Chapter 2

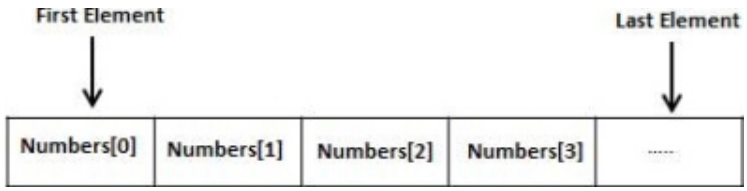
ARRAY

Definition of Array- An array is defined as **finite ordered** collection of **homogenous** data elements which are stored in contiguous memory locations.

Here the words,
finite means data range must be defined.
ordered means data must be stored in continuous memory addresses.
homogenous means data must be of similar data type.

There are two types of Array:
1. Single or One Dimensional Array
2. Multi Dimensional Array

Single or One Dimensional array: A list of items can be given one variable name using only one subscript and such a variable is called single sub-scripted variable or one or Single dimensional array.



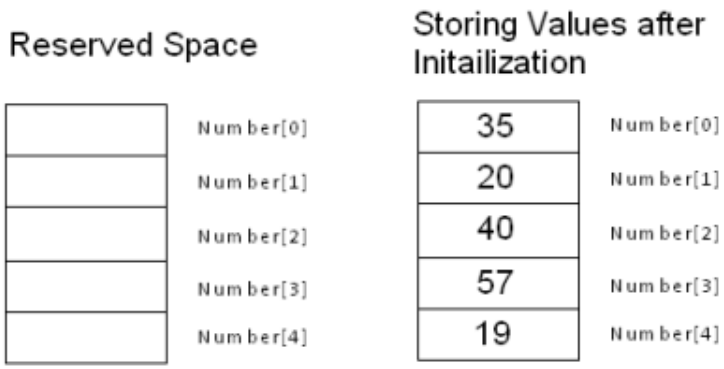
Declaration of One Dimensional array: Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in the memory. The syntax form of array declaration is:
type variable-name[size];

Ex -
```
float height[50];
int group[10];
char name[10];
```

The type specifies the type of the element that will be contained in the array, such as int, float, or char etc. Variable-name specifies the name of array such as height, group and name. The size indicates the maximum number of elements that can be stored inside the array. C programming language also treats character strings simply as arrays of characters.

Now declare an array for five elements
int number[5];

Then the computer reserves five storage locations as the size of the array as shown below –



Initialization of Single or One Dimentional Array: After an array is declared, it's elements must be initialized. In C programming an array can be initialized at either of the following stages:
- At compile time
- At run time

Compile Time initialization: Array can be initialized when it is declared. The general form of initialization of array is:
type array-name[size] = {list of values };

The values in the list are separated by commas. For example
int number[3] = {0,5,4 };

The above statement will declare the variable "number"as an array of size 3 and will assign the values to each element. If the number of values in the list are less than the number of elements, then only that many elements will be initialized. The remaining elements values will be set to zero automatically.

Remember, if we have more initializers than the declared size, the compiler will produce an error.

Run time Initialization: An array can also be explicitly initialized at run time. For example consider the following segment of a C program.

```
 for(i=0;i<10;i++)
    {
        scanf(" %d ", &x[i] );
    }
```

Above example will initialize array elements with the values entered through the keyboard. In the run time initialization of the arrays, looping statements are almost compulsory. Looping statements are used to initialize the values of the arrays one by one by using assignment operator or through the keyboard by the user.

Sample One Dimensional Array Program:

/* Simple C program to store the elements in the array and to print them from the array */

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

{

    int array[5],i;

      printf("Enter 5 numbers to store them in array \n");

    for(i=0;i<5;i++)

    {
        scanf("%d",&array[i]);

    }

        printf("Element in the array are - \n \n");

    for(i=0;i<5;i++)

    {
        printf("Element stored at a[%d] = %d \n",i,array[i]);

    }
        getch();

