Lecture 5
Arrays

Dr. Hacer Yalım Keleş

Ref: Programming in ANSI C, Kumar

BASICS OF ARRAYS

An ordered finite collection of data items, each of the same type, is called an *array*, and the individual data items are its *elements*.

Only one name is assigned to an entire array and individual elements are referenced by specifying a *subscript*. A subscript is also called an *index*.

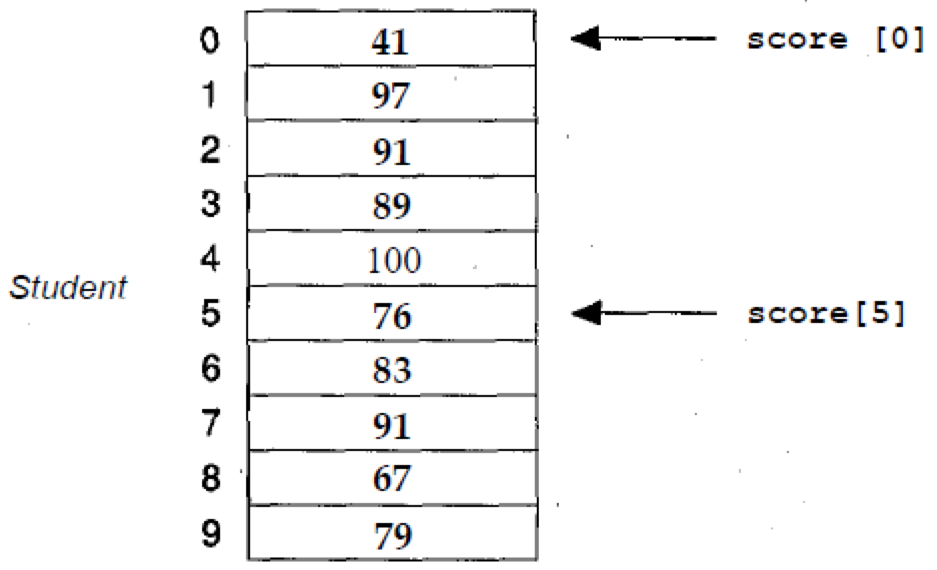
In C, subscripts start at 0, and cannot be negative.

The single group name and the subscript are associated by enclosing the subscript in square brackets to the right of the name.

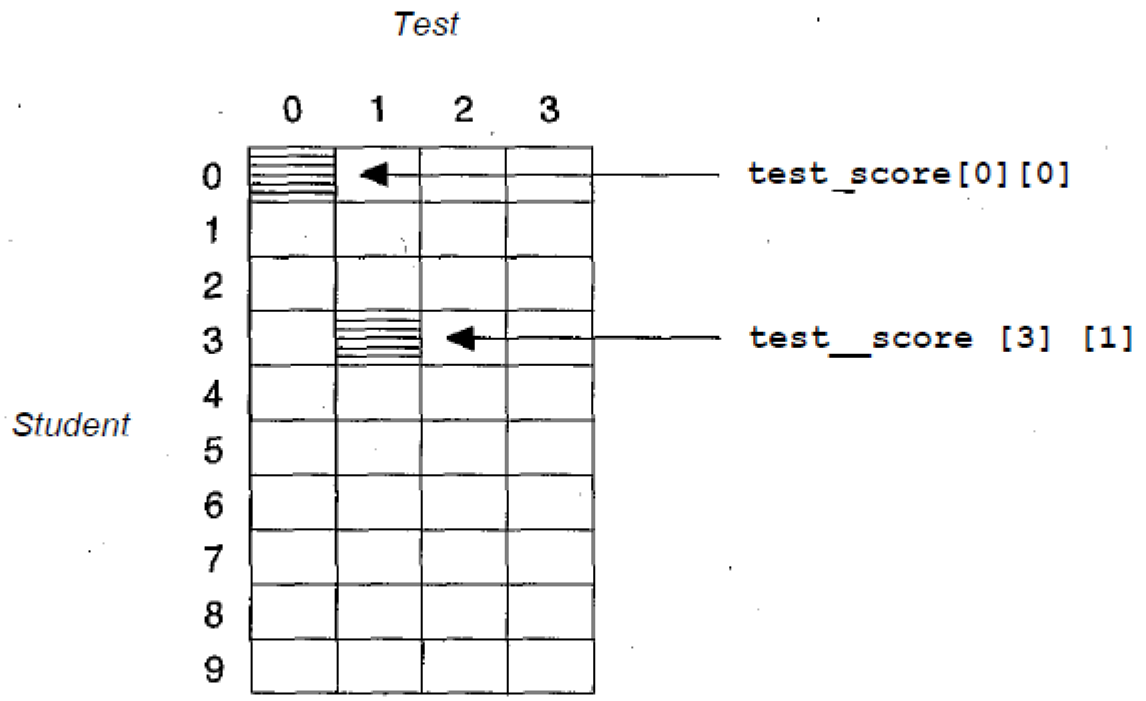
Thus, if prices have been stored in an array named `price`, then `price [0]` refers to the price of the first item, `price [ 4 ]` to the price of the fifth item, etc.

