CP MODULE 2(Part 1of 2)

Arrays and strings- example programs. Two dimensional arrays - matrix operations.

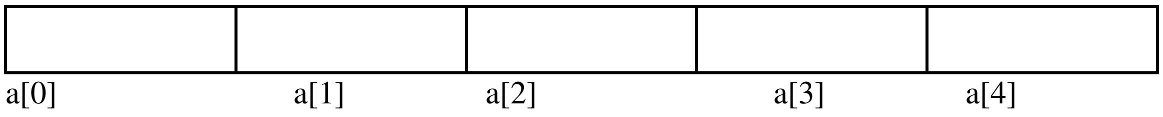
Arrays

- An **array** is a collection of variables of homogeneous (same) type.
- Array is a collection of items of same type that have same name.

Suppose we want to store the mark of 2 students, we can declare 2 variables e.g. a and b.
E.g. int a , b;
But if we want to store marks of more students e.g.10 students, it is not a good practice to declare 10 variables. In situations like this, it is better to declare an array of size 10.
E.g. int a[10];

- Array can be integer array, floating point array, character array etc.
- Individual data item in array is called ***array element***.
- Each array element is referred to by specifying the array name followed by one or more subscripts. Each **subscript (index) is enclosed in square brackets**. Subscript start from 0 and ends at size -1.
- Each subscript must be expressed as a *nonnegative integer*. In an array x with n elements, the array elements are x[0], x[1] , x[2], . . . ,x[n - 1]

Example: int a[5];



Here array **a** can store maximum 5 numbers. First element is in index 0 i.e. a[0]. Second is a[1]. Third is a[2]. Fourth is a[3]. Fifth is a[4].
One-dimensional array can be thought of as a list of values.

ARRAY DECLARATION AND ARRAY DEFINITION

Array name must be accompanied by a size specification (i.e., the number of elements). ***One-dimensional array*** has only *one pair of square brackets*.

The size is specified by a positive integer expression(or constant), enclosed in square brackets.

One-dimensional **ARRAY DECLARATION** is represented as

storage-class data- type array[expression] ;

where *storage -class(is optional)* refers to the storage class(auto,extern,static) of the array, *data- type* is the data type of the elements, *array* is the array name, and expression is a positive-valued integer expression which indicates the number of array elements. The storage-class is optional; default storage class values are ***automatic(auto)*** for arrays that are *defined within a function or a block*, and ***external(extern)***for arrays that are *defined outside of a function.*

Some one-dimensional array definitions are:**.**

```
float b[5];
char x[15];
static int p[10];
```

Here first line defines b to be a floating point array of size 5. x is character array of size 15. p is a static integer array of size 10.

We can also define an *array size in terms of a symbolic constant.* Suppose we have so many arrays with same array size and it is denoted as integer quantity. If we want to change(increase or decrease) the size of all arrays, it is difficult - because we have to change sizes of all arrays.

E.g.
```
#include <stdio.h>
main ( )
{
char a[10];
...
int c[10],d[10];
….
}
```

Here if we want to change the size of all arrays to 100, then we have to change the sizes of all arrays separately.

But if we use symbolic constant as size of array, it is it easier to modify the array size. Whenever we want to change array size we shall simply change the symbolic constant value.

```
#include <stdio.h>
#define SIZE 80
main ( )
{
char a[SIZE];
...
int c[SIZE],d[SIZE];
….
}
```

Here SIZE is a symbolic constant and it stores value 80. We can change it if we want to change the array size.

E.G. **#define SIZE 100**

This will change the array size to 100.

If a program requires a one-dimensional **array declaration**(but the array is defined elsewhere in the program), the declaration is written in the same manner as the array definition with the following exceptions.

1. The *sq*uare brackets may be empty, since the array size will have been specified as a part of the array definition.
2. Initial values cannot be included in the declaration.

```
extern int c [ ] ;          / * external array DECLARATION */
main()
{
static char message[];  /* static array DECLARATION */
..
}
```

Array initialization/definition

Automatic arrays(storage class is auto), cannot be initialized.

