UNIT-IV

Topics: Arrays –Single and Multi-dimensional arrays, Initializing arrays, computing address of an element in array, row major and column major form of an array, character strings and arrays, segmentation fault, bound checking, Sorting Algorithms – Bubble sort, insertion sort, selection sort. –10 Hrs

ARRAYS & CHARACTER STRINGS

BASIC CONCEPT OF ARRAYS

To store a single integer value into a variable an integer variable is declared and then the value can be stored.
For ex. int var;
var=100;
But if more than one value needs to be stored then the above declaration is limited or not sufficient.

- Assume that the program needs to store 5 integer numbers into variables.
- Then, the first possibility is by declaring five different variable of the same type to store the values.

i.e by declaring five different variables:
int var1,var2,var3,var4,var5;
then assigning each variable with a single value i.e
var1=11,var2=22,var3=33,var4=44, var5=55;
The above solution is not a practical when the variables to be declared increase in numbers. Say, if 1000 values have to be stored then certainly with above method one would require 1000 variables to be declared. Such a program will have redundant/repeated statements and length of such a program would be too large.
Alternatively, the arrays could be used to store the values with minimum declaration.
For ex. in the above case to store five variables an array of size five may be declared.
i.e int num[5]; ➔ num is an array of size five of integer
type, which has the capability

to store five different variables
of integer type (same type).

num[0]=11, num[1]=22,num[2]=33, num[3]=44, num[4]=55;

Note the first index of num its 0, in C arrays start with a 0 index.

The following program demonstrates to store five values into the variables without using an array:

```
#include<stdio.h>
void main()
{
int var1;   /* declares five variable var1,var2….var5 */
int var2;
int var3;
int var4;
int var5;
var1= 11;
var2= 22;
var3= 33;
var4= 44;
var5= 55;
printf( "var1: %d \n", var1);
printf( "var2: %d \n", var2);
printf( "var3: %d \n", var3);
printf( "var4: %d \n", var4);
printf( "var5: %d ", var5);
}
```

Output:

```
var1: 11
var2: 22
var3: 33
var4: 44
var5: 55
```

The above program can be written to store five values into a single variable by using an array:

```
#include<stdio.h>
void main()
```

```
{
int num[5];  /*declares only one variable num[5], which*/
/*can store 5 variables from num[0],num[1] till num[4] */
num[0]= 11;
num[1]= 22;
num[2]= 33;
num[3]= 44;
num[4]= 55;
printf( "num1: %d \n", num[0]);
printf( "num2: %d \n", num[1]);
printf( "num3: %d \n", num[2]);
printf( "num4: %d \n", num[3]);
printf( "num5: %d ", num[4]);
}
```

Output:

```
num1: 11
num2: 22
num3: 33
num4: 44
num5: 55
```

Hence, whenever the need arises to store many items of the same type then arrays are preferred compared to the above first approach of multiple variables.

During many occasions while writing a program arrays are the most suitable type of storage for storing data such as

- **Rollno of students in a class**
- **List of colors,**
- **Marks secured by students in a class,**
- **Names of the menu items in a canteen/restaurant**
- **Set of Natural numbers etc.**

ARRAY

• Array is a collection of elements having same data type, same name and stored sequentially in a contiguous memory locations.

• Any element can be accessed by using

→ name of the array and

→ location of element in the array

• Arrays are of two types:

1) Single dimensional array

2) Multi dimensional array

SINGLE DIMENSIONAL ARRAY

• Single dimensional array facilitates to declare similar elements of same type with a single value enclosed in brackets that indicates the size of single dimensional array.

• Inside the computer memory, all the elements of the array are stored contiguously in the memory locations as shown below.

num[0]	num[1]	num[2]	num[3]	num[4]
11	22	33	44	55

In the computer memory location num[0] stores 11
In the computer memory location num[1] stores 22
In the computer memory location num[2] stores 33
In the computer memory location num[3] stores 44
In the computer memory location num[4] stores 55

Declaration of Single Dimensional Arrays

• The general declaration syntax is:

data_type array_name[size_of_array];

where data_type can be int, float or char etc.
array_name is name of the array and **size_of_array** indicates number of elements in the array.

