UNIT – IV

Factoring methods: Finding the square root of a number, the smallest divisor of a number, the greatest common divisor of two integers, generating prime numbers.

Pointers and arrays: Pointers and addresses, pointers and function arguments, pointers and arrays, address arithmetic, character pointers and functions, pointer array; pointers to pointers, Multi-dimensional arrays, initialization of arrays, pointer vs. multi-dimensional arrays, command line arguments, pointers to functions, complicated declarations.

Array Techniques: Array order reversal, finding the maximum number in a set, removal of duplicates from an order array, finding the kth smallest element

---

Note: Refer Additional Supplement for Factoring methods algorithms

Finding the square root of a number:

Program:

```
/**
 * C program to find square root of a number
 */

#include <stdio.h>
#include <math.h>

int main()
{
    double num, root;

    /* Input a number from user */
    printf("Enter any number to find square root: ");
    scanf("%lf", &num);

    /* Calculate square root of num */
    root = sqrt(num);

    /* Print the resultant value */
    printf("Square root of %.2lf = %.2lf", num, root);

    return 0;
}
```

Output:

```
Enter any number to find square root: 25
Square root of 25.00 = 5.00
```

The Smallest Divisor of a Number:

Program:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
        int num,i;
        printf("Enter an integer");
        scanf("%d",&num);
        for(i=2;i<num;i++)
        {
                if((num%i)==0)
                {
                        printf("Smallest Divisor :%d",i);
                        break;
                }
        }
        return 0;
}
```

Output:

```
Enter an integer36
Smallest Divisor :2
```

The Greatest Common divisor of two integers:

Program:

```
#include <stdio.h>
int main()
{
   int n1, n2;

   printf("Enter two positive integers: ");
   scanf("%d %d",&n1,&n2);
   while(n1!=n2)
   {
      if(n1 > n2)
         n1 -= n2;
      else
         n2 -= n1;
   }
   printf("GCD = %d",n1);
   return 0;
}
```

Output:

```
Enter two positive integers: 32 16
GCD = 16
```

Generating Prime Numbers

Program:

```
#include<stdio.h>
int main()
{
   int n,i,fact,j;
   printf("Enter the Number");
   scanf("%d",&n);
   printf("Prime Numbers are: \n");
   for(i=1; i<=n; i++)
   {
      fact=0;
      for(j=1; j<=n; j++)
      {
         if(i%j==0)
            fact++;
      }
      if(fact==2)
         printf("%d " ,i);
   }
   return 0;
}
```

Output:

```
Enter the Number20
Prime Numbers are:
2 3 5 7 11 13 17 19
```

Pointers and Arrays

Introduction to Arrays:

In a programming language the data that has to be processed is loaded in memory and held in variables. When we want to process an integer value, we declare an int variable and assign a value to it.

Assume we want to write a program that prints the total marks scored by students in a class (say, for a total of 30 students).

One way of doing it is to declare 30 int variables, which is not scalable when we think of multiple classes or a complete school.

In such cases, when we want to store multiple values of the same data type, C provides us a derived data type called array.

An array is an ordered sequence of finite data items of the same data type and that shares a common name. The common name is the array name and each individual data item is known as an element of array.

The elements of the array are stored in contiguous memory locations starting from a memory location allocated/stored in the array variable.

For example, an array of size 10 (int arr[10];) can be visualized as shown below.

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]	arr[6]	arr[7]	arr[8]	arr[9]

Essentially an array can be thought of as a sequence of buckets. The first bucket is identified with number 0, the second bucket with 1 and so on. This number is called the index (or) subscript.

For example, if we want to store a value 369 at the first index, the code is arr[0] = 369;.
Similarly, if we want to store a value 567 in the second bucket, the code will be arr[1] = 567; and so on.
If we want to store a value 111 in the last bucket, the code will be arr[9] = 111;.

An element of an array is retrieved/accessed using its index. For example, the value stored in the first bucket can be printed using the below code:
printf("%d", arr[0]); // remember that array's **index** starts from **0** and not **1**

Example:

```
#include <stdio.h>

void main() {

        int arr[10];

        arr[0] = 10;

        arr[1] = 20;

        arr[2] = 100;

        arr[3] = 200;

        printf("The value in arr[0] : %d\n",arr[0] ); // Print the 0th element of arr

        printf("The value in arr[1] : %d\n",arr[1] ); // Print the 1st element of arr

        printf("The value in arr[2] : %d\n",arr[2] ); // Print the 2nd element of arr

        printf("The value in arr[3] : %d\n",arr[3] ); // Print the 3rd element of arr

}
```

Understanding One dimensional Arrays:

There are one-dimensional and multi-dimensional arrays.

A one-dimensional array can be used to represent a list of data elements and is also known as a vector.

A two-dimensional array is used to represent a table of data items consisting of rows and columns and is also known as a matrix.

A three-dimensional array can be used to represent a collection of pages. We will learn more about these in multi-dimensional arrays in later sections.

The syntax for declaring a one-dimensional array is given below:

```
data_type arrayname[size]; //a single array is declared

data_type arrayname1[size1], arrayname2[size2]...arraynameN[sizeN]; ////multiple arrays are declared in the same line
```

Here, data_type refers to the data type of the elements in the array and it can be a primitive data type.
arrayname1, arrayname2, ...etc refers to the identifiers which represent the array names.

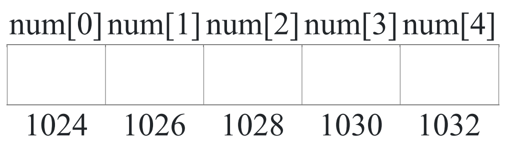
size is an integer expression representing total number of elements in the array.

Let us consider an example

```
int num[5];
```

The above one-dimensional array declaration defines an integer array by name num of size 5, meaning it represents a block of 5 consecutive storage locations that store int values.

Here each element in the array can be accessed by num[0], num[1], num[2], num[3], num[4], where 0, 1, 2, 3, 4 represent the subscripts or indices of the respective elements in the array.



Each element in an array can be accessed by the name of the array followed by the subscript or directly by the address. The array variable holds the base address of that entire array.

Example:

```
#include <stdio.h>

void main() {

        int a[10], i, n;

        printf("Enter how many values you want to read : ");

        scanf("%d",&n ); // Complete the code

        for (i = 0 ; i < n ; i++) { // Complete the code

                printf("Enter the value of a[%d] : ", i);

                scanf("%d",&a[i] ); // Complete the code

        }

        printf("The array elements are : ");

        for (i = 0 ; i < n ; i++) { // Complete the code

                printf(" %d ", a[i] ); // Complete the code

        }

}
```

One Dimensional Array Accessing with base address:

Let us consider an example declaration of a int array of size 10.

