UNIT – V

Sorting and Searching: Sorting by selection, sorting by exchange, sorting by insertion, sorting by partitioning, binary search.

Structures: Basics of structures, structures and functions, arrays of structures, pointers to structures, self-referential structures, table lookup, typedef, unions, bit-fields.

Some other Features: Variable-length argument lists, formatted input-Scanf, file access, Error handling-stderr and exit, Line Input and Output, Miscellaneous Functions.

5.1 SORTING BY SELECTION

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

How Selection Sort Works?

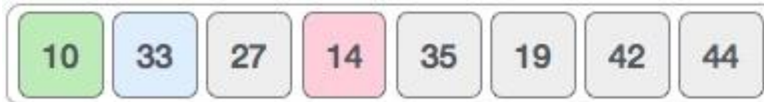
Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

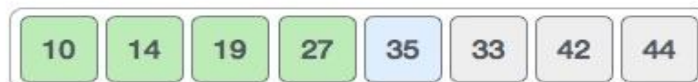
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

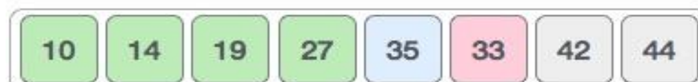


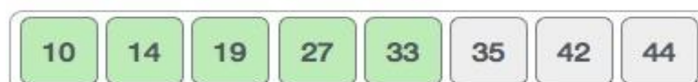
After two iterations, two least values are positioned at the beginning in a sorted manner.

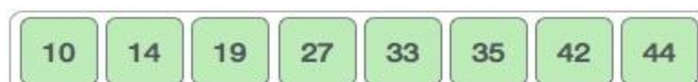
The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −









Now, let us learn some programming aspects of selection sort.

Algorithm

Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

Program:

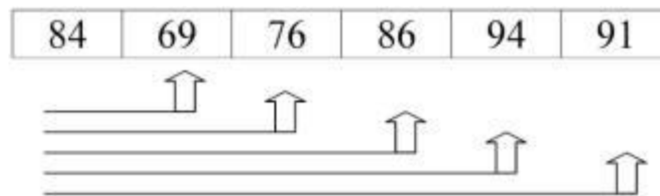
```
#include<stdio.h>
void main(void)
{
     int a[100],n,i,j,first,temp,k;
     printf("\n enter the size");
     scanf("%d",&n);
       for (i = 0; i < n; i++)
       {
             printf("\n enter %d element",i+1);
             scanf("%d",&a[i]);
       }
       // algorithm
     for (i= 0; i <n-1; i++)
         {
             first = i;
             for (j=i+1;j<n; j++)
             {
                    if (a[j] < a[first])
                    first = j;
             }
             temp = a[first];
             a[first] = a[i];
             a[i] = temp;
         printf("\n AFTER %d pass",i+1);
             for (k = 0; k < n; k++)
             {
                 printf(" %d",a[k]);
             }
       }//end algorithm

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

5.2 SORTING BY EXCHANGE

Exchange Sort

The **exchange sort** is similar to its cousin, the bubble sort, in that it compares elements of the array and swaps those that are out of order. (Some people refer to the "exchange sort" as a "bubble sort".) The difference between these two sorts is the manner in which they compare the elements. **The exchange sort compares the first element with each following element of the array, making any necessary swaps.**



When the first pass through the array is complete, the exchange sort then takes the second element and compares it with each following element of the array swapping elements that are out of order. This sorting process continues until the entire array is ordered.

Let's examine our same table of elements again using an exchange sort for descending order. Remember, a "pass" is defined as one full trip through the array comparing and if necessary, swapping elements

Array at beginning:	84	69	76	86	94	91
After Pass #1:	94	69	76	84	86	91
After Pass #2:	94	91	69	76	84	86
After Pass #3:	94	91	86	69	76	84
After Pass #4:	94	91	86	84	69	76
After Pass #5 (done):	94	91	86	84	76	69

The exchange sort, in some situations, is slightly more efficient than the bubble sort. It is not necessary for the exchange sort to make that final complete pass needed by the bubble sort to determine that it is finished.

Program:

```
#include<stdio.h>

void main()
{
      int a[100],n,i,j,k,temp;
```

```
    printf("\n enter n value  ");
    scanf("%d",&n);
    for (i = 0; i < n; i++)
      {
            printf("\n enter %d element",i+1);
            scanf("%d",&a[i]);
      }
      //Algorithm
      for(i = 0; i < (n -1); i++)
      {
            for (j=(i + 1); j < n; j++)
            {
                  if (a[i] > a[j])
                  {
                        temp = a[i];
                        a[i] = a[j];
                        a[j] = temp;
                  }
            }
            printf("\n AFTER %d pass",i+1);
            for (k = 0; k < n; k++)
            {
                printf(" %d",a[k]);
            }
      }
      //Some output
      for (i = 0; i < n; i++)
      {
            printf("\n a[%d] =%d",i,a[i]);
      }
}
```

5.3 SORTING BY INSERTION

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the
value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

Program:

```
#include<stdio.h>
void main()
{
     int a[100],size,i,j,key,k;

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

     for(i = 1; i < size; i++)
     {
          key = a[i];
          for (j = i - 1; (j >= 0) && (a[j] > key); j--)
```