External and static array definitions can be initialized by assigning some initial values if desired. The initial values must appear in the order in which they will be assigned to the

individual array elements. Values should be enclosed in braces and separated by commas.
The general form is

storage- class data- type array[expression] = { value1, value2, . . . , valuen} ;

where value1 refers to the value of the first array element, value2 refers to the value of the second element, and so on. The expression indicates the number of array elements (is optional) when initial values are present.
E.g.

```
extern int digits[] = (1, 2, 3, 4, 5 , 6, 7, 8, 9, 10);
static float x(6] = (0, 0.25, 0, -0.50, 0, 0);
char color[3] = { 'R’ , 'E', ‘D’' } ;
static int d[l0] = {3, 4,3);
digits[0] = 1          x[0] = 0           color[0] = 'R'
digits[l] = 2          x[l] = 0.25        color[l] = 'E'
digits[2] = 3          x[2] = 0           color[2] = ' D '
digits[3] = 4          X[3] = -0.50
digits[4] = 5          x[4] = 0
digits[5] = 6          x[5] = 0
digits[6] = 7
digits[7] = 8
digits[8] = 9
digits[9] = 10

d[0] = 3
d[l] = 4
d[2] = 3
d[3] = 0
d[4] = 0
d[5] = 0
d[6] = 0
d[7] = 0
d[8] = 0
d[9] = 0
```

All individual array elements that are ***not assigned*** explicit initial values will automatically be set to zero.

- The array size need not be specified explicitly when initial values are included as a part of an array definition.

- In a numerical array, the array size will automatically be equal to the number of initial values included within the definition.

```
int digits[] = (1, 2, 3, 4, 5 , 6);
static float x[] = (0, 0.25, 0, - 0 . 5 ) ;
```

Thus, digits will be a six-element integer array, and x will be a static, four-element floating-point array.

digits[0] = 1	x[0] = 0
digits[l] = 2	x[l] = 0.25
digits[2] = 3	x[2] = 0
digits[3] = 4	x [3] = -0.5
digits[4] = 5	
digits[5] = 6	

PROCESSING AN ARRAY

Single operations which involve entire arrays are not permitted in C. Thus, if a and b are similar arrays (i.e., same data type, same dimensionality and same size), assignment operations, comparison operations, etc. must be carried out on an element-by-element basis.

```
int a[10],b[10];
```

<u>We cannot do arithmetic operations on full array.</u> E.g. a+b a=b etc are not allowed .

But we can do operations in array elements like a[0]+b[0]

For performing operations in array, we have to use loops. Each pass through the loop can process one array element .The number of passes through the loop will therefore equal the number of array elements to be processed.

```
/* calculate the average of n numbers, then compute the deviation of each number about the average  d =xi - avg*/
#include <stdio.h>
main ( )
{
int n , count;
float avg, d, sum = 0;
int list[100] ;
/* read a value for n */
printf (“ \n How many numbers are there? “) ;
scanf ("%d",&n) ;
printf “\n”) ;

for (count = 0; count < n; count++)
{
        scanf ( "%d" , &list[count] ) ;
        sum =sum+ list[count];
}
avg = sum / n;
printf("\nThe average is %5.2f\n\n", avg);

for (count = 0; count < n; count++)
{
d = list[count] - avg;
printf ( "\ncount =%d  d = %5.2f  \n",count, d);
}
}
```