```
int num[10];
```

Each element in the array can be accessed by num[0], num[1],....,num[9], where 0, 1, 2,..,9 represents subscripts or indices of the elements in the array.

num[0]	num[1]	num[2]	num[3]	num[4]	num[5]	num[6]	num[7]	num[8]	num[9]
3024	3026	3028	3030	3032	3034	3036	3038	3040	3042

In the above example, the array variable num contains the base address of the entire array. Let us assume the base address as some random number : 3024.

Whenever 1 is added to the array variable num, it gives the next location of the array i.e.,

num + 1 is same as &num[1]

3024 + 1 = 3024 + 1 * [scale factor] = 3024 + 1 * 2 = 3024 + 2 = 3026

In the above line, 2 is the size of data type int in 16-bit machines. 2 is also referred as scale factor.

The formula for finding the ith location of array element is

Address of the ith element = base_address + i * scale_factor

The scale factor is automatically calculated by the system, which is a value representing the size of the data type of the array.

Note : In the below program, the symbol * in *(a + i) is used to find the value at the given address. We will learn more about it while learning about pointers in the later sections.

Example:

```
#include <stdio.h>

void main() {

        int a[10], i, n;

        printf("Enter how many values you want to read : ");

        scanf("%d", &n);

        for (i = 0; i < n; i++) {
                printf("Enter the value of a[%d] : ", i);

                scanf("%d", a + i);

        }

        printf("The array elements are : ");

        for (i = 0; i < n; i++) {

                printf(" %d ", *(a + i)); //Correct the code

        }

}
```

Two Dimensional Array accessing with base address:

Let us consider an example of a two-dimensional array:

int num[3][3]; // declares an array with 3 rows and 3 columns representing a block of 9 consecutive storage locations

num[0][0]	num[0][1]	num[0][2]	num[1][0]	num[1][1]	num[1][2]	num[2][0]	num[2][1]	num[2][2]
row - 0 column - 0	row - 0 column - 1	row - 0 column - 2	row - 1 column - 0	row - 1 column - 1	row - 1 column - 2	row - 2 column - 0	row - 2 column - 1	row - 2 column - 2
9024	9026	9028	9030	9032	9034	9036	9038	9040

In the above example, the array variable num contains the base address of the entire array. Let us assume the base address as some random number : 9024.

The formula for finding the location of element in the ith row and jth column is :

Address of the ith row and jth column element
= *(base_address + i) + j
= base_address + i * total_columns * scale_factor + j * scale_factor

Using the above formula, the location of num[0][1] can be calculated as below:

*(num + 0) + 1 // where 0 is the row's index and 1 is the column's index= 9024 + 0 * total_columns * [scale factor] + 1 * [scale factor]
= 9024 + 0 * 3 * 2 + 1 * 2
= 9024 + 0 + 2
= 9026

In the above line, 2 is the size of data type int in 16-bit machines. 2 is also referred as scale factor.

The scale factor is automatically calculated by the system, which is a value representing the size of the data type of the array.

Note : In the below program, the code snippet *(num + i) + j is used to find the address of ith row and jth column.
And the code snippet * (*(num + i) + j) is used to find the value at ith row and jth column
We will learn more about the usage of * symbol while learning about pointers in the later sections.

Example:

```
#include <stdio.h>

void main() {

        int num[5][5], i, j, rows, cols;

        printf("Enter row and column sizes : ");

        scanf("%d %d", &rows, &cols);

        for (i = 0; i < rows; i++) {

                for (j = 0; j < cols; j++) {

                        printf("Enter the value of num[%d][%d] : ", i, j);

                        scanf("%d", *(num + i) + j);

                }

        }

        printf("The given matrix is\n");

        for (i = 0; i < rows; i++) {

                for (j = 0; j < cols; j++) {

                        printf("%d ", *(*(num + i) + j)); // Correct the code

                }

                printf("\n");

        }

}
```

Understanding three dimensional Array:

A three-dimensional array can be visualised as an array of two-dimensional arrays. The 3-D array can also be visualised as pages which containing a 2-D array with elements in rows and columns.

Below is an example of a three-dimensional array:

Syntax:

data_type arrayname[size1][size2][size3];

Example declaration:

int num[2][2][2];

For example the above 3-D array can be visualised as 2 pages, each containing a 2-D array of 2 rows and columns each. totalling to 8 elements, which are stores in consecutive storage locations.

num[0][0][0]	num[0][0][1]	num[0][1][0]	num[0][1][1]	num[1][0][0]	num[1][0][1]	num[1][1][0]	num[1][1][1]
page - 0 row - 0 column - 0	page - 0 row - 0 column - 1	page - 0 row - 1 column - 0	page - 0 row - 1 column - 1	page - 1 row - 0 column - 0	page - 1 row - 0 column - 1	page - 1 row - 1 column - 0	page - 1 row - 1 column - 1
1024	1026	1028	1030	1032	1034	1036	1038

Example:

```
#include <stdio.h>

void main() {

        int a[3][3][3], i, j, rows, cols, pages, k;

        printf("Enter page, row and column sizes : ");

        scanf("%d %d %d", &pages, &rows, &cols );

        for (i = 0 ; i < pages ; i++) {

                for (j = 0 ; j < rows ; j++) {

                        for (k = 0 ; k < cols ; k++) {

                                printf("Enter the value of a[%d][%d][%d] : ", i, j, k);

                                scanf("%d", &a[i][j][k] );

                        }
                }
        }
        for (i = 0 ; i < pages ; i++) { // Correct the code
                for (j = 0 ; j < rows ; j++) { // Correct the code
                        for (k = 0 ; k < cols; k++) { // Correct the code
                                printf("The value of a[%d][%d][%d] : %d\n",i,j,k, a[i][j][k]); //
                        }
                }
        }
}
```

Initialization of Arrays:

The elements of an array can also be initialized with the values during declaration instead of reading them by the I/O functions.

Note that an array can be initialized in its declaration only. The lists of values should be enclosed within braces. The values should be separated by commas and must be constants or constant expressions.

One-dimensional array can be initialized as int a[4] = {10, 20, 30, 40};. The values within the braces are scanned from left to right and assigned them to a[0], a[1], a[2] and a[3] respectively.

A two-dimensional array can be initialized as int a[3][2] = {{20, 30}, {40, 50}, {60, 70}};. Here, the values within the inner braces are assigned as elements to each row.

If the number of values initialised for an array is less than the size mentioned, the missing elements are assigned to zero.

For example in int a[3][4] = {{1, 2}, {3, 4, 5}};, the elements 1 and 2 are stored in a[0][0], a[0][1] of the 0th row and elements 3, 4 and 5 are stored in a[1][0], a[1][1] and a[1][2] of the 1st row respectively. All the other elements are initialised to zero.

In the example, int a[3][4] = {1, 2, 3, 4, 5};, the values or elements 1, 2, 3, 4 and 5 are assigned starting from left-side to a[0][0], a[0][1], a[0][2], a[0][3] and a[1][0] respectively.

Important Points in Arrays:

Below are some important points to remember regarding arrays:

An array is a derived data type.

It can store one or more elements of a same data type.
An element in an array can be referred as arr[index], where arr is the name of the array and index is an integer constant which specifies the location of the element in the array.

- An array subscript (index) always starts at zero. Hence, it is said that the array uses zero based addressing. The last element's index is always size-1.
- Elements within an array are stored in consecutive memory locations.
- The array name holds/stores the starting memory location of the array, this address is known as the base address of the array.
- Arrays can be one-dimensional or multi-dimensional.
- Array declaration must contain the size without fail.
- The array size must be of integer type which is greater than zero.
- In a multi-dimensional array, each dimension uses a separate pair of square brackets.
- It is a good programming practice to mention the size using symbolic constants which facilitate easier program maintenance.
- It is illegal to refer to the elements that are out of the array bounds. For an array a[10], the array bounds are from a[0] to a[9] only. The subscript or index values other than values between 0 to 9 are illegal hence will throw an error saying the array access is out of bounds.
- In C the compiler does not check on the bounds of an array, hence we will see an error only during execution.

Understanding Functions with one dimensional Arrays:

Arrays can also be passed as arguments to the functions like normal variables.

The main difference between passing the normal variable and an array is that, the copies of the values of the normal variables are passed from the calling function to called function, but where as the passing parameter is an array, then the base address of that entire array has to be passed.

In the case of array, elements are not copied, but the function works with a different array name on the original array itself.

Hence, any modifications done to the array with in the function, are reflected in the array of the caller too.

In the below example a[10] is an array declared with in the main() function and the function calls read(a, n) and display(a, n) are made in the main() function.

In the function call read(a, n), a is the array name which represents the base address of the entire array a[10] and n is the local variable in the main() function .

The function definition of read() receives the base address of the array a from main(), i.e., placed an alias name for the array a in the read(), so any modifications made to the array in the read() function will also reflected to the array in the main() function.

The function definition of read() also receives the the copy of the value of n, i.e., it receives only value, so any modifications made to the value does not reflect the value with in the main() function.

Example:

```
#include <stdio.h>
void read(int [], int);
void display(int [], int);
void main() {
        int a[10], n;
        printf("Enter n value : ");
        scanf("%d", &n);
        read(a, n);
        display(a, n);
}
void read(int x[10], int n) {
        int j;
        printf("Enter %d elements : ", n);
        for (j = 0; j < n; j++) {
                scanf("%d", &x[j]);
        }
}
void display(int y[10], int n) {
        int j;
        printf("Elements in the array are : ");
        for (j = 0; j < n; j++) {
                printf("%d ", y[j]);
        }
}
```

Functions with Two dimensional Arrays:

Like one-dimensional arrays, a two-dimensional array can also be passed as an argument to the functions.

Either one-dimension or two-dimension array is passed as an argument, it will send the base address of the entire array.

In the case of arrays, elements are not copied, but the function works with a different array name on the original array itself.

Hence, any modifications done to the array with in the function, are reflected in the array of the caller too.

In the below sample code a[5][5] is an array declared with in the main() function and the function calls read(a, m, n) and display(a, m, n) are made in the main() function.

In the function call read(a, m, n), a is the array name which represents the base address of the entire array a[5][5] and m, n are the local variables in the main() function representing row and column sizes.

The function definition of read() receives the base address of the array a from main(), i.e., placed an alias name for the array a in the read(), so any modifications made to the array in the read() function will also reflected to the array in the main() function.

The function definition of read() also receives the the copy of the values of m and n, i.e., it receives only values, so any modifications made to the value does not reflect the value with in the main() function.

In the function declaration, representation of the row and column sizes of a matrix are mandatory.

Example:

```
#include <stdio.h>
void read(int [5][5], int, int);
void display(int [5][5], int, int);
void main() {
        int a[5][5], m, n;
        printf("Enter row and column sizes of a matrix : ");
        scanf("%d%d", &m, &n);
        read(a, m, n);
        display(a, m, n);
}
void read(int x[5][5], int m, int n) {
        int i, j;
        printf("Enter %d elements : ", m*n);
        for (i = 0; i < m; i++) {
                for (j = 0; j < n; j++) {
                    scanf("%d", &x[i][j]);
            }
        }
}
void display(int y[5][5], int m, int n) {
        int i, j;
        printf("Elements in the matrix are\n");
        for (i = 0; i < m; i++) {
                for (j = 0; j < n; j++) {
                        printf("%d ", y[i][j]);
                }
                printf("\n");
        }
}
```

Basics of Pointers:

Pointer:
A pointer is a variable which refers to a memory location, which can be an address of another variable or a valid memory address.

A pointer is used to indirectly access the value referred by a variable.

Below are a few usages of pointers :
To modify more than one value inside a function
To conveniently manipulate strings
To manipulate elements stored inside arrays
To store the address of dynamically allocated memory
While creating data structures, such as linked lists, stacks, queues and trees

Operations involved in pointers

The below two operators are exclusively used in connection with pointers:
address of operator (&)
indirection or dereference operator (*)
In the below code:

```
int x = 3;      // Line 1
int y = 5;      // Line 2
int *z, *k;     // Line 3
z = &x;                 // Line 4
```

```
k = &y;                    // Line 5
int sum = *z + *k;  // Line 6
```

In Line 4, the address of operator (&) is used to return the address of the variable x and store it in the pointer variable z.

In Line 6, the indirection operator (*) is used to return the values stored at the addresses referred by the pointer variables z and k.

Below are a few invalid usages of & and * operators:

- &10   // & cannot be used with constants, as they do not have addresses
- &(a + b)   // an expression does not have any address
- register int r; &r;   // it is invalid to access address of register variables
- *1024   // asterisk cannot be used with constants
- int n, p; p = *n;   // Unlike &, asterisk (*) cannot be used on non-pointer variables

Declaration of pointer variable:

The syntax for declaring a pointer variable is given below:

data_type *var1, *var2, ..., *varn;

Where data_type is any valid data type and var1, var2, ..., varn are the names of the pointer variables.

The declaration informs the compiler that var1, var2, ..., varn are used to store the addresses of the variables whose data type matches the current pointer's data type.

The data_type of the pointer defines what type of variables the pointer can point to.

The size of a pointer variable varies depending on the Operating System bit architecture (i.e., 16-bit, 32-bit or 64-bit).

In a 16-bit OS, a pointer variable occupies 2 bytes of memory, 4 bytes in a 32-bit OS and 8 bytes in a 64-bit OS respectively.

Below are a few examples of pointer variable declarations:

int *a; // a is a pointer variable of int type which points to an address of an int value float *f; // f is a pointer variable of float type, which points to an address of a float valuechar *c; // c is a pointer variable of char type, which points to an address of a char value

Example:

```
#include <stdio.h>
void main() {
        int *intPtr, info;
        info = 5;
        intPtr = &info;
        printf("Value in info = %d\n", info);
        printf("Value referred by *intPtr = %d\n", *intPtr);
        *intPtr = *intPtr + 5;
        printf("Value in info = %d\n", info);
        printf("Value referred by *intPtr = %d\n", *intPtr);
}
```

Size of char pointer variable:

Let us consider the below example:

```
char ch = 'A';
char *p;
p = &ch;
printf("Size of a char pointer variable = %zu\n", sizeof(p));
printf("Size of a char variable = %zu\n", sizeof(*p));
```

There is a difference between sizeof(p) and sizeof(*p). The sizeof(p) returns 8 bytes while sizeof (*p) returns 1 byte.

Note: The sizeof operator on any pointer variable always returns the same value. While the sizeof operator on a *pointer_variable (meaning dereferenced pointer variable) returns the size of the data type of the variable pointed to by the pointer_variable.

Parameter passing Methods:

Call by Value:

An argument can be passed to a function in any of the below two ways:

1. By sending the value of the argument
2. By sending the address of the argument

When a function is called by passing values directly, such function calls are referred as call by value.

In a call by value scenario, the value of each actual argument in the calling function is copied into the corresponding formal parameters of the called function.

In such cases, any changes made to the formal parameters in the called function do not reflect back in the calling function.

Example:

```
#include <stdio.h>
void swap(int, int);
void main() {
        int firstNum, secondNum;
        printf("Enter two integer values : ");
        scanf("%d %d", &firstNum, &secondNum);
        printf("Before swapping : firstNum = %d secondNum = %d\n", firstNum, secondNum);
        swap(firstNum, secondNum);
        printf("After swapping : firstNum = %d secondNum = %d\n", firstNum, secondNum);
}
void swap(int firstValue, int secondValue) {
        int temp;
        temp = firstValue;
        firstValue = secondValue;
        secondValue = temp;
        printf("In swap() : firstValue = %d secondValue = %d\n", firstValue, secondValue);
}
```

Call by Reference:

While calling a function, if we pass the address of a variable as argument to a function, it is known as call by address.
In such scenarios, we should declare the parameters in the called function as pointer variables.
The addresses of actual arguments in the calling function are copied into the pointer variables of formal parameters of the called function.
Using the addresses, the called function can access the actual arguments from the formal parameters. Meaning, the code inside the called function can directly access and modify the actual arguments in the calling function.

Example:

```
#include<stdio.h>

#include<conio.h>

void write(int );

void main( )
```

```
{
    int i;
    int a[5]={33, 44, 55, 66, 77};
    clrscr( );
    for (i=0; i<5; i++)
    write (&a[i]);
    getch( );
}
void write(int *n)
{
   printf ("%d\n", *n);