```
            {
                 a[j+1] = a[j];
            }
            a[j+1] = key;
        printf("\n AFTER THE %d PASS",i);
        for(k=0;k<size;k++)
          printf(" %d",a[k]);
      }
      printf("\n elemets after sorting");
      for (i = 0; i < size; i++)
      {
            printf(" \n a[%d]=%d",i,a[i]);
      }
}
```

5.4 SORTING BY PARTITIONING / Quick Sort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

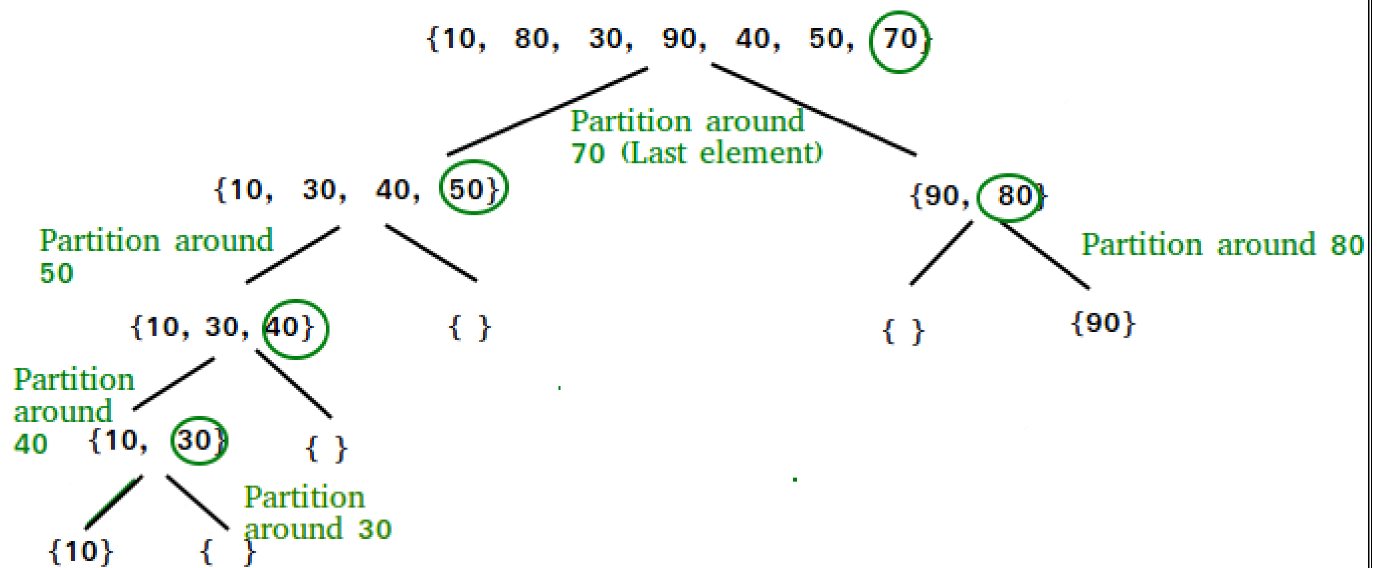
1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[pi] is now
         at right place */
      pi = partition(arr, low, high);

      quickSort(arr, low, pi - 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
}
```



Partition Algorithm

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
```

```
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Illustration of partition() :

```
arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes:  0   1   2   3   4   5   6


low = 0, high =  6, pivot = arr[h] = 70
Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j
                                     // are same

j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
```

arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

5.5 BINARY SEARCH

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

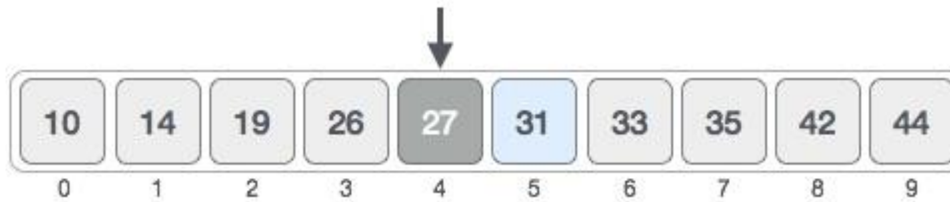
How Binary Search Works?

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

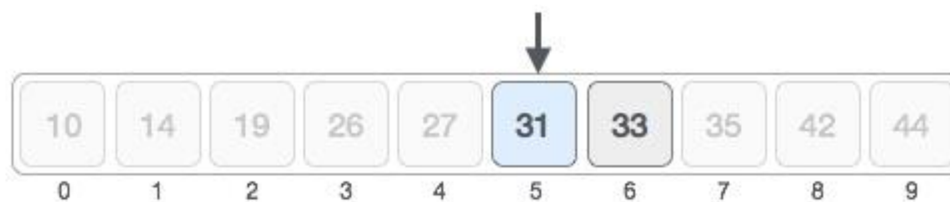
We change our low to mid + 1 and find the new mid value again.

low = mid + 1
mid = low + (high - low) / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudocode

The pseudocode of binary search algorithms should look like this −

```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exists.

    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

    if A[midPoint] < x
      set lowerBound = midPoint + 1

    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while

end procedure
```

