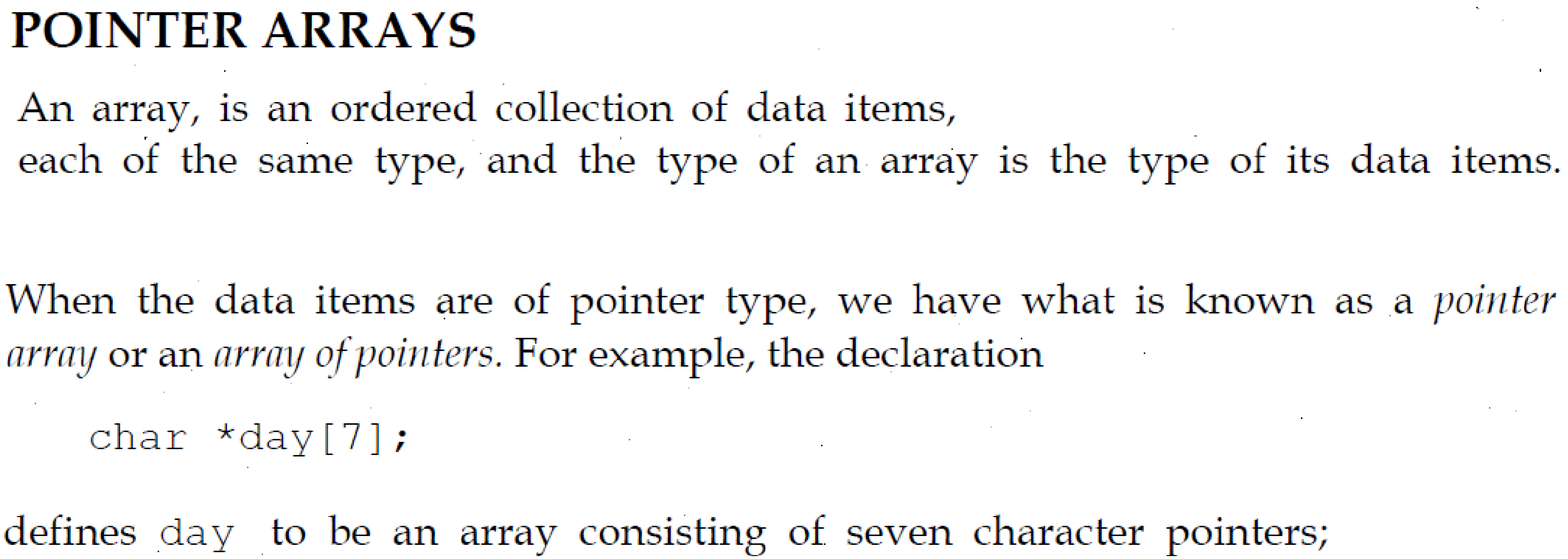


```
for (j = 1; j <= STUDENTS; j++)
     printf("student %d: average score = %f\n",
          j, (float)(*(*(score+0)+j))/TESTS);
```



```
for (i = 1; i <= TESTS; i++)
    printf("test %d: average score = %f\n",
        i, (float)(*(*(score+i)+0))/STUDENTS);
```

```
printf("class average = %f\n",
     (float)(*(*(score+0)+0))/(STUDENTS*TESTS));
```



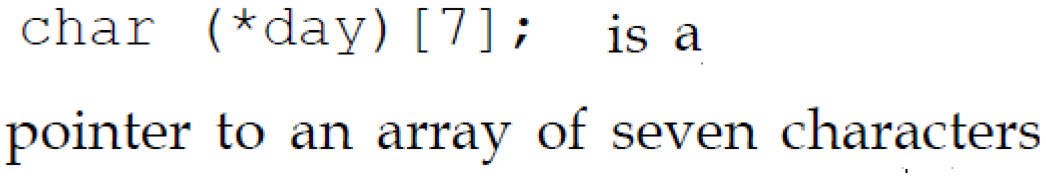
POINTER ARRAYS

An array, is an ordered collection of data items, each of the same type, and the type of an array is the type of its data items.

