Chapter 6

Arrays & Strings

Dr. Niroj Kumar Pani

nirojpani@gmail.com

Department of Computer Science Engineering & Applications

Indira Gandhi Institute of Technology

Sarang, Odisha

Chapter Outline…

- Introduction
- One Dimensional Arrays
  - What is a One-Dimensional Array?
  - Declaration
  - Initialization
  - Accessing the Array Elements
  - Programming Examples
  - Assignments - I

- **Two Dimensional Arrays**
  - Two Dimensional Arrays: Why & What?
  - Declaration
  - Initialization
  - Accessing the Array Elements
  - Programming Examples
  - Assignments - II
- **Strings**
  - What is a String?
  - Declaration
  - Initialization
  - String Input & Output
  - String Handling Functions
  - Programming Examples
  - Assignments - III

Introduction

Why Arrays?

- To understand why do we need arrays, let's consider a situation:
  Suppose, we want to store the roll numbers of 50 students (for some processing). In such a situation, we have got two options:

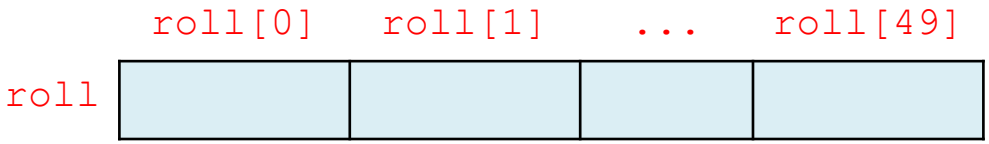
  1. Declare 50 different variables (normal variables that we have used so far), each one to store the roll number of one student. e.g.,

```
int roll1;
int roll2;
....
int roll50;
```

2. **Declare just one variable capable of storing all the fifty roll nos. e.g.,**

```
int roll[50];
```

So that, `roll[0]` automatically corresponds to the 1st roll no., `roll[1]` automatically corresponds to the 2nd roll no...., `roll[49]` automatically corresponds to the 50th roll no.



[NOTE]: Here, the numbers 0,... 49 that are within the square brackets are called subscripts / indexes.

Obviously, the second alternative is better, because it always much easier to handle just one variable instead of handling 50 different variables.

- **Such a single variable, which can store more than one values of same type** (in our case 50 integers) **is called an array or subscripted variable or indexed variable** (because the individual elements can be accessed by using a subscript).

What is an Array (The Formal Definition) ?

- **An array is a finite, ordered, collection of homogeneous data elements / data items that share a common name (variable name).**

  Notable characteristics of an array (from the above definition):

  - **Finite: The number of elements is limited. An array has a fixed size.**
  - **Ordered: The elements are sequenced; there is a 1st element, there is a 2nd element, .. there is a last element.**
  - **Collection: An array refers to a group of (more than one) elements.**
  - **Homogeneous: All the elements are of same data type i.e., all are integers, or all are characters.. etc. , but there can't be a mix.**

Few Important Terms Associated with Arrays

- **Size / Length:** The maximum number of elements that can be stored in an array.
- **Type:** Data types of the elements (all the elements) in an array.
- **Base Address:** Address (of the memory location) of the 1st element in the array.
- **Index / Subscript:** The unique integer used to locate each element in an array.

  For example: In `roll[15]`, the number `15` is called the subscript.

Types of Arrays

Arrays can be classified as:

- **One dimensional array (Linear array / Single subscripted variable / Single indexed variable / Vector):**
  - It is an array in which, the elements are organized as a linear list.

    Hence, all the elements can be uniquely accessed by using just one subscript (that specifies the relative position of the element in the list).

    e.g., the array that we have used to store the 50 roll numbers.

- **Multi Dimensional Array:**
  - It is an array in which, the elements are organized in some nonlinear fashion. Hence, more than one subscripts are required to access all the elements uniquely.
  - Can have the following variations:
    - Two-dimensional array (Double subscripted variable / Double indexed variable / Matrix):
      - The elements are organized as a table.
      - Hence, 2 subscripts are required (one for the rows and the other for the columns).
    - Three-dimensional array:
      - The elements are organized as a box (list of tables).
      - 3 subscripts are required.