• For ex:
    int num[5];

is represented in the following way before initializing.

num[0]	num[1]	num[2]	num[3]	num[4]

while it stores garbage value in the memory locations when not initialized for the first time after declaration of the array.
Accessing an Array:
Array can be accessed by name and the index of the element.
For ex. int data[3]={555,888,999};
  int var= data[2];
The above statement shows how the 3rd element can be accessed using the index and then stored into a variable var.

Initializing Arrays:
• The Arrays can be initialized in two ways:

Initialization/Assigning values explicitly through program.
At compile time
At run time
Assigning input values accepted from the keyboard.

Initialization One-Dimensional Array at Compile Time

data_type ArrarrayName[size_of_array]={v1,v2,v3,….};
where v1, v2, v3 are values

• For ex:
    int marks[5]={65,74,63,95,88};

• The above statement can be as shown below:

num[0]	num[1]	num[2]	num[3]	num[4]
65	74	63	95	88

•Example-1: Program to illustrate compile time initialization of one-dimensional array of marks secured by a student in five subjects, initialized at the time of declaration of the variable.

```
#include<stdio.h>
void main()
{
int marks[5]={65,74,63,95,88};
printf("Marks in subject-1: %d \n", marks[0]);
printf("Marks in subject-2: %d \n", marks[1]);
printf("Marks in subject-3:%d \n", marks[2]);
printf("Marks in subject-4: %d \n", marks[3]);
printf("Marks in subject-5: %d ",marks[4]);
}
```

Output:
Marks in subject-1:65
Marks in subject-2:74
Marks in subject-3:63
Marks in subject-4:95
Marks in subject-5:88

•Example-2: Another method
Program to illustrate compile time initialization of one-dimensional array of marks secured in five subjects by a student after declaration of the variable.

```
#include<stdio.h>
void main()
{
	int marks[5];
marks[0]=65;
```

```
marks[1]=74;
marks[2]=63;
marks[3]=95;
marks[4]=88;
printf("Marks in subject-1: %d \n", marks[0]);
printf("Marks in subject-2: %d \n", marks[1]);
printf("Marks in subject-3:%d \n", marks[2]);
printf("Marks in subject-4: %d \n", marks[3]);
printf("Marks in subject-5: %d ",marks[4]);
}
```

Output:

```
Marks in subject-1:65
Marks in subject-2:74
Marks in subject-3:63
Marks in subject-4:95
Marks in subject-5:88
```

Initializing One dimensional array at run time
Arrays when assigned with values by accepting from the user during the execution of the program are known as Run time initialization.

- For ex: int marks[3]

```
scanf("%d", &marks[0]); //read marks from the keyboard
//into array at location 1

scanf("%d", &marks [1]); //read marks from the keyboard
 //into array at location 2

scanf("%d", &marks [2]); //read marks from the keyboard
 // into location 3
```

- The following statement reads & initializes marks in 3 subjects.

```
for(i=0;i<3;i++)
{
```

```
    scanf("%d", &marks[i]);
}
```

• To display the subject marks stored in the array:

```
for(i=0;i<3;i++)
{
    printf("%d", marks[i]);
}
```

• Example: Program to illustrate runtime initialization of one dimensional array.

```
#include<stdio.h>
void main()
{
int marks[3], i;
printf("Enter marks of a student in three subjects:");
for(i=0;i<3;i++)
{
scanf("%d", &marks[i]);
}
for(i=0;i<3;i++)
{
printf("\nMarks in subject %d is %d", i+1,marks[i]);
}
}
```

Output:

```
Enter marks of a student in three subjects: 88 65 97
Marks in subject 1 is 88
Marks in subject 2 is 65
Marks in subject 3 is 97
```

MULTI DIMENSIONAL ARRAYS:

Arrays with more than single dimension are called multi- dimensional arrays.

• For ex:

```
int mat[3][3]; // declares a two dimensional array of size three
               // rows and three columns
```

The above declaration inside the computer memory can be represented as follows:

	Column-1	Column-2	Column-3
Row-1	mat[0][0]	mat[0][1]	mat[0][2]
Row-2	mat[1][0]	mat[1][1]	mat[1][2]
Row-3	mat[2][0]	mat[2][1]	mat[2][2]

Two dimensional array's are arranged in rows and columns form in the form of a matrix.