When the data items are of pointer type, we have what is known as a *pointer array* or an *array of pointers*. For example, the declaration

```
    char *day[7];
```

defines `day` to be an array consisting of seven character pointers;

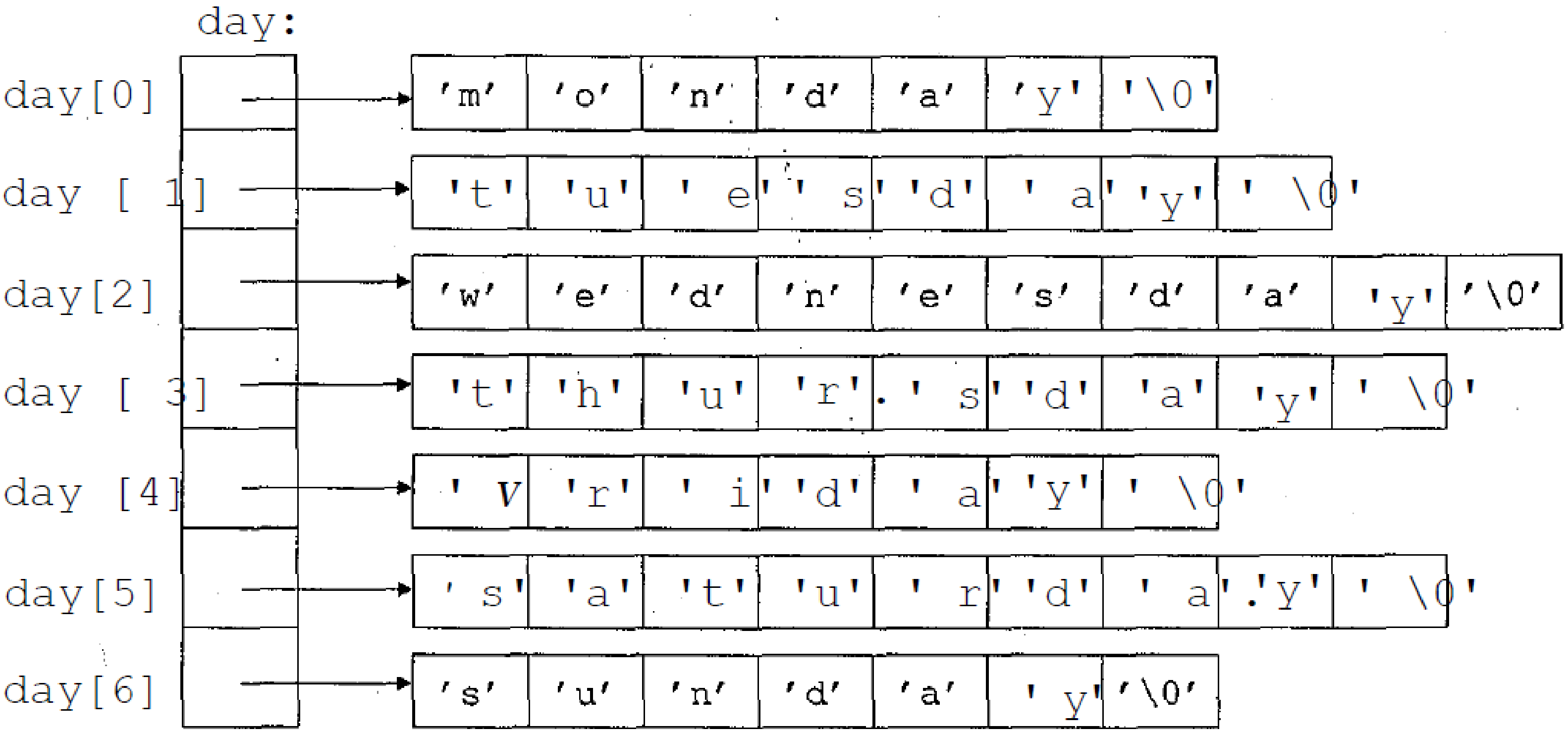


`char (*day)[7];` is a pointer to an array of seven characters

```
char *day[7] =
   {
      "monday", "tuesday", "Wednesday",
      "thursday", "friday", "Saturday", "sunday"
   };
```