MULTIDIMENSIONAL ARRAYS- TWO Dimensional

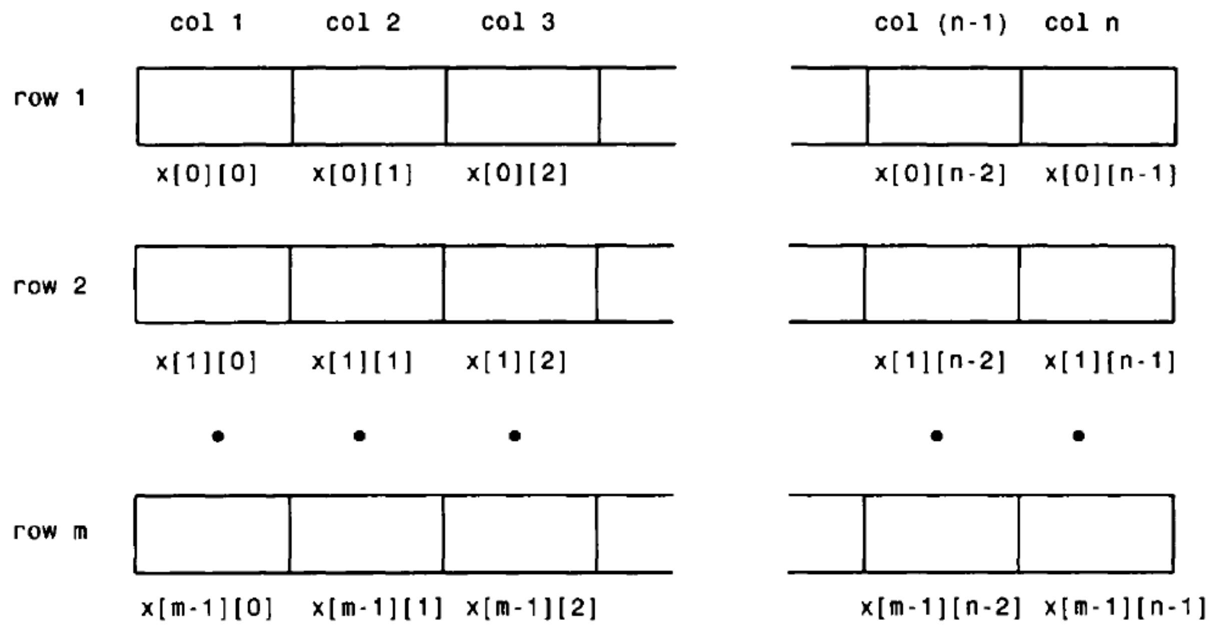
- Multidimensional arrays are defined in much the same manner as one-dimensional arrays, except that a separate pair of square brackets is required for each subscript.
- So 2D arrays have ***two set of square brackets.*** [ ][ ]

Syntax of 2 dimensional array is:

storage-class data- type array[expressionI] [expression2];

Expression1 and expression2 are for giving array size.

- One-dimensional array can be thought of as a list of values,
- Similarly, an m x n, two-dimensional array can be thought of as a table of values having Representation of an array x with m rows and n columns



E.g.

float table[50][50];
char page[24][80];
static double records[100][66][255];

static double records[L][M][N];
Here L, m and N can be initialized using symbolic constants.
Initialization of 2D array is based on specific rule. The rule is that the second(rightmost) subscript increases most rapidly, and the first (leftmost) subscript increases least rapidly. Thus, the elements of a two-dimensional array will be assigned by rows; i.e.,

- the elements of the first row will be assigned first, then the elements of the second row, and so on.
- E.g
  int values[3][4] = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);

```
values[0][0] = 1     values[0][l] = 2     values[0][2] = 3     values[0][3] = 4
values[l][0] = 5     values[l][l] = 6     values[l][2] = 7     values[l][3] = 8
values[2][0] = 9     values[2][1] = 10    values[2][2] = 11    values[2][3] = 12
```

Here the first subscript(3) ranges from 0 to 2, and the second subscript(4) ranges from 0 to 3.

```
int values[3][4] = {
{1,2,3,4},
{5,6,7,8},
{9,10,11,12}
};
```

This is also same as before

But the following

```
int values[3][4] = {
{1,2,3},
{4,5,6},
{7,8,9}
};
```

```
values[0][0] = 1     values[0][l] = 2     values[0][2] = 3     values[0][3] = 0
values[l][0] = 4     values[l][l] = 5     values[l][2] = 6     values[l][3] = 0
values[2][0] = 7     values[2][1] = 8     values[2][2] = 9     values[2][3] = 0
```

E.g
int values[3][4] = {1,2,3,4,5,6,7,8,9};

```
values[0][0] = 1     values[0][l] = 2     values[0][2] = 3     values[0][3] = 4
values[l][0] = 5     values[l][l] = 6     values[l][2] = 7     values[l][3] = 8
values[2][0] = 9     values[2][1] = 0     values[2][2] = 0     values[2][3] = 0
```

E.g.

```
int values[3][4] = {
                        {1,2,3,4,5},
                        {6,7,8,9,10},
                        {11, 12, 13, 14, 15}
                };
```

This will result in a COMPILATION ERROR, since *array size is four elements in each row* but the number of values in each inner pair of braces (five values in each pair.that is 5 elements in a row) exceeds the defined array size(four elements in each row).

