Programming in ANSI C

Dr. Dharm Raj Singh
Assistant Professor, Jagatpur
P. G. College, Varanasi

Outline

MODULE 1

- Unit 1 : Array
- Unit 2 : String

MODULE 2

- Unit 1 : Pointer

MODULE 3

- Unit 1 : Structures
- Unit 2 : Unions

MODULE 4

- Unit 1 : Macros and Preprocessors

MODULE 5

- Unit 1: File handling

C Array

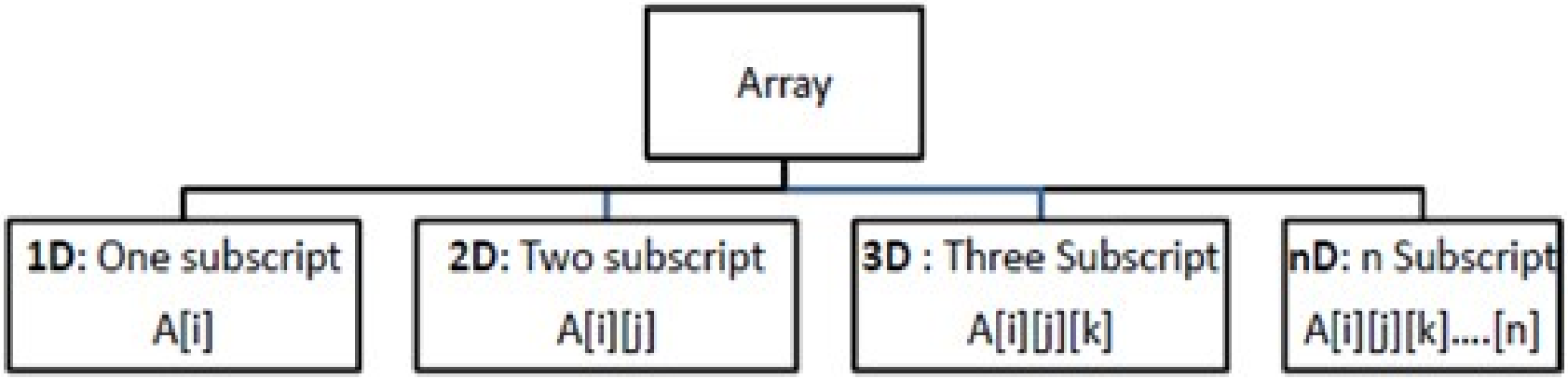
- An array is defined as the collection of similar type of data items stored at contiguous memory locations.
- Arrays are the **derived data type** in C programming language which can store the primitive type of data such as int, char, double, float, etc.
- It also has the capability to **store the collection of derived data types, such as pointers, structure**, etc.
- The array is the simplest data structure where each data element can be randomly accessed by using its index number. example

Price of *n* items: Shares a common attribute that they are numeric value.

	item1p	item2p	item3p	item4p	item5p	item6p	item7p	item8p	-------	itemnp
Index	0	1	2	3	4	5	6	7	-------	n

This can be represented using a common name (say price) and index / subscript, as different terms represented in a sequence.

e.g. price[5] gives the price of the fifth item in the list.



Declaration One Dimensional Array: arrays must be declared explicitly before they are used. The general form of declaration is:

Data_type Array_Name[Array_Size];

int marks[5];

- Here, int is the *data_type*, marks are the *array_name*, and 5 is the *array_size*.

Initialization of One Dimensional Array

- **Once an array is declared, it must be initialized. Otherwise array will contain the garbage values**. There are two different ways in which we can initialize the static array

1. Compile time
2. Run time

- **Compile Time Initialization**

1. Initializing Arrays during Declaration

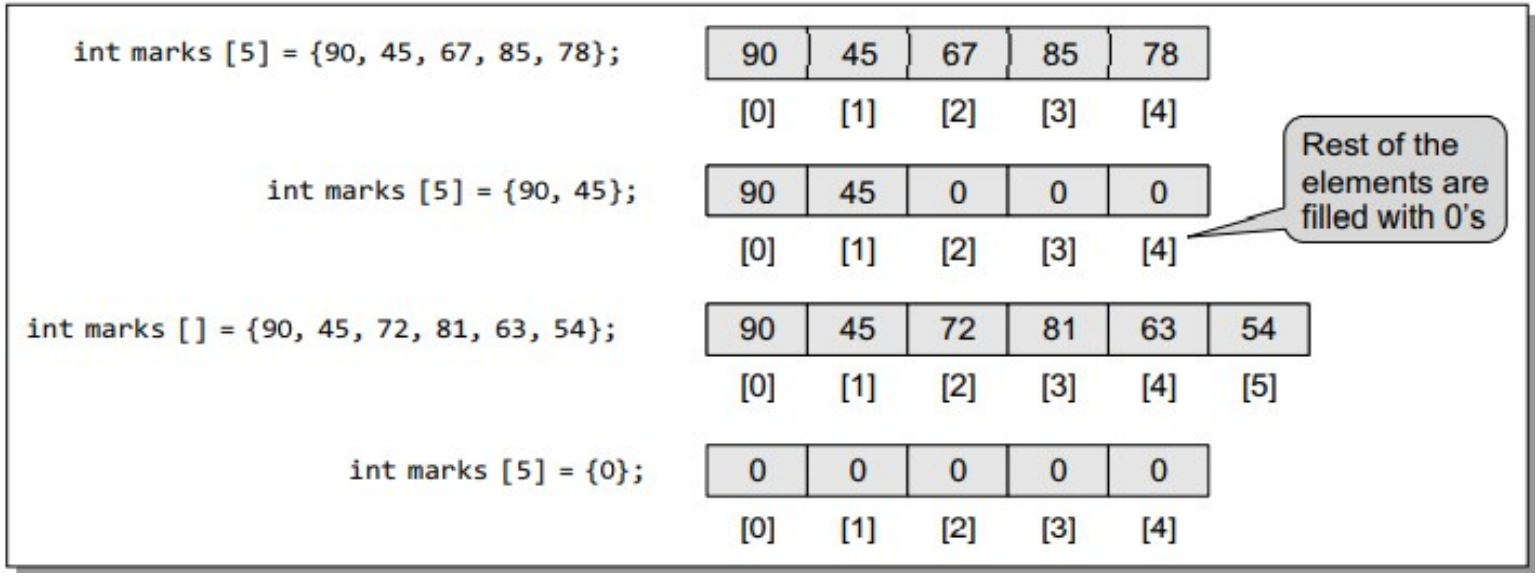
data-type array-name[size] = { list of values separated by comma };

For example:

int marks[5]={90, 82, 78, 95, 88};

marks[0]	90
marks[1]	82
marks[2]	78
marks[3]	95
marks[4]	88

- **Note** that if the number of values provided is less than the number of elements in the array, the un-assigned elements are filled with zeros.



initialize an array is by using the index of each element: We can initialize each element of the array by using the index.

```
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
```

Run Time Initialization

- An array can initialized at run time by the program or by taking the input from the keyboard. Generally, the large arrays are declared at run time in the program itself. Such as –

```
int i, marks[10];
printf("Enter any ten marks: ");
    for(i=0;i<10;i++)
    scanf("%d", &marks[i]);
```

Accessing array elements: An element is accessed by indexing the array name.

Array_name[index];

for example marks[0];

marks[1];

```
main()
{
   int i;
   int x[10];
   for (i=0; i<10; i++)
  {
     x[i] = i + 1;
  printf( “x=%d.\n", x[i]);
   }
 }
```

```
main( )
{
int val[10], i, total=0;
printf(“Enter any ten numbers: ”);
for(i=0;i<10;i++)
scanf(“%d”, &val[i]);
for(i=0;i<10;i++)
total = total + val[i];
printf(“\nTotal is: %d”, total);
}
```

Classification of Array: Array are divided into two part-

One Dimensional Array

Multi Dimensional Array

One Dimensional Array: Dimension is nothing but no of pairs of square brackets placed after the name of the array. It can be defined as one dimensional array has only one subscript specification is required for specify a particular element of an array. The syntax of the declaring one dimensional array is as follows:

Data_Type Array_Name[Expression];

For Example: int marks[10];

Where int is the type of the array, marks is name of the array and expression [10] is shows the maximum number of the element in the array.

```
//Write a C program to print Element of Array.
#include<stdio.h>
#include<conio.h>
void main()
{
   int i, array[30], n;
   printf("\n Enter the no of elements :");
   scanf("%d", &n);
   printf("\n Enter the values :");    //Read values into Array
   for (i = 0; i < n; i++)
  {
            scanf("%d", &array[i]);
  }
   for (i = 0; i < n; i++)             //Printing of all elements of array
  {
      printf("\n array[%d] = %d", i, array[i]);
    }
 getch( );
}
```

Write a program to find the mean of n numbers using arrays.

```
#include <stdio.h>
#include <conio.h>
int main()
{
int i, n, arr[20], sum =0;
float mean = 0.0;
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n arr[%d] = ", i);
scanf("%d",&arr[i]);
}
for(i=0;i<n;i++)
sum += arr[i];
mean = (float)sum/n;
printf("\n The sum of the array elements = %d", sum);
printf("\n The mean of the array elements = %.2f", mean);
return 0;
}
```

Multi Dimensional Array: It can be defined as 'Array which holds more than one subscript is known as Multi Dimensional Array. Generally 2-D array is called as Matrix. It is more suitable for the processing of table and matrix manipulations. C language allow to programmers to use arrays with more than two dimensions. On behalf of this it can be divided into two sub section:

Two Dimensional Array

N-Dimensional Array

Data_Type Array_Name[row size][column size];

For Example: int n[3][4];

Where int is the type of the array, n is the array name and row size is the number of rows and column size the number of column. To find the total number of element of an array, multiply the total number of row with the total number of column.

For Example: int a[3][4];

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form a[i][j], where a of the array, and i and j are the subscripts (index) that uniquely identify each element in a.

Initializing Two-Dimensional Arrays:

- Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
 {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
 {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
 {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

Accessing Two-Dimensional Array Elements:

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

int val = a[2][3];

The above statement will take 4th element from the 3rd row of the array.

Example

```
#include <stdio.h>
int main ()
{
int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}}; /* an array with 5 rows and
2 columns*/
int i, j;
                                        /* output each array element's value */
for ( i = 0; i < 5; i++ )
{
for ( j = 0; j < 2; j++ )
{
printf("a[%d][%d] = %d\n", i,j, a[i][j] );
}
}
return 0;
}
```

Write a program to print a matrix of order m * n where

```
void main( )
{       clrscr( );
        int mat[50][50] i, j, m, n;
        printf("Enter the number of Rows");
        scanf("%d",&m);
        printf("Enter the number of Columns");
        scanf("%d",&n);
        printf("Enter the element for the Matrix");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        scanf("%d", &mat[i][j]);
                }
        }
        printf("The Matrix is:");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        printf("\t%d",mat[i][j]);
                }
                printf("\n");
        }
        getch( );
}
```

Array with more than two dimentions

- Three-D array as 2-D arrays. For example

int x[2][3][4];

This array consist of two 2-D arrays and each of those 2-D array has 3 rows and 4 columns.

[0]

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

[1]

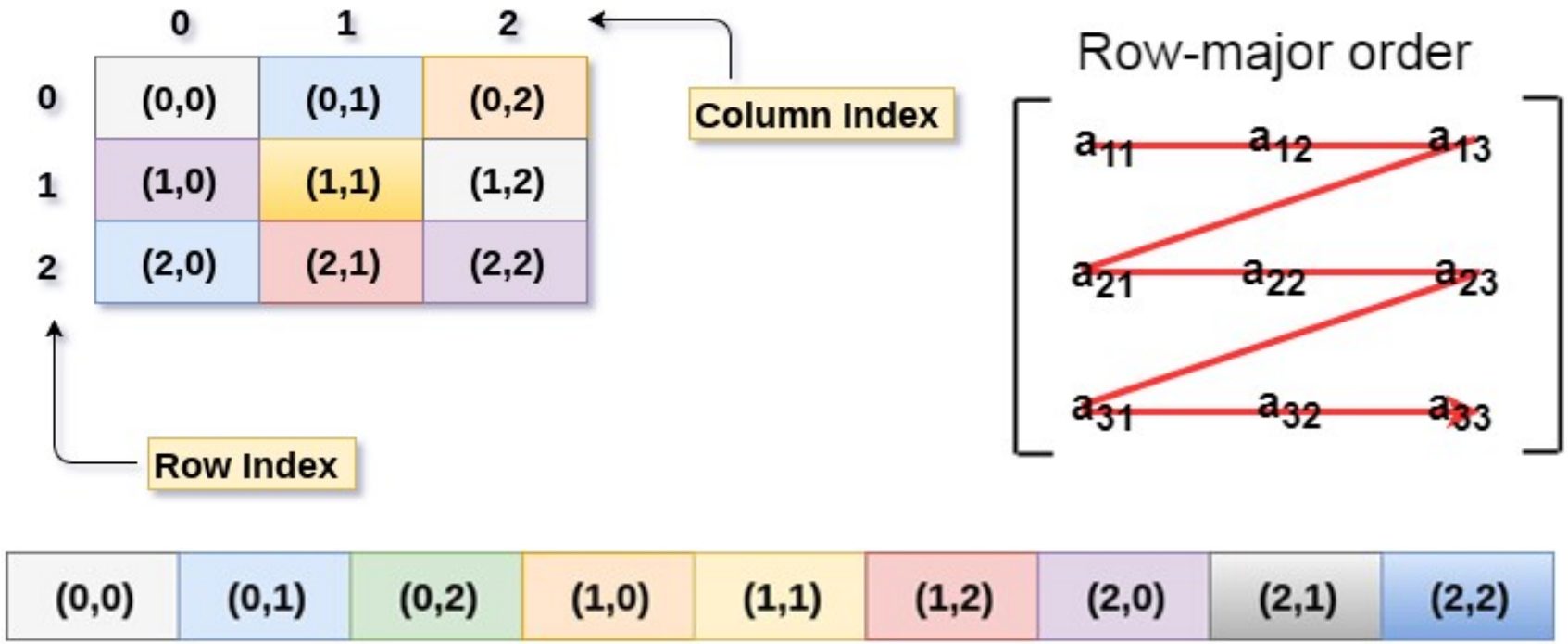
	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

```
#include<stdio.h>
int i,j,k; //variables for nested for loops
int main()
{
  int arr[2][3][4]; //array declaration
 printf("enter the values in the array: \n");
 for(i=1;i<=2;i++) //represents block
{
  for(j=1;j<=3;j++) //represents rows
{
   for(k=1;k<=4;k++) //represents columns
 {
   printf("the value at arr[%d][%d][%d]: ", i, j, k
     );
    scanf("%d", &arr[i][j][k]);
  }
 }
}
```

```
printf("print the values in array: \n");
for(i=1;i<=2;i++)
{
  for(j=1;j<=3;j++)
 {
  for(k=1;k<=4;k++)
  {
   printf("%d ",arr[i][j][k]);
  }
  printf("\n");
 }
 printf("\n");
}
 return 0;
 }
```

Row Major ordering

- In row major ordering, all the rows of the 2D array are stored into the memory contiguously. Considering the array shown in the image, its memory allocation according to row major order is shown as follows.
- first, the 1st row of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last row.



Calculating the Address of the random element of a 2D array

- **By Row Major Order:** If array is declared by a[m][n] where m is the number of rows while n is the number of columns, then address of an element a[i][j] of the array stored in row major order is calculated as,

$$\text{Address}(a[i][j]) = \text{B. A.} + (n*(i-l_1)+(j-l_2))* \text{size}$$

where, B. A. is the base address or the address of the first element of the array a[0][0] , n=(u_2 –l_2+1), and l_1 is lower bound of row, l_2 is lower bound of column, u_1 is upper bound of row, u_2 is upper bound of column.

Example :

a[10...30, 55...75], base address of the array (BA) = 0, size of an elemen t = 4 bytes .

Find the location of a[15][68].

l_1=10, l_2=55, u_1=30, u_2=75, n=(75-55+1)=21

Address(a[15][68]) = 0 + (21(15 - 10) * + (68 - 55)) * 4

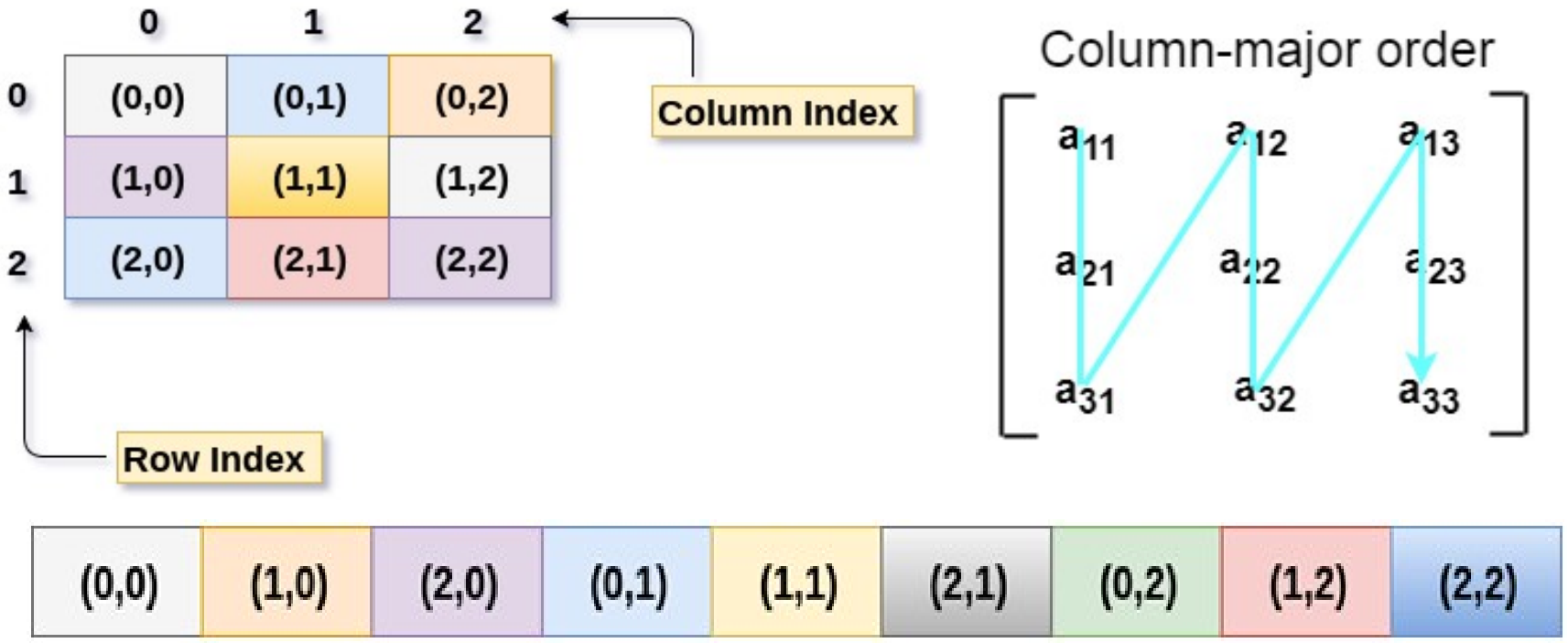
= (21*5 + 13) * 4

= 118 * 4

= 472 answer

Column Major ordering

- According to the column major ordering, all the columns of the 2D array are stored into the memory contiguously. The memory allocation of the array which is shown in the above image is given as follows.
- first, the 1st column of the array is stored into the memory completely, then the 2d row of the array is stored into the memory completely and so on till the last column of the array.