The elements of the array index start with 0,0 for ex. in the above figure the first element of integer type is stored at row=0 and col=0 index i.e at mat[0][0] and the second element is stored at row 0 and col=1 i.e at mat[0][1] & so on.

The General form of the 2D array is as follows:

data_type array_name[size_of_rows][size_of_cols];

data_type could be any basic types such as int,char or float etc.

array_name: any valid identifier name

size_of_rows :number of rows in 2D array

size_of_cols :number of columns in 2D array

The 2D array's can be also be used to store string of characters:

char names[3][10];

The meaning of above declaration is the compiler reserves 60 bytes of storage (assuming integer takes two bytes) such that first element of char type is stored at row=0 and col=0 index i.e at mat[0][0] and the second element is stored at row 0 and col=1 i.e at mat[0][1] & so on.

Ex. char names[3][6]={"Amul", "Ramesh", "Vijay"};

The computer represents the above declaration in the following way:

	Col-1	Col-2	Col-3	Col-4	Col-5	Col-6	Col-7
Row-1	‘A’	‘m’	‘u’	‘l’	‘\0’		
Row-2	‘R’	‘a’	‘m’	‘e’	‘s’	‘h’	‘\0’
Row-3	‘V’	‘l’	‘j’	‘a’	‘y'	‘\0’	

Note the names are represented horizontally row wise where each character is stored in a row, column combination. Ex. the letter ‘u’ of Amul is stored at row=1 and col=3 (i.e names[1][3]).
The maximum names that can be stored in the above array is three coz. array has maximum three rows while the maximum number of characters in a name can’t be more than 5 coz. maximum column size is 6 (including the null character).
The last character is always null terminated string represented as ‘\0’ (backslash zero). Hence, while declaration of 2D array should also consider the number of columns i.e column size should be maximum characters expected in a string plus one.

Initialization of Two Dimensional Arrays
At Compile Time:
For ex:
int matrix[3][3]= {{1,2,3},{4,5,6},{7,8,9}}; or
int matrix[3][3]= {1,2,3,4,5,6,7,8,9};
The above code can be represented as shown below:

	Col-1	Col-2	Col-3
Row-1	1	2	3
Row-2	4	5	6
Row-3	7	8	9

Compile Time Initialization:
For ex:

```
#include<stdio.h>
void main()
{       int matrix[3][3]= {1,2,3,4,5,6,7,8,9};
```

```
        int i,j;
                printf("\The elements of the matrix are:\n");
        for(i=0;i<3;i++)
        for(j=0;j<3;j++){
                printf("%2d", matrix[i][j]);
        }
}
```

At Run Time:

• For ex:

```
int matrix[3][3];
scanf("%d", &matrix[0][0]); //reads values entered
                                        //into matrix at row=0 col=0
scanf("%d", &matrix[0][1]); //reads values entered into
                                                //row=0 and col=1.
scanf("%d", &matrix[0][2]); //reads values entered into
                                                //row=0 and col=2
scanf("%d", &matrix[1][0]);
scanf("%d", &matrix[1][1]);
scanf("%d", &matrix[1][2]);
scanf("%d", &matrix[2][0]);
scanf("%d", &matrix[2][1]);
scanf("%d", &matrix[2][2]); //reads values entered into
                                                //row=2 and col=2
```

• The following loop reads 9 values entered by the user:

```
for(i=0;i<2;i++){
for(j=0;j<3;j++){
        scanf("%d", &matrix[i][j]);
}
}
```

• The following loop displays 9 values to the output screen:

```
for(i=0;i<3;i++){
for(j=0;j<3;j++){
```

```
        printf("%d ", matrix[i][j]);
}
}
```

```
#include<stdio.h>
void main()
{int matrix[3][3];
 int i,j;
 printf("\nEnter the elements of the matrix of order [3x3]:\n");

for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
scanf("%d", &matrix[i][j]);
}
}
printf("\nThe elements of the matrix are:\n");
for(i=0;i<3;i++)
  {
for(j=0;j<3;j++)
{
printf("%2d", matrix[i][j]);
}
printf("\n");
}
}
```