[NOTE]: Although in theory it is possible to have n-dimensional array (that requires 'n' subscripts), arrays beyond 3-D are impractical. In fact, in computer science, arrays above 2-D are very rarely required. So, in this chapter, we will limit out discussion up to the 2-D array.

One-Dimensional Arrays

What is a One-Dimensional Array ?

- **[Definition]: As just discussed, a 1-D array (also called, linear array / single subscripted variable / single indexed variable / vector) is an array in which, the elements are organized as a linear list. Hence, all the elements in the array can be uniquely accessed by using just one subscript** (that specifies the relative position of the element in the list).

  e.g., the array that we have used to store the 50 roll numbers.

Declaration

- **Since array is also a variable, it needs to be declared before its use.**
- **Syntax:**

```
data_type array_name[size];
```

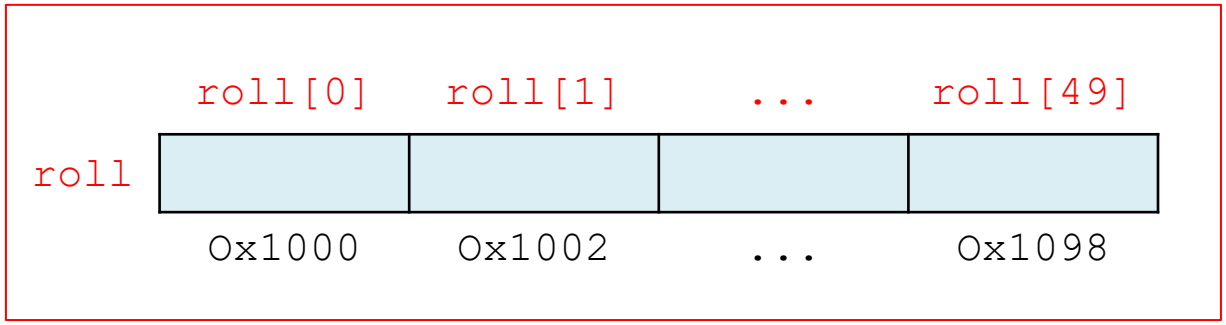
e.g.,

```
int roll[50];
char grade[30];
```

- **What happens when we make a declaration like `int roll[50];`?**

This declaration tells the compiler to,

- **Reserve** 100 bytes **of *consecutive* memory space** (considering that an `int` takes 2 bytes of memory).
- **Associate the name `roll` with this location.**



Memory layout of the array `int roll[50]`

- **Notes:**

1. **In the array declaration, the position of the square brackets is optional** (we will know its reason in the chapter "Pointers") **i.e.,**

   `int rollNos[50];` is same as: `int [50]rollNos;`

   However, some compilers may not allow the second type declaration. So, it is always safe to go with the 1st type declaration (i.e., `int rollNos[50];`).

2. **No matter how big an array is, the array elements are always stored in consecutive / contiguous memory locations.**

3. **In C, the index / subscript of a 1-D array always starts from '0'. its range is '0' to 'array size-1'.**

4. **Initially (when the array not initialized), the memory spaces are filled with garbage values.**

5. Address Calculation of an Element in a 1-D Array:

Let, Base address of the array = B

Word size of the array (size of each element in bytes) **=** W

Then, the location / address of the element $A[i] = B + W*i$

Example: Given an integer array $A[100]$**, with base address** 2000**, find the location of** $A[25]$**?**

Solution: $B = 2000, W = 2.$

$\Rightarrow$ Address of $A[25] = 2000 + 2*25 = 2050$

Initialization

- A 1-D array could be initialized at the place declaration.
- **Syntax 1:**

```
data_type array_name[size] = {elements};
```

e.g.,

```
int roll[4] = {7, 2, 9, 6};
char grad[3] = {'A', 'B', 'C'};
```

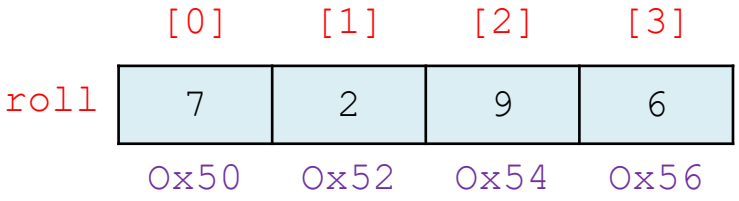
- **Syntax 2 (Preferred):**

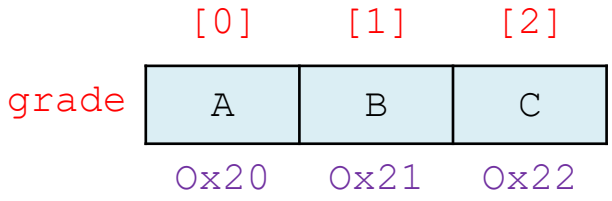
```
data_type array_name[] = {elements};
```