When passing multidimensional arrays to a function, the formal argument declarations within the function definition must include explicit size specifications in all of the subscript positions *except the first*. These size specifications must be consistent with the corresponding size specifications in the calling program. The

```
void readinput(int a[ ][3], int r, int c)
{
……………...
}
main()
{
int a[2][3],r,c;
………
…..
readinput(a,r,c);
…………..
}
```

STRINGS

STRING

When a string constant is assigned to an external or a static character array as a part of the array definition, the array size specification is usually omitted.

The proper array size will be assigned automatically.

Null character \0 will be automatically added at the end of every string.

So size of string includes the size of all elements in string and null character \0.

The following initialization is wrong(incorrect) because the array named color is having size 3. It has three elements R, E, D. Space is not allotted to store null character \0. So this is***incorrect*** *representation.*

char color[3] = "RED';

The following is the **correct representation**
char color[] = "RED";
Here size is not explicitly given. So the null character, \0 is automatically added at the end of the second string. Thus the elements of the this array are
color[0] = ‘R’
color[l] = ' E’
color[2] = ‘D’
color[3] = ' \ 0 '
We can also write as
char color[4] = "RED";
This is same as array without size specification.
We cannot separate the initialization from declaration. Thus the statements
char str[20];
str="program";
will result in a compilation ERROR.
For assigning strings to string variables, we need string function strcpy().
strcpy(str,”program”);

```
#include<string.h>
…
char str[20];
strcpy(str,”program”);
….
```

<u>Inputting and printing strings</u>
Strings are character arrays, we need not input the characters one by one. We can get the entire string at a single step. This is achieved by using the format specifier %s in the scanf().,
No need not use & in scanf() while inputting strings.
Similarly, to print a string, we use %s in the printf() and then specify the array name.

E.g. Write code that inputs the user’s first name and then prints a greeting message.

```
#include<stdio.h>
main()
{
char name[15];
printf("Enter your first name\n");
scanf("%s",neme);
printf("Welcome %s, \n",name);
}
```

<u>OUTPUT</u>

```
Enter you first name
Arun Sekhar
Welcome Arun
```

But, %s in scanf() cannot handle multi-word strings.
Here we entered Arun Sekhar but %s does not read after space(space is delimiter in %s) So only Arun is stored in name.
Even if we try to assign a multi-word string to a character array with scanf() and %s, o*nly the first word will get assigned*. To get around this limitation, we have to use **gets()** or use %[∧₩n] instead of %s
E.g.

```
main()
{
char a[20];
scanf("%[^\n]",a);
…
}
```

For printing strings in such cases, you can use either puts() or printf().
E.g.
To input a **multi-word string.**

```
#include<stdio.h>
main()
{
char name[25];
printf("Enter your name\n");
gets(name);
printf("Welcome %s,\n",name);
}
```

OUTPUT

```
Enter you first name
Arun Sekhar
Welcome Arun Sekhar
```

String functions
String functions are declared inside the header file string.h

Function	Purpose
strcpy(s1, s2)	Copies **s2** into **s1**
strcat(s1, s2)	Concatenates **s2** onto the end of **s1**
strlen(s1)	Returns the length of **s1**
strcmp(s1, s2)	Returns 0 if **s1** and **s2** are the same; a value less than 0 if **s1** < **s2**; a value greater than 0 if **s1** > **s2**
strcmpi(s1, s2)	Compares two strings **s1** and **s2** without regard to case ("**i**" denotes that this function ignores case)

We cannot separate the initialization from declaration. Thus the statements
char str[8];
str="program";
will result in a compilation ERROR.
For assigning strings to string variables, we need string function strcpy().
strcpy(str,"program");
#include<string.h>
…
main()
{...
char str[20];
strcpy(str,"program");
….
}
If sl precedes s2 in lexicographical (alphabetical) order, denoted by sl < s2, then a negative value will be returned by strcmp function. Otherwise a positive value is returned. Thus strcmp("their","there") will return −9, i.e. ASCII(i) − ASCII(r).
the use of some of the string functions.
#include <stdio.h>
#include <string.h>
main()
{
char s1[30], s2[30];
printf("Enter two strings");
gets(s1);
gets (s2);
printf("The length of the strings are %d and %d\n", strlen(s1), strlen(s2));
if(strcmp(s1, s2)==0)
{printf("The strings are equal\n");}
strcat(s1, s2);

```
printf("The concatenated string is %s\n", s1);
strcpy(s1,"beautiful\n");
printf("The first string is now modified to %s",s1);
}
```