or through assignment statements as in

```
    char *day[7];

    day[0] =   "monday";
    day[1] ='"tuesday";
    day[2] =  "Wednesday";
    day[3] =  "thursday";
    day[4] =   "friday";
    day[5] =   "Saturday";
    day [6] =   "sunday"';
```



day:

day[0] → 'm' 'o' 'n' 'd' 'a' 'y' '\0'

day [1] → 't' 'u' 'e' 's' 'd' 'a' 'y' '\0'

day[2] → 'w' 'e' 'd' 'n' 'e' 's' 'd' 'a' 'y' '\0'

day [3] → 't' 'h' 'u' 'r' 's' 'd' 'a' 'y' '\0'

day [4] → 'V 'r' 'i' 'd' 'a' 'y' '\0'

day[5] → 's' 'a' 't' 'u' 'r' 'd' 'a' 'y' '\0'

day[6] → 's' 'u' 'n' 'd' 'a' 'y' '\0'

Given the pointer array `day`, the following function converts a day number into a pointer to that day's name:

```
char *dayname(int n)
  {
    return n >= 0 && n <= 6 ? day[n] : NULL;
  }
```

Command-Line Arguments

`main` can be defined with formal parameters so that the program may accept command-line arguments, that is, arguments that are specified when the program is executed. Thus, one could compute the factorial of a desired number by executing the program `factorial` as

```
    factorial 5
```

The function main is then defined as having two parameters, customarily called argc and argv, and appears as

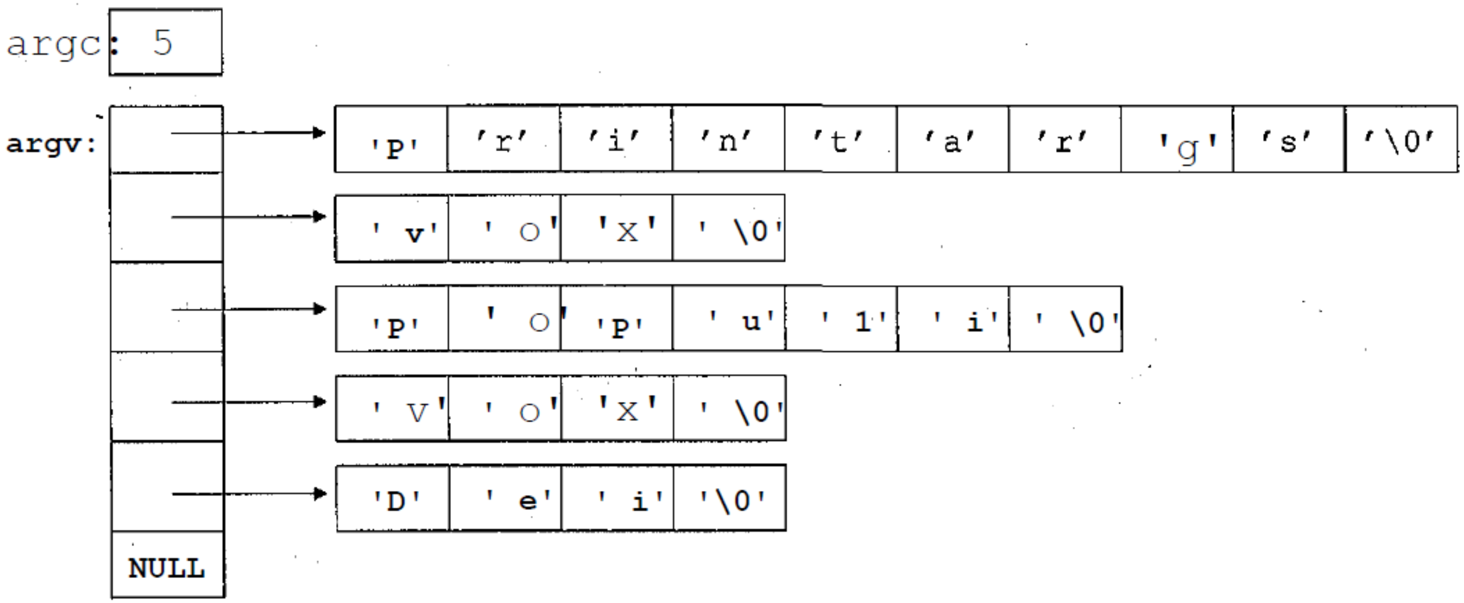
```
int main(int argc, char *argv[])
  {
     ...
  }
```