Output:

Enter the elements of the matrix of order [3x3]:

11 22 33 44 55 66 77 88 99

The elements of the matrix are:

11 22 33

44 55 66

77 88 99

Q. Write a C program to add the elements of square matrices. Display the resultant matrix to the output screen.

Solution:

```
#incude<stdio.h>
void main()
{
int matrix1[3][3], matrix2[3][3], matrix3[3][3];
int rows1,cols1, rows2,cols2;
printf("Enter the number of rows and columns of first matrix:");
scanf("%d%d",&rows1,&cols1);
printf("Enter the number of rows and columns of second matrix:");
scanf("%d%d",&rows2,&cols2);
if(rows1!=rows2 && cols1!=cols2)
{
        printf("Not a square matrix. Can't add elements !!!");
        exit(1);
}
printf("Enter the elements of first matrix:\n");
for(i=0;i<rows1;i++)
for(j=0;j<cols1;j++) {
        scanf("%d", &matrix1[i][j]);
}

printf("Enter the elements of second matrix:\n");
for(i=0;i< rows1;i++)
for(j=0;j< cols;j++){
        scanf("%d", &matrix2[i][j]);
}

printf("The elements of first matrix:\n");
for(i=0;i< rows1;i++){
for(j=0;j< cols;j++){
printf("%3d", matrix1[i][j]);
}
printf("\n");
}
printf("\nThe elements of second matrix:\n");
```

```
for(i=0;i< rows;i++){
for(j=0;j< cols;j++){
printf("%3d", matrix2[i][j]);
}
printf("\n");
}
for(i=0;i< rows;i++){
for(j=0;j< cols;j++){
        matrix3[i][j]= matrix1[i][j]+ matrix2[i][j];
}
}
printf("\nThe elements of resultant matrix:\n");

for(i=0;i< rows;i++){
for(j=0;j< cols;j++){
        printf("%3d", matrix3[i][j]);
}
        printf("\n");
}
}
```

Output:

Enter the number of rows and columns of first matrix: **2 2**
Enter the number of rows and columns of second matrix: **2 2**
The elements of first matrix: **1 1 1 1**
The elements of second matrix: **1 2 3 4**
The elements of resultant matrix:
2 3
4 5

Enter the number of rows and columns of first matrix: **3 3**
Enter the number of rows and columns of second matrix: **3 3**
The elements of first matrix: **1 1 1 1 1 1 1 1 1**
The elements of second matrix: **1 2 3 4 5 6 7 8 9**
The elements of resultant matrix:
2 3 4

5 6 7
8 9 10

I) Computing the address of an element of an Array in row major.

Address of [I,J][I,J]th element in row major

order =B+W[n(I−Lr)+[J−Lc]]=B+W[n(I−Lr)+[J−Lc]]

where B denotes base address, W denotes element size in bytes, n is the number of

columns; Lr is the first row number, Lc is the first column number.

Ex:

int mat[3][4];

Address of mat[1][2] === mat[0][0] +2[4(2-0)+[3-0]]

mat[0][0]+2(4(1)+2))= mat[0][0]+12= 12

1	2	3	4
(0,0)	**(0,1)**	**(0,2)**	**(0,3)**
5	6	7	8
(1,0)	**(1,1)**	**(1,2)**	**(1,3)**
10	11	12	13
(2,0)	**(2,1)**	**(2,2)**	**(2,3)**

II) Computing the address of an element of an Array in column major.