An array has the following properties:

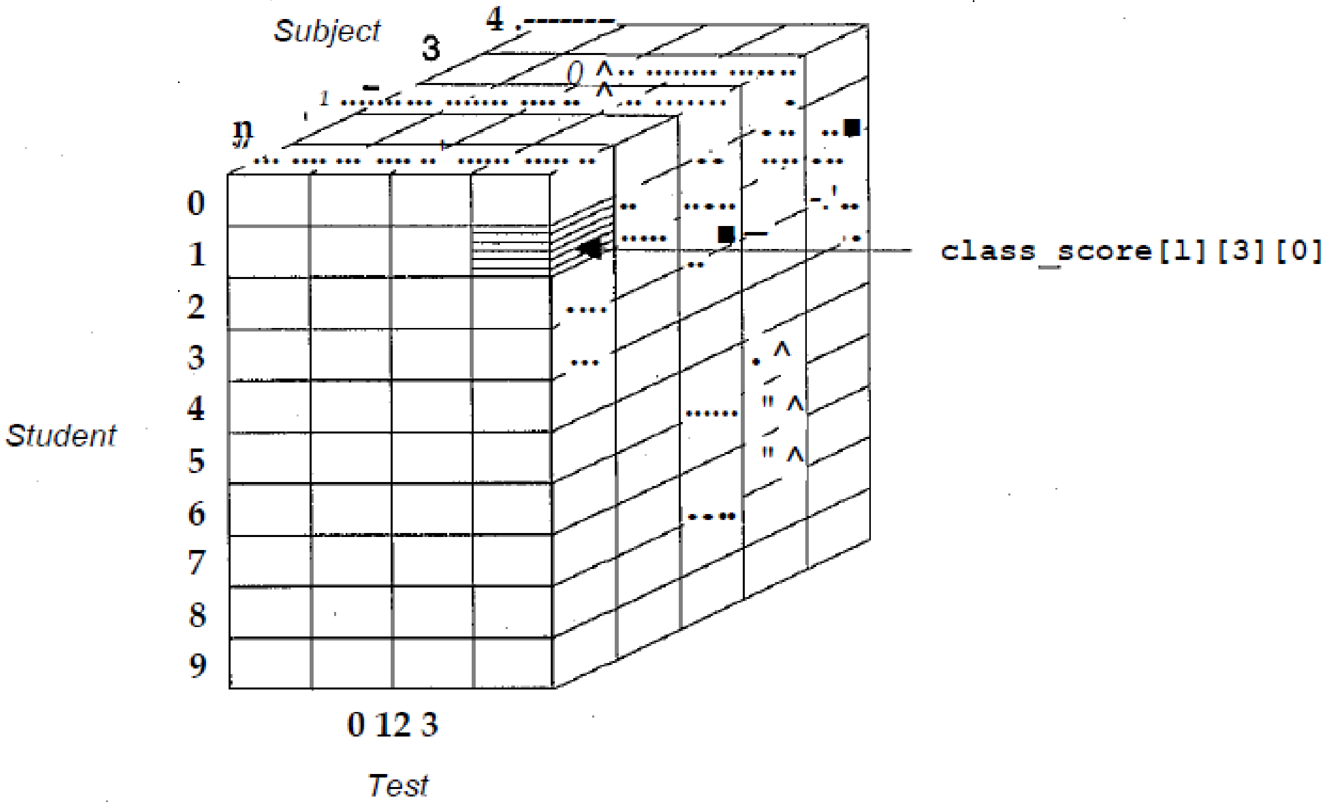
1. The type of an array is the data type of its elements.
2. The *location* of an array is the location of its first element.
3. The *length* of an array is the number of data elements in the array.
4. The *size* of an array is the length of the array times the size of an element.