}
```

Dynamic Memory Allocation: (DMA)

In C, memory allocation can happen statically (during complie-time), automatically or dynamically (during program execution or runt-ime).

The memory allocation for all global and static (variables declared using static keyword) is done statically.

The memory for automatic variables is allocated on the stack and is done automatically during function calls.

C provides below 4 the library functions to manually manage dynamically allocated memory:

1. malloc() - used to allocate a specified number of bytes as a block of memory from the free store (which is also referred as heap)
2. calloc() - used to allocate a specified number of bytes from the free store and also initialize them to zero.
3. realloc() - used to increases or decreases the size of the specified block of memory.
4. free() - used to release the specified block of memory to the system.

For static-duration and automatic-duration variables, the size of the allocation must be available at compile-time. However, there may be scenarios where we may want to allocate memory depending on user input during run-time. In such cases dynamic memory allocation (DMA) is required.

malloc() Function:

The malloc() function is used to allocate memory during execution. It takes only one integer argument which represents the number of bytes to be allocated.

The malloc() function returns the base address to the the block of memory allocated on the heap.

The return type of that address is void *, hence in the code we need to typecast that void * type to the primitive data type of the pointer variable.

Below code shows the usage if malloc() syntax:

```
(void *) malloc(number of bytes);
```

Let us consider an example:

```
float *p; // Line 1
p = (float *) malloc (sizeof (float)); // Line 2
```

In the Line 1, the pointer variable p of type float occupies 2 bytes in memory.

In Line 2, the call to malloc() function returns 4 bytes of heap memory.

Since the default return type of malloc() is void *, we are typecasting it to float.

If it is unable to find the requested amount of memory, malloc() function returns NULL. The macro NULL means that the pointer does not refer to a valid object.

The memory which is allocated by malloc() is by default uninitialized, meaning can contain garbage value.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main( )
{
        int *p,i,n;
        clrscr( );
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        p=(int*)malloc(n*sizeof(int));
        printf("\nEnter %d Elements:",n);
        for(i=0;i<n;i++)
                scanf("%d",p+i);
        printf("\nArray Elements Are:");
        for(i=0;i<n;i++)
                printf("%3d",*(p+i));
   getch( );
}
```

OUTPUT:

```
Enter how many numbers:5
Enter 5 Elements: 1 2 3 4 5
Array Elements Are:  1  2  3  4  5
```

calloc() Function:

void *calloc(size_t *num*, size_t *size*);

The calloc( ) function that is declared in the <stdlib.h> header offers a couple of advantages over the malloc( ) function. First, it allocates memory as an array of elements of a given size, and second, it initializes the memory that is allocated so that all bits are zero. The calloc( ) function requires you to supply two argument values, the number of elements in the array, and the size of the array element, both arguments being of type **size_t**. The function still doesn't know the type of the elements in the array so the address of the area that is allocated is returned as type void *.

Here's how you could use calloc( ) to allocate memory for an array of 75 elements of type int:

int *pNumber = (int *) calloc(75, sizeof(int));

The return value will be NULL if it was not possible to allocate the memory requested, so you should still check for this. This is very similar to using malloc( ) but the big plus is that you know the memory area will be initialized to zero.

```
#include<conio.h>
#include<stdlib.h>
void main( )
{
        int *p,i,n;
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        p=(int*)calloc(n,sizeof(int));
        printf("\nEnter %d Elements:",n);
        for(i=0;i<n;i++)
                scanf("%d",p+i);
        printf("\nArray Elements Are:");
        for(i=0;i<n;i++)
           printf("%3d",*(p+i));
   getch( );
}
```

OUTPUT:

```
Enter how many numbers:5
Enter 5 Elements: 1 2 3 4 5
Array Elements Are:  1  2  3  4  5
```

Realloc() Function:

void *realloc(void ****ptr*, size_t** ***size*****);**

The realloc( ) function enables you to reuse memory that you previously allocated using malloc( ) or calloc( ) (or realloc( )).The realloc( ) function expects two argument values to be supplied: a pointer containing an address that was previously returned by a call to malloc( ), calloc( ) or realloc( ), and the size in bytes of the new memory that you want allocated.

The realloc( ) function releases the previously allocated memory referenced by the pointer that you supply as the first argument, then reallocates the same memory area to fulfill the new requirement specified by the second argument. Obviously the value of the second argument should not exceed the number of bytes that was previously allocated. If it is, you will only get a memory area allocated that is equal to the size of the previous memory area.

Note:

I. The new memory block may or may not be begin at the same place as the old one. In case, it is not able to find additional space in the same region, it will create the same in an entirely new region and moves the contents of the old block into the new block.

II. If the function is unsuccessful to allocate the memory space, it returns a NULL pointer and the original block is lost.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main( )
{
        int *p,i,n;
        clrscr( );
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        p=(int*)malloc(n*sizeof(int));
        printf("\nEnter %d Elements:",n);
        for(i=0;i<n;i++)
                scanf("%d",p+i);
        p=(int*)realloc(p,(n+2)*sizeof(int));
        printf("\nEnter %d Elements:",n+2);
        for(i=0;i<n+2;i++)
                scanf("%d",p+i);
        printf("\nArray Elements Are:");
        for(i=0;i<n+2;i++)
                printf("%3d",*(p+i));
   getch( );
}
```

OUTPUT:

```
Enter how many numbers:3
Enter 3 Elements:1 2 3
Enter 5 Elements:4 5 6 7 8
```