}
```

Input – Enter 5 numbers to store them in array – 23 45 32 25 45
Output – Element in the array are –
Element stored at a[0]-23
Element stored at a[1]-45
Element stored at a[2]-32
Element stored at a[3]-25
Element stored at a[4]-45

Multi Dimensional Array:

Array of an array known as multidimensional array.The general form of a multidimensional array declaration –

type name[size1][size2]...[sizeN];

The simplest form of multidimensional array is the two-dimensional array.Example

int x[3][4];

Here, x is a two-dimensional (2d) array and can hold 12 elements. You can think the array as table with 3 row and each row has 4 column.

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Initialization of Two Dimensional(2D)Array: Like the one dimensional array, 2D arrays can be initialized in two ways; the compile time initialization and the run time initialization.

Compile Time initialization – We can initialize the elements of the 2D array in the same way as the ordinary variables are declared. The best form to initialize 2D array is by using the matrix form. Syntax is as below –

```
int table[2][3] = {
                    {0, 2, 5}
                    {1, 3, 0}
                };
```

Run Time initialization – As in the initialization of 1D array we used the looping statements to set the values of the array one by one. In the similar way 2D array are initialized by using the looping structure. To initialize the 2D array by this way, the nested loop structure will be used; outer for loop for the rows (first sub-script) and the inner for loop for the columns (second sub-script) of the 2D array. Below is the looping section to initialize the 2D array by using the run time initialization method –

```
for(i=0;i<3;i++)
   {
       for(j=0;j<3;j++)
       {
           scanf("%d",&ar1[i][j]);
       }
   }
```

Sample 2D array Program:

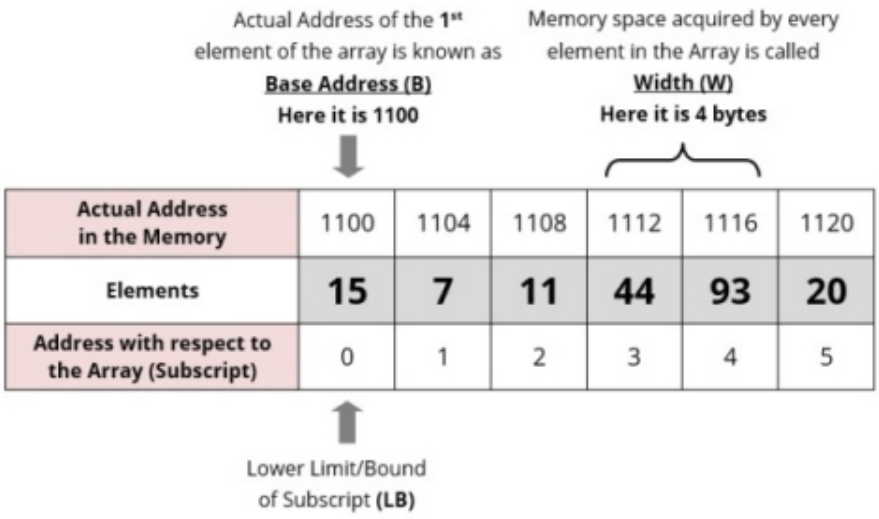
```
/* Sample 2-D array C program */

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

```
{
    int array[3][3],i,j,count=0;

     /* Run time Initialization */
   for(i=1;i<=3;i++)

    {
          for(j=1;j<=3;j++)
      {
          count++;
          array[i][j]=count;
          printf("%d \t",array[i][j]);
       }
        printf("\n");
    }

    getch();

}
```

Output –

```
1    2    3
4    5    6
7    8    9
```

Address Calculation in Single (One) Dimentional Array:



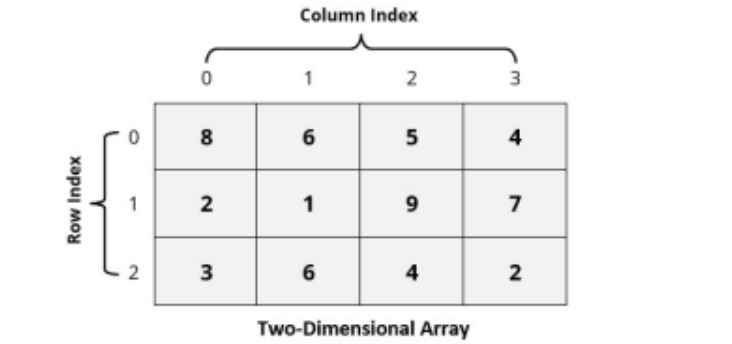
Array of an element of an array say "A[I]" is calculated using the following formula:

Address of A [I] = B + W * (I – LB)
Where,
B = Base address
W = Storage Size of one element stored in the array (in byte)
I = Subscript of element whose address is to be calculate
LB = Lower limit / Lower Bound of subscript, if not specified assume 0 (zero)

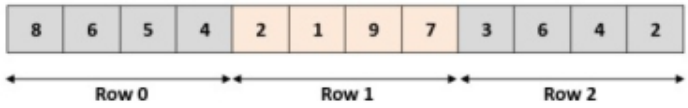
Example:
Given the base address of an array B[1300…..1900] as 1020 and size of each element is 2 bytes in the memory. Find the address of B[1700].

Solution:
The given values are: B = 1020, LB = 1300, W = 2, I = 1700
Address of A [I] = B + W * (I – LB)
= 1020 + 2 * (1700 – 1300)
= 1020 + 2 * 400
= 1020 + 800
= 1820 [Ans]

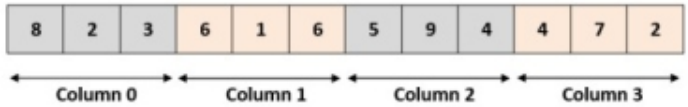
Address Calculation in Multi (Two) Dimensional Array:
While storing the elements of a 2-D array in memory, these allocations are contiguous memory locations. Therefore, a 2-D array must be linearized their storage. There are two alternatives to achieve linearization: Row-Major and Column-Major.



Row-Major (Row Wise Arrangement)



Column-Major (Column Wise Arrangement)



Address of an element of any array say "A[I][J]" can be calculated by two types as given below:
(a) Row Major System
(b) Column Major System

Row Major System:
The address of a location in Row Major System is calculated using the following formula:
Address of A [I][J] = B + W * [N * (I – Lr) + (J – Lc)]

Column Major System:
The address of a location in Column Major System is calculated using the following formula:
Address of A [I][J] Column Major Wise = B + W * [(I – Lr) + M * (J – Lc)]

Where,
B = Base address
I = Row subscript of element whose address is to be calculate
J = Column subscript of element whose address is to be calculate
W = Storage Size of one element stored in the array (in byte)
Lr = Lower limit of row/start row index of matrix, if not given assume 0 (zero)
Lc = Lower limit of column/start column index of matrix, if not given assume 0 (zero)
M = Number of row of the given matrix
N = Number of column of the given matrix

Note: Usually number of rows and columns of a matrix are given (like A[20][30] or A[40][60]) but if it is given as A[Lr- – – – – Ur, Lc- – – – – Uc]. In this case number of rows and columns are calculated using the following methods:
Number of rows (M) will be calculated as = (Ur – Lr) + 1
Number of columns (N) will be calculated as = (Uc – Lc) + 1

And rest of the process will remain same as per requirement (Row Major Wise or Column Major Wise).

Examples:
An array X [-15…….10, 15……40] requires one byte of storage. If beginning location is 1500 determine the location of X [15][20].

Solution:
As you see here the number of rows and columns are not given in the question. So they are calculated as:
Number or rows say M = (Ur – Lr) + 1 = [10 – (- 15)] +1 = 26
Number or columns say N = (Uc – Lc) + 1 = [40 – 15)] +1 = 26

(i) Column Major Wise Calculation of above equation

The given values are: B = 1500, W = 1 byte, I = 15, J = 20, Lr = -15, Lc = 15, M = 26

Address of A [I][J] = B + W * [(I – Lr) + M * (J – Lc)]
= 1500 + 1 * [(15 – (-15)) + 26 * (20 – 15)] = 1500 + 1 * [30 + 26 * 5] = 1500 + 1 * [160]

= 1660 [Ans]

(ii) Row Major Wise Calculation of above equation

The given values are: B = 1500, W = 1 byte, I = 15, J = 20, Lr = -15, Lc = 15, N = 26

Address of A [I][J] = B + W * [N * (I – Lr) + (J – Lc)]
= 1500 + 1* [26 * (15 – (-15))) + (20 – 15)] = 1500 + 1 * [26 * 30 + 5] = 1500 + 1 * [780 + 5] = 1500 + 785
= 2285 [Ans]

Basic Operations on Array: Follwing operations can be performed on array
(a) Traverse – access all the array elements one by one.
(b) Insertion – Adds an element at the given index.
(c) Deletion – Deletes an element at the given index.
(d) Search – Searches an element using the given index or by the value.
(e) Update – Updates an element at the given index.