Program:

```
#include<stdio.h>
void main()
{
      int a[100],n,i,j,k,temp,key,low,high,mid,flag=0,ch;
     printf("\n enter n value  ");
     scanf("%d",&n);
     for (i = 0; i < n; i++)
```

```
	{
		printf("\n enter %d element",i+1);
		scanf("%d",&a[i]);
	}
	for(i = 0; i < (n -1); i++)
	{
		for (j=(i + 1); j < n; j++)
		{
			if (a[i] > a[j])
			{
				temp = a[i];
				a[i] = a[j];
				a[j] = temp;
			}
		}
	}
	printf("\n ELEMENTS AFTER SORTING \n");
	for (i = 0; i < n; i++)
	{
		printf(" %d",a[i]);
	}

ss: flag=0;
	printf("\n enter KEY element ");
	scanf("%d",&key);
	low=0;
	high=n-1;
	while(low<=high)
	{
	   mid=(low+high)/2;
	   if(a[mid]==key)
	    {
	      flag=1;
	      break;
	    }
	   else if(key>a[mid])
	     low=mid+1;
	   else if(key<a[mid])
	     high=mid-1;
	}
	if(flag==1)
	  printf("\n %d is found in %d location",key,mid);
	else
	  printf("\n KEY NOT FOUND");
	printf("\n DO U WANT TO CONTINUE 1.YES  2. NO");
	scanf("%d",&ch);
	if(ch==1)
	goto ss;
}
```

5.6 BASICS OF STRUCTURES

A Structure is a **derived data type** to organize a group of related data items of **different data types** referring to a single entity. i.e., a single variable capable of holding data items of different data types.

The data items in a structure are usually related like different kinds of information about a **person** or about a **part** or about an **account**, etc.

Each data item in a structure is called a **member**, sometimes these members are also called **fields**.

The keyword used to create a structure is struct.

The **advantage** of using a structure is that the accessibility of members becomes easier since all the members of a specific structure gets allocation of continuous memory and therefore it minimize the memory access time.

Structures can be **declared** and **defined** in many number of ways.

Generally a **structure** can be **declared** as:

```
struct tag_name {
        data_type1 var1;
        data_type2 var2;
        ....
        ....
        data_typen varn;
};
```

The declaration begins with the keyword struct. The list of declaration of its members must be enclosed in braces, the **tag_name** is an identifier which specifies the new structure name.

The **declaration** of a structure does not reserve any storage space. But the **definition** of the structure creates structure variables.

The **structure variables** can be defined as:

```
struct tag_name svar1, svar2 ...svarn;
```

Let us consider an example:

```
struct example {
        int a;
        float b;
```

```
        char c;
};
struct example s1, s2, s3; // structure definition
```

In the above code **example** is a new user defined data type which consists of 3 variables of different data types. The structure definition allocates a 7 bytes (2 + 4 + 1) memory to each structure variable s1, s2, s3.

Another way of declaring and defining a structure is:

```
struct {
        data_type1 var1;
        data_type2 var2;
        ....
        ....
        data_typen varn;
}svar1, svar2;
```

In the above type of syntax variables are declared directly with the declaration of a structure **without choosing a name** for the the structure and only **two** structure variables can only be defined. So, It is not possible to create new variables of this structure type later in the program.

Another syntax is:

```
struct tag_name {
        data_type1 var1;
        data_type2 var2;
        ....
        ....
        data_typen varn;
}svar1, svar2;
....
struct tag_name svar3, svar4;
```

Here, two structures variables are declared along with the structure tag, they can be used in the program where they are declared. A new structure variables can also be defined by using **tag_name** and used in the program.

Processing of a structure refers to the ways of accessing the members of that structure.

Basically there are **two operators** available for accessing the members of a structure depending on whether a normal variable or a pointer variable is declared to the structure.

They are:

1. Dot (period) operator
2. Arrow operator

To access a member of a structure using **dot** (.) operator, the syntax is:

```
structure_variable.member
```

Let us consider an example:

```
struct example {
        int a;
        float b;
        char c;
} ;
struct example e;
e.a = 25;
e.b = 45.67;
e.c = 'M';
```

In the above code e is a structure variable which has the memory of 3 variables a, b and c. Each member can be accessed by using the operator dot (.) as e.a, e.b and e.c.

Let us consider another example:

```
struct employee {
        int empId;
        char empName[20];
} emp1;
emp1.empId = 201;
emp1.empName = "APJ.Abdul Kalam"; // Gives an error because a string can not be assigned directly
strcpy(emp1.empName, "APJ.Abdul Kalam"); // Correct way of assigning a string to the array
```

A **structure variable** can be initialized in its definition itself with a list of initializers enclosed within the braces. The initialization of values will be taken in an order.

Let us consider an example:

```
struct student {
        int roll_number;
        int total_marks;
        float attendance_percentage;
} ;
struct student s = {101, 450, 72.45};
```

In the above example 101 is assigned to **s.roll_number**, 450 is assigned to **s.total_marks** and 72.45 is assigned to **s.attendance_percentage**.

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

In arrays we can not do assignment of one array variable to the another array variable of the same type also. Below is an example of assigning one array variable to another array variable of same type:

```
int a[5] = {10, 20, 30};
int b[5];
b = a; // Gives an error incompatible assignment of arrays
b[0] = a[0];
b[1] = a[1];
b[2] = a[2]; // This type of assignment is correct,i.e., array values can be assigned individually by using index
```

Below is an example of assigning one structure variable to another structure variable of same type:

```
struct book {
        char name[50];
        float price;
        int pages;
} ;
struct book b1 = {"C-Language", 155.95, 375};
struct book b2;
b2 = b1; // This type of assignment is correct with respect to structure variables
```

In the above example the contents of a structure variable b1 are copied into another structure variable b2 of same type.