Calculating the Address of the random element of a 2D array

- **By Column major order:** If array is declared by a[m][n] where m is the n umber of rows while n is the number of columns, then address of an eleme nt a[i][j] of the array stored in row major order is calculated as,

$$\textbf{Address(a[i][j]) = B. A. + ((i-l}_1\textbf{)+(j-l}_2\textbf{)*m)* size}$$

- where, B. A. is the base address or the address of the first element of the array a[0][0] , $m=(u_1 - l_1+1)$, and l_1 is lower bound of row, l_2 is lower bound of column, u_1 is upper bound of row, u_2 is upper bound of column.
- **Example :**
- A [5 ... 20][20 ... 70], BA = 1020, Size of element = 8 bytes. Find the locat ion of a[0][30].
- $l_1=5$, $u_1=20$, $l_2=20$, $u_2=70$, $m=(20-5+1)=16$
- Address [A[0][30]) = ((0-5) + (30-20)*16)*8) + 1020
- = (-5+160)* 8 + 1020

  = 2260 bytes

Passing Arrays as Function Arguments

- If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received.
- **Way-1 Passing Addresses or Formal parameters as a pointer as follows:** we can pass the address of an individual array element by preceding the indexed array element with the address operator. There fore, to pass the address of the fourth element of the array to the called function, we will write &arr[3].

```
void myFunction(int *param)
{
.
.
}
```

```c
#include <stdio.h>
void disp( int *num)
 {
 printf("%d ", *num);
 }

int main()
{ int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
for (int i=0; i<10; i++)
{ /* Passing addresses of array elements*/
disp (&arr[i]);
}
return 0;
}
```

Passing array to a function as a pointer

```
#include <stdio.h>
void printarray(char *arr)
{
   printf("Elements of array are : ");
   for(int i=0;i<5;i++)
   {
      printf("%c ", arr[i]);
   }
}
int main()
{
  char arr[5]={'A','B','C','D','E'};
  printarray(arr);
  return 0;
}
```

➢ Way-2: Passing Individual Elements (Passing Data Values):

Individual elements can be passed in the same manner as we pass variables of any other data type. The condition is just that the data type of the array element must match with the type of the function parameter.

```
#include <stdio.h>
void disp( char ch)
{
  printf("%c ", ch);
 }
int main()
{ char arr[] = {'a', 'b', 'c', 'd', 'e'};
for (int x=0; x<5; x++)
{
/* passing each element one by one using subscript*/
disp (arr[x]);
}
return 0;
}
```

Passing the Entire Array

- array name refers to the address of first element of the array in the memory. The address of the remaining elements in the array can be calculated using the array name and the index value of the element. Therefore, when we need to pass an entire array to a function, we can simply pass the name of the array.

```
include <stdio.h>
float Sum(float num[]);
int main()
 { float result, num[] = {23.4, 55, 22.6, 3, 40.5, 18};
result = Sum(num);
printf("Result = %.2f", result);
 return 0;
}
float Sum(float num[])
{ float sum = 0.0;
for (int i = 0; i < 6; ++i)
{ sum += num[i];
 }
 return sum; }
```

Example

```
double getAverage(int arr[], int size);
int main () {
int balance[5] = {1000, 2, 3, 17, 50};
double avg;
avg = getAverage( balance, 5 ) ;
printf( "Average value is: %f ", avg );
return 0;
}
double getAverage(int arr[], int size)
{
int i;
double avg;
double sum;
for (i = 0; i < size; ++i)
{
sum += arr[i];
}
avg = sum / size;
return avg;
}
```

SORTING

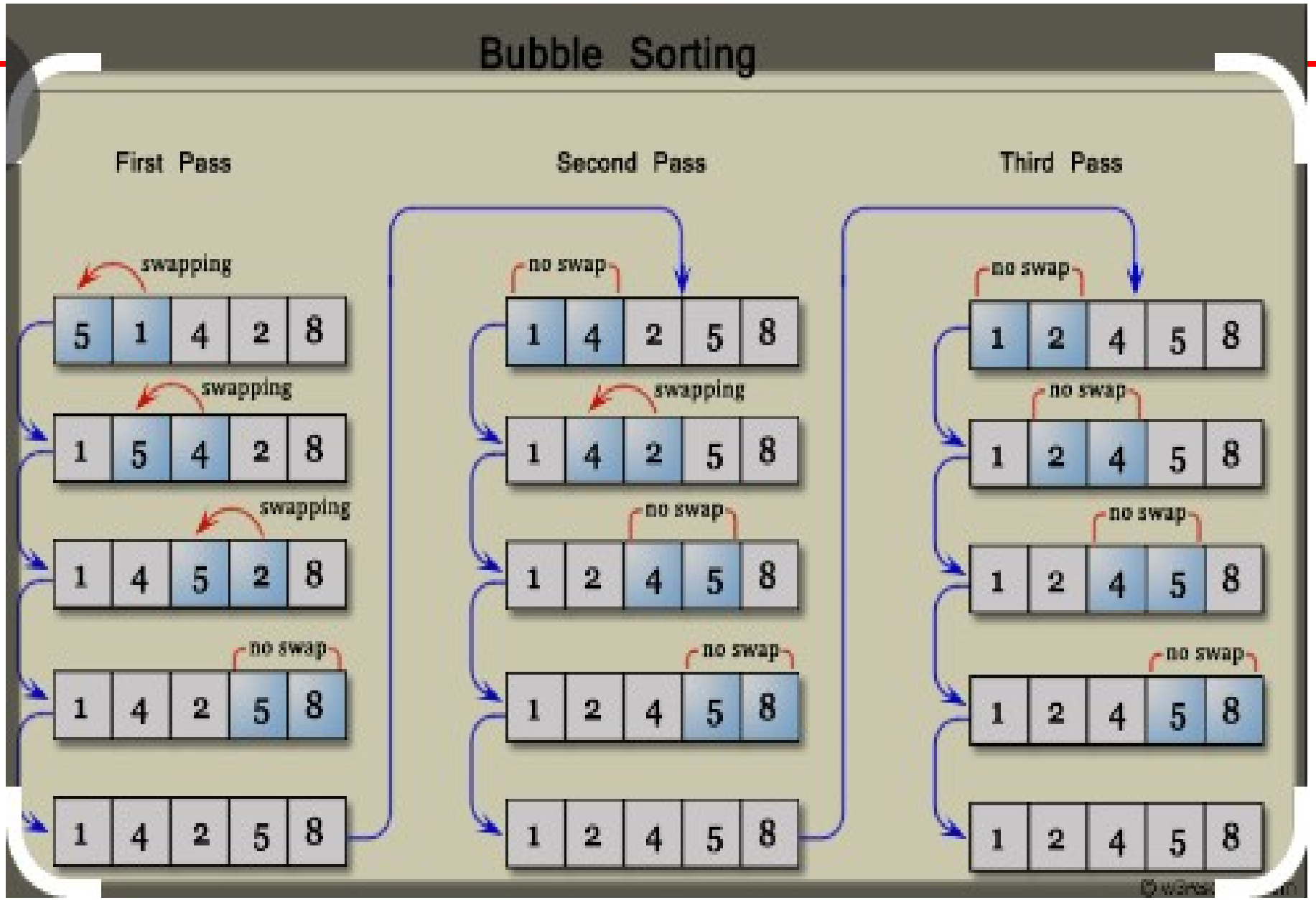
- Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending.
- There are two types of sorting:
- **Internal sorting:** which deals with sorting the data stored in the computer's memory.
- **External sorting** which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in the memory.

Bubble sort

- Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order).
- In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one.
- This process will continue till the list of unsorted elements exhausts.

```
BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For 1 = 0 to N-1
Step 2:     Repeat For J = 0 to N - I
Step 3:                 IF A[J] > A[J + 1]
                        SWAP A[J] and A[J+1]
            [END OF INNER LOOP]
        [END OF OUTER LOOP]
Step 4: EXIT
```


Bubble Sorting
First Pass
Second Pass
Third Pass
swapping
no swap
5 1 4 2 8
1 5 4 2 8
1 4 5 2 8
1 4 2 5 8
1 4 2 5 8
1 4 2 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8

Write a program to enter n numbers in an array. Redisplay the array with elements being sorted in ascending order.

```
int main()
{ int array[100], n, i, j, swap;
printf("Enter number of elements n");
scanf("%d", &n);
printf("Enter %d Numbers:n", n);
for(i = 0; i < n; i++)
scanf("%d", &array[i]);
for(i = 0 ; i < n - 1; i++)
{
for(j = 0 ; j < n-i-1; j++)
{
```

```
if(array[j] > array[j+1])
{ swap=array[j];
array[j]=array[j+1];
array[j+1]=swap;
}
 }
 }
printf("Sorted Array:n");
for(i = 0; i < n; i++)
printf("%dn", array[i]);
return 0;
 }
```

Insertion Sort

- Insertion sort is a simple sorting algorithm.
- This sorting method sorts the array by shifting elements one by one.
- It builds the final sorted array one item at a time.
- This sort is efficient for smaller data sets but it is insufficient for larger lists.
- It **has less space complexity like bubble sort**.
- It **requires single additional memory space.**
- Insertion sort does not change the relative order of elements with equal keys because it is stable.

INSERTION-SORT (ARR, N)
Step 1: Repeat Steps 2 to 5 for K=1 to N–1
Step 2: SET TEMP = ARR[K]
Step 3: SETJ=K-1
Step 4: Repeat while TEMP <= ARR[J] && j >=0
SET ARR[J + 1] = ARR[J]
SETJ=J-1
[END OF INNER LOOP]
Step 5: SET ARR[J + 1] = TEMP
[END OF LOOP]
Step 6: EXIT

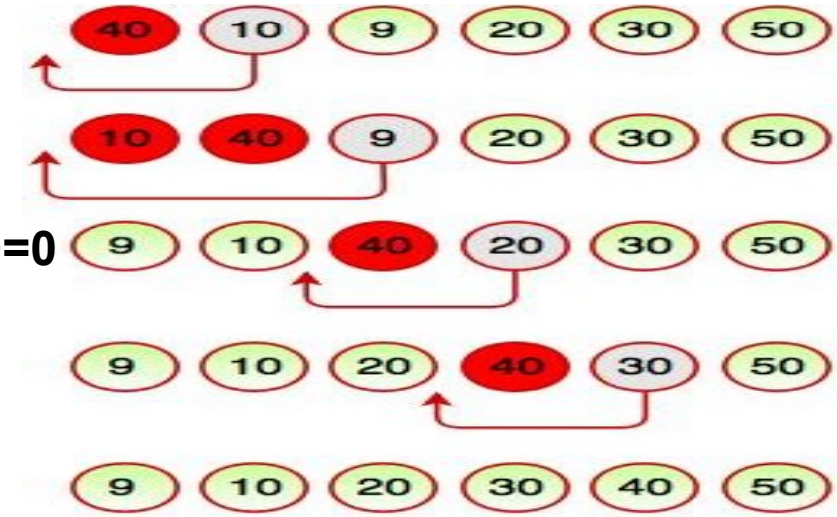


Fig. Working of Insertion Sort

Selection Sort

- Selection sort is a simple sorting algorithm which finds the smallest element in the array and exchanges it with the element in the first position. Then finds the second smallest element and exchanges it with the element in the second position and continues until the entire array is sorted.

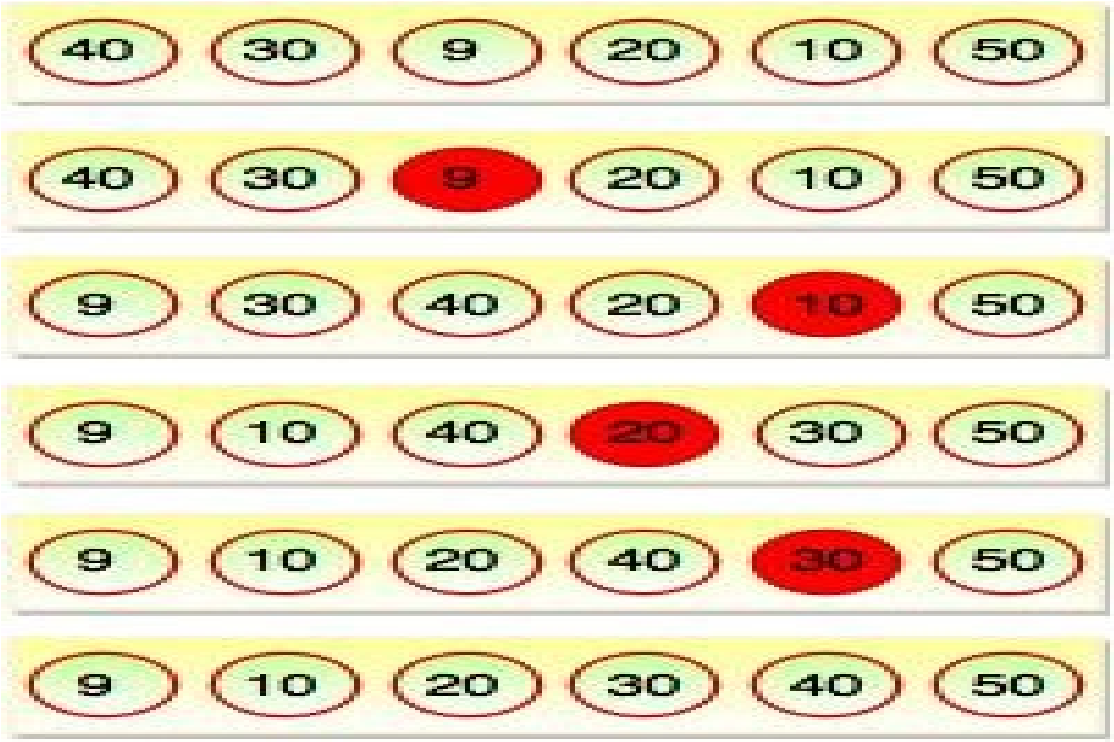


Fig. Working of Selection Sort

Selection sort

Step 1: For I = 0 TO N – 2
Step 2: [Assume i^{th} elements as smallest] Small = A[I]
Step 3: POS = I
Step 4: [Find the smallest element in the array and its position]
For J = I + 1 TO N – 1
Step 5: If(A[J] < small) Then
Step 6: small = A[J]
Step 7: POS = J
[EndIf]
[End of Step 4 For loop]
Step 8: [Exchange i^{th} element with smallest element]
A[POS] = A[I]
Step 9: A[I] = small
[End of Step 1 For loop]
Step 10: Exit

Exercise

1. Write a program in C to merge two arrays of same size sorted in descending order.
2. Write a program in C to print all unique elements in an array.
3. Write a program in C to read n number of values in an array and display it in reverse or der.
4. Write a program in C to find the sum of all elements of the array.
5. Write a program in C to count a total number of duplicate elements in an array.
6. Write a program in C to find the maximum and minimum element in an array.
7. Write a program in C to separate odd and even integers in separate arrays
8. Write a program in C for a 2D array of size 3x3 and print the matrix.
9. Write a program in C for addition of two Matrices of same size.
10. Write a program in C for multiplication of two square Matrices.
11. Write a program in C to find transpose of a given matrix.
12. Write a program in C to calculate determinant of a 3 x 3 matrix.
13. Write a program in C to generate a random permutation of array elements.
14. Write a program that interchanges the odd and even components of an array.
15. Write a program to find the second largest of n numbers using an array.
16. Write a program to find whether the array of integers contains a duplicate number.
17. Write a program to read an array of n numbers and then find the smallest number.

Strings

- The string is sequence of characters that is treated as a single data item. The character array ended with a null character \0 is implemented as string. A series of characters enclosed in double quotes (" ") is called a **string constant, a string or string literal.**
- The C compiler can automatically add a null character (\0) at the end of a string constant to indicate the end of the string. For example, "hello" is considered a string constant.

Declaring a string: C does not support string as the data type. However is allows us to represent strings as character arrays. It can be declared as – **char string_name[size];**

Initializing a String: strings can also be initialized at compile time and at run time. The process of initializing the strings at compile time is to write the string literal within double quotes. Compile time initialization can be done in three ways.

Type 1. char name[9] = “Sri Rama”;

Type 2. char stream[7] = {`H', `e', `l', `l', `o', `!', `\0'};

Type 3. char str[] = "I like C.";

Type 1: Consider a character array name that is initialized with a string constant “Mr JOHN”. This is written as

char name[9] = “Mr JOHN”;

This string is stored in the memory as given below.