Traverse: Traversing means accessing the each and every element of array exactly once. Following is the algorithm for traversing a linear array

Here A is a linear array with lower bound LB and upper bound UB. This algorithm traverses array A and applies the operation PROCESS to each element of the array.
1. Repeat For I = LB to UB
2. Apply PROCESS to A[I]
   [End of For Loop]
3. Exit

Insertion: Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array. Following is the algorithm for Insertion an element in to a linear array.

Algorithm: Let LA be a Linear Array (unordered) with N elements and K is a positive integer such that K<=N. Following is the algorithm where ITEM is inserted into the Kth position of LA –
1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop

C Program for Insertion:

```
#include <stdio.h>

main() {
```

```
  int LA[] = {1,3,5,7,8};
  int item = 10, k = 3, n = 5;
  int i = 0, j = n;

  printf("The original array elements are :\n");

  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }

  n = n + 1;

  while( j >= k) {
    LA[j+1] = LA[j];
    j = j - 1;
  }

  LA[k] = item;

        printf("The array elements after insertion :\n");

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

When we compile and execute the above program, it produces the following result –

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8

The array elements after insertion :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8
```

Deletion: Deletion refers to removing an existing element from the array and re-organizing all elements of an array. Following is the algorithm for Deletion an element from a linear array.

Consider LA is a linear array with N elements and K is a positive integer such that K<=N.

Following is the algorithm to delete an element available at the Kth position of LA.

1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J-1] = LA[J]
5. Set J = J+1
6. Set N = N-1
7. Stop

C Program for Deletion:

```
#include <stdio.h>
main() {
  int LA[] = {1,3,5,7,8};
  int k = 3, n = 5;
  int i, j;

  printf("The original array elements are :\n");

  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }

  j = k;

  while( j < n) {
    LA[j-1] = LA[j];
    j = j + 1;
  }

  n = n -1;

  printf("The array elements after deletion :\n");

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

When we compile and execute the above program, it produces the following result –
Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :
```

LA[0] = 1
LA[1] = 3
LA[2] = 7
LA[3] = 8

Search: Searching refers the finding out an element using the given index or by the value. There are two types of searching in linear array
1.Linear Search
2. Binary Search

Linear Search:
A linear search is the basic and simple search algorithm. A linear search searches an element or value from an array till the desired element or value is not found. It searches in a sequence order. It compares the element with all the other elements given in the list and if the element is matched it returns the value index else it return -1. Linear Search is applied on the unsorted or unordered list when there are fewer elements in a list.

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to find an element with a value of ITEM using sequential search
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop

C Program for Searching:

```
#include <stdio.h>
 Void main() {
   int LA[] = {1,3,5,7,8};
   int item = 5, n = 5;
   int i = 0, j = 0;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   while( j < n){
      if( LA[j] == item ) {
         break;
      }

      j = j + 1;
   }
```

```
   printf("Found element %d at position %d\n", item, j+1);
}
```

When we compile and execute the above program, it produces the following result –
Output

```
The original array elements are:
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
Found element 5 at position 3
```

Binary Search:

Binary Search is applied on the sorted array or list. In binary search, we first compare the value with the elements in the middle position of the array. If the value is matched, then we return the value. If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array. We repeat this procedure on the lower (or upper) half of the array. Binary Search is useful when there are large numbers of elements in an array.

We basically ignore half of the elements just after one comparison.

Compare x with the middle element.
If x matches with middle element, we return the mid index.
Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
Else (x is smaller) recur for the left half.