Nested Structures:

There are two ways to define nested structures in **C**:

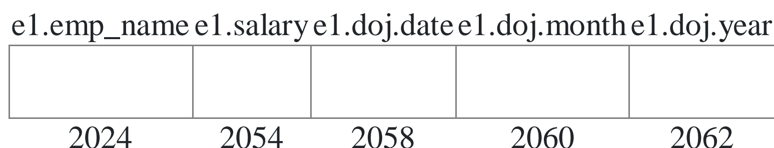
1. Define one structure variable within the declaration of another structure.
2. Declare one structure within the declaration of another structure.

Below is an example of defining one **structure variable** within another structure:

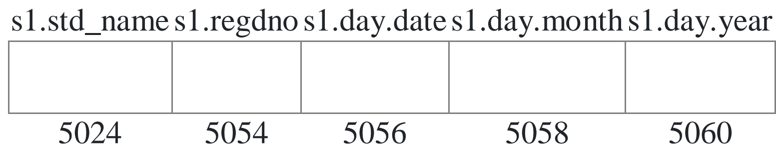
```
struct date {
        int date;
        int month;
        int year;
};
struct employee {
        char emp_name[30];
        float salary;
        struct date doj;
}e1;
struct student {
        char std_name[30];
        int regdno;
        struct date day;
}s1;
```

In the above example, date is one user defined data type which is declared outside of all structures. The structure variables of date is defined with in two structures employee and student.

The structure variable e1 is defined for the type structure **employee**, which allocates the memory to the each and every field including the structure variable doj of date.



The structure variable s1 is defined for the type structure **student**, which allocates the memory to the each and every field including the structure variable day of date.



When the structure date is declared in the global scope, the **structure variables** of date can be defined any where in the program. We can access the member of nested structure as given below:

Syntax:

Outer_Structure.Nested_Structure.member

Examples:

e1.doj.date

s1.day.month

Another way to define a **nested structure** is to declare one structure within the declaration of another structure.

Below is an example of declaring one **structure** within the declaration of another structure:

```
struct student {
        char std_name[30];
        int regdno;
        struct date {
                int date;
                int month;
                int year;
        }day;
}s1;
```

In the above example, date is one user defined data type which is declared inside the structure student. So, the structure variables for date can not be defined outside of structure student.

The inner structure date can not be used outside of the outer structure student.

The structure variable s1 is defined for the type structure **student**, which allocates the memory to the each and every field including the structure variable day of date.

s1.std_name	s1.regdno	s1.day.date	s1.day.month	s1.day.year
5024	5054	5056	5058	5060

5.7 STRUCTURES AND FUNCTIONS

Like an ordinary variable, a **structure variable** can also be passed to a **function**. User may either pass individual structure elements or the entire structure at once.

If a structure variable is **passed by value**, changes made to the structure variable inside the function definition does not reflect in the originally passed structure variable.

If a structure variable is **passed by address**, changes made to the structure variable inside function definition reflects in the originally passed structure variable.

If a programmer wants to pass a structure variable to the function, first the programmer has to **define the structure** in the **global** section then only the function can access that structure variable.

A pointer pointing to a structure is known as **structure pointer**. A pointer to the structure can be defined with an asterisk (*) symbol.

Remember that on the left hand side of **dot** (.) operator there must always be a structure variable, whereas on the left hand side of the **arrow** (->) operator there must be always a pointer to the structure.

A pointer variable to the structure can hold the address of a structure variable or address of a heap memory allocated at run-time for the structure type.

The **heap memory** for the structure type can also be created at the run-time using malloc() and the address of that allocated memory can be assigned to a **pointer to that structure**.

The syntax of allocating memory for the structure using malloc() is:

```
ptr_variable_to_structure = (type_cast *) malloc(size_of_structure);
```

Below is an example of a pointer to the structure:

```
struct student {
        int rollno;
        float percentage;
};
struct student *p; // p is a pointer variable to the structure
```

```
p = (struct student *) malloc(sizeof(struct student)); // Assigning the address of a heap memory to the pointer p
p->rollno = 22; // Assigning value 9 to the member rollno of the structure
p->percentage = 72.54; // Assigning value 72.54 to the member percentage of the structure
printf("%d %f", p->rollno, p->percentage);
```

User can create heap memory for array of structures at run-time by using malloc(), which can be handled using a pointer variable of the structure.

Below is an example to create heap memory for array of structures:

```
struct student {
        int regdno;
        int maths_marks, c_marks, java_marks;
        int total;
        float avg;
};
p = (struct student *) malloc(3 * sizeof(struct student)); // Creates heap memory of 3 structure variables
```

From the above declaration, &(p + i)->regdno returns the address of the ith student's regdno which is referencing through the pointer p.

(p + i)->total gives the value of the ith student's total which is referencing through the pointer p.

5.8 ARRAYS OF STRUCTURES

When the programmer wants to define more number of structure variables, it is better practice to define **array of structures** rather than individual structure variables.

Creation of **array of structures** is same as creation of **integer arrays**, **float arrays** and so on.

The syntax of defining an array of structure is:

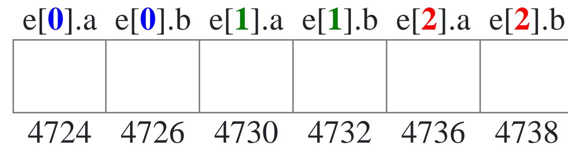
```
struct tag_name structure_variable[size]; // The size specifies the number of structure variables to be created
```

Below is an example of array of structure:

```
struct example {
        int a;
        float b;
```

```
};
struct example e[3];
```

In the above example the definition of a structure will create an array of structures with **3** structure variables of type **structure example**. An array of structure variables should get a memory of continuous blocks and they can be accessed sequentially.