M	**r**		**J**	**O**	**H**	**N**	**\0**	

- The compiler can automatically append a null character (\0) to the end of the array and treat the character array as a character string.
- Note that the size of the array is specified to hold up to nine elements, although the string constant has only eight characters enclosed in double quotes. The extra space is reserved for the null character that the compiler will add later.

- **Type 2:** array can be declared as string when null(\0) is appended at the end by user and array size is made sufficient to hold all characters. This is shown below.

  char stream[7] = {`H', `e', `l', `l', `o', `!', `\0'}; /*string */

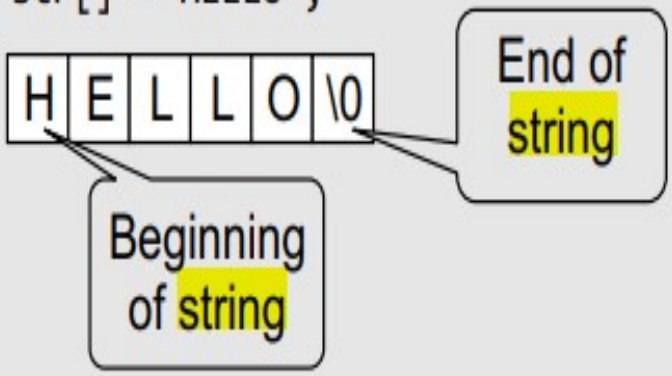
- **Type 3:** The third type of initialization is not specifying the array size. For example,

  char str[] = "I like C.";

```
#include <stdio.h>
int main ()
{
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
printf("Greeting message: %s\n", greeting );
return 0;
}
```

Difference between character storage and string storage

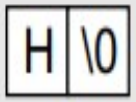
```
char str[] = "HELLO";
```




```
char ch = 'H';
```

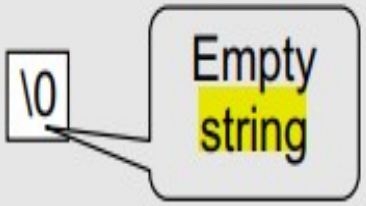
Here H is a character not a string.
The character H requires
only one memory location.


```
char str[] = "H";
```



Here H is a string not a character. The string H requires two memory locations. One to store the character H and another to store the null character.

```
char str[] = "";
```



Although C permits empty string,
it does not allow an empty character.

counting the number of vowels by using the null character.

```
#include<stdio.h>
void main ()
{
   char s[11] = "javatpoint";
   int i = 0;
   int count = 0;
   while(s[i] != NULL)
   {
      if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
      {
         count ++;
      }
      i++;
   }
   printf("The number of vowels %d",count);
}
```

Some of the String Functions in string.h

Function name	Description
strcpy(str1,str2)	copies str2 to str1 including the null (\0).
strcat(str1,str2)	Appends str2 to end of string str1
strlen(string)	Returns the length of string. Does not include the null(\0)
strcmp(str1,str2)	Compares str1 with str2 returns integer result.If str1<str2 returns negative integer.Returns zero when str1==str2,and Returns a positive integer when str1>str2.
Strncpy(str1,str2,n)	It copies at most n characters of str2 to str1.If str2 has fewer than n characters, it pads str1 with null('\o') characters.
Strchr(string,char)	Locates the position of the first occurrence of char within string and returns the address of the characterif it finds. and null if not.For ex-(" Hello", 'l')
strlwr()	converts all characters in a string from uppercase to lowercase.
strupr()	converts all characters in a string from lower case to uppercase.

Example of strcpy() function:

```
 #include <stdio.h>
#include <string.h>
int main ()
{
char str1[12] = "Hello";
char str3[12];

strcpy(str3, str1);                    /* copy str1 into str3 */
printf("strcpy( str3, str1) : %s\n", str3 );
return 0;
}
```

Output

strcpy(str3, str1) : Hello

Example of strlen() function:

```
#include <stdio.h>
#include <string.h>
void main()
{
char name[30] = "This is string in C";
int len;
len= strlen( name );
printf( "Length of string <%s> is %d.\n", name ,len);
}
```

Output

Length of string <This is string in C> is 19.

Example of strcat() function:

```
 #include <stdio.h>
#include <string.h>
int main ()
{
char str1[12] = "Hello";
char str2[12] = "World";
strcat( str1, str2);          /* concatenates str1 and str2 */
printf("strcat( str1, str2): %s\n", str1 );
return 0;
}

Output
strcat( str1, str2): HelloWorld
```

Example of strcmp() function:

```
#include <stdio.h>
#include <string.h>
main( )
{
char string1[ ] = "Jerry" ;
char string2[ ] = "Ferry" ;
int i, j, k ;
i = strcmp ( string1, "Jerry" ) ;
j = strcmp ( string1, string2 ) ;
k = strcmp ( string1, "Jerry boy" ) ;
printf ( "\n%d %d %d", i, j, k ) ;
}

Output
0 1 -1
```

Write a program to reverse a string without using strrev() function.

```
#include <stdio.h>
#include <string.h>
int main()
{
   char str[100], temp;
   int len, L,R;
   printf("Enter a string:");
   gets(str);
   len = strlen(str);
   L = 0;
   R = len -1;
   while(L < R)
   {
       temp = str[L];
       str[L] = str[R];
       str[R] = temp;
       L++;
       R--;
     }
      printf("Reversed string is: %s\n", str);
}
```

Output

```
Enter a string:raj
Reversed string is: jar
```

Write a program to check given string is palindrome or not.

```
#include <stdio.h>
#include <string.h>
int main()
{
   char str1[25], str2[25];
   printf("Enter the string :");
   gets(str1);
   strcpy(str2,str1);
   strrev(str2);
    if (strcmp(str1,str2) == 0)
      printf("String is a palindrome ");
   else
      printf("String is not a palindrome ");
}
```

Output

```
Enter the string : ramesh
String is not a palindrome
```

Write a program to find the length of a string.

```
#include <stdio.h>
#include <string.h>
int main()
 { char str[100], i = 0, length;
  printf("\n Enter the string : ");
gets(str);
 while(str[i] != '\0')
 i++;
 length = i;
printf("\n The length of the string is : %d", length);
return 0;
}
```

Output

```
Enter the string : ramesh
 The length of the string is : 6
```

Write a program check given string is palindrome or not without using string functions.

```
#include <stdio.h>
#include <string.h>
int main()
{
   char str[100];
   int start, middle, end, len = 0;
   printf("Enter the string :");
   gets(str);
    while (str[len] != '\0')
      len++;
    end = len - 1;
   middle = len/2;
   for (start = 0; start < middle; start++)
   {
      if (str[start] != str[end])
      {
         printf("String is not palindrome.\n");
         break;
      }
      end--;
   }
   if (start == middle)
      printf("String is palindrome.\n");
}
```

Output:
Enter the string :rar
String is palindrome.

ARRAYS OF STRINGS

- We have seen that a string is an array of characters.
- An array of strings would store 20 individual strings. An array of strings is declared as **char names[20][30];**
- Here, the first index will specify how many strings are needed and the second index will specify the length of every individual string. So here, we will allocate space for 20 names where each name can be a maximum 30 characters long. **For example**
- char name[5][10] = {"Ram", "Mohan", "Shyam", "Hari", "Gopal"};
- Then in the memory, the array will be stored as shown

name[0]	R	A	M	'\0'					
name[1]	M	O	H	A	N	'\0'			
name[2]	S	H	Y	A	M	'\0'			
name[3]	H	A	R	I	'\0'				
name[4]	G	O	P	A	L	'\0'			

Write a program to sort the names of students.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
char names[5][10], temp[10];
int i, n, j;
clrscr();
printf("\n Enter the number of students : ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n Enter the name of student %d : ", i+1);
gets(names[i]);
}
```

```
for(i=0;i<n;i++)
{
for(j=0;j<n–i–1;j++)
 {
if(strcmp(names[j], names[j+1])>0)
 {
strcpy(temp, names[j]);
strcpy(names[j], names[j+1]);
strcpy(names[j+1], temp);
}
}
}
printf("\n Names of the students in
alphabetical order are : ");
for(i=0;i<n;i++)
puts(names[i]);
getch();
return 0;
}
```

fflush(stdin) function

```
#include <stdio.h>
#include <conio.h>
int main()
{  char data[20];
    char city[20];
   int n;
printf("input any number");
scanf("%d", &n);
    printf("Enter Name ");
    gets(data);
    printf("Enter City ");
    gets(city);
    return 0;
}
```

Output:
input any number3
Enter Name
Enter City varanasi

➢**In above example, we have used gets() function to take input from the user. Here we don't have to use fflush(stdin) function because gets() function always clear the input buffer after taking input.**

fflush(stdin) function

- **Note: If you are taking int or float or double type input and after that, you are taking any character array input then use fflush(stdin) function before taking character array input. For example:**

```
#include <stdio.h>
#include <conio.h>
int main()
{   char data[20];
    char city[20];
int n;
printf("input any number");
scanf("%d", &n);
fflush(stdin);
    printf("Enter Name ");
    gets(data);
    printf("Enter City ");
    gets(city);
    return 0;
}
```

```
Output
input any number3
Enter Name raj
Enter City varanasi
```

Exercise

1. Write a program to count the number of vowels in a given string.
2. Write a program to count the number of words in a given string. Two words are separated by one or more blank spaces.
3. Write a program that replaces two or more consecutive blanks in a string by a single blank.
4. Write a program to copy contents of one array into another array.
5. Read a string from keyboard and inverse the case of it. That is, convert lower case letters to upper-case and vice versa.
6. Write a program to find out the length of a given string without using the library function strlen().
7. Write a program to find out the length of a given string using the library function strlen().
8. Write a program to insert a string in the main text.
9. Write a program to delete a substring from a text.
10. Write a program to replace a pattern with another pattern in the text.
11. Write a program to count the number of vowels in a given string.
12. Write a program to read multiple lines of text and then count the number of characters, words, and lines in the text.

Exercise

- Q13.write a program print the following output.

```
16 15 14 13 12
17  4  3  2 11
18  5  0  1 10
19  6  7  8  9
20 21 22 23 24
```

- Q14.write a program print the following output.

```
H
E H
L E H
L L E H
O L L E H
```

- Q15 Write a program to convert string lower case to upper case without using string functions.
- Q16 Write a program to search the occurrence of character in string.
- Q17 Write a program to c concatenate two Strings without using library function.

Address operator &

- The actual location of a variable in the memory is depending on system. How we know the address of a variable? This can be done with the help of **operator(&) in C.** The **& operator (ampersand)** is also known as "**Address of**" Operator. The **&** operator can be used only with a simple variable or an array element. Some of the following:-

1. **&125** (pointing at constants).
2. **int x[10],&x (pointing at array names.)**

```
#include <stdio.h>
int main()
{ int a;
printf("Input any no.:\n");
scanf("%d",&a);
printf("Your data is %d.\n",a);
printf("Variable 'a' is stored in memory at %u\n", a);
 }
/* %u is used for display the location of variable in memory decimal system */
```

Pointer

- A pointer is a derived data type in 'C'. It is built from any one of the primary data type available in 'C' programming language.
- A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location or Pointer is a variable that holds the Address of another variable.

Declaring Pointer Variable:-

Pointer declaration is similar to other type of variable except asterisk (*) character before pointer variable name. it is also called dereference operator.

"Value at Address"(*) Operator

The * Operator is also known as Value at address **operator.**

syntax to declare a pointer:- ***data_type *variable_name;***

```
int  *p;
char *pch;   float *pf;
```

Example

```
#include<stdio.h>
int main()
{ int*p;
int v=10;
p=&v;   /* Assigning the address of variable v to the pointer p*/
printf("Value of variable v is: %d", v);
printf("\n Value of variable v is: %d",*p);
printf("\n Address of variable v is:%p", &v);
printf("\n Address of variable v is: %p", p);
printf("\nAddress of pointer p is: %p",&p);
}
```

Note: *Pointers can be outputted using %p, since, most of the computers store the address value in hexadecimal form using %p gives the value in that form. But for simplicity and understanding we can also use %u to get the value in Unsigned int form.*

BENIFITS OF POINTER:-

1. Pointers **reduce the length and complexity of a program**.
2. **Execution time** with pointers is **faster because** data are manipulated **with the address, that is, direct access to memory location.**
3. A pointer enables us to access a variable that is defined outside the function.
4. Pointers are more efficient in handling the data tables.
5. The use of a pointer array of character strings results in **saving of data storage space in memory.**
6. We can **return multiple values from a function** using the pointer.
7. **Pointers save memory space.**
8. Pointers are used to allocate memory dynamically.

%x -> Hexadecimal value represented with lowercase characters (unsigned integer type)
%X -> Hexadecimal value represented with uppercase characters (unsigned integer type)
%p -> Displays a memory address (pointer type) hexadecimal system are depending by our system bit.

```
include <stdio.h>
int main()
{
int x = 12;
int *ptr;
ptr = &x;
printf("Address of x: 0x%p\n", ptr);
printf("Address of x: 0x%x\n", &x);
printf("Address of ptr: 0x%x\n", &ptr);
printf("Value of x: %d\n", *ptr);
return 0;
}
```

OUTPUT:
Address of x: 0x0065FBFA
Address of x: 0x65fbfa
Address of ptr: 0x65fbfa
Value of x: 12

INITIALIZATION OF POINTER VARIABLE:-

- The process of assigning the address of a variable to a pointer variable is known as **initialization.** Once a pointer variable has been declare we can use the assignment operator to initialize the variable. EXAMPLE:

```
int quantity;
int  *p;
p= &quantity;
```

```
#include<stdio.h>
int main()
{ int a;
  int *ptr;
  a = 10;
  ptr = &a;
  printf("Value of ptr:%u", ptr);
  return (0);
 }
```

Dereferencing Pointer:

- Dereferencing is an operation performed to access and manipulates data contained in the memory location pointed to by a pointer.
- The operator * **is used to dereference pointers**. A pointer variable is dereference when the unary operator *, in this case called the indirection operator, is used as a prefix to the pointer variable.

```
int main()
{
int *p, v1, v2;
p=&v1;
*p=25;
*p+=10;
printf(" value of v1= %d\n", v1);
v2=*p;
printf("value of v2= %d\n", v2);
p=&v2;
*p+=20;
printf(" now the value of  v2=%d \n", v2);
}
```

Accessing the Address of a Variable through its Pointer:-

- **O**nce a pointer has been assigned the address of variable, the question is how to access a value of a variable through its pointer? This is done by using another unary operator *(asterisk),Consider the following statements.

```
int ds, *p, n;
ds = 133;
p =  &ds;
n =  *p;
```

- **Here** we can seen that ds and **n** declare as integer and **p** pointer variable that point to an integer. When the operator * is placed before the pointer variable in an expression, the pointer returns the values of variable of which the pointer value is the address. In this case, ***p** returns the value of variable **ds.** Thus the value of **n** would be **133.** The two statements

```
p  =&ds;
n   =   *p; are equivalent to
n   =  *&ds; which is turn is equivalent to  n    =   ds;
```

```
int main()
{ int *p, q;
  q = 19;
  p = &q;      /* assign p the address of q */
  printf(" Vlaue of q=%d\n",q);
  printf("Contents of p=%d\n ", *p);
  printf("Address of q stored in p=%d ",p);
  return 0;
  }
```

OUTPUT:

Vlaue of q=19

Contents of p=19

Address of q stored in p=6487572

Pointer Arithmetic

- The size of data type which the pointer variable points to is the number of bytes accessed in memory.
- The size of the pointer variable is dependent on the data type of the variable pointed by the pointer.
- Some arithmetic operations can be performed with pointers.
- C language supports the flowing arithmetic operators which can be performed with pointers .They are
  1. **Pointer increment(++)**
  2. **Pointer decrement(- -)**
  3. **addition (+)**
  4. **subtraction (-)**
  5. **Subtracting two pointers of the same type**
  6. **Comparison of pointers**

Pointer increment and decrement:-

- Integer, float, char, double data type pointers can be incremented and decremented. For all these data types both prefix and postfix increment or decrement is allowed.
- Integer pointers are incremented or decremented in the multiples of two. Similarly character by one, float by four and double pointers by eight etc For 32-bit machine.
- **Note** that **pointer arithmetic cannot be performed on void pointer**, since they have no data type associated with them.
- Let int *x;

```
x++  /*valid*/
++x  /*valid*/
x--  /*valid*/
--x  /*valid*/
```

- The Rule to increment the pointer is given as

  new_address= current_address + i * size_of(data type)
- Where i is the number by which the pointer get increased.
- For 32-bit int variable, it will be incremented by 2 bytes.
- For 64-bit int variable, it will be incremented by 4 bytes.
- The Rule to decrement the pointer is given as

  new_address= current_address - i * size_of(data type)

Note: If we have more than one pointer pointing to the same location , then the change made by one pointer will be the same as another pointer.

```
void main()
{
int *p1,x;
float *f1,f;
char *c1,c;
p1=&x;
f1=&f;
c1=&c;
printf("Memory address before increment:\n int=%p\n,float=%p\n, char=%p\n",p1,f1
    ,c1);
p1++;
f1++;
c1++;
printf("Memory address after increment:\n int=%p\n, float=%p\n,char=%p\n",p1,f1,
    c1);
}
```

C Pointer Addition

- We can add a value to the pointer variable. The formula of adding value to pointer is given

new_address= current_address + (number * size of (data type))

```
#include<stdio.h>
int main(){
int number=50;
int *p;             //pointer to int
p=&number;   //stores the address of number variable
printf("Address of p variable is %u \n",  p);
p=p+3;                                  //adding 3 to pointer variable
printf("After adding 3: Address of p variable is %u \n", p);
return 0;
}
```

Output

Address of p variable is 3214864300

After adding 3: Address of p variable is 3214864312

C Pointer Subtraction

- Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given as:

new_address= current_address - (number * size of (data type))

```
#include<stdio.h>
int main()
{
int number=50;
int *p;    //pointer to int
p=&number;       //stores the address of number variable
printf("Address of p variable is %u \n",  p);
p=p-3;    //subtracting 3 from pointer variable
printf("After subtracting 3: Address of p variable is %u \n", p);
return 0;
}
```

Output

- **Address of p variable is 3214864300**
- **After subtracting 3: Address of p variable is 3214864288**

Subtraction of one pointer from another:-

- One pointer variable can be subtracted from another provided both variables point to elements of the same array. **The resulting value indicates the number of bytes separating the corresponding array elements.**

Example: Two integer pointers say **ptr1(address:1000)** and **ptr2(address:1004)** are subtracted. The difference between addresses is 4 bytes. Since the size of int is 4 bytes, therefore the **increment between ptr1 and ptr2** is given by **(4/4) = 1**.