e.g.,

```
int roll[] = {7, 2, 9, 6};
char grade[] = {'A', 'B', 'C'};
```

- Both the syntaxes result in the same array structures:



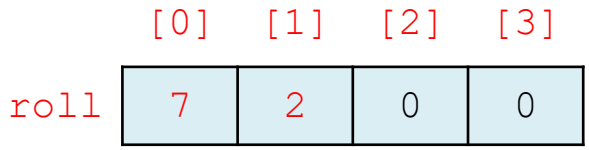


- **Notes:**

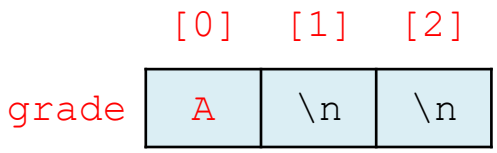
1. **In both the syntaxes, the data type of the `elements` should be same as (or at least compatible with) the data type of the array.**
2. **In Syntax 1, it is required that the `size` of the array should be *exactly* same as the number of `elements` (which is not applicable for Syntax 2).**

   - **If** size > number of elements**, then only that many elements will be initialized (in order). The remaining elements will be set to zero** (if the array is of int/float) **or NULL** (if the array is of char). **e.g.,**

```
int roll[4] = {7, 2};
```




```
char grade[3] = {'A'};
```



   - **If** size < number of elements**, then a run time error may occur** (the compiler will NOT show any error message, a warning might be displayed).

3. **Initialization and declaration CAN'T be separated. i.e., the followings will generate a compilation error.**

```
int roll[3];
roll = {7, 2, 5};
```

```
int roll[];
roll = {7, 2, 5};
```

4. **Initializing a 1-D array with another 1-D array is not allowed (like we were doing with ordinary variables) i.e., the followings will generate a compilation error.**

```
int a[3] = {7, 2, 5};
int b[3];
b = a;
```

Accessing the Array Elements

- **An element in a 1-D array can be uniquely accessed by using just one *appropriate* index (that specifies the relative position of the element in the list).**
- **For a 1-D array declared as:** `int roll[50];` **the** $(i+1)^{th}$ **element** (the element at position i) **can be accessed by using the expression:** `roll[i].`

  For Example:

  - **The 1st element is accessed by** `roll[0].`
  - **The 2nd element is accessed by** `roll[1].`
  - **The 37th element is accessed by** `roll[36].`
  - **The last (50th) element is accessed by** `roll[49].`

Programming Examples

- **Programming Example 1:**

```
/* PR6_1.c: Program that accepts ten integers as input and displays them in reverse order */

# include <stdio.h>
# include <conio.h>

void main()
{
    int i, numbers[10];
    printf("\nEnter 10 integers: ");
    for(i=0;i<10;i++)
    {
        scanf("%d", &numbers[i]);
    }
    printf("\nThe numbers in the reverse order are: ");
    for(i=9;i>=0;i--)
    {
        printf("%d ", numbers[i]);
    }
    getch();
}
```

Output

```
Enter 10 integers: 1 2 3 4 5 6 7 8 9 10
The numbers in the reverse order are:
10 9 8 7 6 5 4 3 2 1
```

Programming Example 2:

```
/* PR6_2.c: Program that accepts 10 integers as input and displays their sum and average */

# include <stdio.h>
# include <conio.h>

void main()
{
    int i, numbers[10], sum = 0;
    float avg;

    printf("\nEnter 10 integers: ");
    for(i=0;i<10;i++)
        scanf("%d", &numbers[i]);
    for(i=0;i<10;i++)
        sum = sum + numbers[i];
    avg = (float)sum/10;

    printf("\nSum of the 10 integers: %d", sum);
    printf("\n\nAverage of the 10 integers: %.2f", avg);
    getch();
}
```