5.9 POINTERS TO STRUCTURES

A **pointer** pointing to a **structure** is known as a **structure pointer**. If a pointer to the structure variable is declared, then **arrow** (->) notation is used to access the members of the structure.

Remember that on the left hand side of **dot** operator there must always be a **structure variable**, whereas on the left hand side of the **arrow** operator there must always be a **pointer to a structure**.

To access a member of a structure using **arrow** (->) operator, the syntax is:

```
structure_pointer_variable -> member
```

Let us consider an example:

```
struct student {
        int roll_number;
        int total_marks;
        float attendance_percentage;
} ;
struct student s = {201, 350, 92.75};
struct student *p;
p = &s;
printf("%d %d %f\n", p->roll_number, p->total_marks, p->attendance_percentage);
```

In the above example p is a **pointer** to the structure student, so it can store the address of a structure variable of same type.

Here, **p->roll_number** refers the value of s.roll_number, **p->total_marks** refers the value of s.total_marks and **p->attendance_percentage** refers the value of s.attendance_percentage.

5.10 SELF-REFERENTIAL STRUCTURES

Let us consider the below code:

```
struct student {
            int rollno;
            float percentage;
            struct student s; // It gives an error
} ;
struct student s1;
```

In the above code without completing the declaration of a structure, a structure variable has been defined, which does not allocate any memory and it gives an error message because the structure student is not declared yet.

But we can define a pointer to the structure with in the declaration of the same structure because a pointer can allocate 2 bytes of memory without depending on the type.

It is sometimes desirable to include one member that is **pointer** to the parent structure type within a structure.

A structure is called as **self referential structure** if it contains a pointer to itself. A self-referential structure is one of the data structures which refer to the pointer to point to another structure of the same type.

The general syntax of a self referential structure can be given as:

```
struct tag_name {
            datatype1 var1;
            datatype2 var2;
            .......
            struct tag_name *p;
};
```

In the above example the pointer variable p is referring to its parent structure. So, the structure of the type **tag_name** will contain a member that points to another structure of same type **tag_name**.

The major application areas of this kind of structures includes construction of **non-linear data structures** such as trees, graphs and **linear data structures** such as linked lists, stacks, queues.

5.12 TYPEDEF

typedef is a C keyword implemented to tell the compiler for assigning an alternative name to C's already exist data types. This keyword, typedef typically employed in association with user-defined data types in cases if the names of datatypes turn out to be a little complicated or intricate for a programmer to get or to use within programs. The typical format for implementing this typedef keyword is:

Syntax:

```
typedef <existing_names_of_datatype> <alias__userGiven_name>;
```

A structure can also be declared by using the keyword typedef as:

```
typedef struct tag_name {
        data_type1 var1;
        data_type2 var2;
        ....
        ....
        data_typen varn;
}new_name;
new_name svar1, svar2, ..., svarn;
```

Here, **typedef** is used to create an alias name **new_name** for the user defined structure **struct tag_name**, so **new_name** can be used to define structure variables of the given type.

5.13 UNIONS

A union is also a collection of different data types in **C** but that allows to store different data types in the same memory location.

User can define a union with many members, but only **one member** can contain a value at any given time.

Unions provide an efficient way of using the **same memory location** for multiple-purpose.

A union is same as structures but the difference is that only **one member** can be accessed at a time because the memory is created only for one member which has the **highest number of bytes** in size.

A union is declared by using the keyword union and members of the union can be accessed by using dot (.) operator.

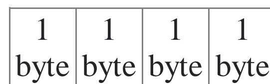
The declaration and definition of the union is:

```
union tag_name {
            data_type1 var1;
            data_type2 var2;
            ......
            data_typen varn;
};
union tag_name uvar1, uvar2,....uvarn;
```

Below code is an example for union:

```
union example {
            int number;
            float price;
            char ch;
};
union example e;
```

In the above example **4 bytes** of memory is allocated to the union variable e, the members can be accessed as e.number, e.price, e.ch butt only one member can be accessed at a time because the same memory is used for all the 3 members.



1 byte	1 byte	1 byte	1 byte

A Structure is a **derived data type** to organize a group of related data items of **different data types** referring to a single entity. i.e., a single variable capable of holding data items of different data types.

A Union is also a collection of **different data types** but only one member can be accessed at a time.

5.14 BIT-FIELDS

In **C**, the user can specify the size in bits for the members of a **structure** and a **union**.

The idea behind usage of bits is to use **memory efficiently** when the value of a field does not exceed a limit of storing the bits.

Bit fields can be used to reduce memory consumption when a program requires a number of integer variables which will always consists of low values.

For example, consider the following example:

```
struct date {
        int day;
        int month;
        int year;
};
void main() {
        struct date d = {31, 12, 2001};
        printf("Given date is: %d/%d/%d\n", d.day, d.month, d.year);
}
```

The size of the above structure is **12 bytes** (in 32-bit processor) because each int type occupies 4 bytes of memory.

Here, the value of **day** is always from **1 to 31** and the value of **month** is from **1 to 12**. hence, the space can be optimized by using bit fields..

The field day will take maximum of 5 bits to store the value between 1 and 31, as well as month will take maximum of 4 bits to store the value between 1 and 12. So, represent them in the bits and not in bytes.