```
#include <stdio.h>
main()
{ int x[] = {10, 20, 30, 45, 67, 56, 74} ;
int *i, *j ;
i = &x[1] ;
j = &x[5] ;
 printf("i=%d and j=%d\n",i,j);
printf ( "%d %d", j - i, *j - *i ) ;
}
Output:
i=6487540 and j=6487556
4          36
```

```
#include <stdio.h>
main()
{
int x = 5, y = 7;
int *p = &y;
int *q = &x;
printf("p is %d\n q is %d\np - q is %d", p, q, (p - q));
}
```

Output

```
p is 6487560
q is 6487564
p - q is -1
```

Comparison of Pointers

We can compare pointers using operators like >, >=, <, <=, ==, and !=. These operators return true for valid conditions and false for unsatisfied conditions.

```
#include <stdio.h>
int main()
{
    int arr[5];
    int *ptr2 = &arr[0];
    int *ptr1 =arr;
    if (ptr1 == ptr2)
     {
        printf("Pointer to Array Name and First Element are Equal.");
    }
    else
     {
        printf("Pointer to Array Name and First Element are not Equal.");
    }
    return 0;
}
```

Null Pointers

- A pointer variable is a pointer to a variable of some data type. However, in some cases, we may prefer to have a null pointer which is a **special pointer value and does not point to any value.**
- This means that a *null pointer does not point to any valid memory address.*
- 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.

```
#include <stdio.h>
 int main ()
 {
int *ptr = NULL;
 printf("The value of ptr is : %x\n", &ptr );
return 0;
}
```

Uses of NULL Pointer in C

- Following are some most common uses of the NULL pointer in C:
- To initialize a pointer variable when that pointer variable hasn't been assigned any valid memory address yet.
- To check for a null pointer before accessing any pointer variable. By doing so, we can perform error handling in pointer-related code, e.g., dereference a pointer variable only if it's not NULL.
- To pass a null pointer to a function argument when we don't want to pass any valid memory address.
- A NULL pointer is used in data structures like trees, linked lists, etc. to indicate the end.

Pointers does not allow the flowing operation

(a) Addition of two pointer
(b) Multiplication of a pointer with a constant
(c) Division of a pointer with a constant

There are various operations which can not be performed on pointers.

Address + Address = illegal

Address * Address = illegal

Address % Address = illegal

Address / Address = illegal

Address & Address = illegal

Address ^ Address = illegal

Address | Address = illegal

~Address = illegal

void Pointer

void Pointer: A generic pointer is a pointer variable that has **void** as its data type.

- The void pointer, or the generic pointer, is a special type of pointer **that can point to variables of any data type.**
- For example if we have a pointer to int, then it would be incorrect to assign the address of a float variable to it. But an exception to this rule is a pointer to void.
- Syntax of declaration of a void type:

<void> *<variable name>;

- A void pointer cannot be dereferences simple by using indirection operator. Before dereferencing, it should be **type cast** to appropriate pointer data type.
- **Pointer arithmetic cannot be performed on void pointers** without typecasting.

```
int main ()
    {
    int  a=3;
    float b=3.4, *fp=&b;
    void *vp;
    vp=&a;
    printf ("value of a= %d\n", *(int *)vp);
    *(int *)vp=12;
    printf ("value of a= %d\n", *(int *)vp);
    vp = fp;
    printf ("value of b= %f\n", *(float *)vp);
    }
```

Output:

value of a= 3

value of a= 12

value of b=3.400000

Precedence of dereferencing Operator and Increment /decrement Operators

- The precedence level of * operator and increment/ decrement operators is same and their associativity is from right to left.
- Suppose ptr is an integer and x in an integer variable.

➢ **X=*ptr++;**

- The expression ***ptr++** is equivalent to ***(ptr++)**, since these operators associate from right to left. The increment operator is postfix, so first the value of ptr will be used in the expression and then it will be incremented.

➢ **X=*++ptr;**

- The expression ***++ptr** is equivalent to ***(++ptr),** since these operators associate from right to left. The increment operator is prefix, so first ptr will be incremented

➢ **X=++*ptr;**

- The expression **++*ptr** is equivalent to **++(*ptr),** since these operators associate from right to left.

➢ **X=(*ptr)++;**

- Here increment operator is applied over **(*ptr),** since it is postfix increment hence first the value of ***ptr** will be assigned to x and then it will be incremented.

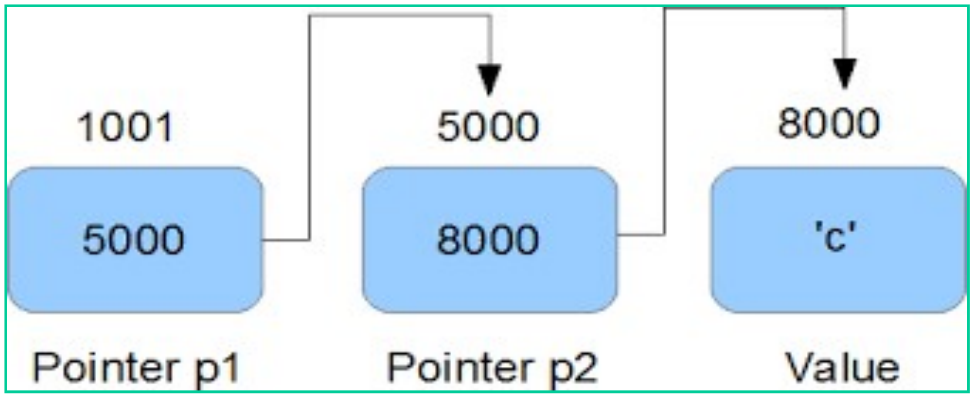
```
#include <stdio.h>
int main() {
   int nums[] = {10, 20, 30, 40, 50};
   int *p = nums;      //pointer storing the address of nums array
   int var;
   var = ++*p;         //value at index 0 incremented by 1
   printf(" *p = %d, var = %d \n", *p, var);
   var = *p++;
   printf(" *p = %d, var = %d \n", *p, var);
   var = *++p;         //pointer address get inc and then value is stored
   printf(" *p = %d, var = %d \n", *p, var);
   return 0;
}
```

Output

```
*p = 11, var = 11
*p = 20, var = 11
*p = 30, var = 30
```

Pointer to pointer

- It is possible to make a pointer to point to another pointer, (this is also known as **Pointer to pointer** and **Double pointer**), thus creating a chain of pointer as shown.



- Here, the pointer variable **p1** contains the address of the pointer variable **p2.** This is known as ***multiple indirections.***
- A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name.
- The general syntax for declaring a pointer to pointer is:

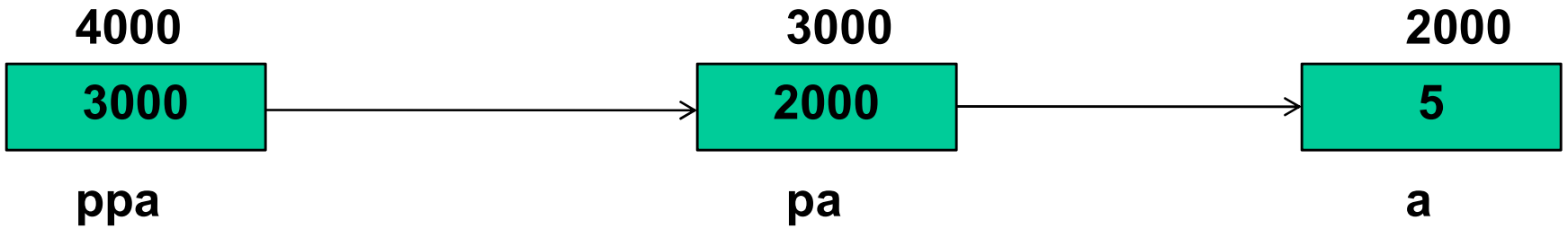
<data type> **< pointer to pointer variable name>.

Example**: int **p1;**

```
int main ()
{
int  var;
int  *ptr;
int  **pptr;
var = 3000;
ptr = &var;      /* take the address of var */
pptr = &ptr;     /* take the address of ptr using address of operator &  */
printf("Value of var = %d\n", var );
printf("Value available at *ptr = %d\n", *ptr );
printf("Value available at **pptr = %d\n", **pptr);  /* take the value using pptr */
 }
```

OUTPUT:-

```
 Value of var = 3000
 Value available at *ptr = 3000
 Value available at **pptr = 3000
```



Value of a	a	*pa	**ppa	5
Address of a	&a	pa	*ppa	2000
Value of pa	&a	pa	*ppa	2000
Address of pa		&pa	ppa	3000
Value of ppa		&pa	ppa	3000
Address of ppa			&ppa	4000

Use of pointers to pointers

- Double pointers are often used in situations where we need to allocate memory for a two-dimensional array dynamically or when we need to pass a pointer to a pointer to a function to modify the original pointer value.
- Note that if we used a single pointer to hold the address of the row pointer array, we would not be able to allocate memory for the columns of the array dynamically, since the single pointer would only hold the address of the first row pointer, and we would not be able to access the other rows of the array.
- If we used a single pointer in the modify Pointer function, we would only be able to modify the copy of the original pointer value that was passed to the function, not the original pointer value itself. Therefore , using a double pointer is necessary to modify the original pointer value directly.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int m,n;
printf("input number of row and column\n"
    );
scanf("%d \n %d",&m, &n);
    int** arr;    // Allocate memory for rows
arr = (int**)malloc(m * sizeof(int*));
for (int i = 0; i < m; i++)
 arr[i] = (int*)malloc(n * sizeof(int));
printf("input element in matrix \n");
    for (int i = 0; i < m; i++)
 {   for (int j = 0; j < n; j++)
  {  scanf("%d",&arr[i][j]);
        }
    }
    // Print the array
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }

    // Free allocated memory
    for (int i = 0; i < m; i++) {
        free(arr[i]);
    }
    free(arr);

    return 0;
}
```

```
#include <stdio.h>
  // Function that takes array of strings as argument
void print(char** arr, int n) {
      for (int i = 0; i < n; i++)
          printf("%s\n", *(arr + i));
}

int main() {
    char* arr[10] = {"Geek", "Geeks",
                     "Geekfor"};
    print(arr, 3);
    return 0;
}
```

Pointers and one dimensional Arrays:-

- The elements of an array are store in contiguous memory location. When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e address of the first element of the array is also allocated by the compiler.

Suppose we declare an array x,

int x[5] = { 1, 2, 3, 4, 5 };

- Assuming that the base address of x is 1000 and each integer requires two bytes, the five elements will be stored as follows:

Element	x[0]	x[1]	x[2]	x[3]	x[4]
Address	1000	1002	1004	1006	1008

- Here variable **x** will give the base address, which is **a constant pointer** pointing to the first element of the array**, x[0].** Hence **x** contains the address of **x [0]** i.e. **1000**.

We can also declare a pointer of type int to point to the array **x**.

```
int *p;  p = x;    // or,
p = &x[0];       //both the statements are equivalent.
```

- Now we can access every element of the array **x** using **p++** to move from one element to another.
- When using pointers, an expression like x[i] is equivalent to writing *(x+i).

NOTE: array name is a constant pointer variable so you can't incre ment or decrement it.

- Many beginners get confused by thinking of array name as a pointer.
- For example, while we can write ptr = x; // ptr = &x[0]
- **we cannot write x = ptr**;
- This is because while x is a variable, x is a constant. The location at which the first element of x will be stored cannot be changed once x[] has been declared. Therefore, an array name is often known to be a constant pointer.
- **Note: x[i], i[x], *(x+i), *(i+x)** gives the same value.

```
#include<stdio.h>
int main()
{
    int a[]={1,2,3,4,5,6};
    int *b=a;
    b=a++;  //error
    printf("%d\n",*b);
}
```

```
#include<stdio.h>
int main()
{
        int a[]={1,2,3,4,5,6};
        int *b=a;
        b=a+1;
        printf("%d\n",*b);
}
```

Pointer to Array :

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

```
int main()
{
int i;
int a[5]={1,2,3,4,5};
int *p = a;    // same as int *p = &a[0]
for(i =0; i <5; i++)
{
  printf("%d\n",*p);
  p++;
}
}
```

Output:

1
2
3
4
5

Pointer to an Array:

- We have seen that a pointer that pointed to the first element of array.
- We can also declare a pointer that can point to the whole array.
- For example int (*p)[10]; here p is point to an array of 10 integer.
- **Note** that it is necessary to enclose the **pointer name inside parentheses**.

```
int main()
{ int *p;
int x[5];
int (*pa) [5];
 p=x;
 pa=x;
printf(" p=%u,   pa=%d\n", p, pa);
 p++;
 pa++;
printf(" p=%u,   pa=%d\n", p, pa);
 }
```

Output:

P=3000, pa=3000
P=3002, pa=3010

Arrays Of Pointers

- An array of pointers can be declared as int *ptr[10];
- The above statement declares an array of 10 pointers where each of the pointer points to an integer variable.

```
#include <stdio.h>
int main()
 { int arr1[]={1,2,3,4,5};
int arr2[]={0,2,4,6,8};
int arr3[]={1,3,5,7,9};
int *parr[3] = {arr1, arr2, arr3};
int i;
for(i = 0;i<3;i++)
printf("%d", *parr[i]);
 return 0;
 }
```

Output 1 0 1

- parr[0] stores the base address of arr1 (or, &arr1[0]). So writing *parr[0] will print the value stored at &arr1[0]. Same is the case with *parr[1] and *parr[2]

Pointer to Two dimensional Array:

- In two dimensional arrays, we can access each element by using two subscripts, where first subscript represents row number and second subscript represents the column number.
- A two dimensional array is of form, x[i][j]. Lets see how we can make a pointer point to such an array.
- we know now, name of the array gives its base address.
- In x[i][j], x will give the base address of this array, that is the address of x[0][0] element.
- In the case of 2-D arrays, first element is a 1-D array, so the name of 2-D array represents a pointer to 1-D array.
- x[i] or (x+i) or *(x+i) point to i^{th} row of 1-D array of the base address where **x** is two dimension array.
- Individual elements of the array mat can be accessed using either:

  x[i][j] or *(*(x + i) + j) or *(x[i]+j);

```
main( )
{
    int s[4][2] = {{ 12, 56 }, { 12, 33 }, { 14, 80 }, { 13, 78 }} ;
     int i, j ;
     for ( i = 0 ; i <= 3 ; i++ )
     {
     printf("\n Address of %d th row one D array=%u  %u\n", i, s[i], *(s+i));
     printf ( "\n" ) ;
     for ( j = 0 ; j <= 1 ; j++ )
     printf ( "%d ", *( *( s + i ) + j ) ) ;
     }}
```

Output:

```
Address of  0th row one D array=2293400   229340012 56
Address of  1th row one D array=2293408    229340812 33
Address of  2th row one D array=2293416    229341614 80
Address of  3th row one D array=2293424    229342413 78
```

- The golden rule to access an element of a two-dimensional array can be given as arr[i][j] = (*(arr+i))[j] = *((*arr+i))+j) = *(arr[i]+j)
- Therefore, arr[0][0] = *(arr)[0] = *((*arr)+0) = *(arr[0]+0)

If we declare an array of pointers using, data_type *array_name[SIZE]; Here SIZE represents the number of rows and the space for columns that can be dynamically allocated	If we declare a pointer to an array using, data_type (*array_name)[SIZE]; Here SIZE represents the number of columns and the space for rows that may be dynamically allocated

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

Output 1 2 3 4

Dynamic Memory Allocation

- The process of allocating memory to the variables during execution of the program or at run time is known as dynamic memory allocation.
- C language has four library routines which allow this function.

1. **malloc():** malloc is declared in <**stdlib.h**> , so we include this header file in any program that calls malloc. The malloc **function reserves a block of memory of specified size** and returns a pointer of type void.