C Prgram for Binary Search:

```
#include <stdio.h>
#include<conio.h>
#define MAX 20

// array of items on which linear search will be conducted.
int intArray[MAX] = {1,2,3,4,6,7,9,11,12,14,15,16,17,19,33,34,43,45,55,66};

void printline(int count) {
   int i;

   for(i = 0;i <count-1;i++) {
      printf("=");
   }

   printf("=\n");
}
```

```
int find(int data) {
   int lowerBound = 0;
   int upperBound = MAX -1;
   int midPoint = -1;
   int comparisons = 0;
   int index = -1;

   while(lowerBound <= upperBound) {
      printf("Comparison %d\n" , (comparisons +1) );
      printf("lowerBound : %d, intArray[%d] = %d\n",lowerBound,lowerBound,
         intArray[lowerBound]);
      printf("upperBound : %d, intArray[%d] = %d\n",upperBound,upperBound,
         intArray[upperBound]);
      comparisons++;

      // compute the mid point
      // midPoint = (lowerBound + upperBound) / 2;
      midPoint = lowerBound + (upperBound - lowerBound) / 2;

      // data found
      if(intArray[midPoint] == data) {
         index = midPoint;
         break;
      }else {
         // if data is larger
         if(intArray[midPoint] < data) {
            // data is in upper half
            lowerBound = midPoint + 1;
         }
         // data is smaller
         else {
            // data is in lower half
            upperBound = midPoint -1;
         }
      }
   }
   printf("Total comparisons made: %d" , comparisons);
   return index;
}

void display() {
   int i;
   printf("[");

   // navigate through all items
   for(i = 0;i<MAX;i++) {
      printf("%d ",intArray[i]);
   }
```

```
  printf("]\n");
}

main() {
  printf("Input Array: ");
  display();
  printline(50);

  //find location of 1
  int location = find(55);

  // if element was found
  if(location != -1)
    printf("\nElement found at location: %d" ,(location+1));
  else
    printf("\nElement not found.");
}
```

If we compile and run the above program then it would produce following result –
Output

```
Input Array: [1 2 3 4 6 7 9 11 12 14 15 16 17 19 33 34 43 45 55 66 ]

Comparison 1
lowerBound : 0, intArray[0] = 1
upperBound : 19, intArray[19] = 66
Comparison 2
lowerBound : 10, intArray[10] = 15
upperBound : 19, intArray[19] = 66
Comparison 3
lowerBound : 15, intArray[15] = 34
upperBound : 19, intArray[19] = 66
Comparison 4
lowerBound : 18, intArray[18] = 55
upperBound : 19, intArray[19] = 66
Total comparisons made: 4
Element found at location: 19
```

Update: Update operation refers to updating an existing element from the array at given index.

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to update an element available at the Kth position of LA.

1. Start
2. Set LA[K-1] = ITEM
3. Stop

C Program for Updation:

```
#include <stdio.h>
#include<conio.h>
main() {
  int LA[] = {1,3,5,7,8};
  int k = 3, n = 5, item = 10;
  int i, j;

  printf("The original array elements are :\n");

  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }

  LA[k-1] = item;

  printf("The array elements after updation :\n");

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

When we compile and execute the above program, it produces the following result –
Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after updation :
LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8
```

Character String in C: Strings are actually one-dimensional array of characters terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.

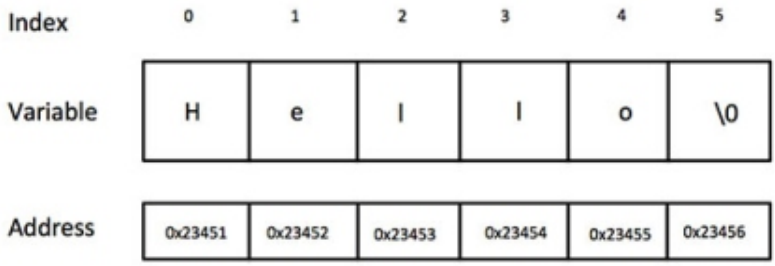
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows −

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above defined string in C/C++ −



Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string −

```
#include <stdio.h>

int main () {

  char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
  printf("Greeting message: %s\n", greeting );
  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Greeting message: Hello
```

C supports a wide range of functions that manipulate null-terminated strings −

S.N.	Function	Purpose
1	strcpy(s1, s2);	Copies string s2 into string s1.
2	strcat(s1, s2);	Concatenates string s2 onto the end of string s1.
3	strlen(s1);	Returns the length of string s1.
4	strcmp(s1, s2);	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	strchr(s1, ch);	Returns a pointer to the first occurrence of character ch in string s1.
6	strstr(s1, s2);	Returns a pointer to the first occurrence of string s2 in string s1.