```
Array Elements Are:  4  5  6  7  8
```

Free()

void free (void *_ptr_);

When you allocate memory dynamically, you should always release the memory when it is no longer required. Memory that you allocate on the heap will be automatically released when your program ends, but it is better to explicitly release the memory when you are done with it, even if it's just before you exit from the program. In more complicated situations, you can easily have a **memory leak**. A memory leak occurs when you allocate some memory dynamically and you do not retain the reference to it, so you are unable to release the memory. This often occurs within a loop, and because you do not release the memory when it is no longer required, your program consumes more and more of the available memory and eventually may occupy it all.

Of course, to free memory that you have allocated using malloc( ) or calloc( ), you must still be able to use the address that references the block of memory that the function returned. To release the memory for a block of dynamically allocated memory whose address you have stored in the pointer pNumber, you just write the statement**: free (pNumber);**

The free ( ) function has a formal parameter of type void *, and because any pointer type can be automatically converted to this type, you can pass a pointer of any type as the argument to the function. As long as pNumber contains the address that was returned by malloc( ) or calloc( ) when the memory was allocated, the entire block of memory that was allocated will be freed for further use.

If you pass a null pointer to the free( ) function the function does nothing. You should avoid attempting to free the same memory area twice, as the behavior of the free( ) function is undefined in this instance and therefore unpredictable. You are most at risk of trying to free the same memory twice when you have more than one pointer variable that references the memory you have allocated, so take particular care when you are doing this.

<u>Here are some basic guidelines for working with memory that you allocate dynamically:</u>

• Avoid allocating lots of small amounts of memory. Allocating memory on the heap carries some overhead with it, so allocating many small blocks of memory will carry much more overhead than allocating fewer larger blocks.

• Only hang on to the memory as long as you need it. As soon as you are finished with a block of memory on the heap, release the memory.
• Always ensure that you provide for releasing memory that you have allocated. Decide where in you code you will release the memory when you write the code that allocates it.
• Make sure you do not inadvertently overwrite the address of memory you have allocated on the heap before you have released it; otherwise your program will have a memory leak. You need to be especially careful when allocating memory within a loop.

```
int main() {
        int *p;
        p = (int *) malloc (sizeof (int));
        if (p == NULL) { /* failed to reserve memory */
                printf ("Failed to allocate space for %d bytes", sizeof (int));
                return 1;
        }
        *P = 15;
        printf ("%d\n", *p);
        free(p); /* memory is released */
        return 0;
}
```

Pointers with one dimensional Arrays:

An array variable name itself represents a pointer to the first element of that array.

In the statement int num[10];, the address of the first array element can be expressed as either &num[0] or num. Similarly, the address of the second array element can be written as &num[1] or as num +1, and so on.

In general, the address of the ith array element can be expressed as either &num[i] or (num +i).

Thus, the address of an array element can be expressed in two ways: either by writing the actual array element proceeded by the ampersand or by writing an expression in which the subscript is added to the array variable name.

The expressions &num[i] and (num + i) both represent the address of the ith element of num, and num[i] and *(num +i) both represent the value at that address.

The address of the given array element can be find by using the formula:

Address of num[i] = base address + i * scale factor // scale factor is the size of the data type used to create an array

By using heap memory, A programmer can create a new one-dimensional array at the run time and it can be accessed by using the pointers.

To create heap memory, programmer can use either malloc() or calloc().

Example:

```
#include <stdio.h>
#include <stdlib.h>
void main() {
        int *p, n, i;
        printf("Enter n value : ");
        scanf("%d", &n);
        p = (int *) malloc(n * sizeof(int));
        printf("Enter %d values : ", n);
        for (i = 0; i < n; i++) {
                scanf("%d", p + i);
        }
        printf("The given array elements are : ");
        for (i = 0; i < n; i++) {
                printf("%d ", *(p + i));
        }
}
```

Pointers with two dimensional Arrays:

We know that an array variable name itself represents a pointer to the first element of that array.

In the statement int num[5][5];, the address of the first row first column element can be expressed as either &num[0][0] or num. Similarly, the address of the first row second column element can be written as &num[0][1] or (num + 0 * 5 + 1), and so on.

The address of the given array element can be find by using the formula:

Address of num[i][j] = base address + i * total number of columns * scale factor + j * scale factor

// scale factor is the size of the data type of the array

By using heap memory, A programmer can create a new two-dimensional array at the run time and it can be accessed by using the pointers.

To create heap memory, programmer can use either malloc() or calloc().

Example:

```
#include <stdio.h>
#include <stdlib.h>
void main() {
        int *p, m, n, i, j;
        printf("Enter row and column size : ");
        scanf("%d %d", &m, &n);
        p = (int *) calloc(m * n, sizeof(int));
        printf("Enter %d matrix elements : ", m * n);
        for (i = 0; i < m; i++) {
                for (j = 0; j < n; j++) {
                        scanf("%d", p + i * n + j);
                }
        }
        printf("The given matrix is\n");
        for (i = 0; i < m; i++) {
                for (j = 0; j < n; j++) {
                        printf("%d ", *(p + i * n + j));
                }
                printf("\n");
        }
}
```

Array of Pointers:

Whenever addresses are stored as array elements, such an array is called as array of pointers.

The format for declaration of an array of pointers is
data_type *array_name [size];
        (or)
data_type *(array_name [size]);
The following declaration creates an array named ptr_array containing 5 elements, each of which is of the type int *.

int *ptr_array[5];

The above declaration creates an array of pointers with size 5, named as ptr_array. In each location user needs to store the addresses of the another variables of the same data type int.

Example:

```
#include <stdio.h>
#include <stdlib.h>
void main() {
        float *a[5], f1, f2;
        f1 = 10.24;
        f2 = 12.54;
        a[0] = &f1;
        a[1] = &f2;
        printf("First value = %f\n", *(a[0]));
        printf("Second value = %f\n", *(a[1]));
}
```

2D Array as Array of Pointers:

A multidimensional array can be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays.

In such cases, the newly defined array will have one less dimension than the original array. Each pointer will indicates the beginning of a separate (n - 1)th dimensional array.

In general, a two-dimensional array can be defined as a one-dimensional array of pointers as Data_type *array_name [size1]; rather than Data_type array_name[size1][size2];.

Similarly, an n-dimensional array can be defined as an (n - 1)th dimensional array of pointers as Data_type *array_name[size1] [size2] ---------- [sizen-1]; instead of Data_type array_name[size1] [size2][size3]-----------[sizen];.

Here the preceding asterisk (*) establishes that the array will contain pointers.
Consider the following example:

```
int *num[10];
```

Here num[0] points to the beginning of the first row, num[1] points to the beginning of the second row, and so on. In this kind of definition, the number of elements in each row is specified at the run time.

An individual array element, such as array[2][4] can be accessed by *(array[2] + 4). In this expression, array[2] is a pointer to the first element in row 2, so that (ary[2] + 4) points to element 4 (actually the fifth element).

The heap memory is allocated individually for each pointer of the array in array of pointers.