- This means that we can assign it to any type of pointer. It is used to dynamically allocate a single large block of memory with the specified size. it has initialized each block with the **default garbage value initially**.
- The general syntax of malloc() is

ptr =(cast-type*)malloc(byte-size);

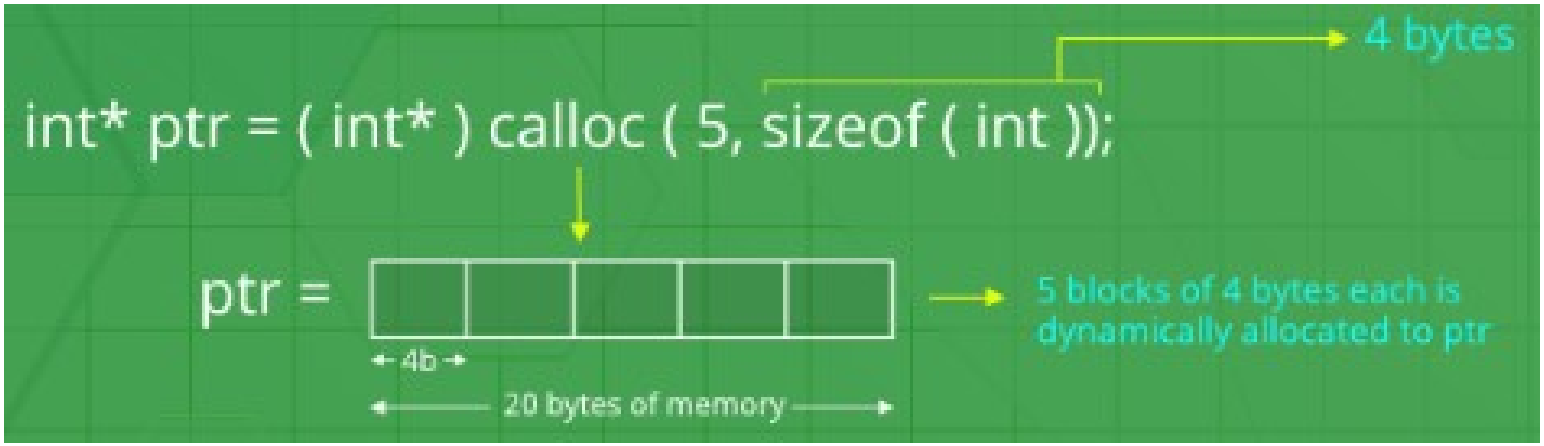
- where ptr is a pointer of type cast-type. malloc() returns a pointer (of cast type) to an area of memory with size byte-size.
- For example, **arr=(int*)malloc(10*sizeof(int));**

```
#include <stdio.h>
#include<stdlib.h>
 int main()
{ int i, n;
int *arr;
printf("\n Enter the number of elements ");
scanf("%d", &n);
arr = (int*)malloc(n * sizeof(int));
for(i=0;i <n;i++)
{
printf("\n Enter the value %d of the array: ", i);
scanf("%d",&arr[i]);
}
 printf("\n The array contains \n");
 for(i=0;i <n;i++)
printf("  %d ",arr[i]);
}
```

- **calloc():** calloc() function is another function that reserves memory at the run time. It is normally used to request **multiple blocks of storage each of the same size** and then **initializes all bytes to zero**. calloc() stands for contiguous memory allocation and is primarily used to allocate memory for arrays. The syntax of calloc() can be given as:

ptr=(cast-type*) calloc(n, elem-size);

- The above statement allocates contiguous space for n blocks each of size elem-size bytes.



```
#include <stdio.h>
#include<stdlib.h>
  int main()
{ int i, n;
int *arr;
printf("\n Enter the number of elements ");
scanf("%d", &n);
arr = (int*) calloc(n, sizeof(int));
for(i=0;i <n; i++)
{
printf("\n Enter the value %d of the array: ", i);
scanf("%d", &arr[i]);
}
 printf("\n The array contains \n");
 for(i=0;i <n;i++)
printf("  %d  ", arr[i]);
}
```

- **realloc() :**At times the memory allocated by using calloc() or malloc() might be insufficient or in excess. In both the situations we can always use realloc() to change the memory size already allocated by calloc() and malloc(). This process is called reallocation of memory.

ptr = realloc(ptr, newsize);

Releasing the Used Space(Free()): When a variable is allocated space during the compile time, then the memory used by that variable is automatically released by the system in accordance with its storage class. But when we dynamically allocate memory then it is our responsibility to release the space when it is not required.

- we must release that block of memory for future use, using the free function.

free(ptr);

```
#include <stdio.h>
#include <string.h>
#include<stdlib.h>
 int main()
{
 char *str;
str = (char *)malloc(4);
strcpy(str, "ram");
printf("\n STR = %s", str);
str = (char *)realloc(str,8);        /*Reallocation*/
printf("\n STR size modified\n");
printf("\n STR = %s\n", str);
printf("\n input next string \n");
gets(str);
printf("\n STR = %s", str);
  free(str);         /*freeing memory*/
 return 0;
}
```

output
STR = ram
STR size modified
STR = ram
input next string
ramayan
STR = ramayan

Returning array by passing an array which is to be returned as a parameter to the function

```
#include <stdio.h>
int *getarray(int *a)
{   printf("Enter the 5 elements in an array : ");
    for(int i=0;i<5;i++)
    {
        scanf("%d", &a[i]);
    }
    return a;
}
int main()
{
  int *n;
  int a[5];
  n=getarray(a);
  printf("\nElements of array are :");
  for(int i=0;i<5;i++)
    {
        printf("%d", n[i]);
    }
    return 0;  }
```

Output

```
Enter the elements in an array : 5
2
3
5
6
Elements of array are :52356
```

Returning array by passing an array which is to be returned as a parameter to the function and function as pointer because it return multiple value.

```
#include <stdio.h>
int  getarray(int *a)  //    replace it  int  *getarray(int *a)
{   printf("Enter the 5 elements in an array : ");
    for(int i=0;i<5;i++)
    {
        scanf("%d", &a[i]);
    }
    return a;
}
int main()
{
  int *n;
  int a[5];
  n=getarray(a);
  printf("\nElements of array are :");
  for(int i=0;i<5;i++)
    {
        printf("%d", n[i]);
    }
    return 0;  }
```

Output with warning if we do not use function as pointer:

returning 'int *' from a function with return type 'int' makes integer from pointer without a cast

Enter the elements in an array : 5

2

3

5

6

Elements of array are :52356

How to dynamically allocate a 2D array in C

1) **Using a single pointer and a 1D array with pointer arithmetic:** A simple way is to allocate a memory block of size r*c and access its elements using simple pointer arithmetic.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{    int r = 3, c = 4;
     int* ptr = (int *)malloc((r * c) * sizeof(int));
                          /* Putting 1 to 12 in the 1D array in a sequence */
     for (int i = 0; i < r * c; i++)
       ptr[i] = i + 1;
                          /* Accessing the array values as if it was a 2D array */
    for (int i = 0; i < r; i++) {
       for (int j = 0; j < c; j++)
           printf("%d ", ptr[i * c + j]);
       printf("\n");
   }
   free(ptr);
   return 0;
}
```

2) Using an array of pointers

- We can create an array of pointers of size r. Note that from C99, C language allows variable sized arrays. After creating an array of pointers, we can dynamically allocate memory for every row.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{   int r = 3, c = 4, i, j, count;
    int* arr[r];
    for (i = 0; i < r; i++)
     arr[i] = (int*)malloc(c * sizeof(int));   // Note that arr[i][j] is same as *(*(arr+i)+j)
    count = 0;
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            arr[i][j] = ++count;     // Or *(*(arr+i)+j) = ++count
    for (i = 0; i < r; i++)
    {      for (j = 0; j < c; j++)
            printf("%d ", arr[i][j]);
            printf("\n");
    }
    for (int i = 0; i < r; i++)
        free(arr[i]);
    return 0;
}
```

3) Using pointer to a pointer

- We can create an array of pointers also dynamically using a double pointer. Once we have an array pointers allocated dynamically, we can dynamically allocate memory and for every row like method 2.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{    int r = 3, c = 4, i, j, count;
     int** arr = (int**)malloc(r * sizeof(int*));
    for (i = 0; i < r; i++)
        arr[i] = (int*)malloc(c * sizeof(int));   // Note that arr[i][j] is same as *(*(arr+i)+j)
    count = 0;
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            arr[i][j] = ++count;    // OR *(*(arr+i)+j) = ++count
    for (i = 0; i < r; i++)
    {   for (j = 0; j < c; j++)
            printf("%d ", arr[i][j]);
           printf("\n");
    }
    for (int i = 0; i < r; i++)
        free(arr[i]);
    free(arr);
    return 0;
}
```

Pointers and Character Strings:

- The string in C programming language is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus a **null-terminated** Pointer can also be used to create strings. Pointer variables of char type are treated as string.

  char *str="Hello";

- The above code creates a string and stores its address in the pointer variable str. The pointer str now points to the first character of the string "Hello". Another important thing to note here is that the string created using char pointer can be assigned a value at **runtime**.

```
 #include <stdio.h>
int main()
{
char str[] = "Hello";
char *pstr;
pstr = str;
printf("\n The string is : ");
while(*pstr != '\0')
{
printf("%c", *pstr);
 pstr++;
}
 return 0;
}
```

Output:

Greeting message:=Hello

```
int main()
 {
char *x[]={"Hello","World"};
printf("The Array of String is = %s,
%s \n", x[0], x[1]);
}
```

Output:

The Array of String is = Hello, World

Dangling Pointers in C

- The most common **bugs related to pointers** and memory management **is dangling/wild pointers**. Sometimes the programmer fails to initialize the pointer with a valid address, then this type of initialized pointer is known as a dangling pointer in C.
- Dangling pointer occurs at the time of the object destruction when the **object is deleted or de-allocated from memory without modifying the value of the pointer**.
- In this case, the pointer is pointing to the memory, which is de-allocated.
- The dangling pointer can point to the memory, which contains either the program code or the code of the operating system.
- If we assign the value to this pointer, then it overwrites the value of the program code or operating system instructions; in such cases, the program will show the undesirable result or may even crash.
- If the memory is re-allocated to some other process, then we dereference the dangling pointer will cause the segmentation faults.

Using free() function to de-allocate the memory.

```
#include <stdio.h>
#include<stdlib.h>
int main()
{
   int *ptr=(int *)malloc(sizeof(int));
   int a=560;
   ptr=&a;
   free(ptr);
   printf("%d",*ptr);
   return 0;
}
```

- In the above code, we have created two variables, i.e., *ptr and a where 'ptr' is a pointer and 'a' is a integer variable.
- we know that **malloc()** function returns void, so we use int * to convert void pointer into int pointer.
- The statement **int *ptr=(int *)malloc(sizeof(int));** will allocate the memory with 4 bytes

- The statement **free(ptr)** de-allocates the memory, then **'ptr' pointer becomes dangling as it is pointing to the de-allocated memory.**
- If **we assign the NULL value to the 'ptr', then 'ptr' will not point to the deleted memory.** Therefore, we can say that ptr is not a dangling pointer.
- **Note:** When there is a **pointer pointing to a memory address of any variable, but after some time the variable was deleted from the memory location while keeping the pointer pointing to that location is known as a dangling pointer in C.**

Variable goes out of the scope

- When the variable goes out of the scope then the pointer pointing to the variable becomes a **dangling pointer.**

```
#include<stdio.h>
int main()
{
   char *str;
   {
      char a = 'A';
      str = &a;
   }                              // a falls out of scope
      printf("%s", *str);      // str is now a dangling pointer
}
```

- In the inner scope, we declare a character variable. The str pointer contains the address of the variable 'a'.
- When the control comes out of the inner scope, 'a' variable will no longer be available, so str points to the de-allocated memory. It means that the str pointer becomes the dangling pointer.

Exercise

1. Write a program in C to find the maximum number between two numbers using a pointer.
2. Write a program in C to find the largest element using Dynamic Memory Allocation.
3. Write a program in C to find the factorial of a given number using pointers.
4. Write a program to find out whether a given string is palindrome or not using pointers.
5. Write a C program to swap two arrays using pointers.
6. Write a C program to add two matrix using pointers.
7. Write a C program to multiply two matrix using pointers.
8. Write a C program to compare two strings using pointers.
9. Write a C program to sort array using pointers.
10. Write a program in C to store n elements in an array and print smallest elements using pointer.
11. Write a program in C to compute the sum of all elements in an array using pointers.

Structure & Union

- Structure is a user-defined data type. It allows us to combine the different data types under a single name for better handling. Structure represent a record such as student record which contain name of student, city of student, roll number age and date of birth. Keyword struct is use to define a structure. The syntax of a struct is as following-

```
struct  structure_name
{
 Statements
};
```

Example of Structure:

```
struct Student
{
 char[20] name;
 int rollno;
 int age;
};
```

Declaring Structure Variables

- Members of the structure cannot be accessed directly.
- To access the member of a structure within a program, a variable has to be declared. The following format is used.

```
struct book
{
char title[20];
char author[15];
float price;
};
struct book book1, book2, book3;
```

- The other way to declare a structure variable is to combine both the structure definition and variables declaration in one statement.

```
struct book
{
char title[20];
char author[15];
float price;
} book1, book2, book3;
```

Accessing Structure Members

The individual members of a structure can be accessed through the structure variable only. The link between a member and a variable is established through the **operator '.'** is called as the dot operator or member operator or period operator. The syntax is

Structure variable. member name

For example,

```
struct book
{
char title[20];
char author[15];
float price;
} b1, b2, b3;

b1.price;
b2.author;
b1.title;
```

Assigning Values to the Members

Members of the structure can be assigned the values as given below.

strcpy(b1.title, " Java Programming");

b1. author = "raj";

b1.price = 275.00;

scanf can also be used to give values through the keyboard at Run Time.

scanf("%s", b1.title);

scanf("%s", b1.author);

scanf("%f", &b1. price);

Structure Initialization

- Like any other data type, a structure variable can be initialized at compile time. The general format for structure initialization is

```
struct time
{
int hrs;
int mins;
int secs;
}t1 = {4, 52, 29},t2 = {10, 40, 21};
```

or

```
struct time t1 = {4, 52, 29};
struct time t2 = {10, 40, 21};
```

```
main()
{
struct student
{
int roll_no;
char name[10];
int age;
}s;
printf("Enter roll, name and age: ");
scanf("%d %s %d", &s.roll_no,
s.name, &s.age);
printf("\nEntered information: \n");
printf("Roll number: %d", s.roll_no);
printf("\nName: %s", s.name);
printf("\nAge: %d",s.age);
}
```

Initialization individual Structure member

```
int main (void)
{
struct date
{
int month;
int day;
int year;
};
struct date today;
today.month = 4;
today.day = 25;
today.year = 2009;
printf ("Today's date is %i / %i / %.2i \n", today.month, today.day,
today.year % 100);
return 0;
}     // %.2d or %.2i if output one digit then add prefix one digit Zero.
 output
Today's date is 4 / 25 / 09
```

Array of Structures

- A better approach would be to use an array of structures. We can declare array of structures. Example **struct student s[10];**

```
 int main( )
{
struct book
{ char name[20] ;
  float price ;
  int pages ;
} ;
struct book b[3] ;
```

```
int i ;
for ( i = 0 ; i <= 2 ; i++ )
{
printf ( "\nEnter name, price and pages " ) ;
scanf ( "%s %f %d", b[i].name, &b[i].price, &b[i].pages ) ;
}
for ( i = 0 ; i <= 2 ; i++ )
printf ( "\n%s%f %d", b[i].name, b[i].price, b[i].pages ) ;
return 0;
}
```

- The above program may not work correctly and will give the message **"floating point formats not linked"** at run time. This problem occurs because the floating point formats (**for scanf() and other related functions**) are not always linked, to reduce the size of executable file. **Borland C suggests that to avoid the above problem is insert a definition of function like this.**

```
Void dummy()
{ float x, *ptr=&x;}
```

```
void dummy()
{ float x, *ptr=&x;
}
struct student
{
char name[20];
int rollno;
float marks;
}s[2];
void main()
{  int i;
 for(i=0;i<2;i++)
  {  printf(" enter name, rollno, and marks "\n);
   scanf("%s %d %f", s[i].name, &s[i].rollno, &s[i].marks);
  }
  printf("\nThe name, rollno, and marks=");
  for(i=0;i<2;i++)
{
  printf("\n%s %d %f", s[i].name, s[i].rollno, s[i].marks);
  }
getch();
}
```

```
int main( )
{
struct book
{ char name[20] ;
  float price ;
  int pages ;
} ;    struct book b[2]={ {“math”, 222.5, 120},{“chemistry”, 350.22, 320}};
       int i ;
       for ( i = 0 ; i <2 ; i++ )
       printf ( "\n%s%f %d", b[i].name, b[i].price, b[i].pages ) ;
  }
```

Advantages of Structure over Array:

- The structure can store different types of data whereas an array can only store similar data types.
- Structure does not have limited size like an array.
- Structure elements may or may not be stored in contiguous locations but array elements are stored in contiguous locations.
- In structures, object instantiation is possible whereas in arrays objects are not possible.

Pointer to structures

- We can have a pointer to structure which can point to the starting address of a structure variable. These pointers are called structure pointers.
- There are two ways of accessing the members of structure through the structure pointer . Let ptr is a pointer to a structure, to access members of structure variable as

(*ptr). Variblename

- Here parentheses are necessary because dot operator has higher precedence than * operator.
- We can use the arrow operator (->) for accessing structure members through pointer as

ptr-> Variblename

The arrow operator has same precedence as that of dot operator.

Example
```
struct student{
        char name[20];
        int rollno;
        } ;
struct student s, *ptr;
        ptr=&s;
```
(*ptr).name or ptr->name

```
#include<stdio.h>
main( )
{
struct employee
{
char name[10] ;
int age ;
float salary ;
} *ptr;
struct employee e1 = { "Sanjay", 30, 5500.50 } ;
    //struct employee *ptr ;
ptr=&e1;
printf( "\n%s %d %f", ptr->name, ptr->age, ptr->salary ) ;
printf ( "\n%s %d %f", (*ptr).name, (*ptr).age, (*ptr).salary ) ;
}
```

Pointer within structure

- Pointer can also be used as a member of structure . To access value pointed to by the member as pointer that in structure as

structure_variable . Member_of_pointer_variable

or

structure_pointer_variable -> Member_of_pointer_variable

```
#include<stdio.h>
main( )
{
struct employee
{
char *name ;
int age ;
} *ptr;
struct employee e1={"ram", 25};
ptr=&e1;
printf( "\n%s %d ", ptr->name, ptr->age ) ;
printf ( "\n%s %d", e1.name, e1.age ) ;
}
```

Additional Features of Structures

- The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator.

```
 main( )
{
struct employee
{
char name[10] ;
int age ;
float salary ;
} ;
struct employee e1 = { "Sanjay", 30, 5500.50 } ;
struct employee e2, e3 ;
e2=e1;
e3 = e1 ;              /* copying all elements at one go */
printf ( "\n%s %d %f", e1.name, e1.age, e1.salary ) ;
printf ( "\n%s %d %f", e2.name, e2.age, e2.salary ) ;
printf ( "\n%s %d %f", e3.name, e3.age, e3.salary ) ;
}
```

Additional Features of Structures (nested structures)

- One structure can be nested within another structure. Using this facility complex data types can be created. The following program shows nested structures.