The following example uses some of the above-mentioned functions −

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

int main () {
```

```
   char str1[12] = "Hello";
   char str2[12] = "World";
   char str3[12];
   int  len ;

   /* copy str1 into str3 */
   strcpy(str3, str1);
   printf("strcpy( str3, str1) :  %s\n", str3 );

   /* concatenates str1 and str2 */
   strcat( str1, str2);
   printf("strcat( str1, str2):   %s\n", str1 );

   /* total length of str1 after concatenation */
   len = strlen(str1);
   printf("strlen(str1) :  %d\n", len );

   return 0;
}
```

When the above code is compiled and executed, it produces the following result−

```
strcpy( str3, str1) :  Hello
strcat( str1, str2):   HelloWorld
strlen(str1) :  10
```

Static and Dynamic Memory Allocation: Dynamic memory allocation is at runtime. Static memory allocation is before run time, but the values of variables may be change at run time.

Static memory allocation saves running time, but can't be possible in all cases.
Dynamic memory allocation stores it's memory on heap, and the static memory allocation stores it's data in the “data segment” of the memory.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
 //static allocation example using integer array.
  int arr[5]; /* static memory allocation memory allocated before execution, the size of array
should be initialized*/
 for ( int j = 0; j<5; j++) //Waste of memory can be occured.
 {
 printf("Enter number for Static Array %d: " ,j);
 scanf("%d", &arr[j]);
 }
 printf("\nThe Static Array is: \n");
 for ( int j = 0; j<5; j++)
 {
 printf("The value of %d is %d\n", j, arr[j]);
```

```
}

//dynamic allocation example using integer array
 int* array;
 int n, i;
 printf("\n----------\n\nDynamic Allocation\n");
 printf("Enter the number of elements: ");
 scanf("%d", &n);
  array = (int*) malloc(n*sizeof(int)); //memory is allocated during the execution of the
program
 //Less Memory space required.
 for (i=0; i<n; i++) {
 printf("Enter number %d: ", i);
 scanf("%d", &array[i]);
 }

printf("\nThe Dynamic Array is: \n");

for (i=0; i<n; i++) {
 printf("The value of %d is %d\n", i, array[i]);
 }
 printf("Size= %d\n", i);

system("PAUSE");
 return 0;
}
```

Memory Allocation Functions:

Programming language provides several functions for memory allocation and management. These functions can be found in the <stdlib.h> header file.

S. No.	Function & Description
1	**void *calloc(int num, int size);** This function allocates an array of num elements each of which size in bytes .
2	**void free(void *address);** This function releases a block of memory specified by address.
3	**void *malloc(int num);** This function allocates an array of num bytes and leave them uninitialized.
4	**void *realloc(void *address, int newsize);** This function re-allocates memory extending it upto newsize.

Following are examples of dynamic memory allocation using functions:

1.

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

int main() {

  char name[100];
  char *description;

  strcpy(name, "Zara Ali");

  /* allocate memory dynamically */
  description = malloc( 200 * sizeof(char) );

  if( description == NULL ) {
     fprintf(stderr, "Error - unable to allocate required memory\n");
  }
  else {
     strcpy( description, "Zara ali a DPS student in class 10th");
  }

  printf("Name = %s\n", name );
  printf("Description: %s\n", description );
}
```

When the above code is compiled and executed, it produces the following result.

```
Name = Zara Ali
Description: Zara ali a DPS student in class 10th
```

Same program can be written using calloc(); only thing is you need to replace malloc with calloc as follows −

```
calloc(200, sizeof(char));
```

So you have complete control and you can pass any size value while allocating memory, unlike arrays where once the size defined, you cannot change it.

2.

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

int main() {

  char name[100];
```

```
   char *description;

   strcpy(name, "Angad");

   /* allocate memory dynamically */
   description = malloc( 30 * sizeof(char) );

   if( description == NULL ) {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   }
   else {
      strcpy( description, "Angad is a cute Boy.");
   }

   /* suppose you want to store bigger description */
   description = realloc( description, 100 * sizeof(char) );

   if( description == NULL ) {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   }
   else {
      strcat( description, "He is in 1st Class");
   }

   printf("Name = %s\n", name );
   printf("Description: %s\n", description );

   /* release memory using free() function */
   free(description);
}
```

When the above code is compiled and executed, it produces the following result.

Name = Angad
Description: Angad is a cute Boy.He is in 1st Class.