The declaration of a bit field in the structure is as follows:

```
struct tag_name {
        data_type member_name : width_in_bits;
        // width_in_bits specifies the number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type
};
```

The above example can be written as:

```
struct date {
        unsigned int day : 5; // These are unsigned because day, month, year are not specified in negative'sunsigned int month : 4;
        unsigned int year;
};
void main() {
        struct date d = {31, 12, 2001};
        printf("Given date is: %d/%d/%d\n", d.day, d.month, d.year);
}
```

The size of the above structure is **8 bytes** because **day** and **month** are represented in bits and the field **year** can take 4 bytes of memory.

Within **4 bytes** slot **, day** and **month** are allocated 5 bits, 4 bits and the remaining bits are free. Hence,the user can use up to a total of 32 bits.

5.15 VARIABLE-LENGTH ARGUMENT LISTS

Sometimes, you may come across a situation, when you want to have a function, which can take variable number of arguments, i.e., parameters, instead of predefined number of parameters. The C/C++ programming language provides a solution for this situation and you are allowed to define a function which can accept variable number of parameters based on your requirement. The following example shows the definition of such a function.

```
int func(int, ... ) {
   .
   .
   .
}
int main() {
   func(1, 2, 3);
   func(1, 2, 3, 4);
}
```

It should be noted that the function **func()** has its last argument as ellipses, i.e. three dotes (...) and the one just before the ellipses is always an **int** which will represent the total number variable arguments passed. To use such functionality, you need to make use of **stdarg.h** header file which provides the functions and macros to implement the functionality of variable arguments and follow the given steps −

- Define a function with its last parameter as ellipses and the one just before the ellipses is always an int which will represent the number of arguments.
- Create a **va_list** type variable in the function definition. This type is defined in stdarg.h header file.
- Use **int** parameter and **va_start** macro to initialize the **va_list** variable to an argument list. The macro va_start is defined in stdarg.h header file.
- Use **va_arg** macro and **va_list** variable to access each item in argument list.
- Use a macro **va_end** to clean up the memory assigned to **va_list** variable.

Now let us follow the above steps and write down a simple function which can take the variable number of parameters and return their average −

Example

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

```
double average(int num,...) {
   va_list valist;
   double sum = 0.0;
   int i;
   va_start(valist, num); //initialize valist for num number of arguments
   for (i = 0; i < num; i++) {
      //access all the arguments assigned to valist
      sum += va_arg(valist, int);
   }
   va_end(valist); //clean memory reserved for valist
   return sum/num;
}
int main() {
   printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2,3,4,5));
   printf("Average of 5, 10, 15 = %f", average(3, 5,10,15));
}
```

Output

```
Average of 2, 3, 4, 5 = 3.500000
Average of 5, 10, 15 = 10.000000
```

5.15 FORMATTED INPUT-SCANF

Formatted Input and Output allows programmers to perform input and output in a particular fashion.

Formatting integer input
%wd
Here %d is the conversion specification for integer and w denotes the maximum width of the input data. If the length of the input is more than the width then values are not stored correctly.
Let's take some examples:

```
scanf("%2d%3d", &a, &b);
```

In this case variable, a has a width of 2 and b has a width of 3.
The values of a and b can be entered in the following ways:

Case 1: When the length of the data entered is less than the field width, then the input values are stored correctly in the given variables.
Input: 4 34
In this case, 4 is stored in a and 34 is stored in b.
Case 2: When the length of the data entered is equal to the field width, then the input values are stored correctly in the given variables.
Input: 23 456
In this case, 23 is stored in a and 456 is stored in b.

Case 3: When the length of the data entered is greater than the field width, then input values are not stored correctly in the given variables.
Input: 234 99
Since a has a width of 2, only 23 is stored in a and 4 is stored in b, while the rest of the input is ignored.
Formatting integer output
%wd
In this case, the w denotes the minimum width of the data and d is for integers. If the length of the variable is less than the width then the value is printed right-justified with leading spaces. For e.g:

Case 1: When the length of the variable is less than the specified width.

```
printf("a=%2d,b=%3d", a, b);
```

If a = 4 and b = 23, then the output will be:
Expected Output:

```
a=•4,b=•23
```

In this case, width specified for the first variable is 2 and the length of the output is only 1 digit(since the number is 4), as a result, one leading space is added before 4. The space character is represented using a • character. Similarly, the width of the second variable is 3 while the length of the output is only 2 digit (since the number is 23), so once again a leading space is added before 23.

Case 2: When the length of the variable is equal to the width specified no leading space is added.

```
printf("a=%3d,b=%4d", a, b);
```

If a = 456 and b = 2234, then
Expected Output:

```
a=456,b=2234
```

Case 3: When the length of the variable is more than the width specified then the output is printed correctly despite the length of the variable.

```
printf("a=%2d,b=%3d", a, b);
```

If a = 1221 and b = 19234, then

Expected Output:

a=1221,b=19234

Formatting Floating Point Input

%wf

Here w is an integer number specifying the maximum width of the input data including digits before and after decimal points and the decimal itself.

Case 1: When the length of the input data is less than the given width, then the values are stored correctly in the given variables.

```
scanf("%3f%4f", &a, &b);
```

Input: 4 1.2

In this case the maximum width of the first variable is 3, while the length of the input is 1, similarly, the width of the second variable is 4, while the length of the input is 3. So the values are stored correctly in the variables. i.e a = 4 and b = 1.2.