Address of [I,J][I,J]th element in column major

order =B+W[(I−Lr)+n[J−Lc]]=B+W[(I−Lr)+n[J−Lc]]

where B denotes base address, W denotes element size in bytes, n is the number of rows; Lr is the first row number, Lc is the first column number.
mat[3][4];

Address of mat[1][2] === mat[0][0] +2[(1-0)+3[2-0]]

mat[0][0]+2(4(1)+2))= mat[0][0]+14= 2+14=16th Byte

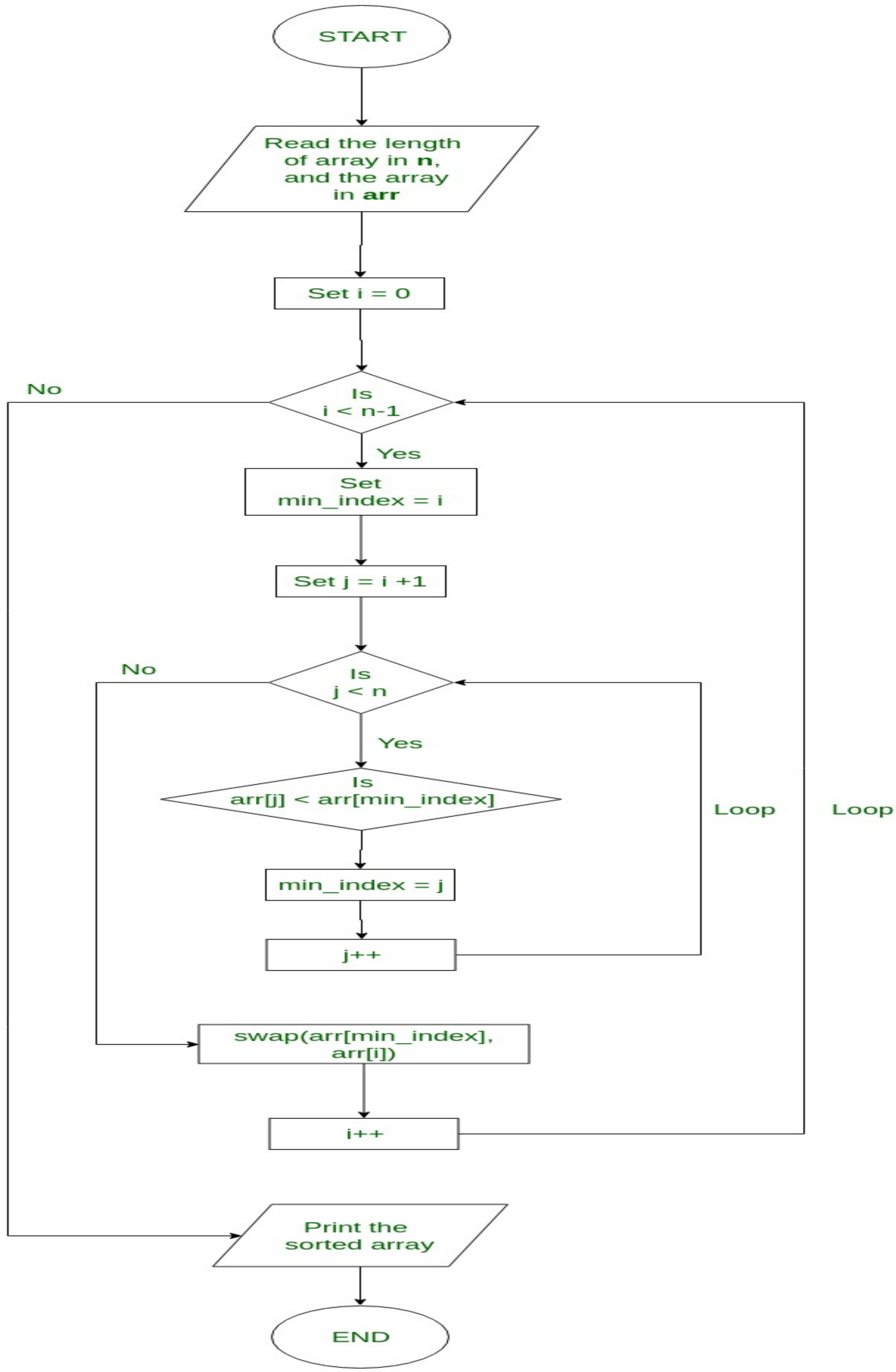
1	2	3	4
(0,0)	**(0,1)**	**(0,2)**	**(0,3)**
5	6	7	8
(1,0)	**(1,1)**	**(1,2)**	**(1,3)**
10	11	12	13
(2,0)	**(2,1)**	**(2,2)**	**(2,3)**