If the user inputs the strings "hello" and "world", the output is

```
The length of the strings are 5 and 5
The concatenated string is helloworld
The first string is now modified to beautiful
```

Table of strings

```
char subjects[][6]= {

"MA102",
"PH100",
"BE100",
"BE102",
"EE100",
"CS100"
};
```

To access the name of ith subject in the list, we just write subjects[i].
Thus subjects[0] = "MA102", subjects[1] = "PH100" and so on.
A table of strings is usually used to sort a list of strings in alphabetical order.

PROGRAMMING EXAMPLES

```
##ADD TWO MATRICES
#include <stdio.h>
main()
{
int r1, c1,r2,c2, a[10][10], b[10][10], sum[10][10], i, j;
printf("Enter the number of rows and columns in first matrix\n");
scanf("%d %d", &r1,&c1);
printf("Enter the number of rows and columns in second  matrix\n");
scanf("%d %d", &r2,&c2);
if(r1!=r2 || c1!=c2)
{
printf("ADDITION NOT POSSIBLE\n");

}
else{
        printf("Enter elements of 1st matrix\n");
        for(i=0; i<=r1-1; i++)
        for(j=0; j<=c1-1; j++)
        scanf("%d",&a[i][j]);

        printf("Enter elements of 2nd matrix:\n");
        for(i=0; i<=r2-1; i++)
        for(j=0; j<=c2-1; j++)
        scanf("%d",&b[i][j]);

        for(i=0;i<=r1-1;i++)
        for(j=0;j<=c1-1;j++)
        sum[i][j]=a[i][j]+b[i][j];

        printf("Sum of the two matrices is\n");
        for(i=0;i<=r1-1;i++)
        {
                for(j=0;j<=c1-1;j++)
                {printf("%d ",sum[i][j]);
                printf("\n");
                }
        }
    }
}
```

2.MATRIX MULTIPLICATION

```
#include <stdio.h>
main()
{
int r1, c1,r2,c2, a[10][10], b[10][10], c[10][10], i, j,k;
printf("Enter the number of rows and columns in first matrix\n");
scanf("%d %d", &r1,&c1);

printf("Enter the number of rows and columns in second  matrix\n");
scanf("%d %d", &r2,&c2);

if(c1!=r2 )
{
printf("MULTIPLICATION NOT POSSIBLE\n");

}
else{
        printf("Enter elements of 1st matrix\n");
        for(i=0; i<=r1-1; i++)
        for(j=0; j<=c1-1; j++)
        scanf("%d",&a[i][j]);

        printf("Enter elements of 2nd matrix:\n");
        for(i=0; i<=r2-1; i++)
        for(j=0; j<=c2-1; j++)
        scanf("%d",&b[i][j]);
        for(i=0;i<r1;i++)
        {       for(j=0;j<c2;j++)
                {
                c[i][j]=0;
                        for(k=0;k<r2;k++)
                        {
                        c[i][j]=c[i][j]+(a[i][k]*b[k][j]);
                        }
                }
        }


        printf("PRODUCT of the two matrices is\n");
        for(i=0;i<=r1-1;i++)
        {
                printf("\n");
                for(j=0;j<=c2-1;j++)
                {printf("%d\t",c[i][j]);

                }
        }
  }

}
```

3. Matrix transpose

```
#include <stdio.h>
main()
{
int r, c,a[10][10], b[10][10],i,j;
printf("Enter the number of rows and columns in matrix\n");
scanf("%d %d", &r,&c);

printf("Enter elements of matrix\n");
        for(i=0; i<=r-1; i++)
        for(j=0; j<=c-1; j++)
        scanf("%d",&a[i][j]);

printf("Transpose of matrix is \n");
        for(i=0; i<=c-1; i++)
        {
                ("\n");
                for(j=0; j<=r-1; j++)
                {
                b[i][j]=a[j][i];
                printf("%d\t",b[i][j]);
                }

        }
}
```