Case 2: When the length of input data is equal to the width, then the values are stored correctly in the variables.

```
scanf("%3f%4f", &a, &b);
```

Input: 1.2 33.1

In this case, the width and length of the input are same, so the values are stored correctly in the variables. i.e a = 1.2 and b = 33.1.

Case 3: When the length of input data is greater than the width specified then the values are not stored correctly in the variables.

```
1 scanf("%3f%4f", &a, &b);
```

Input: 5.21 983.71

As the width of the first variable is 3 only 5.2 is stored in the variable a while 1 is stored in b, and the rest of the input is ignored.

Formatting Floating Point Output

%w.nf

The w is the minimum width of output data and n is the digits to be printed after the decimal point. Note that width includes digits before and after the decimal point and the decimal itself.

Case 1: When the length of the output data is less than width specified, then the numbers are right justified with the leading spaces.

```
printf("a=%5.1f, b=%5.2f", a, b);
```

where a = 3.1 and b = 2.4

Expected Output:

a=••3.1, b=•2.40

In this case width of the variable, a is 5 and length of output data is 3, that's why two leading spaces are added before 3.1 . Similarly, the width of the variable b is 5 and length of output data is 3 , but since the number of digits to be printed after the decimal point is 2 only one leading space is added before 2.4.

Case 2: When the length of data is equal to the width specified, then the numbers are printed without any leading spaces.

```
printf("a=%4.2f, b=%4.2f", a, b);
```

where a = 32.1 and b = 45.11.

Expected Output:

```
a=32.10, b=45.11
```

Case 3: When the length of data is greater than the width specified, then the numbers are printed without any leading spaces.

```
printf("a=%5.2f, b=%4.3f", a, b);
```

where a = 34189.313 and b = 415.1411.

Expected Output:

```
a=34189.31, b=415.141
```

Formatting String Input

%ws

Here w specify the length of the input to be stored in the variable.

```
char str[20];

scanf("%4s", str)
```

Note: Strings in C is declared as an array of characters, we will learn more about arrays and string in lesson String Basics in C.

If the input is earning then only earn will be stored in the variable str.

Formatting String Output

%w.ns

The w is the width of the string. The dot (.) character after w and n are optional. If present only n characters will be displayed and (w-n) leading spaces will be added before the string. On the other hand, if only width of the string(i.e w) is specified and the length of the string is less than the width specified then the output will be right-justified with leading spaces. Otherwise, no leading space is added.

Case 1:

```
printf("%4s", "codeindepth");
```

Expected Output:

```
codeindepth
```

Here width of the string is less than the length of the input, so the string will be printed with no leading spaces.

Case 2:

```
printf("%10s", "code");
```

Expected Output:

```
•••••••code
```

Here width of the string is 10 and the length of the string is 4, so the string will be printed with 6 leading spaces.

Case 3:

```
1 printf("%10.3s", "code");
```

Expected Output:

```
1 •••••••cod
```

Here width of the output is 10 but .3 indicates that only 3 characters will be displayed. The length of the string is 4, so only "cod" will be displayed along with 7 (10-3=7) leading spaces.

Case 4:

```
1 printf("%.6s", "codeindepth");
```

Expected Output:

```
1 codein
```

Here width of the input is not specified but .6 indicates that whatever be the length of input string only the first 6 character from the string will be displayed.

5.16 FILE ACCESS

There are **four** basic **file operations**:

1. Opening a file
2. Reading data from a file
3. Writing data to a file
4. Closing a file

A **file** has to be opened before performing read and write operations. On opening a file,the operating system provides a handle to perform various **file operations** using different functions.

A request made to the operating system for opening a file can either **succeed** or **fail**. If the request **succeeds**, the operating system returns a file descriptor which is a pointer to the structure FILE. If the request **fails**, a **NULL** is returned.

A **file** can be opened by using fopen(). This function returns the file descriptor, as **FILE** pointer.

The declaration of fopen() as available in **stdio.h** is:

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

Where filename is a sequence of characters that make up a valid filename on the given operating system.
Note: The **filename** can also include the complete path of the file.
The parameter mode determines the way in which the opened file will be used (i.e, for reading, writing or appending, etc.)

The **mode** parameter also allows us to read or write data either in text or binary format.

Different modes for **text files** are:

Mode	Description
r	Opens a text file only for reading
w	Opens a text file only for writing
a	Opens a text file for appending
r+	Opens a text file for reading and writing
w+	Opens a text file for reading and writing
a+	Append a text file for read/write

Different modes for **binary files** are:

Mode	Description
rb	Opens a binary file only for reading
wb	Opens a binary file only for writing
ab	Opens a binary file for appending
rb+	Opens a binary file for reading and writing
wb+	Opens a binary file for reading and writing

ab+	Append a binary file for read/write

The data structure of a file descriptor is defined as FILE in the standard library **stdio.h**. Hence, all files are declared as type FILE before they are used.

Consider an example:

```
FILE *fp; // fp is a FILE pointer variable that represents the file descriptor
fp = fopen("sample.txt", "r"); // fopen() function opens a file "sample.txt" in read mode
```

Note the difference in the **fopen()** operation in the different **modes**:

- When the mode is **write only**, a file with a specified name is created if the file does not exist and the contents are **deleted** if the file already exists.
- When the mode is **read only**, the file is opened with the current contents, if the file already exists, else an error occurs.
- When the mode is **append**, the file is opened with the current contents if the file exist, else a new file with the give name and path is created.
- The file that is opened by using fopen() should be closed after the work is over on that file. A file close will **flush out** all the entries from buffer.