One-dimensional array `score`



Two-dimensional array test score



Three-dimensional array class score

```c
#include <stdio.h>
#define STUDENTS 10

int main(void)
  {
     int i, sum;
     int score[STUDENTS];

     for (i = 0; i < STUDENTS; i++)
         scanf("%d", &score[i]);

     for (sum =0, i = 0 ;  i <  STUDENTS; i++)
         sum += score[i];

     printf("average score = %f\n",
             (float)sum/STUDENTS);

    return 0;
  }
```

Array Declaration

Arrays, like simple variables, need to be declared before use. An array declaration is of the form

type array-name [expr-1] *[expr-2]* ■ ■ ■ *[expr-n]* ;

where *array-name* is the name of the array being declared, *type* is the type of the elements that will be contained in the array, and *expr-i* is a constant integral expression specifying the number of elements in the ith dimension of the array.

```
int score[10];
```

declares `score` to be an array containing `10` integer elements; the declaration

```
float test_score[10][4];
```

declares the array `test_score` to be a two-dimensional array with `10` rows, `4` columns, and `10` x `4` = `40` single-precision floating point elements; and the declaration

```
double class_score[10][4][5];
```

declares the array `class_score` to be a three-dimensional array with `10` rows, `4` columns, `5` levels, and `10` x `4` x `5` = `200` double-precision floating point elements.

```
#define CHARS_PER_LINE 80
#define TOTAL_LINES  100

char line[CHARS_PER_LINE];
char text[CHARS_PER_LINE * TOTAL_LINES];
```

While declaring an array, the type can be prefixed with `auto`, `static`, or `ext ern`. We çan therefore write

```
auto int score[10];
static float test_score[10] [4] ;
extern double class_score[10][4][5];
```

The length may be omitted in an `extern` declaration of a one-dimensional array, since this declaration does not allocate storage but refers to an object defined elsewhere. The `extern` declaration of an n-dimensional array (n > 1), however, must include the last *n*-1 dimensions so that the accessing algorithm can be determined. Thus, the following are legal declarations:

```
extern int score[];
extern float test_score[][4];
extern double class_score[] [4] [5];
```

Accessing Array Elements

A particular element of an array can be accessed by specifying appropriate subscripts with the array name. Any integral expression can be used as a subscript. Thus, if `score` has been declared to be a one-dimensional array and `offset` an `int`, the following are all valid references to the elements of `score`:

```
score[offset]
score[3*offset]
score[offset/5]
```

Accessing Array Elements

Subscripts should not have values outside the array bounds. C does not automatically check subscripts to lie within the array bounds. It is imperative, therefore, that you check the value of a subscript before using it to access an array element whenever the characteristics of a program make it possible for the subscript to have a value outside the array bounds.

An individual array element can be used anywhere that a simple variable could be used. For example, we can assign a value to it, display its value, or perform arithmetic operations on it.

```
matrix[0] [5] = 1 ;
```

```
p = (price[0] + price[9]) / 2;

--matrix[0][5];


i = 5;
p = price [++i];


i = 5;
p = price[i++];
```

An array cannot be copied into another array by assigning it to the other array. Thus, an array `from` declared as

```
int from[10];
```

cannot be copied into an array to declared as

```
int to[10];
```

by writing assignment statements such as

```
to = from;          /* illegal */
```

or

```
to[] = from[];      /* illegal */
```

It should be copied element-by-element through individual assignments by setting up an appropriate loop such as

```
for(i = 0; i < 10; i++)
        to[i] = from[i];
```

Array Initialization

Elements of an array, be it an external, static, or automatic array, can be assigned initial values by following the array definition with a list of initializers enclosed in braces and separated by commas. For example, the declaration

```
    int score[5] = { 41, 97, 91, 89, 100 };
```

	0	1	2	3
0	95	98	97	96
1	79	89	79	85
2	99	98	99	99
3	90	89	83	86
4	70	72	79	69

```
int test_score[5][4] =
  {
    { 95, 98, 97, 96 },
    { 79, 89, 79, 85 },
    { 99, 98, 99, 99 },
    { 90, 89, 83, 86 },
    { 75, 72, 79, 69 }
  };
```

The inner pairs of braces are optional. Thus, the above declaration can equivalently be written as

```
int test_score[5][4] =
  {
    95, 98, 97, 96, 79, 89, 79, 85, 99, 98,
    99, 99, 90, 89, 83, 86, 75, 72, 79, 69
  };
```