Selection Sort Algorithm:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.
1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.
In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

```
arr[] = 64 25 12 22 11


// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Flowchart of the Selection Sort:



Flowchart for Selection Sort

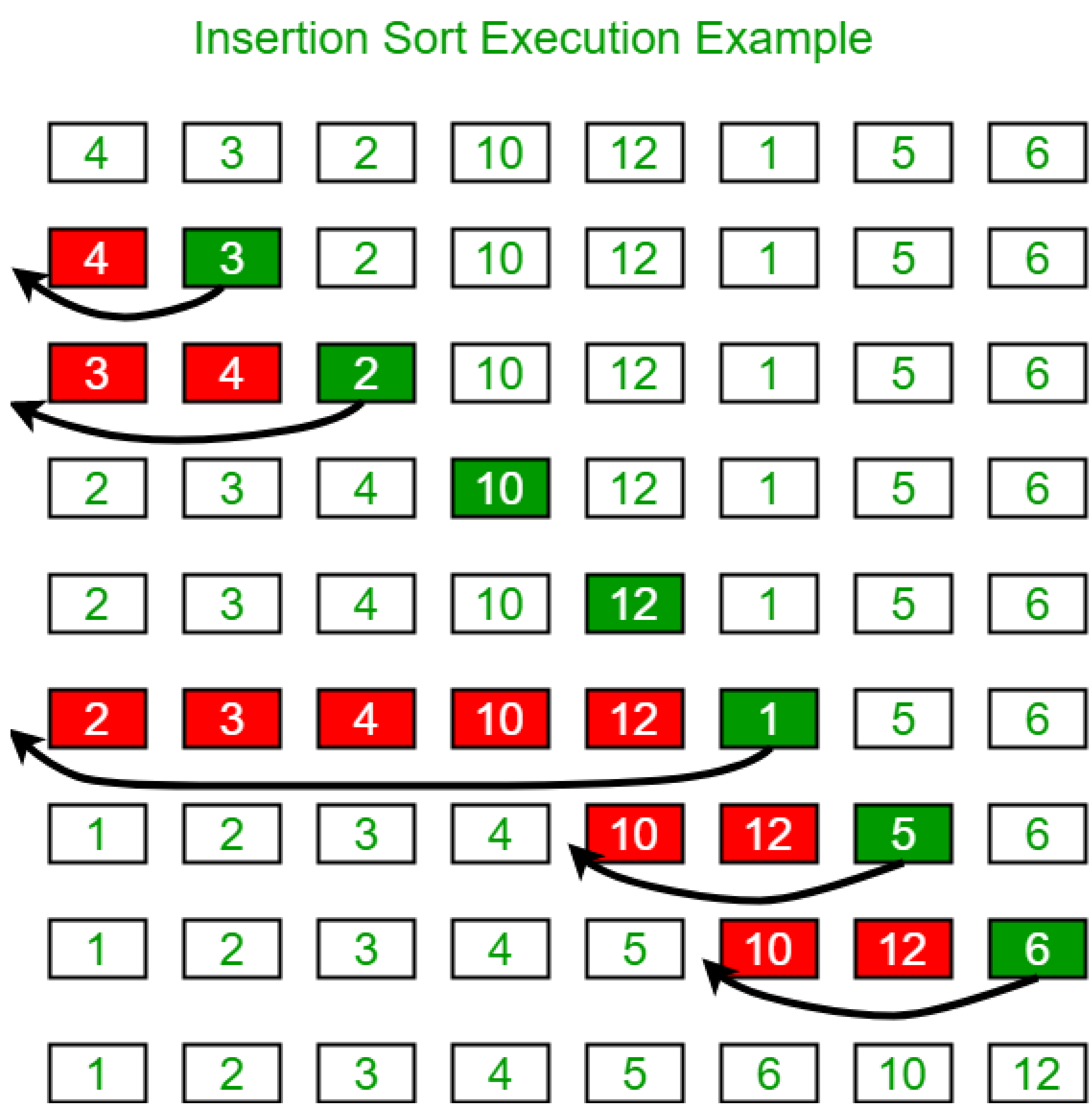
/* Selection Sort */