Pointers in 'C': A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is −
type *var-name;

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. Take a look at some of the valid pointer declarations −

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
```

```
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

The following example shows use of pointer variable −

```
#include <stdio.h>

int main () {

  int  var = 20;   /* actual variable declaration */
  int  *ip;        /* pointer variable declaration */

  ip = &var;  /* store address of var in pointer variable*/

  printf("Address of var variable: %x\n", &var  );

  /* address stored in pointer variable */
  printf("Address stored in ip variable: %x\n", ip );

  /* access the value using the pointer */
  printf("Value of *ip variable: %d\n", *ip );

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program −

```
#include <stdio.h>

int main () {
```

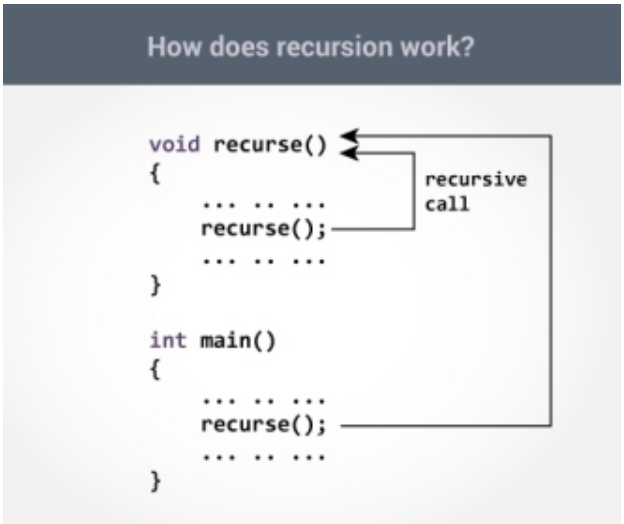
```
  int  *ptr = NULL;

  printf("The value of ptr is : %x\n", ptr  );

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

The value of ptr is 0

To check for a null pointer, you can use an 'if' statement as follows −

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```



Recursion in 'C': Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function. Or in orher words when a function is calling to itself is known as recursive function.

Recursion works as follows:

```
void recurse()
{
   ... .. ...
   recurse();
   ... .. ...
}

int main()
{
```

```
    ... .. ...
    recurse();
    ... .. ...
}
```

Conditions for recursive function:
1.Every function must have a **Base Criteria (Termination condition)** and for that it should not call to itself.
2. Whenever a function is calling to itself it must be closer to the **Base Criteria.**

Following are some examples of recursions-
(a) Fibonacci Series
(b) Binomial coefficient
(c) GCD

(a) Fibonacci Series

Fibonacci series are the numbers in the following integer sequence
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ....
the first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent term is the sum of the previous two terms. In mathematical terms, the Nth term of Fibonacci numbers is defined by the recurrence relation:

fibonacci(N) = Nth term in fibonacci series
fibonacci(N) = fibonacci(N - 1) + fibonacci(N - 2);
whereas, fibonacci(0) = 0 and fibonacci(1) = 1

Below program uses recursion to calculate Nth fibonacci number. To calculate Nth fibonacci number it first calculate (N-1)th and (N-2)th fibonacci number and then add both to get Nth fibonacci number.
For Example : fibonacci(4) = fibonacci(3) + fibonacci(2);

C program to print fibonacci series till Nth term using recursion

In below program, we first takes the number of terms of fibonacci series as input from user using scanf function. We are using a user defined recursive function named 'fibonacci' which takes an integer(N) as input and returns the Nth fibonacci number using recursion as discussed above. The recursion will terminate when number of terms are less then 2 because we know the first two terms of fibonacci series are 0 and 1.