Some special rules govern array initialization:

1. If the number of initializers is less than the number of elements in the array, the remaining elements are set to zero. Thus, the initializations

```
int score[5] = { 41, 97, 91 };

int test_score[5][4] =
  {
     { 0, 98 },
     { 79, 89, 79 },
     { 0, 0, 0, 99 },
     { 90 },
  };.
```

are equivalent to

```
int score[5] = { 41, 97, 91, 0, 0 };

int test_score[5][4] =
  {
     { 0, 98, 0, 0 },
     { 79, 89,. 79, 0 },
     { 0, 0, 0, 99 },
     { 90, 0, 0, 0 },
     {0, 0, 0, 0 }
  };
```

It is an error to specify more initializers than the number of elements in the array.

2. If initializers have been provided for an array, it is not necessary to explicitly specify the array length, in which case the length is derived from the initializers. For example,

```
float sqroot[] = { 0, 1.0, 1.414, 1.732, 2.0 };
```

is equivalent to

```
float sqroot[5] = { 0, 1.0, 1.414, 1.732, 2.0 };
```

3. A character array may be initialized by a string constant, resulting in the first element of the array being set to the first character in the string, the second element to the second character, and so on. The array also receives the terminating ' \ 0' in the string constant. Thus,

```
char os [4]        = "AIX";
char computer[] = "sierra";
```

are equivalent to

```
char os [ 4 ]     = { 'A', 'I', 'X'., '\0' };
char computer[7] =
    { 's', 'i', 'e', ' r', 'r', 'a', '\0' };
```

It is also possible to initialize a character array with an explicit length specification by a string constant with exactly as many characters. The array does not receive the terminating ' \ 0' in that case. For example,

```
char os [3]        = "AIX";
```

is equivalent to

```
char os[3]         = { 'A', 'I', 'X' };
```

However, the string constant must not have more characters than the length of the character array being initialized. Thus, it is an error to initialize os as

```
char os [3]        = "ultrix";
```

Passing Array Elements as Arguments

```
#include <math.h>
double cuberoot(double x)
  {
    return pow(x, 1.0/3.0);
  }
```

and the declarations

```
double  z [10];
double zzz [10] [5] [20];
```

the function call

```
cuberoot(z[5])
```

returns the cube root of the value of the element z [5], and the function call

```
cuberoot (zzz [5] [0] [10])
```

returns the cube root o/ the value of the element zzz[5] [0] [10].

Individual array elements, like simple variables, are passed by value. Their values are copied into the corresponding parameters and cannot be changed by the called function. If the type of the array element is different from the argument type expected by the function, the same conversion rules as for simple variables apply.

Passing Arrays as Arguments

```
void array_cuberoot(double x[10])
  {
     int i;

     for(i =0; i < 10; i++)
          x[i] = cuberoot(x[i]);
  }
```

```
double z[10];
```

can be passed to array_cuberoot as

```
array_cuberoot(z) ;
```

This function call will alter every element of z to the cube root of its current value.

```
double y[100];

void array_cuberoot(double x[], int length)
  {
     int i ;

       for(i =0; i < length; i++)
           x[i] = cuberoot(x[i]);
  }

array_cuberoot(z, 10);

array_cuberoot(y, 100);
```

When declaring a multi-dimensional array as a parameter, it is necessary to specify the lengths of all but the first dimension. Thus, the prototype of a function to compute the cube root of every element of a two-dimensional array can be written as

```
void matrix_cuberoot(double xx[10][5]);
```

or as

```
void matrix_cuberoot(double xx[][5], int rows);
```

but not as

```
void matrix_cuberoot(double xx[][],int rows, int cols);
```

Given a sequence x1,x2,x3,...,xn , its cyclic permutation is defined to be the sequence x2,x3,...,xn,x1 The desired program is as follows.

```
#include <stdio.h>
#define LAST 10

int main(void)
  {
    float array[LAST], temp;
    int i;

    /* read array */
    for (i = 0; i < LAST; i++)
        scanf("%f", &array[i]);

    /* permute array */
    temp = array [0];
    for (i = 1; i < LAST; i++)
        array [i - 1] = array[i];
    array[LAST-1]= temp;

    /* output the permuted array */
    for (i = 0; i < LAST; i++)
        printf("%f\n", array[i]);

    return 0;
  }
```