The parameter argc (for argument count) is the count of the number of command-line arguments, and the parameter argv (for argument vector) is a pointer to a one-dimensional array of pointers to character strings representing arguments. By convention, argv [0] points to a string which is the name of the program, argv [i], where i = 1, 2,. . ., argc-1, points to the ith argument, and argv [argc] is NULL. The argument count argc is at least one, since the first argument is the name of the program itself.

For example, if the command line for a program `printargs` is

```
printargs vox populi vox Dei
```

then, when the function `main` is called, `argc` will be 5 and `argv` a null-terminated array of pointers to strings as shown below:



argc: 5

argv:

'p' 'r' 'i' 'n' 't' 'a' 'r' 'g' 's' '\0'

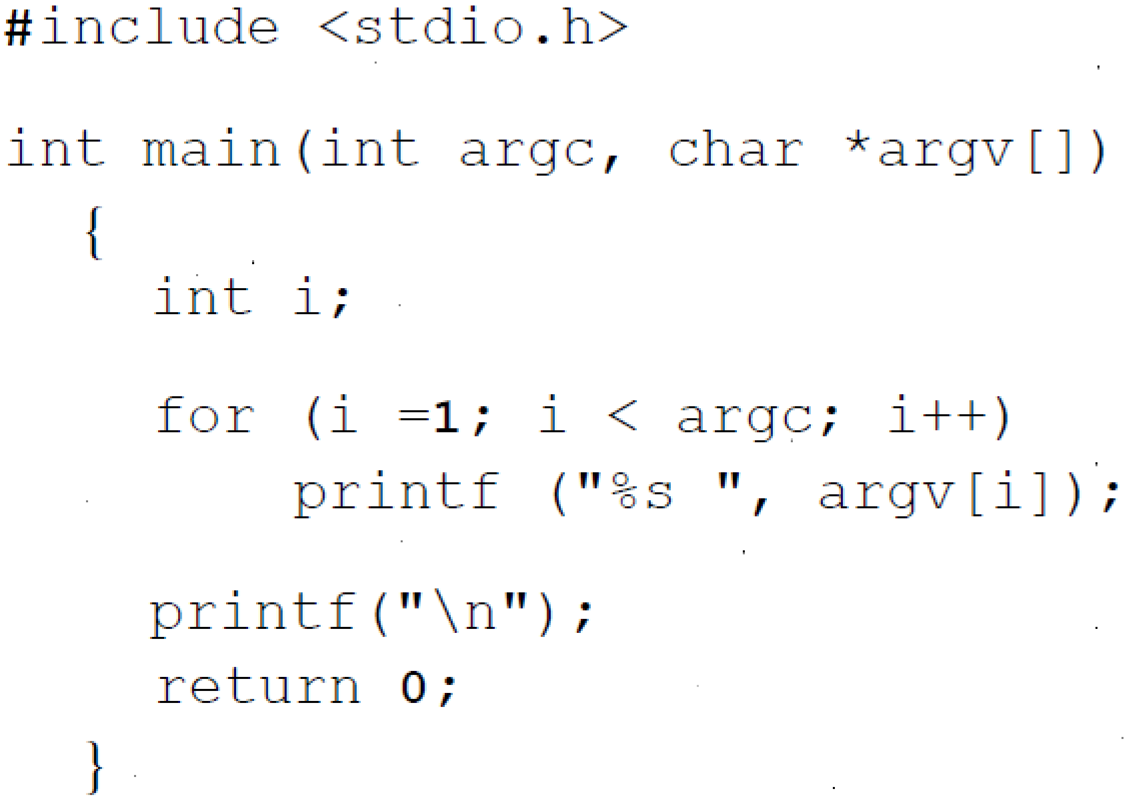
'v' 'o' 'x' '\0'

'p' 'o' 'p' 'u' '1' 'i' '\0'

'v' 'o' 'x' '\0'

'D' 'e' 'i' '\0'