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

int fibonacci(int term);
int main(){
   int terms, counter;
   printf("Enter number of terms in Fibonacci series: ");
   scanf("%d", &terms);
    /*
     Nth term = (N-1)th therm + (N-2)th term;
```

```
    */
printf("Fibonacci series till %d terms\n", terms);
   for(counter = 0; counter < terms; counter++){
      printf("%d ", fibonacci(counter));
   }
   getch();
   return 0;
}
/*
 Function to calculate Nth Fibonacci number
 fibonacci(N) = fibonacci(N - 1) + fibonacci(N - 2);
 */
int fibonacci(int term){
   /* Exit condition of recursion*/
   if(term < 2)
      return term;
   return fibonacci(term - 1) + fibonacci(term - 2);
}
```

Program Output

```
Enter number of terms in Fibonacci series: 9
Fibonacci series till 9 terms
0 1 1 2 3 5 8 13 21
```

(b) Binomial Coefficient Program:

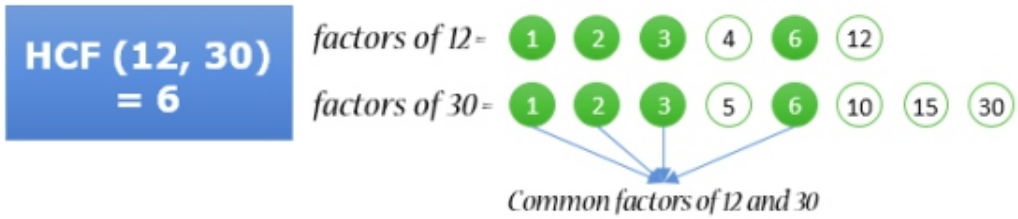
```
#include<stdio.h>

int fact(int);
void main()
{
int n,r,f;
printf("enter value for n & r\n");
 scanf("%d%d",&n,&r);
if(n<r)
printf("invalid input");
 else f=fact(n)/(fact(n-r)*fact(r));
printf("binomial coefficient=%d",f);
}

 int fact(int x)
{
if(x>1)
return x*fact(x-1);
else return 1;
 }
```

(c) GCD of Two numbers:

```
Input first number: 10
```

Input second number: 15
Output GCD: 5

Logic to find GCD using recursion



Euclidean algorithm to find GCD:
Begin:
function gcd(a, b)
  If (b = 0) then
    return a
  End if
  Else
    return gcd(b, a mod b);
  End if
End function
End

Program to find GCD using recursion:

```
/**
 * C program to find GCD (HCF) of two numbers using recursion
 */

#include <stdio.h>


/* Function declaration */
int gcd(int a, int b);



int main()
{
    int num1, num2, hcf;

    /* Reads two numbers from user */
    printf("Enter any two numbers to find GCD: ");
    scanf("%d%d", &num1, &num2);

    hcf = gcd(num1, num2);
```

```
    printf("GCD of %d and %d = %d\n", num1, num2, hcf);

    return 0;
}



/**
 * Recursive approach of euclidean algorithm to find GCD of two numbers
 */
int gcd(int a, int b)
{
    if(b == 0)
        return a;
    else
        return gcd(b, a%b);
}
```

Output:
Enter any two numbers to find GCD: 12
30
GCD of 12 and 30 = 6

Important Points

- An array is defined as finite ordered collection of homogenous data elements which are stored in contiguous memory locations.
- While storing the elements of a 2-D array in memory, these are allocated contiguous memory locations.
- Traversing means accessing the each and every element of array exactly once.
- A pointer is a variable whose value is the address of another variable.

Exercise

Objective type questions.

Q1. In linear search algorithm worst case occurs when
   a. The item is somewhere in the middle of the array
   b. The item is not in the array at all
   c. The item is the last element in the array
   d. The item is the last element in the array or is not there at all

Q2. The complexity of linear search algorithm is
   a. O(n) b. O(logn)
   c. O(n2) d. O(nlogn)

Q3. Average case occur in linear search algorithm
   a. When item is somewhere in the middle of the array
   b. When item is not in the array at all

c. When item is the last element in the array
d. When item is the last element in the array or is not there at all

Q4. Finding the location of the element with a given value is:
a. Traversal b. Search
c. Sort d. None of above

Q5. Which of the following case does not exist in complexity theory
a. Best case b. Worst case
c. Average case d. Null case

Short answer type questions.

Q1. What is the time complexity of binary search ?
Q2. What do you mean by Array ?
Q3. What is string ?
Q4. What do you mean by pointer ?
Q5. What is dynamic memory allocation ?

Essay type questions.

Q1. Explain two Dimentional array with example ?
Q2. Explain Malloc function in detail ?
Q3. Which data structure is used to perform recursion ?
Q4. Why binary search is better then linear search ?
Q5. Explain character string ?

Answers

Ans1. d Ans2. d Ans3. a
Ans4. b Ans4. d