Output

```
Enter 10 integers: 1 2 3 4 5 6 7 8 9 10
Sum of the 10 integers: 55
Average of the 10 integers: 5.50
```

Programming Example 3:

```
/* PR6_3.c: Program that accepts 10 integers as input and displays the largest among them */

# include <stdio.h>
# include <conio.h>

void main()
{
   int i, numbers[10], largest;

   printf("\nEnter 10 integers: ");
   for(i=0;i<10;i++)
   {
      scanf("%d", &numbers[i]);
   }
   largest = numbers[0];
   for(i=1;i<10;i++)
   {
       if(numbers[i]>largest)
           largest = numbers[i];
   }
   printf("\nThe largest is: %d", largest);
   getch();
}
```

Output

```
Enter 10 integers: 1 10 55 2 3 4 0 7 7 1
The largest is: 55
```

Assignments - I

Complete the experiments given in "Lab Manual - Section 6"**.**

Two-Dimensional Arrays

Two Dimensional Arrays: Why & What ?

- **The 1-D array, that we have discussed so far, can store a *list* of items. However, there could be situations where a *table* of data items(of same type) need to be stored and processed.**

  For example: Suppose we want to store the roll numbers and marks (out of 100) of 4 students side by side in a table, as shown here

  (the 1st column corresponds the roll numbers and the 2nd column corresponds to the marks):

1	62
2	75
3	52
4	96

  For such a situation, we need a 2-D array.

- **[Definition]: A 2-D array (also called, double subscripted variable / double indexed variable / matrix) is an array in which, the data items are organized as a table. Hence, all the elements in the array can be uniquely accessed by using two subscripts** (one specifies the row and other specifies the column).

Declaration

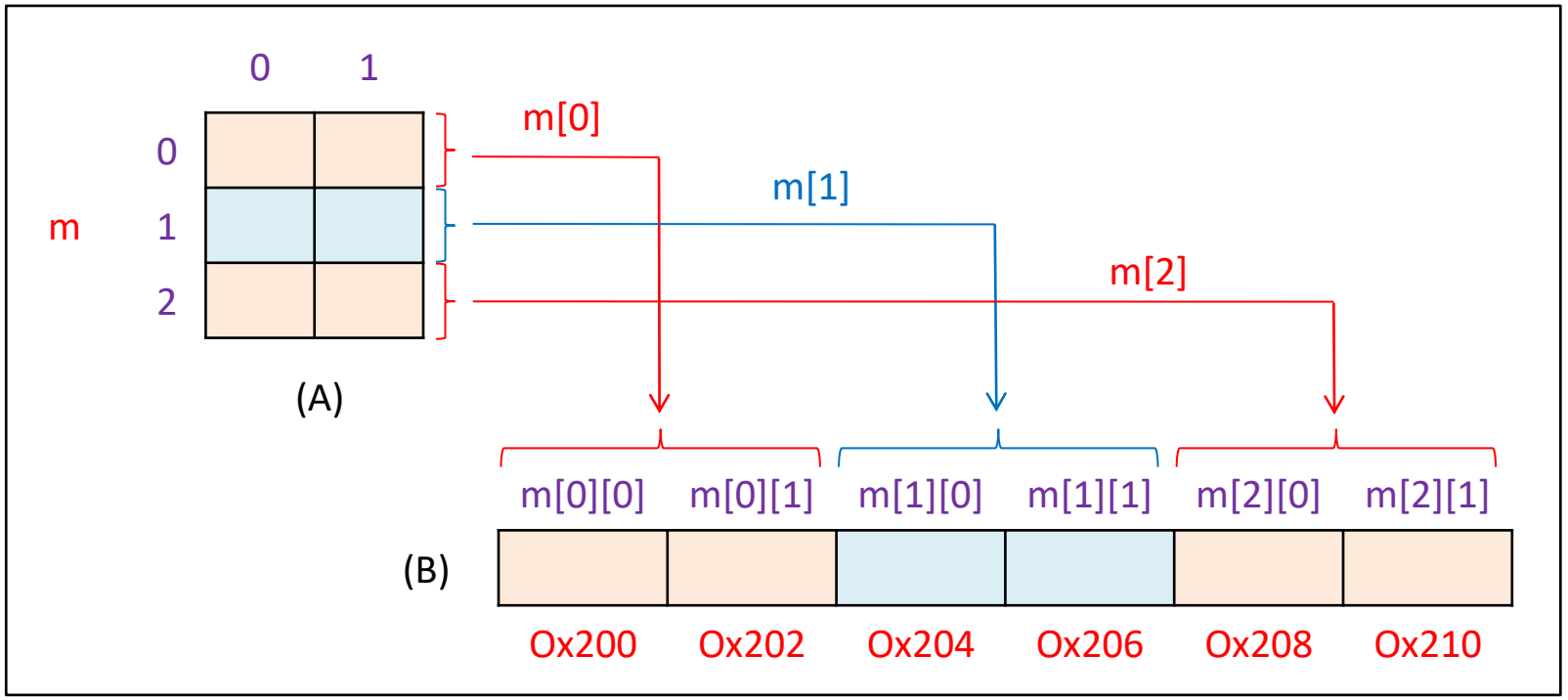
- **Like a 1-D array, a 2-D array also needs to be declared before its use.**
- **Syntax:**

```
data_type array_name[row_size][column_size];
```