```
#include <stdio.h>
#include <stdlib.h>
void main() {
        int *p[10], n, m, i, j;
        printf("Enter row and column sizes : ");
        scanf("%d %d", &m, &n);
        for (i = 0; i < m; i++) {
                p[i] = (int *) malloc(n * sizeof(int));
        }
        printf("Enter %d elements : ", m * n);
        for (i = 0; i < m; i++) {
                for (j = 0; j < n; j++) {
                        scanf("%d", p[i] + j);
                }
        }
        printf("The given matrix is\n");
        for (i = 0; i < m; i++) {
                for (j = 0; j < n; j++) {
                        printf("%d ", *(p[i] + j));
                }
                printf("\n");
        }
}
```

Pointer to an Array:

we can use a pointer to point to an array, and then we can use that pointer to access the array elements. Lets have an example,

```
#include<stdio.h>
int main()
{
int i;
int a[5] = {1,2,3,4,5}
int *p =a;
for(i=0;i<5;i++)
{
printf("%d", *p);
p++;
}
return 0;
}
```

In the above program, the pointer *p will print all the values stored in the array one by one. We can also use the Base address (a in above case) to act as a pointer and print all the values.

Character Pointers:
A character pointer variable can be declared as char *ptr_var;.

User can initialize a character pointer with the string directly without mentioning the size of the character array, like char *message = "It is CodeTantra";.

In the above initialization the string literal is used to create a little block of characters somewhere in memory (anonymous location in the form of array) which the pointer message is initialized to point to.

User may reassign message to point somewhere else, but as long as it points to the string literal, we can't modify the characters it points to.

Example:

```
#include <stdio.h>
void main() {
        char *ch = "I love coding";
        printf("The given string = %s\n", ch);
        ch = "Learn coding from CodeTantra";
        printf("The given string = %s\n", ch);
}
```

Example:

```
#include <stdio.h>
void main() {
        char *ch1 = "I love coding";
        char ch2[20];
        int i = 0;
        printf("The given string = %s\n", ch1);
        while(*ch1 != '\0') {
                ch2[i++] = *ch1++;
        }
        ch2[i] = '\0';
        printf("The copied string = %s\n", ch2);
}
```

Multiple Indirections or Pointer to Pointer

It is possible to have a pointer point to another pointer variable that points to the normal variable. This is called multiple indirection or pointers to pointers.

A single pointer variable can hold the address of a normal variable which contains the value.

In case of a pointer to a pointer, the double pointer contains the address of the single pointer variable which points to the normal variable that contains the desired value.

Multiple indirections can be carried on to whatever extent desired, but more than a pointer to a pointer is rarely needed. In fact, excessive indirection is difficult to follow and difficult to identify the errors.

A double pointer can be declared by placing an additional asterisk (*) in front of the single pointer variable.

The general format for double pointer variable is:

```
data_type **ptr_var;
```

Let us consider the below example:

```
float **a;
float *b;
float c = 2.5;
b = &c; // b is a single pointer which holds the address of a normal variable
a = &b; // a is a double pointer which holds the address of a single pointer variable
```

Example:

```
#include <stdio.h>
void main() {
        int **a, *b, c = 22;
        b = &c;
        a = &b;
        printf("Accessing through double pointer the given value = %d\n", **a);
        printf("Accessing through single pointer the given value = %d\n", *b);
```

```
    printf("The given value = %d\n", c);
}
```

Challenges in Pointers:

The main challenges while using pointers are:
Dangling pointers
Memory leakages
Dangling pointer is a pointer which points to a memory that has been freed or deallocated. Such a pointer still points to the memory location of the deallocated memory, accessing such memory might cause segmentation fault error.

In simple terms, a pointer pointing to a non-existing memory location is called a dangling pointer.

Let us consider an example:

```
void main() {
    int *p = (int *) malloc(sizeof(int)); // allocating heap memory
    ....
    free(p); // after deallocating the heap memory, *p now becomes a dangling pointer
    *p = 12; // trying to access non-existing memory location can result in error
}
```

Let us consider another example for dangling pointer:

```
void main() {
    int *q;
    int *p = (int *) malloc(sizeof(int)); // allocating heap memory
    q = p; // store the address containing in p to q i.e., p and q are referring the same heap
memory
    free(q); // after deallocating the heap memory through q
    *p = 12; // trying to access non-existing memory location through p which is also a
dangling pointer, can result in an error
}
```

Memory leak occurs when programmers create a memory in heap and forget to delete it. Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.
A memory leak reduces the performance of the computer by reducing the amount of available memory.

A memory leak is that memory which hasn't been freed and there is usually no way to free it anymore. This can result when a pointer which was the only reference to a memory location (that was dynamically allocated and not freed) now points somewhere else.

Let us consider an example:

```
void main() {
        int *p = (int *) malloc(sizeof(int)); // allocating heap memory
        *p = 12;
        printf("The given value = %d\n". *p);
        *p = (int *) malloc( 3 * sizeof(int)); // allocating a new memory, the previously allocated
memory results in a memory leak
}
```

Dangling Pointer:

Dangling pointer is a pointer which points to a memory that has been freed or deallocated. Such a pointer still points to the memory location of the deallocated memory, accessing such memory might cause segmentation fault error.

In simple terms, a pointer pointing to a non-existing memory location is called a dangling pointer.

Let us consider an example:

```
void main() {
        int *p = (int *) malloc(sizeof(int)); // allocating heap memory
        ....
        free(p); // after deallocating the heap memory, *p now becomes a dangling pointer
        *p = 12; // trying to access non-existing memory location can result in error
}
```

Let us consider another example for dangling pointer:

```
void main() {
        int *q;
        int *p = (int *) malloc(sizeof(int)); // allocating heap memory
        q = p; // store the address containing in p to q i.e., p and q are referring the same heap
memory
        free(q); // after deallocating the heap memory through q
        *p = 12; // trying to access non-existing memory location through p which is also a
dangling pointer, can result in an err
```

Command Line Arguments:

In environments that support C, there is a way to pass command-line arguments or parameters to a program when it begins executing. When main is called, it is called with two arguments. The first (conventionally called **argc**, for argument count) is the number of command-line arguments the program was invoked with; the second (**argv**, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string. We customarily use multiple levels of pointers to manipulate these character strings.

Note that **argv[0]* is the name of the program invoked, which means that **argv[1]* is a pointer to the first argument supplied, and **argv[n]* is the last argument. If no arguments are supplied, *argc* will be one. Thus for n arguments, *argc* will be equal to n + 1. The program is called by the command line.

Example:

```
#include<stdio.h>
#include<conio.h>
main(int argc, char * argv[])
{
            int i;
            clrscr();
            printf("Number of arguments: %d\n",argc);
            printf("Name of the program :%s\n",argv[0]);
            for(i=1;i<argc;i++)
            printf("user value no %d is : %s\n",i,argv[i]);
}
```

The program is to be run at command line. The above program is executed using following steps:

a) compile the program
b) makes its exe file
c) switch to the command prompt(C:\TC>)
d) make sure that the exe file is available in the current directory.
e) type the following

```
C:\TC\cmd Farooq c faculty
Number of arguments: 4
Name of the program :C:\TC\CMD.EXE
```

```
user value no 1 is : Farooq
user value no 2 is : c
user value no 3 is : faculty
```

Example: Addition of given numbers using Command Line Arguments

```
#include<string.h>
main(int argc,char *argv[])
{
                int x,y;
                clrscr();
                x=atoi(argv[1]);
                y=atoi(argv[2]);
                printf("\nNumber of arguments:%d",argc);
                printf("\nAddition Result:%d",x+y);
}
```

Array Techniques:

Array Order Reversal:

```
#include <stdio.h>
int main() {
  int array[100], n, c, t, end;
  printf("Enter the number of elements");
  scanf("%d", &n);
  printf("The array elements are");
  end = n - 1;
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);

  for (c = 0; c < n/2; c++)
  {
    t          = array[c];
    array[c]   = array[end];
    array[end] = t;
    end--;
  }

  printf("Reversed array elements are:\n");

  for (c = 0; c < n; c++)
    printf("%d\n", array[c]);

  return 0;
}
```

Finding the maximum number in a set:

```
#include <stdio.h>
int main() {
    int i, n;
    float arr[100];
    printf("Enter the number of elements (1 to 100): ");
    scanf("%d", &n);
    for (i = 0; i < n; ++i) {
        printf("Enter number%d: ", i + 1);
        scanf("%f", &arr[i]);
    }
    // storing the largest number to arr[0]
    for (i = 1; i < n; ++i) {
        if (arr[0] < arr[i])
            arr[0] = arr[i];
    }
    printf("Largest element = %.2f", arr[0]);
    return 0;
}
```

Output:

```
Enter the number of elements (1 to 100): 5
Enter number1: 5
Enter number2: 6
Enter number3: 7
Enter number4: 89
Enter number5: 5
Largest element = 89.00
```

Removal of duplicates from an ordered Array:

```
#include<stdio.h>
#include<conio.h>
int main()
{
   int a[20], i, j, k, n;
   printf("\nEnter array size: ");
   scanf("%d", &n);

   printf("\nEnter %d array element: ", n);
   for(i = 0; i < n; i++)
   {
      scanf("%d", &a[i]);
   }

   printf("\nOriginal array is: ");
   for(i = 0; i < n; i++)
   {
      printf(" %d", a[i]);
   }

   printf("\nNew array is: ");
   for(i = 0; i < n; i++)
   {
      for(j = i+1; j < n; )
      {
         if(a[j] == a[i])
         {
            for(k = j; k < n; k++)
            {
               a[k] = a[k+1];
            }
            n--;
         }
```

```
            else
            {
               j++;
            }
        }
    }

    for(i = 0; i < n; i++)
    {
       printf("%d ", a[i]);
    }
    getch();
}
```

Output:

```
Enter array size: 6

Enter 6 array element: 1
2
2
3
5
5

Original array is:  1 2 2 3 5 5
New array is: 1 2 3 5
```

Finding the Kth Smallest Element:

```
#include<stdio.h>
#include<stdlib.h>
```

```
void main() {
        int a[10], i, j, n, index = 0, ele, n1;
        printf("Enter how many values you want to read : ");
        scanf("%d", &n);
        for(i = 0; i < n; i++) {
        printf("Enter the value of a[%d] : ", i);
        scanf("%d", &a[i]);
        }
        printf("Enter which smallest element you want: ");
        scanf("%d",&ele);
        for(i=0;i < n;i++)      {
                for (int j = 0; j < n; j++) {
                        if (a[i] < a[j]) {
                                int temp = a[j];
                                a[j] = a[i];
                                a[i] = temp;
                        }
                }
        }
                printf("%d is the %dth smallest element",a[ele], ele);
}
```

Output:

```
Enter how many values you want to read : 5
Enter the value of a[0] : 3
Enter the value of a[1] : 2
Enter the value of a[2] : 1
Enter the value of a[3] : 5
Enter the value of a[4] : 4
Enter which smallest element you want: 2
3 is the 2th smallest element
```