NULL

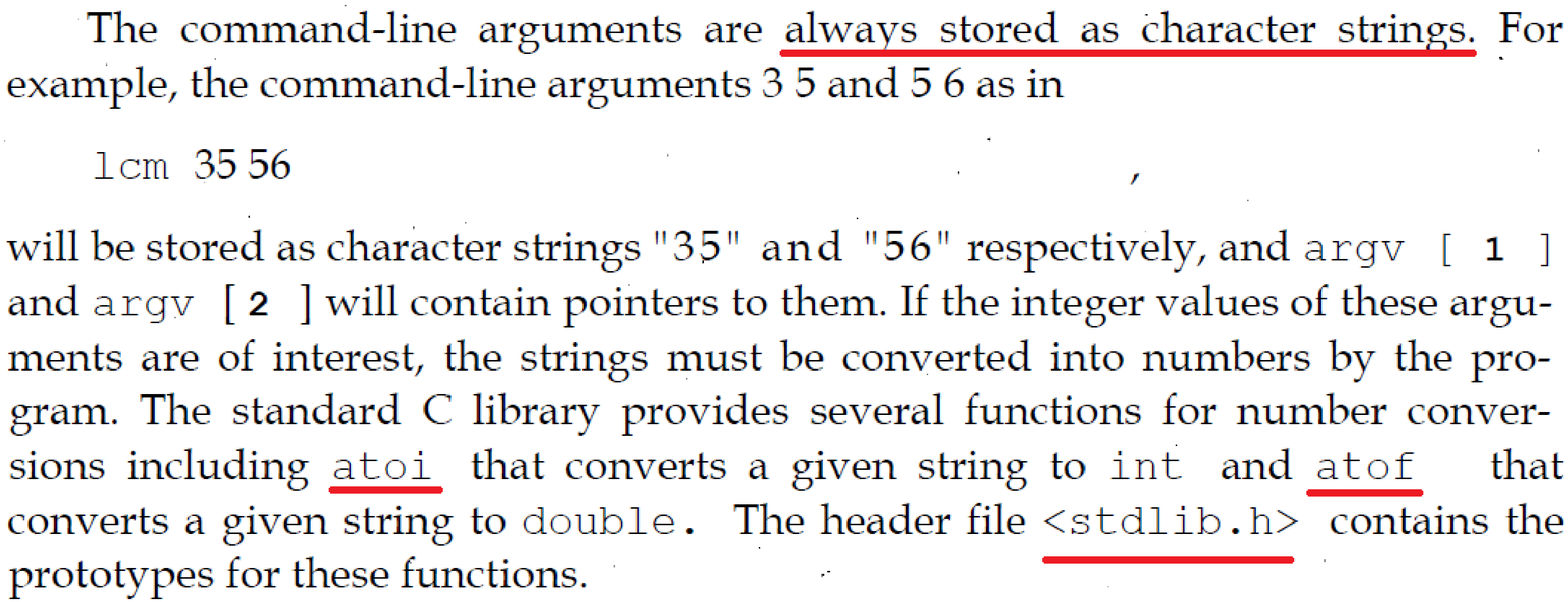


```
#include <stdio.h>

int main(int argc, char *argv[])
  {
    int i;

    for (i =1; i < argc; i++)
        printf ("%s ", argv[i]);

    printf("\n");
    return 0;
  }
```

Whitespaces are used to delimit the command-line arguments. If an argument contains whitespaces, it must be placed within quotation marks. Thus, if `printargs` is invoked as

```
printargs "vox populi" "vox Dei"
```

then `argc` will be 3 and `argv` an array of three pointers pointing to the strings `"printargs"`, `"vox populi"`, and `"vox Dei"` respectively.



The command-line arguments are always stored as character strings. For example, the command-line arguments 3 5 and 5 6 as in

```
lcm 35 56
```

will be stored as character strings "35" and "56" respectively, and `argv [ 1 ]` and `argv [ 2 ]` will contain pointers to them. If the integer values of these arguments are of interest, the strings must be converted into numbers by the program. The standard C library provides several functions for number conversions including `atoi` that converts a given string to `int` and `atof` that converts a given string to `double`. The header file `<stdlib.h>` contains the prototypes for these functions.