e.g.,

```
int m[3][2];
char a[4][7];
```

- **The memory lay out of the array declared as `int m[3][2];` is shown below:**



The table representation of a 2-D array (as shown in 'A') is *only* conceptually true. This is because memory doesn't contain rows and columns.

So far, the actual memory representation is concerned, the array elements, whether it is a 1-D array, or a 2-D array or a n-D array, are always stored in contagious memory locations (C supports the row major order as shown in 'B'.).

- **Notes:**

1. **The index of a 2-D array ranges from is "0,0" to "row size-1, column size-1".**
2. **Initially (when the array not initialized), the memory spaces are filled with garbage values.**
5. **Address Calculation of an Element in a 1-D Array:**

   Let, Base address of the array = B

   Array size = R*C

   Word size of the array (size of each element in bytes) **=** W

   Then,

   - **For row-major order, the address of the element** A[i][j] = B + W(C*(i-1)+(j-1))
   - **For column-major order, the address of the element** A[i][j] = B + W((i-1)+R*(j-1))

Example 1: **Given an integer array** $A[3][4]$ **stored in row-major order, with base address** 100**, find the location of** $A[2][3]$**?**

Solution: $B = 100$, Array size $= 3*4$, $W = 2$.

$\Rightarrow$ Address of $A[2][3] = 100 + 2(4*(2-1)+(3-1)) = 112$.

Example 2: **Given an integer array** $A[3][4]$ **stored in column-major order, with base address** 100**, find the location of** $A[2][3]$**?**

Solution: $B = 100$, Array size $= 3*4$, $W = 2$.

$\Rightarrow$ Address of $A[2][3] = 100 + 2((2-1)+3*(3-1)) = 114$

Initialization

- Like a 1-D array, a 2-D could also be initialized at the place declaration. The syntaxes are demonstrated with the help of following examples.
- Syntaxes:

(1)
```
int m[3][2] = {
                 {1,2}
                 {3,4}
                 {5,6}
              };
```

(2)
```
int m[3][2] = {1,2,3,4,5,6};
```

(3)
```
int m[][2] = {1,2,3,4,5,6};
```
(Preferred)

- All the syntaxes result in the same array structure:

m

	0	1
0	1	2
1	3	4
2	5	6

- **Notes:**

1. **As it can be noticed, in all the syntaxes, the data type of the *elements* should be same as (or compatible with) the data type of the array.**
2. **In Syntax 1 & 2, it is required that the `size` of the array should be *exactly* same as the number of *elements*** (which is not applicable for Syntax 3).

   - **If** size > number of elements**, then only that many elements will be initialized (in order). The remaining elements will be set to zero** (if the array is of int/float) **or NULL** (if the array is of char). **e.g.,**

```
char c[2][3] = {'a', 'b', 'c', 'd'};
```

a	b	c
d	\0	\0

   - **If** size < number of elements**, then a run time error may occur** (the compiler will NOT show any error message, a warning might be displayed).