  Files are closed with the function fclose(). By closing a file after operations performed on that file will save all the modifications done on the file.

  The general syntax of fclose() is:

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

- The fclose() function closes the file that is being pointed by file pointer fp.

  An important point regarding fclose() is closing a file will prevent misuse of the **FILE**.

  If an error occur in closing a file then fclose() returns EOF (End Of File). For successful close operation it returns 0.

  Let us consider an example:

```
FILE *fp;
fp = fopen("sample.txt", "r");
........... // perform operations on file
fclose(fp); // disassociate the file with the stream
```

- The functions getc() and fgetc() are used to **read a character** from the file which is opened for reading purpose. Both functions have same syntax and their working is also same.

  The general syntax of getc() is:

- int getc(FILE *fp);
- The getc() function returns the **ASCII** equivalent of the character that is being read. If an error occurs in reading,then it returns **EOF**.

  Consider the following example:

- FILE *fp;
- fp = fopen("sample.txt", "r");
- ch= getc(fp);
- Here, a single character is read from the file with logical name fp. That character is assigned to character variable ch.

  The functions putc() and fputc() are used to **write a character** into a file which is opened for writing.

  The general syntax of putc() is:

- int putc(int c, FILE *fp);
- If an error occurs, it returns **EOF**.

  Consider the following example:

- putc(ch, fp);
- Here, the statement will write the character specified by ch into the file pointed by fp.

5.17 ERROR HANDLING-STDERR AND EXIT

It is possible that an error may occur while performing **input** or **output** operations on a file. The typical situations includes:

- Trying to read beyond the end of the file mark
- Device overflow
- Trying to use a file that has not been opened for that purpose
- Trying to perform an operation on a file when it is opened for another type
- Opening a file with an invalid filename
- Attempting to write onto a write protected file

If the user fails to check for such **read** and **write** errors, program may terminate **abnormally** For this reason **C** language provides some **library functions** to detect errors If any, while read/write operations are carried out.

Some such functions are as given below:

1.feof(): The feof() function can be used to check the **end of file** condition.

It takes a file pointer as its argument and returns **non-zero** if the file pointer is at the end of file, Else it returns **zero**.

For example,Consider the program given below:

```
if (feof(fp)) {
        printf("End of data\n");
}
```

2.ferror(): The ferror() function is used to detect errors that might occur during **read/write** operation on a file. It returns a **zero** when the attempt is successful and **non-zero** in case of failure.

Consider the following example:

```
if (ferror(fp)) {
        printf("Error occurred during read/write\n" );
}
```

3.perror(): The perror() is a standard library function which prints the error message specified by the **compiler**. To print the error message user can use **perror()** instead of **printf**().

Consider the following example:

```
if (ferror(fp)) {
        perror("Error occurred during read/write\n");
}
```

5.18 LINE INPUT AND OUTPUT

The fgets **function** reads a sequence of character, i. e., a character string from an input stream. Its prototype is given below.

char *fgets(char *s , int n, FILE*fp);

The characters from the input stream are read into a character arrays until a newline character is read, n - 1 characters are read or the end-of-file is reached. If a newline character is read, it is retained in the character array. The function then adds a terminating null character to strings and returns it. However, the function returns a NULL value if an end-of-file is encountered before any character is read or an error occurs during input.

The fputs function writes a character string to an output stream. Its format is given below.

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

All the characters in strings except the null terminator are written to stream fp. The function returns the numbers of characters written to the stream or EOF if an error occurs during output.

These functions are similar to putchar () and getchar () which we have already used for standard input/output. The functions may also be used for standard input/output by changing the file stream. The header file <stdio.h> should be included in the program which uses these functions.

A program to display a specified text file using the fgets and fputs functions is given below.

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
  //Header fileincluded for function exit()
  FILE* Fptr;
  char ch;
  Fptr = fopen("Myfile","w");
  clrscr();
  if(Fptr ==NULL)
      //open myfilefor writing.
     // If return value is NULL exit
     {
      printf("File could not be opened.");
      exit (1);
     }
  else
   {
      printf("File is open. Enter the text.\n");
      while ((ch= getchar()) != EOF) /*get characters from keyboard*
      fputc(ch, Fptr); //one by one and write it to file.
      fclose(Fptr);
    } //this code closes the file
    Fptr = fopen("Myfile","r"); // open file for reading
    if(Fptr ==NULL) // if return value is NULL, exit.
      {
        printf("File could not be opened");
        exit(1);
      }
    else
       {
         printf("File is opened again for reading.\n");
         while ((ch= fgetc(Fptr)) != EOF) // get characters from file
           printf("%c",ch);
           fclose(Fptr);
```

```
    }  // till EOF and display it.
}         // This line closes the file.
```

The expected output is as given below. The EOF is included by pressing Ctrl and Z. It is indicated by ^Z in the third line of output.

5.19 MISCELLANEOUS FUNCTIONS.

Descriptions and example programs for C environment functions such as getenv(), setenv(), putenv() and other functions perror(), random() and delay() are given below.

Miscellaneous functions Description

getenv( ): This function gets the current value of the environment variable
setenv( ): This function sets the value for environment variable
putenv(): This function modifies the value for environment variable
perror( ): Displays most recent error that happened during library function
callrand( ): Returns random integer number range from 0 to at least 32767
delay(): Suspends the execution of the program for particular t