```
#include <stdio.h>
int main()
{
   int a[] = {64, 25, 24, 22, 11};
   int i, j, min,temp,k;

   // Move one by one the boundary of unsorted subarray
   for (i = 0; i <5; i++)
   {
      // Find the minimum element in unsorted array
      min = i;
      for (j = i+1; j < 5; j++)
      {
         if (a[min] > a[j])
         {
             temp=a[j];
             a[j]=a[min];
             a[min]=temp;
         }
      }
 }
   printf("\nSorted list:\n");
   for (i = 0; i < 5; i++)
   {
      printf("%3d\t",a[i]);
   }
   return 0;
}
```

Insertion Sort:
Consider the set of unsorted elements stored in an array.

Example:
14, 13, 15, 7, 8
Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 13 is smaller than 14, move 14 and insert 13 before 14
13,14, 15,7, 8
i = 2. 15 will remain at its position as all elements in A[0..I-1] are smaller than 15
13,14, **15**,7, 8
i = 3. 7 will move to the beginning and all other elements from 13 to 15 will move one position ahead of their current position.
7,**13,14**, **15**,8
i = 4. 8 will move to position after 7, and elements from 13 to 15 will move one position ahead of their current position.
7,**8,13,14**, **15**



Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

Algorithm
// Sort an arr[] of size n
Loop from i = 1 to n-1.
……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

Insertion Sort Program:

```
#include <stdio.h>
int main()
{  int a[] = { 5,8,3,6,2};
   int i, j, temp,min;
   for (i = 1; i <5; i++)

 {
      // Assume the second element as min in an unsorted array
         min=a[i];
         j=i;
      // move the min element till it finds its position
         if (min < a[j-1])
         {
            while(a[j] < a[j-1] && j > 0 )
            {
               // insert the element by moving other elements
                temp=a[j];
                a[j]=a[j-1];
                a[j-1]=temp;
                j--;
            }
         }
   }
   printf("\nSorted list:\n");
   for (i = 0; i < 5; i++)
   {
      printf("%3d\t",a[i]);
   }
   return 0;
}
```

Segmentation fault Error:

Segmentation fault is a specific kind of error caused by accessing memory that "***does not belong to you***."

When a program code tries to do read or write operation in a read only location in memory, it leads to a core dump and results in a Segmentation fault error.
It is an error indicating accessing memory that is inaccessible.

Examples:

a) **Accessing out of array index bounds :**

```
// C program to demonstrate segmentation fault when array out of bound is accessed.
#include<stdio.h>
void int main()
{
  int arr[4];
  arr[5] = 10;  // Accessing out of bound
}
```

b) **Improper use of scanf() :**

scanf() function expects address of a variable as an input In this program **X** takes value of 6 and assume it's address as 2000. When scanf() reads **X**, input fetched from **STDIN** is placed in invalid memory location 6 which should be 2000 instead. It's leads to memory corruption and hence to Segmentation fault.

```
// C program to demonstrate segmentation fault when value is passed to scanf
#include <stdio.h>
int main()
{
  int X = 6;            //Assume x  is allocated the address 2000 by the compiler
  scanf("%d", X);
  return 0;
}
```

Output:

Segmentation fault

Bound Checking: Accessing array out of bounds

In high level languages such as Java, there are functions which prevent from accessing array out of bound by generating a exception. But in case of C, there is no such functionality, but programmer needs to take care of this situation.

C doesn't provide any clue to deal with this problem of accessing invalid index. An undefined behavior (UB) is a result of executing computer code whose behavior is not prescribed by the language specification. This generally happens when the compiler of the source code makes certain assumptions, but these assumptions are not satisfied during execution.

Access non allocated location of memory: The program can access some piece of memory which is owned by it.

```
// Program to demonstrate
// accessing array out of bounds
#include <stdio.h>
int main()
{
   int num[3];
   num[0]=11;
   num[1]=22;
   num[2]=33;
  printf("num[0] is %d\n", num[0]);
   // num[6] is out of bound
   printf("num[8] is %d\n", num[6]);
   return 0;
}
```

Output:

```
num[0] is 11
num[6] is -3354   //Garbage Value
```

========================= **End of UNIT-IV** ===========================