3. **Initialization and declaration CAN'T be separated. i.e., the followings will generate a compilation error.**

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

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

4. **Initializing a 2-D array with another 2-D array is not allowed (like we were doing with ordinary variables) i.e., the followings will generate a compilation error.**

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

Accessing the Array Elements

- **An element in a 2-D array can be uniquely accessed by using two indexes** (the 1st index specifies the row number, and the 2nd index specifies the column number).
- **For a 2-D array declared as:** `int m[3][2];` **the element that belongs to row 'i' and column 'j' can be accessed by using the expression:** `m[i][j].`

  For Example:

  - **The 1st element (element of row 0, column 0) is accessed by** `m[0][0].`
  - **The element that belongs row 1, column 2 to is accessed by** `m[1][2].`
  - **The last element (element of row 2, column 1) is accessed by** `m[2][1].`

Programming Examples

- **Programming Example 1:**

```
/* PR6_4.c: Program that reads a matrix of order 3 × 3 and then finds its transpose. */

# include <stdio.h>
# include <conio.h>

void main()
{
   int i, j, a[3][3];

   printf("\nEnter a 3X3 matrix:\n");
   for(i=0;i<3;i++)
   {
       for(j=0;j<3;j++)
       {
           scanf("%d", &a[i][j]);
       }
   }
```

[Cont.]

```
    printf("\n\nThe transposed matrix is:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("%-5d", a[j][i]);
        }
        printf("\n\n");
    }

    getch();
}
```

Output

```
Enter a 3X3 matrix:
4 2 1
6 5 8
3 7 9

The transposed matrix is:
4 6 3
2 5 7
1 8 9
```

■ Programming Example 2:

```
/* PR6_5.c: Program that reads a matrix of order 3 × 3 and then adds its diagonal elements. */

# include <stdio.h>
# include <conio.h>

void main()
{
   int i, j, a[3][3], sum = 0;

   printf("\nEnter a 3X3 marrix:\n");
   for(i=0;i<3;i++)
   {
       for(j=0;j<3;j++)
       {
           scanf("%d", &a[i][j]);
       }
   }
```

[Cont.]

```
    //Calculating the sum
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            if(i!=j)
                continue;
            sum = sum + a[i][j];
        }
    }

    printf("\n\nThe sum of it's diagonal elements are: %d", sum);

    getch();
}
```

Output

```
Enter a 3X3 matrix:
4 2 1
6 5 8
3 7 9

The sum of it's diagonal elements are: 18
```

Programming Example 3:

```
/* PR6_6.c: Program that multiplies two 3 × 3 matrixes. */

# include <stdio.h>
# include <conio.h>

void main()
{
   int i, j, k, a[3][3], b[3][3], c[3][3];

   printf("\nEnter the first 3X3 matrix:\n");
   for(i=0;i<3;i++)
   {
       for(j=0;j<3;j++)
       {
           scanf("%d", &a[i][j]);
       }
   }
```

[Cont.]

```
printf("\nEnter the second 3X3 matrix:\n");
for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        scanf("%d", &b[i][j]);
    }
}

//Multiplying 'a' and 'b' and storing the result in 'c'

for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        c[i][j] = 0;
        for(k=0;k<3;k++)
        {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}
```

[Cont.]

```
    printf("\n\nThe resultant matrix after multiplication is:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("%-4d", c[i][j]);
        }
        printf("\n\n");
    }

    getch();
}
```

Output

```
Enter the first 3X3 matrix:
1 2 3
4 5 6
7 8 9

Enter the second 3X3 matrix:
9 8 7
6 5 4
3 2 1

The resultant matrix after multiplication is:
30  24  18
84  69  54
138 114 90
```

Assignments - II

Complete the experiments given in "Lab Manual - Section 7"**.**

Strings

What is a String ?

- C doesn't support strings as a data type.
- [Definition]: In C, a string is (implemented as) a one-dimensional array of characters terminated by a NULL character ('\0').

  For instance, the string constant "Virus" is actually stored in the memory as:

[NOTE]: The terminating NULL ('\0') is important, because *only this* differentiates a string from a character array.

Declaration

- Syntax:

```
char string_name[size];
```

e.g., `char name[10];`

Here, the `size` should be equal to the maximum number of characters in the string *plus* one (to accommodate the terminating '\0' character.)