```
main( )
{
struct student
{
char phone[15] ;
char city[25] ;
int pin ;
} ;
struct emp
{
char name[25] ;
struct student s1;
} ;
struct emp s2 = { "raj", "531046", "nagpur", 10 };
printf ( "\nname = %s phone = %s", s2.name, s2.s1.phone ) ;
printf ( "\ncity = %s pin = %d", s2.s1.city, s2.s1.pin ) ;
}
```

```
struct address
{
 char address1  [40];
 char city[25] ;
 int pin ;
  struct emp
   {
     char name[25] ;
     char phone[15] ;
    } s1;
}s2;
```

WAP to read and display the information of a student using a nested structure

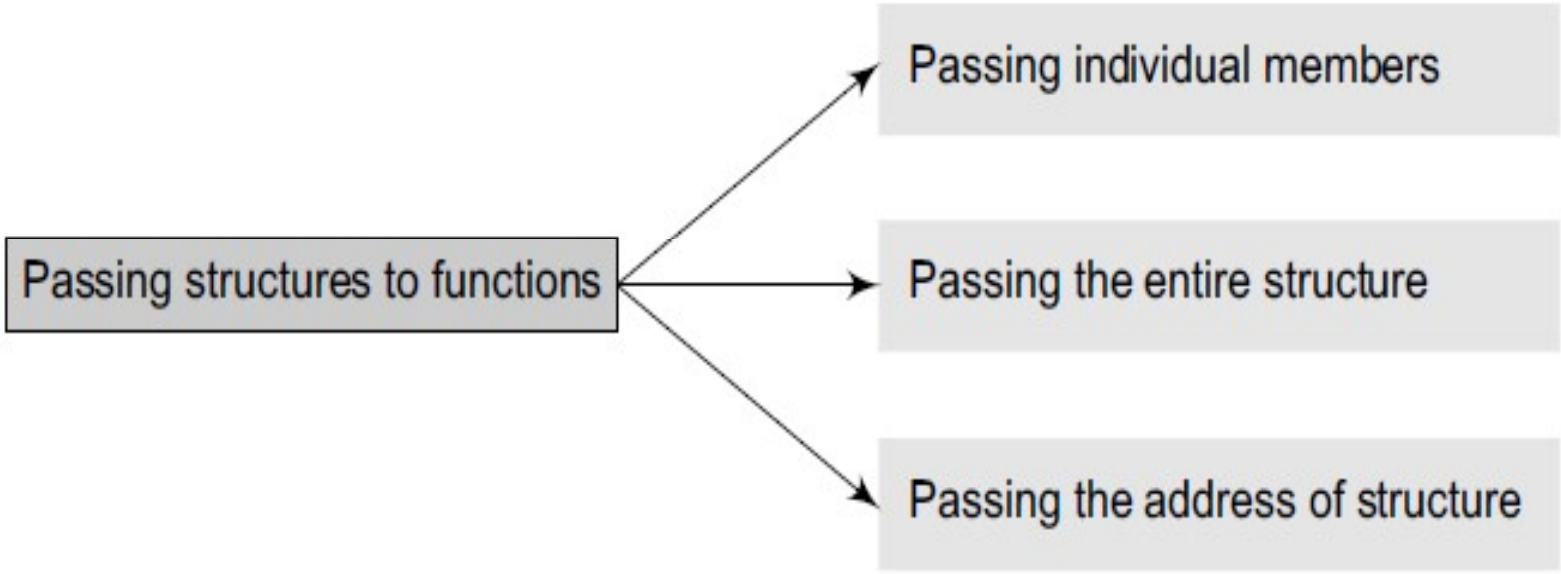
```
void main()  {
struct DOB
{   int day;
    int month;
    int year;
};
struct student
{        char name[100];
          int roll_no;
         struct DOB d1;
};
struct student s1;
clrscr();
printf("\n Enter the name  and roll number : ");
scanf("%s  %d", s1.name, &s1.roll_no);
printf("\n Enter the DOB : ");
scanf("%d %d %d", &s1.d1.day, &s1.d1.month, &s1.d1.year);
printf("\n NAME = %s  ROLL No. = %d", s1.name, s1.roll_no);
printf("\n DOB = %d – %d – %d", s1.d1.day, s1.d1.month, s1.d1.year);
getch();
}
```

WAP to read and display the information of a student using a nested structure

```
#include <stdio.h>
#include <string.h>
struct Employee
{    int id;
   char name[20];
   struct Date
    {  int dd;
       int mm;
       int yyyy;
    }doj;
}e1;
int main( )
{   e1.id=101;     //storing employee information
   strcpy(e1.name, "Sonoo Jaiswal");  //copying string into char array
   e1.doj.dd=10;
   e1.doj.mm=11;
   e1.doj.yyyy=2014;
    printf( "employee id : %d\n", e1.id);
   printf( "employee name : %s\n", e1.name);
   printf( ”date of joining (dd/mm/yyyy) : %d/%d/%d\n", e1.doj.dd,e1.doj.mm,e1.
doj.yyyy);
   return 0;
}
```

STRUCTURES AND FUNCTIONS

- we must have a mechanism to pass them to functions and return them. A function may access the members of a structure in three ways as shown in Fig.



Additional Features of Structures (Passing individual elements)

- Like an ordinary variable, a structure variable can also be passed to a function. We may either pass individual structure elements or the entire structure variable at one go.

```
struct POINT
{
int x;
int y;
};
void display(int, int);
int main()
{
struct POINT p1 ;                    //= {2, 3};
printf("input the value of x and y");
scanf("%d%d", &p1.x, &p1.y);
display(p1.x, p1.y);
return 0;
}
void display(int a, int b)
{
printf(" The coordinates of the point are: %d %d", a, b);
}
```

(Passing individual elements)

```
 void display ( char *, char *, int  );
int main( )
{
struct book
{
char name[25] ;
char author[25] ;
int callno ;
} ;
struct book  b1 = { "Let us C", "YPK", 101 } ;
display ( b1.name, b1.author, b1.callno ) ;
}
void display ( char *s, char *t, int n )
{
printf ( "\n%s %s %d", s, t, n ) ;
}
```

Additional Features of Structures (Passing the Structure variable)

- A better way would be to pass the entire structure variable at a time. This method is shown in the following program.

```
struct book
{
char name[25] ;
char author[25] ;
int callno ;
} ;
void display( struct book b1  );
main( )
{
struct book  b1 = { "Let us C", "YPK", 101 } ;
display ( b1 ) ;
}
void display ( struct book  b )
{
printf ( "\n%s %s %d", b.name, b.author, b.callno ) ;
}
```

Additional Features of Structures

- we can pass the address of a structure variable to a function. note that to access the structure elements using pointer to a structure we have to use the '->' **operator.**

```
/* Passing address of a structure variable */
struct book
{
char name[25] ;
char author[25] ;
int callno ;
} ;
void display ( struct book *b );
main( )
{
struct book b1 = { "Let us C", "YPK", 101 } ;
display ( &b1 ) ;
}
void display ( struct book *b )
{
printf ( "\n%s %s %d", b->name, b->author, b->callno ) ;
}
```

Use of Typedef with Structures

- A **typedef** is a keyword used in C programming to define a new name for exiting data types. But it cannot provide a new data type to the predefined data type. It behaves similarly as we define the alias for the commands.

  typedef <existing_name> <alias_name>

- Example **typedef** unsigned **int** unit;

  unit a, b;

```
struct student
{
char name[20];
int age;
};
typedef struct student p;
p s1, s2;
```

```
#include <stdio.h>
typedef int Length; // typedef provide the int data type to a new name as Length
int main()
{
Length num1, num2, sum; // here Length variable is treated as the int data type
printf(" Enter the first number:");
scanf(" %d", & num1);
printf(" Enter the second number:");
scanf(" %d", & num2);
sum = num1 + num2;
printf(" The sum of the two numbers is: %d", sum);
return 0;
}
```

Typedef struct

- *Putting it simply, typedef means we no longer have to write struct every time in our code. Hence it makes our code look clean and readable. It also makes the code portable and easily manageable.*
- When defining structs, it is more convenient to use typedefs.
- When declaring a struct variable, you must prefix the struct name with the keyword "struct“.
- Writing a struct every time you use a struct is a bit of a hassle.
- **Advantages of Typedef**
- Easy to read source code - With typedef You can make the source code easier to read by creating a type that is easy to understand.
- Easier to write types - typedef by shortening the new type name, it becomes easier to write that type.
- It help in effective documentation of the program thus increasing its clarity.
- Note that typedef statement does not occupy any memory; it simply defines a new type.

Example

```
#include<stdio.h>
#include <string.h>
typedef  struct employee
{
    char name[50];
    int salary;
}emp;   // emp is name of new  type. The type emp does not require the struct
          keyword
int main( )
{
    emp e1;
    printf("\nEmployee name:\t");
    scanf("%s", e1.name);
    printf("\nEnter Employee salary: \t");
    scanf("%d", &e1.salary);
    printf("\nstudent name is %s", e1.name);
    printf("\nroll is %d", e1.salary);
    return 0;
}
```

Example

```
#include <stdio.h>
#include <string.h>
 struct student_str                          //normal structure declaration
{               char name[30];
                int age;
};
typedef struct {                            //structur declaration with typedef
                char name[30];
                int age;
}emp_str;
int main()
{               struct student_str std;
                        emp_str emp;
                strcpy(std.name, "Amit Shukla");     //assign values to std
                std.age = 21;
                strcpy(emp.name, "Abhishek Jain");      //assign values to emp
                emp.age = 27;
                printf("Student detail:\n");
                printf("Name: %s\n",std.name);
                printf("Age: %d\n",std.age);
                printf("Employee detail:\n");
                printf("Name: %s\n",emp.name);
                printf("Age: %d\n",emp.age);
                        return 0;
}
```

SELF-REFERENTIAL STRUCTURES

- Self-referential structures are those structures that contain a reference to the data of its same type. That is, a self-referential structure, in addition to other data, contains a pointer to a data that is of the same type as that of the structure. For example,

```
struct node
 {
 int val;
struct node *next;
 };
```

- Here, the structure node will contain two types of data: an integer val and a pointer next. You must be wondering why we need such a structure. Actually, self-referential structure is the foundation of other data structures.

Uses of Structures

- Where are structures useful? The immediate application that comes to the mind is Database Management. That is, to maintain data about employees in an organization, books in a library, items in a store, financial accounting transactions in a company etc. They can be used for a variety of purposes like:

1. Changing the size of the cursor
2. Clearing the contents of the screen
3. Placing the cursor at an appropriate position on screen
4. Drawing any graphics shape on the screen
5. Receiving a key from the keyboard
6. Checking the memory size of the computer
7. Formatting a floppy
8. Hiding a file from the directory
9. Displaying the directory of a disk
10. Sending the output to printer
11. Interacting with the mouse

Union

- A **union** is a derived data type available in C like structure and it also contain member of different data type and allow us to store them in the same memory location. We can define a union with multiple members, but **only one member can contain a value at a time.** It also provides an efficient way of using the same memory location for multiple-purpose. The syntax of union are as follows-

```
union number
 {
   int x;
   float y;
   char str[15];
};
```

So the memory size occupied by the number union will be 15 byte but if we take a structure of same type it take 21 byte of memory.

Initializing Unions

The difference between a structure and a union is that in case of a union, the fields share the same memory space, so new data replaces any existing data.

```
#include<stdio.h>
#include <string.h>
union Data
{
int i;
float f;
char str[20];
};
int main( )
{
union Data d;
d.i = 10;
printf( "d.i : %d\n", d.i);
d.f = 220.5;
printf( "d.f : %f\n", d.f);
strcpy( d.str, "C Programming");
printf( "d.str : %s\n", d.str);
return 0;  }
```

Initializing Unions

```
typedef struct POINT1
{
int x, y;
};
typedef union POINT2
{
int x;
int y;
};
int main()
{
POINT1 P1 = {2,3};      /*POINT2 P2 ={4,5}; Illegal in case of unions*/
POINT2 P2;
P2.x = 4;
P2.y = 5;
printf("\n The coordinates of P1 are %d and %d", P1.x, P1.y);
printf("\n The coordinates of P2 are %d and %d", P2.x, P2.y);
return 0;
}
```

Output

The coordinates of P1 are 2 and 3

The coordinates of P2 are 5 and 5

Difference between Structure & Union

Structure	Union
In structure **each member get separate space in memory.** struct Student { char[15] name; int rollno; int age; }s1; Therefore total memory required for structure is equal to the sum of the size of all the members that is 19 byte.	But in union, the total memory space is equal to the member with **largest size** and all other members of union **share the same memory space.** This is the main difference between structure and union. union student { char[15] name; int rollno; int age; }s1; Therefore total memory required for structure union is 15bytes.
We can access any member in any sequence.	We can access only that variable whose value is recently used.

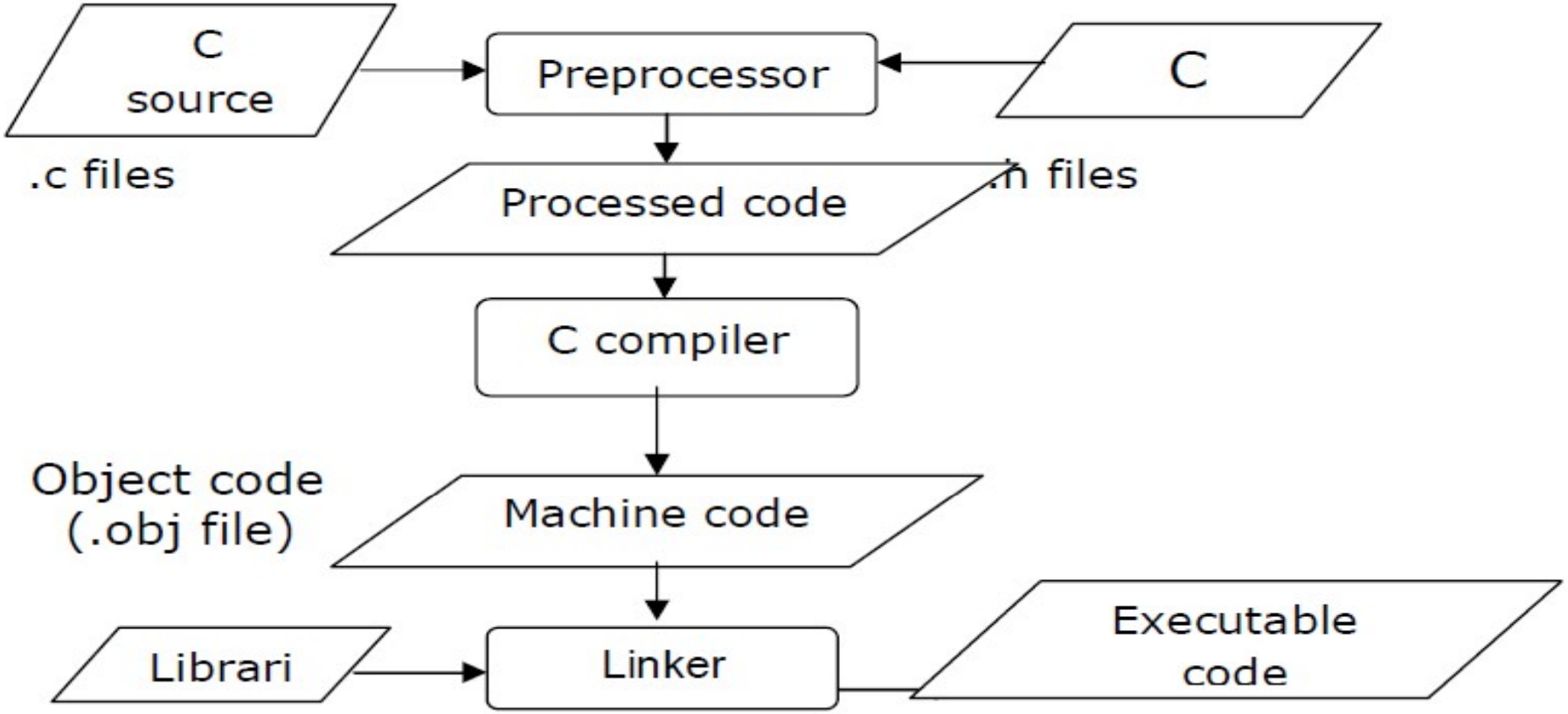
Exercise

1. Develop a program in c using structures to read the following information from the keyboard.
    1. employee name
    2. employee code
    3. designation
    4. age
2. Create a structure to specify data of customers in a bank. The data to be stored is: Account number, Name, Balance in account. Assume maximum of 200 customers in the bank.
3. Write a program to create a book structure having name, author, page and price.
4. Write a program to display Employee Record Keeping System using nested structure.
5. Write a program to find minimum and maximum from an array using structure.
6. Write a C Program to Calculate Difference Between Two Time Periods.

Preprocessors

- A unique feature of c language is the preprocessor. A program can use the tools provided by **preprocessor to make his program easy to read, modify, portable and more efficient.**
- preprocessor is a program that processes our program before it is passed to the compiler.
- The C preprocessor is a collection of special statements, called directives that are executed at the beginning of the compilation process.
- Preprocessor directives follow the special syntax rules that are listed below.
  - ***Executed by the pre-processor.***
  - ***Occurs before a program is compiled.***
  - ***Begin with #.***
  - ***Would not end with semicolon.***
  - ***Can be placed anywhere in the program.***
  - ***Normally placed at the beginning of the program or before any particular function.***

The compilation process can be diagrammatically given as below.



We would learn the following preprocessor directives here:

(a) Macro expansion

(b) File inclusion

(c) Conditional Compilation

(d) Miscellaneous directives

C Macros

- A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive. There are two types of macros:

1. Object-like Macros
2. Function-like Macros

Object-like Macros : The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants. For example:

#define PI 3.14

Here, PI is the macro name which will be replaced by the value 3.14.

- **Function-like Macros:** The function-like macro looks like function call. For example:

#define MIN(a,b) ((a)<(b)?(a):(b))

Here, MIN is the macro name.

Macros versus Functions

- In a macro call the preprocessor replaces the macro template with its macro expansion, in a stupid, unthinking, literal way.
- As against this, in a function call the control is passed to a function along with certain arguments, some calculations are performed in the function and a useful value is returned back from the function.
- This brings us to a question: when is it best to use macros with arguments and when is it better to use a function?
- ***Usually macros make the program run faster but increase the program size,*** *whereas* ***functions make the program smaller and compact.***

Macro Expansion

```
#include <stdio.h>
#define UPPER 25
main( )
{
int i ;
for ( i = 1 ; i <= UPPER ; i++ )
printf ( "\n%d", i ) ;
}
```

- In this program instead of writing 25 in the **for loop we are writing** it in the form of UPPER, which has already been defined before **main( ) through the statement,**
  #define UPPER 25
- This statement is called '**macro definition**' or more commonly, just a '**macro**'.

- using **#define can produce more efficient and more easily** understandable programs.
- A **#define** directive is many a times used to define operators as shown below.