Initialization

- A string can be initialized at the place declaration. The syntaxes are demonstrated with the help of following examples.
- **Syntaxes:**

(1)
```
char name[4] = {'S', 'a', 'n', '\0'};
```

(2)
```
char name[] = {'S', 'a', 'n', '\0'};
```

(3)
```
char name[4] = "San";
```

(4)
```
char name[] = "San";
```
(Preferred)

- All the syntaxes result in the same array structure:

- **Notes:**

1. **In Syntax 1 & 3, it is required that the `size` of the array should be *exactly one more than* the number of characters in the string to accommodate the extra '\0' character** (which is not required in Syntax 3).

   - **If** size > number of characters+1**, then the remaining elements will be set to the NULL character ('\0').**

```
char name[6] = "San";
```

   - **If** size < number of characters+1**, then a run time error may occur** (the compiler will NOT show any error message, a warning might be displayed).

2. **Initialization and declaration CAN'T be separated. i.e., the followings will generate a compilation error.**

```
char name[4];
name = "San";
```

```
char name[];
name = "San";
```

3. **Initializing a string variable with another string variable is not allowed (like we were doing with ordinary variables) i.e., the followings will generate a compilation error.**

```
char name1[4] = "San";
char name2[4];
name2 = name1;
```

String Input & Output

- **String Input:** **We can input a string with the help of following library functions**
  - `gets()`
  - `scanf()`
- **String Input:** **We can print a string with the help of following library functions**
  - `puts()`
  - `printf()`

We have already discussed these functions in detail in the chapter "Console Input/Output".

String Handling Functions

- The C library supports a large number of string-handling functions that can be used to carry out many of the sting manipulations. In this section, we will discuss the most common string-handling functions.
- To use these string-handling functions we must include the header file `string.h` by using the following preprocessor directive.

```
#include <string.h>
```

strcat()

- **Syntax:** `strcat(str1, str2);`
- **Working:** **It concatenates / appends the sting `str2` to the string `str1`. `str1` should be large enough to accommodate the contains of `str2`.**
- **Example:**

```
...
char str1[10] = "Red";
char str2[] = "Apple";
...
strcat(str1, str2); //str1 will now contain "RedApple".
                    //str2 will now contain "Apple"
...
```

strncat()

- **Syntax:** `strncat(str1, str2, n);`
- **Working:** **It concatenates / appends the first 'n' characters of the string `str2` to the string `str1`. `str1` should be large enough to accommodate the appended characters of `str2`.**
- **Examples:**

```
...
char str1[10] = "Red";
...
strncat(str1, "Apple", 3); //str1 will now contain "RedApp".
...
```

```
...
char str1[10] = "Big";
char str2[] = " Apple";
...
strncat(str1, str2, 10); //str1 will now contain "Big Apple".
                         //str2 will now contain "Apple"
...
```

strcmp()

- **C doesn't permit the comparison of two strings directly. i.e., the following statements are NOT permitted in C.**

```
if(str1 == str2)
if(str1 = "ABC")
```

In order to compare two strings, we have to use the built-in function `strcmp()`.

- **Syntax:**

```
strcmp(str1, str2);
```

- **Working: It compares the two strings `str1` and `str2` character by character from left to right.**
  - **If the two strings are found to be identical, then it returns a zero.**
  - **If the two strings are not identical, then it returns the difference between the ASCII values of the 1st non-matching pairs of characters** (in some compilers it may return "-1").

- **Example:**

```
...
int i;
char str1[] = "Their";
...
i = strcmp(str1, "There"); //i will now contain -9 (ASCII "i" minus ASCII "r").
...
```

strncmp()

- **Syntax:** `strncmp(str1, str2);`
- **Working:** **Same as** `strcmp()`**. The only difference is that it compares the 1st "n" characters of** `str2` **with** `str1`**.**
- **Example:**

```
...
int i, j;
char str1[] = "Their";
...
i = strncmp(str1, "There", 3); //i will now contain 0.
j = strncmp(str1, "There", 4); //j will now contain -9
                                  (ASCII "i" minus ASCII "r").
...
```

strcpy()

- **Syntax:** `strcpy(str1, str2);`
- **Working:** **It copies the string `str2` into the string `str1`. `str1` should be large enough to accommodate the contains of `str2`.**
- **Example:**

```
...
char str1[] = "Handsome";
char str2[] = "Stupid";
...
strcpy(str1, str2); //str1 will now contain "Stupid".
                    //str2 will now contain "Stupid"
...
```

strlen()