```
#define AND &&
#define OR ||
main( )
{
int f = 1, x = 4, y = 90 ;
if ( ( f < 5 ) AND ( x <= 20 OR y <= 45 ) )
printf ( "\nYour PC will always work fine..." ) ;
else
printf ( "\nIn front of the maintenance man" ) ;
}
```

- A **#define directive could be used even to replace a condition, as** shown below.

```
#define AND &&
#define ARANGE ( a > 25 AND a < 50 )
main( )
{
int a = 30 ;
if ( ARANGE )
printf ( "within range" ) ;
else
printf ( "out of range" ) ;
}
```

Macros with Arguments

- The macros that we have used so far are called simple macros. Macros can have arguments, just as functions can.

```
#define AREA(x) ( 3.14 * x * x )
main( )
{
float r1 = 6.25, r2 = 2.5, a ;
a = AREA ( r1 ) ;
printf ( "\nArea of circle = %f ", a ) ;
a = AREA ( r2 ) ;
printf ( "\nArea of circle = %f ", a ) ;
}
```

Here's the output

```
Area of circle = 122.656250
Area of circle = 19.625000
```

File Inclusion

- This directive causes one file to be included in another. The preprocessor command for file inclusion looks like this:

  #include "filename "

- If we have a very large program, the code is best divided into several different files, each containing a set of related functions. It is a good programming practice to keep different sections of a large program separate. These files are **#included** at the beginning of main program file.
- Actually there exist two ways to write **#include statement.** These are:

#include "filename"

#include <filename>

- The meaning of each of these forms is given below:

#include "goto.c "

➢ This command would look for the file **goto.c in the current directory** as well as the specified list of directories as mentioned in the include search path that might have been set up.

#include <goto.c>

➢ This command would look for the file **goto.c in the specified list of directories only**.

Conditional Compilation

- if we want, have the compiler skip over part of a source code by inserting the preprocessing commands **#if and #endif,** which have the general form:

```
#if  macroname
statement 1 ;
statement 2 ;
#endif
```

If **macroname has been #defined, the block of code will be** executed as usual; other wise not.

```
#include <stdio.h>
#include <conio.h>
 #define NUMBER  0
void main()
{
#if NUMBER==0
printf("Value of Number is: %d", NUMBER);
#endif
getch();
}
```

#else

- The #else preprocessor directive evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif , #ifdef and #ifndef directives.

- **Syntax:**

```
#if expression
Staement 1
 #else
 Staement 2
 #endif
```

Syntax with #elif

```
#if expression
    Staement 1
#elif expression
    Staement 2
#else
    Staement 2
#endif
```

Example

```
#include <stdio.h>
#include <conio.h>
#define NUMBER 1
void main()
{
 #if NUMBER==0
printf("Value of Number is: %d",NUMBER);
#else
printf("Value of Number is non-zero");
#endif
getch();
}
```

Miscellaneous Directives

- There are two more preprocessor directives available, though they are not very commonly used. They are:

(a) #undef

(b) #pragma

#undef

- To undefine a macro means to cancel its definition. This is done with the **#undef** directive.

Syntax: **#undef token**

```
#include <stdio.h>
#define PI 3.1415
#undef PI
main()
{
  printf("%f", PI);
}
```

Output: Compile Time Error: 'PI' undeclared

#pragma

#pragma startup and #pragma exit: These directives allow us to specify functions that are called upon **program startup (before main( )) or program exit (just before the program terminates)**. Their usage is as follows:

- **Syntax: #pragma token**

```
#include<stdio.h>
#include<conio.h>
void func() ;
#pragma startup func
#pragma exit func
void main()
{  printf("\nI am in main");
   getch();
 }
void func()
 { printf("\nI am in func");
   getch();
  }
Output:  I am in func
         I am in main
         I am in func
```

Stringize (#)

- The stringize or number-sign operator ('#'), when used within a macro definition, *converts a macro parameter into a string constant.* This operator may be used only in a macro that has a specified argument or parameter list. For example:

```
/*P13.12 Program to understand the use of stringizing operator*/
#include<stdio.h>
#define SHOW(var,format)  printf(#var"=%"#format"\n",var);
int main(void)
{           int x=9;
            float y=2.5;
            char z='$';
            SHOW(x,d);
            SHOW(y,0.2f);
            SHOW(z,c);
            SHOW(x*y,0.2f);
            return 0;
}
```

output
x=9
y=2.50
z=$
x*y=22.50

How to write a running C code without main()

- Write a C language code that prints **Raj Singh** without any main function. Logically it's seems impossible to write a C program without using a main() function. Since every program must have a main() function because:-
- It's an entry point of every C/C++ program.
- All Predefined and User-defined Functions are called directly or indirectly through the main.
- Therefore we will use preprocessor directive #define with arguments to give an impression that the program runs without main. But in reality it runs with a hidden main function. Let's see how the preprocessor doing their job:-
- Hence it can be solved in following ways:-

```
#include<stdio.h>
#define fun main
int fun(void)
{    printf("Raj Singh");
   return 0;
}
```

Token Pasting (##)

- The token-pasting operator (##) within a macro definition **combines two arguments**. It permits two separate tokens in the macro definition to be **joined into a single token**. For example:

```
/*P13.13 Program to understand the use of token pasting operator*/
#include<stdio.h>
#define  PASTE(a,b) a##b
#define MARKS(subject) marks_##subject
int main(void)
{
        int k2=14,k3=25 ;
        int marks_chem=89,marks_maths=98;
        printf("%d   %d  ",PASTE(k,2),PASTE(k,3));
        printf("%d   %d\n",MARKS(chem),MARKS(maths));
        return 0;
}
```

Output

14 25 89 98

Bitwise Operators

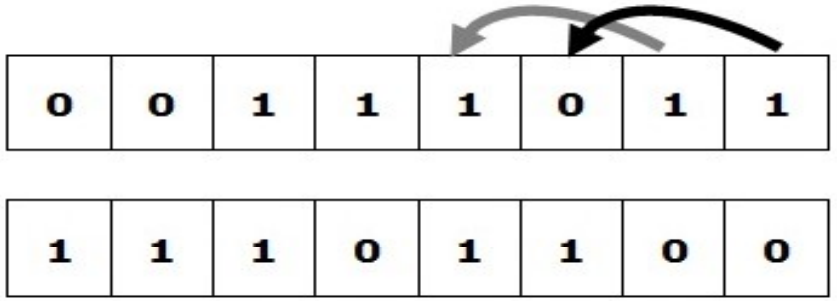
Bitwise operator works on bits and performs bit-by-bit operation.

Operator	Meaning	Remark
&	Bitwise AND	101 & 010 = 000
|	Bitwise OR	101 | 010 = 111
^	Bitwise EX-OR	111 ^ 010 = 101
<<	Shift left	<<100 = 001
>>	Shift right	>>100 = 010
~	Ones complement	~101 = 010

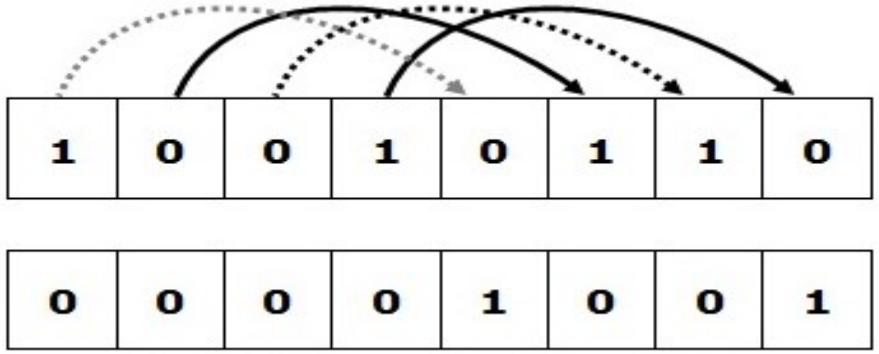
Bitwise Shift Operators (<<, >>)

- There are two shift operators – **Left shift** and **Right shift**. These operators shift the bits by the corresponding value, in other word's move the bits. The sign << for left shift and >> for right shift.

For example C = A << 2; // left shift A by 2



For example D = B >> 4; // right shift B by 4



```
#include <stdio.h>   /*BITWISE*/
#include<conio.h>
void main()
{
  unsigned int a = 60;                    /* 60 = 0011 1100 */
  unsigned int b = 13;                    /* 13 = 0000 1101 */
  int c = 0;
  c = a & b;                              /* 12 = 0000 1100 */
  printf("Line 1 - Value of c is %d\n", c );
  c = a | b;                              /* 61 = 0011 1101 */
  printf("Line 2 - Value of c is %d\n", c );
  c = a ^ b;                              /* 49 = 0011 0001 */
  printf("Line 3 - Value of c is %d\n", c );
  c = ~a;                                 /*-61 = 1100 0011 */
  printf("Line 4 - Value of c is %d\n", c );
  c = a << 2;                             /* 240 = 1111 0000 */
  printf("Line 5 - Value of c is %d\n", c );
  c = a >> 2;                             /* 15 = 0000 1111 */
  printf("Line 6 - Value of c is %d\n", c );
  getch();
}
```

Masking

Masking is an operation in which we can selectively mask or filter the bits of a variable, such that some bits are to keep/change/remove a desired part of information.

Masking using Bitwise AND: More often in practice bits are "masked off" (or masked to 0) than "masked on" (or masked to 1). When a bit is ANDed with a 0, the result is always 0, i.e. Y AND 0 = 0. To leave the other bits as they were originally, they can be ANDed with 1, since Y AND 1 = Y.

```
  10100101 10100101
  00001111 00001111 (AND mask bits)
= 00000101 00000101
```

ch	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
mask	0	0	0	0	1	0	0	0
result = ch & mask	0	0	0	0	b_3	0	0	0

Masking using Bitwise OR: the principle of OR masking is that Y OR 1 = 1 and Y O R 0 = Y. Therefore, to make sure a bit is on, OR can be used with a 1. To leave a bit unchanged, OR is used with a 0.

```
  10010101 10100101
  11110000 11110000  (OR masking bits)
= 11110101 11110101
```

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
ch	1	0	0	0	0	0	1	1
mask	0	0	0	0	1	0	0	0
ch | mask	1	0	0	0	1	0	1	1

XOR masking

- This can be achieved using the XOR (exclusive or) operation. XOR returns 1 if and only if an odd number of bits are 1. Therefore, if two corresponding bits are 1, the result will be a 0, but if only one of them is 1, the result will be 1. Therefore inversion of the values of bits is done by XORing them with a 1. If the original bit was 1, it returns 1 XOR 1 = 0. If the original bit was 0 it returns 0 XOR 1 = 1. Also note that XOR masking is bit-safe, meaning that it will not affect unmasked bits because Y XOR 0 = Y.

10011101 10010101
00001111 11111111 (XOR masking bits)
= 10010010 01101010

```c
#include <stdio.h>

main()
{
        int x = 50, k=10;
        clrscr();
        printf("\n Value of x is %d ", x);
        printf("\n\n Value of k is %d ", k);

        printf("\n\n k = %d is Masking the value of x = %d \n", k, x);
        x = x ^ k;
        printf("\n After XOR Masking the Value of x is %d \n" , x);

        printf("\n k = %d is Masking the Changed Value of x = %d again", k,
        x = x ^ k;
        printf("\n\n Now the Value of x is changed again to %d " , x);
}
```

```
Output
Value of x is 50                //      Bit-Pattern : 00110010
Value of k is 10                //      Bit-Pattern : 00001010
k = 10 is Masking the value of x = 50
After XOR Masking the Value of x is 56          // Resultant    : 00111000
k = 10 is Masking the Changed Value of x = 56 again
Now the Value of x is changed again to 50       // Resultant    : 00110010
```

Bit Fields

- In C, **we can specify size (in bits) of structure and union members**. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is withing a small range.
- For example, consider the following declaration of date without the use of bit fields.

```
#include<stdio.h>
struct date
{unsigned int d;
 unsigned int m;
 float y;
}s;
int main()
{   printf("sizeof  d=%d m=%d y=%d bytes", sizeof(s.d),
sizeof(s.m), sizeof(s.y));
printf("\nSize of date is %u bytes\n",sizeof(struct date));
struct date dt = { 31, 12, 2014 };
printf("Date is %d / %d / %0.0f", dt.d, dt.m, dt.y);
}
Output:   sizeof  d=4 m=4 y=4 bytes
Size of date is 12 bytes
Date is 31 / 12 / 2014
```

Bit Fields

- Each member declaration must now include a specification indicating the size of corresponding bit field. The member name must be followed By a colon(:) and unsigned integer indicating the field size.
- C compilers may **order the bit fields from right to left**, whereas other C Compilers will order them from left to right.
- We will assume right to left Ordering in example

```
#include<stdio.h>
struct da
 { unsigned a : 5;    // a begin first word
   unsigned b : 5;
   unsigned c : 5;
   unsigned d : 5;   // forced to second word
}v={1,2,3,4};
int main()
{printf("v.a= %d v.b= %d v.c= %d  vid=%d", v.a, v.b, v.c, v.d);
printf("\n Size of v is %d bytes\n",sizeof(v));
}
```

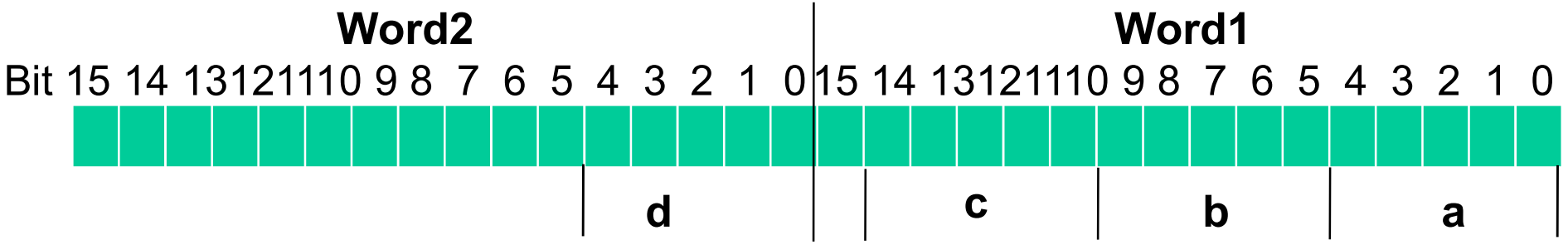
output:

v.a= 1 v.b= 2 v.c= 3 vid=4

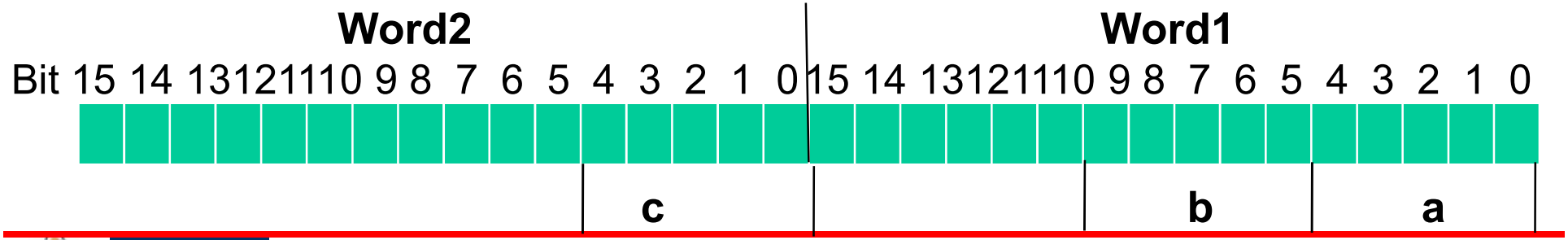
Size of v = 4 bytes

Bit Fields

- In the above example the four fields within v require a total of 20 bits. If the computer only allows 16 bits for an unsigned integer quantity, this structure will require two words of memory. The first three fields will be stored in first word. Since the last field will straddle the word boundary, it is automatically forced to the beginning of the second word.



- Unnamed field can be used to control the alignment of bit fields within a word of memory. Such fields provide padding within the word. The size of the unnamed field determines the extent of the padding.



➢The above representation of 'date' takes 12 bytes on a compiler where an unsigned int takes 4 bytes. Since we know that the value of d is always from 1 to 31, the value of m is from 1 to 12, we can optimize the space using bit fields.

```
#include<stdio.h>
struct date
 {
unsigned int d : 5;    // d has value between 1 and 31, so 5 bits  are sufficient
unsigned int m : 4;    // m has value between 1 and 12, so 4 bits are sufficient
float y;
}s;
int main()
{
printf("\n Size of date is %u bytes\n",sizeof(struct date));
struct date dt = { 31, 4, 2014 };
printf("Date is %d/%d/%0.0f", dt.d, dt.m, dt.y);
}
```

Output: Size of date is 8 bytes
Date is 31/12/2014

Bit Fields

```
#include<stdio.h>
struct da
 {
unsigned a : 5;    // a begin first word
unsigned b : 5;
unsigned : 6;
unsigned c : 5;   // forced to second word
}v={1,2,3};
int main()
{
printf("v.a= %d v.b= %d v.c= %d  ", v.a, v.b, v.c);
printf("\n Size of v = %d bytes\n",sizeof(v));
}
```

Output:

v.a= 1 v.b= 2 v.c= 3

Size of v = 4 bytes

Note: another way to control the alignment of bit fields is to include an unnamed field whose width is zero(0) . This will automatically force the next field aligned with the beginning of a new word.

File I/O

- A file represents a sequence of bytes, does not matter if it is a text file or binary file. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices.

File Operations:

There are different operations that can be carried out on a file.

These are:

Create a File

1. Creation of a new file
2. Opening an existing file
3. Reading from a file
4. Writing to a file
5. Moving to a specific location in a file (seeking)
6. Closing a file

Functions for file handling

There are many functions in the C library to open, read, write, search and close the file. A list of file functions are given:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

Create a File

- Whenever you want to work with a file, the first step is to create a file. A file is nothing but **space in a memory** where data is stored.
- Syntax:

```
FILE *fp;
fp = fopen ("file_name", "mode");
```

```
#include <stdio.h>
int main()
{
FILE *fp;
fp = fopen ("data.txt", "w");
printf("file is created");
}
```

Note : File is created in the same folder where you have saved your code.

You can specify the path where you want to create your file

```
#include <stdio.h>
int main()
 {
FILE *fp;
fp  = fopen ("D://data.txt", "w");
printf("file is created");
}
```

Opening Files: You can use the fopen( ) function to create a new file or to open an existing file, this call will initialize an object of the type FILE is defined in stdio.h, which contains all the information necessary to control the stream.

Syntaxe :

FILE *fopen(const char * filename, const char * mode);

Here, filename is string as arguments, which you will use to name your file and access mode can have one of the following values in next slide:

```
FILE  *fp;
Fp=fopen("R1.txt",  "W");
```

Closing a File: the file must be closed when no more operations are to be performed on it. in C we can use the fclose() function. After closing the file, connection between file and program is broken.

Syntax **int fclose(FILE *fp);**

Mode	Description
r	opens a text file for reading (must exist)
w	Opens a text file for writing**, if it does not exist then a new file is created**. Here your program **will start writing content from the beginning of the file.**
a	Opens a text file for writing in appending mode**, if it does not exist then a new file is created.** Here your program will start appending content in the existing file content
rb	opens a binary file for reading
wb	opens a binary file for writing
ab	appends to a binary file
r+	Opens a text file for reading and writing both.
w+	Opens a text file for reading and writing both. **It first truncate the file to zero length if it exists otherwise create the file if it does not exist.**
a+	Opens a text file for reading and writing both**. It creates the file if it does not exist.** The reading will start from the beginning but writing can only be appended.
rb+	opens a binary file for read/write
wb+	opens a binary file for read/write
ab+	append a binary file for read/write

End of File

- The file reading functions need to know the end of file so that they can stop reading. when the end of file is reached, the operating system sends an end-of-file signal to the program. When the program receives this signal, the file reading functions return **EOF**, **which is a constant defined in the file stdio.h and its value is -1.**
- if a function stops reading we can not make out whether the end of file was reached or an error was encountered. We can check separately for these two conditions with the help of ferror() and feof().
- **The feof() is used to check the end of file condition**. It returns a nonzero value if end of file otherwise it return zero. **Syntax int feof(FILE *fp);**

Character I/O

fgetc(): This function **reads a single character from the given file** and increments the file pointer position. It returns the character after converting it to an int without sign extension.

Syntax int fgetc (FILE *fp) ;

fputc(): This function **writes a character to the specified file at the current file position** and then increments the file position pointer.

Syntax int fputc(int c, FILE *fp);

Reading from a File

```
/* Display contents of a file on screen. */
# include "stdio.h"
main( )
{
FILE *fp ;
char ch ;
fp = fopen ( "PR1.C", "r" ) ;
while ( 1 )
{
ch = fgetc ( fp ) ;
if ( ch == EOF )
break ;
printf ( "%c", ch ) ;
}
fclose ( fp ) ;
}
```

```
# include "stdio.h"
int main()
{
 FILE *fp;
 int x;
  fp=fopen("first.txt","r");
 if(fp==NULL)
 {
  printf("File not found.");
 }
else
{
 while(1)
 {
  x=fgetc(fp);
  if(feof(fp))
  break;
  printf("%c",x);
 }
}
 fclose(fp);
}
```

This function read characters from a file and these characters are stored in the string pointed

```
char *fgets( char *buf, int n, FILE *fp );
```

The functions fgets() reads up to n - 1 characters from the input stream referenced by fp. It copies the read string into the buffer temp, appending a null character to terminate the string.

```
#include <stdio.h>
main()
{
FILE *fp;
char temp[100];
fp = fopen("test.txt", "r");
fscanf(fp, "%s", temp);
printf("1 : %s\n", temp );
fgets(temp, 20, (FILE*)fp);
printf("2: %s\n", temp );
fclose(fp);
}
```

If test file has these two line

This is testing for fprintf...
This is testing for fputs...

Output:

1 : This
2: is testing for fpr

Formatted I/O fscanf() and fprintf()

Functions fprintf() and fscanf() are same as printf() and scanf() difference is that, these functions work on files instead of standard input . These function has one more parameter which is a pointer of file. Syntax of these function are as follows-

fscanf(FILE *fp, "format specifier",&variable1,&variable2,….);

Ex: fscanf(fp, "%d%d%d" , &a, &b, &c);

fprintf(FILE *fp, "format specifier",variable1,variable2……);

Ex: fprintf(fp, "%d\t%d\t%d\t", a, b,c);

```
#include <stdio.h>
int main()
{  FILE *fp;
   char Temp[255];    //creating char array to store data of file
   fp = fopen("file.txt", "r");
   while(fscanf(fp, "%s", Temp)!=EOF)
  {
   printf("%s ", buff );
   }
   fclose(fp);
   return 0; }
```

Writing a File

- Following is the simplest function to write individual characters to a stream:

int fputc(int c, FILE *fp);

- The function fputc() writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise EOF if there is an error.
- You can use the following functions to write a null-terminated string to a stream:

int fputs(const char *s, FILE *fp);

- The function fputs() writes the string s to the output stream referenced by fp.

```
#include <stdio.h>
main()
{
FILE *fp;
fp = fopen("test1.txt", "w+");
fprintf(fp, "This is testing for fprintf...\n");
fputs("This is testing for fputs...\n", fp);
fclose(fp);
}
```

➢When the above code is compiled and executed, it creates a new file test.txt, and writes two lines using two different functions.

```
#include <stdio.h>
int main()
{   int i;  char ch;
    FILE * fptr;
    char fn[50];
    char str[] = "Guru99 Rocks\n";
    fptr = fopen("test3.txt", "w+");
    for (i = 0; str[i] != '\n'; i++) {
    fputc(str[i], fptr);      /* write to file using fputc() function  to character wise*/
}
fclose(fptr);
       fptr = fopen("test3.txt", "r");
while ( 1 )
{
ch = fgetc ( fptr) ;
if ( ch == EOF )
break ;
printf ( "%c", ch ) ;
}
 fclose(fptr);
       return 0;
   }
```

Output:
Guru99 Rocks

getc( ) function and putc( ) function : The getc() and putc() are exactly similar to that of fgetc() and fputc().

- The putc function writes the character to the file associated with the file pointer.
- The getc() function takes the file pointer as its argument, reads the character from the file and returns that character as a integer or EOF when it reaches the end of the file.

```
// define putc() method
        int putc (int ch, FILE *file);
// define getc() method
        int getc (FILE *file);
```

```
#include<stdio.h>
int main () {
   FILE *fp;
   int ch;
   fp = fopen("myfile.txt", "w");
   for( ch = 65 ; ch <= 90; ch++ )
{
      putc(ch, fp);
   }  fclose(fp);
fp=fopen("myfile.txt","r");
while((ch=getc(fp))!=EOF)
{
printf("%c",ch);
}
fclose(fp);
}
```

Write a cprogram to transfer the content of a text file to another text file

```
#include <stdio.h>
#include <stdlib.h>
main()
{   char ch, source_file[20], target_file[20];
   FILE *f1, *f2;
   printf("Enter name of source file to copy\n");
   gets(source_file);
   f1 = fopen(source_file, "r");
   printf("Enter name of target file\n");
   gets(target_file);
   f2= fopen(target_file, "w");
   while( ( ch = fgetc(f1) ) != EOF )
   fputc(ch, f2);
   printf("File copied successfully.\n");
   fclose(f1);
   fclose(f2);
   return 0;
}
```

Write a C program to input students record by keyboard and then write record in a text file

```
#include<stdio.h>
#include<conio.h>
    int main()
     {
         FILE *fp;
          int roll;
          char name[25];
          float marks;
          char ch;
          fp = fopen("file2.txt","w");          //Statement   1
         do
          {     printf("\nEnter Roll : ");
                scanf("%d",&roll);
                printf("\nEnter Name : ");
                scanf("%s",name);
                printf("\nEnter Marks : ");
                scanf("%f",&marks);
                fprintf(fp,"%d\t%s\t %f \n",roll,name,marks);
                printf("\nDo you want to add another data (y/n) : ");
                ch = getche();
          }while(ch=='y' || ch=='Y');
          printf("\nData written successfully...");
          fclose(fp);
          return 0;
     }
```

Block Read/write

- This function is used for writing an entire block to a given file.

Size_t fwrite(const void *ptr, size_t size, size_t n, FILE *fptr);

- Size_t is defined in stdio.h as typedef unsined int size_t; ptr is a pointer which points to the block of memory that contains the information to be written to the file, size denotes the length of each item in bytes, n is the number of items to be written to the file, and fptr is a file pointer which points to the file to witch data is written.
- For using these functions, the file is generally opened in binary mode("wb", "rb").
- Example To write a single float value contained in variable x to the file

  fwrite(&x, sizeof(float), 1, fp);

- To write a only first 5 elements from integer array x[10] to the file

  fwrite(x, sizeof(int), 5, fp);

- To write a structure variable which is defined as **struct record{ char name[20];**
  in roll;
  float marks;
  } s;

  fwrite(&s, sizeof(s), 1, fp);

```
/*P12.10 Program to understand the use of fwrite()*/
#include<stdio.h>
#include<stdlib.h>
struct record
{           char name[20];
            int roll;
            int marks;
}student;
int main(void)
{           int i,n;
            FILE *fp;
            printf("Enter number of records : ");
            scanf("%d",&n);
            for(i=0; i<n; i++)
            {           printf("Enter name : ");
                        scanf("%s",student.name);
                        printf("Enter roll no : ");
                        scanf("%d",&student.roll);
                        printf("Enter marks : ");
                        scanf("%d",&student.marks);
                        fwrite(&student,sizeof(student),1,fp);
            }
            fclose(fp);
            return 0;
}
```

Block Read/write

- This function is used to read an entire block from a given file.

Size_t fread(void *ptr, size_t size, size_t n, FILE *fptr);

- Size_t is defined in stdio.h as typedef unsined int size_t; ptr is a pointer which points to the block of memory which receives the data read from the file, size den otes the length of each item in bytes, n is the number of items to be read from the file, and fptr is a file pointer which points to the file from witch data is read.
- Example To read a single float value from the file and store it in variable x.

```
fread(&x, sizeof(float), 1, fp);
```

- To read 5 integer from the file and store them in first five elements of array x[10]

```
fread(x, sizeof(int), 5, fp);
```

- To read a structure variable that is defined as

```
struct record{  char name[20];
                in roll;
                float marks;
              } s;

fread(&s, sizeof(s), 1, fp);
```

```
/*P12.11 Program to understand the use of fread()*/
#include<stdio.h>
#include<stdlib.h>
struct record
{           char name[20];
            int roll;
            int marks;
}student;
int main(void)
{           FILE *fp;
            fp = fopen("stu.dat","rb");
            printf("\nNAME\tROLLNO\tMARKS\n");
            while(fread(&student,sizeof(student),1,fp)==1)
            {
                        printf("%s\t",student.name);
                        printf("%d\t",student.roll);
                        printf("%d\n",student.marks);
            }
            fclose(fp);
            return 0;
}
```

Random Access to File

- We can access the data stored in the file in two way sequentially or randomly . The standard library also provides a means to move back and forth within a file to any specified location.
- Random accessing of files in C language can be done with the help of the following functions −
- ftell ( )
- rewind ( )
- fseek ( )

rewind( ) : The rewind( ) function **places the pointer to the beginning of the file**, irrespective of where it is present right now.

```
#include<stdio.h>
#include<conio.h>
int main(){
FILE *fp;
char c;
fp=fopen("Raj.txt","r");
 while((c=fgetc(fp))!=EOF){
printf("%c", c);
}
printf("\n");
rewind(fp);          //moves the file pointer at beginning of the file
while((c=fgetc(fp))!=EOF)
{  printf("%c", c);
}
  fclose(fp);
return 0;
}
```

Output:
this is a simple text
this is a simple text

fseek():

➢ The **fseek( )** The fseek() function takes three arguments, first is the **file pointer**, second is the **offset** that specifies the number of bytes to moved and third is the **position** from where the offset will move. It will return zero if successfully move to the specified position otherwise return nonzero value.

int fseek(FILE *fp, long offset, int pos);

There are three positions from where offset can move.

Constant Name	Constant Value	Description
SEEK_SET	0	The begining of file
SEEK_CUR	1	The current position in file
SEEK_END	2	The end of file

Examples of usage of fseek() function

1. fseek(p, 10L, 0); // position pointer is skipped 10 bytes forward from the beginning of the file. For long integer L is used.
2. fseek(p, 5L, 1); // position pointer is skipped 5 bytes forward from the current position.
3. fseek(p, -5L, 1); // position pointer is skipped 5 bytes backward from the current position.
4. fseek(p, 0L, 2); // position pointer is skipped 0 bytes from the end of file.
5. fseek(p, 0L, 0); // position pointer is skipped 0 bytes from the beginning of the file.
6. fseek(p, -5L, SEEK_END); // position pointer is skipped 5 bytes backward from the end of file.

Example

```
#include <stdio.h>
int main ()
{
   FILE *fp;
   fp = fopen("R5.txt","w+");
   fputs("This is C Programming ", fp);

fseek( fp, 7, SEEK_SET );
fputs(" Java Prog. language ", fp);
   fclose(fp);

   return(0);
}
```

//Now let's see the content of the left file using the following program −

```
#include <stdio.h>
int main () {
   FILE *fp;
   int c;
   fp = fopen("R5.txt","r");
   while(1)
{    c = fgetc(fp);
      if( feof(fp) )
    {   break;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

Output

This is Java Prog. language

ftell()

ftell() in C is used to find out the current position of file pointer in the file with respect to starting of the file. Syntax of ftell() is:

```
long ftell(FILE *pointer)
```

```
#include<stdio.h>
      int main()
{
   /* Opening file in read mode */
   FILE *fp = fopen("R5.txt","r");
     /* Reading first string */
   char string[20];
   fscanf(fp,"%s", string);
  // R5.txt file content is This is Java Prog. language
   /* Printing position of file pointer */
   printf("%ld", ftell(fp));
   return 0;
}
 output      4
```

```
#include<stdio.h>
      int main () {
  char str[] = "This is book";
  FILE *fp;
  int ch;                    /* First let's write some content in the file */
  fp = fopen(" R5.txt" , "w" );
  fwrite(str , 1 , sizeof(str) , fp );
  fclose(fp);
  fp = fopen(" R5.txt" , "r" );
  while(1) {
    ch = fgetc(fp);
    if( feof(fp) ) {
      break ;
    }       printf("%c", ch);
  }
  rewind(fp);
  printf("\n");
  while(1) {
    ch = fgetc(fp);
    if( feof(fp) ) {
      break ;
    }
    printf("%c", ch);
     }
  fclose(fp);
  return(0);
}
```

Output

```
This is book
This is book
```

Library Functions in C

- The Standard Function Library in C is a huge library of sub-libraries, each of which contains the code for several functions.
- In order to make use of these libraries, link each library in the broader library through the use of header files. The definitions of these functions are present in their respective header files.
- In order to use these functions, we have to include the header file in the program. This table briefly describes the C library functions.

Function	System Include File	Function Prototype	Description
abort	stdlib.h	void abort(void);	Stops a program abnormally.
abs	stdlib.h	int abs(int *n*);	Calculates the absolute value of an integer argument *n*.
acos	math.h	double acos(double *x*);	Calculates the arc cosine of *x*.
asctime	time.h	char *asctime(const struct tm **time*);	Converts the *time* that is stored as a structure to a character string.

Function	System Include File	Function Prototype	Description
asin	math.h	double asin(double *x*);	Calculates the arc sine of *x*.
atan	math.h	double atan(double *x*);	Calculates the arc tangent of *x*.
atof	stdlib.h	double atof(const char **string*);	Converts *string* to a double-precision floating-point value.
atoi	stdlib.h	int atoi(const char **string*);	Converts *string* to an integer.
fmod	math.h	double fmod(double *x*, double *y*);	Calculates the floating-point remainder of *x/y*.
gamma	math.h	double gamma(double *x*);	Computes the Gamma Function
hypot	math.h	double hypot(double *side1*, double *side2*);	Calculates the hypotenuse of a right-angled triangle with sides of length *side1* and *side2*.
isalnum	ctype.h	int isalnum(int *c*);	Tests if *c* is alphanumeric.
isalpha	ctype.h	int isalpha(int *c*);	Tests if *c* is alphabetic.
isascii[4]	ctype.h	int isascii(int *c*);	Tests if *c* is within the 7-bit US-ASCII range.
isblank	ctype.h	int isblank(int *c*);	Tests if *c* is a blank or tab character.

Function	System Include File	Function Prototype	Description
isdigit	ctype.h	int isdigit(int *c*);	Tests if *c* is a decimal digit.
islower	ctype.h	int islower(int *c*);	Tests if *c* is a lowercase letter.
isupper	ctype.h	int isupper(int *c*);	Tests if *c* is an uppercase letter.
rand	stdlib.h	int rand(void);	Returns a pseudo-random integer.
y0	math.h	double y0(double *x*);	Calculates the Bessel function value of the second kind of order 0.
y1	math.h	double y1(double *x*);	Calculates the Bessel function value of the second kind of order 1.
yn	math.h	double yn(int *n*, double *x*);	Calculates the Bessel function value of the second kind of order *n.*