- **Syntax:** `strlen(str);`
- **Working:** **It returns the length of the string `str` *excluding* the NULL character.**
- **Example:**

```
...
int i;
char str[] = "Hello";
...
i = strlen(str); //i will now contain "5".
...
```

strrev()

- **Syntax:** `strrev(str);`
- **Working:** **It reverses the characters of the string** `str`**.**
- **Example:**

```
...
char str[] = "Hello";
...
strrev(str); //str will now contain "olleH".
...
```

strupr()

- **Syntax:** `strupr(str);`
- **Working:** **It converts the characters of the string `str` to uppercase.**
- **Example:**

```
...
char str[] = "Hello";
...
strupr(str); //str will now contain "HELLO".
...
```

strlwr()

- **Syntax:** `strlwr(str);`
- **Working:** **It converts the characters of the string `str` to lowercase.**
- **Example:**

```
...
char str[] = "Hello";
...
strlwr(str); //str will now contain "hello".
...
```

strstr()

- **Syntax:** `strstr(str1, str2);`
- **Working:** **It searches the string `str2` in the string `str1`.**
  - **If there is a success, the function returns a pointer to the first occurrence of `str2` in `str1`.**
  - **Otherwise, it returns a NULL pointer.**
- **Example:**

```
...
char str1[] = "Big Boss";
char str2[] = "Boss";
char *s;
...
s = strstr(str1, str2);
printf("The substring is: %s", s); //Output: "The substring is: Boss"
...
```

Programming Examples

- **Programming Example 1:**

```
/* PR6_7.c: Program that copies one string into another (without using strcpy()) and
counts the number of characters copied.  */

# include <stdio.h>
# include <conio.h>

void main()
{
   int i, counter = 0;
   char sourceStr[50], targetStr[50];

   printf("\nEnter some text: ");
   gets(sourceStr);

   //Copying to "targetStr"
   for(i=0;sourceStr[i]!='\0';i++)
   {
       targetStr[i] = sourceStr[i];
       counter++;
   }
   targetStr[i] = '\0';
```

[Cont.]

```
    printf("\nAfter copying: %s", targetStr);
    printf("\nNumber of characters copied: %d", counter);
    getch();
}
```

Output

```
Enter some text: Hello, there!!

After copying: Hello, there!!
Number of characters copied: 14
```

Programming Example 2:

```
/* PR6_8.c: Program that reads a string and finds the total number of vowels in it. */

# include <stdio.h>
# include <conio.h>

void main()
{
   int i, vow = 0;
   char str[50];
   printf("\nEnter some text: ");
   gets(str);

   for(i=0;i<strlen(str);i++)
   {
       if(str[i]=='A' || str[i]=='a' || str[i]=='E' || str[i]=='e' ||
          str[i]=='I' || str[i]=='i' || str[i]=='O' || str[i]=='o' ||
          str[i]=='U' || str[i]=='u')
       {
           vow++;
       }
   }
   printf("\nNumber of vowels: %d", vow);
   getch();
}
```

Output

```
Enter some text: Big brother.

Number of vowels: 3
```

Programming Example 3:

```
/* PR6_9.c: Program that reads a string and counts the total number of alphabets, digits, spaces, and
special symbols present in it. */

# include <stdio.h>
# include <conio.h>

void main()
{
    int i;
    int alphabet = 0, digit = 0, space = 0, specSymb = 0;
    char str[100];
    printf("\nEnter some text: ");
    gets(str);
    for(i=0;i<strlen(str);i++)
    {
        if((str[i]>=65 && str[i]<=90) || (str[i]>=97 && str[i]<=122))
            alphabet++;
        else if(str[i]>=48 && str[i]<=57)
            digit++;
        else if(str[i]==32)
            space++;
        else
            specSymb++;
    }
```

[Cont.]

```
    printf("\nNumber of alphabets: %d", alphabet);
    printf("\nNumber of digits: %d", digit);
    printf("\nNumber of spaces: %d", space);
    printf("\nNumber of special symbols: %d", specSymb);
    getch();
}
```

Output

```
Enter some text: Hello, there !!

Number of alphabets: 10
Number of digits: 0
Number of spaces: 2
Number of special symbols: 3
```

Assignments - III

Complete the experiments given in "Lab Manual - Section 8".

End of Chapter 6