1. Define structure with syntax. Explain different types of structures.

Sol: A structure is a collection of one or more declaration of variables of same data type or dissimilar data types, grouped together as a single entity.

Syntax:

```
Struct
{
  type1  member 1 ;
  type2  member 2 ;
  .....  ..........
};
```

Example:

```
Struct
{
  char name[20];
  int  marks ;
  float average;
};
```

⇒ The variables defined inside the structure are called members of the structure.

⇒ All members are logically related data items.

The structures can be classified as

1. Tagged structure
2. Tagless structure
3. Type defined structure

1 Tagged structure:

⇒ In the structure definition, the keyword Struct can be followed by an identifier. This identifier is called tagname.

⇒ The structure definition associated with tagname is called tagged structure or named structure.

Syntax:

```
Struct tag_name
{
  type1 member1;
  type2 member2;
  ....
};
```

Example:

```
Struct student
{
   char name[10];
   int  marks;
   float average ;
};
```

2. tagless or un-named structure

⇒ In the structure definition, the keyword struct is not followed by an identifier.

⇒ The structure definition without tagname is called tagless structure or un-named structure.

Syntax:

```
struct
{
    type1 member1;
    type2 member2;
    ....  .......
};
```

Example:

```
struct
{
    char name[10];
    int marks;
    float average;
};
```

3. type defined structure.

⇒ In the structure definition, the keyword struct is not followed by an identifier.

⇒ It is a type of tagless structure. But, it is preceded by keyword typedef.

⇒ The structure definition with the keyword typedef is called type defined structure.

⇒ using this type defined structure, we can declare variables.

Syntax:

```
typedef struct
{
    type1 member1;
    type2 member2;
    ....  .....
} TYPE ID;

TYPED V1, V2, ... Vn;
```

Example:

```
typedef struct
{
    char name[10];
    int marks;
    float average;
} STUDENT;

STUDENT cse;
```

2. Difference between structure and union.

structure	union
1. Separate memory locations are allocated for allocated for every member of the structure	1. The memory is allocated and its size is equal to maximum size of a member.
2. Each member within a structure is assigned unique address	2. The address is same for all members
3. The address of each member is greater than the address of its previous member	3. The address is same for all members
4. Altering the value of one member will not affect other members of the structure	4. Altering the value of one member affects other member as the memory is shared.
5. Several members of a structure can be intialized	5. Only the first member of the union can be initialized.
6. Size of structure is >= Sum of sizes of its members.	6. Size of union is = size of largest member.

Example:

```
Struct
{
    char name[10];
    int marks;
    double average;
} cse;
```

Example:

```
union
{
    char name[10];
    int marks;
    double average;
} cse;
```

3. Define pointer to structure, Explain how to access member of structure using Star, Dot and Arrow operator with Example.

Sol^n A variable which contains address of a structure variable is called pointer to a structure.

Example

```
typedef struct
{
    char name[10];
    int  marks;
    float average;
} STUDENT;

STUDENT a= {"ram", 25, 24.5};
STUDENT *p;
P = &a;
```

Output:

```
ram  25  24.5
```

⇒ The variable p holds the address of a structure variable. So, the variable p is pointer to a structure.

1. The members of a structure can be accessed using dot operator denoted by . and dereferencing operator denoted by *

⇒ A member of a structure can be accessed by writing * followed by pointer variable but enclosed within parentheses followed by a dot and member name.

Syntax:

```
(*pointer_variable).member
```

```
#include <stdio.h>
void main()
{
    typedef struct
    {
      char name[10];
       int  marks;
       float average;
    } STUDENT;
```

```
STUDENT  a = {"ram", 25, 24.5};
STUDENT  *p;
P = &a;
Printf ("%s", (*p).name);
Printf ("%d", (*p). marks);
Printf ("%f", (*p). average);
}
```

2. The members of a structure can be accessed using arrow operator.

A member of a structure can be accessed by writing pointer variable followed by arrow operator in turn followed by member name.

Syntax:

```
Pointer_variable -> member
```

```
# include <stdio.h>
void main()
{
    typedef struct
    {
        char   name[10];
        int    marks;
        float  average;
    } STUDENT;

    STUDENT  a = {"ram", 25, 24.5};
    STUDENT  *p;
    P = &a;
    Printf ("%s", p -> name);
    Printf ("%d", p -> marks);
    printf ("%f", p -> average);
}
```

Output

```
ram  25  24.5
```

H. Define union. Write the syntax for defining union. How memory is allocated for union.

Sol^n A union is a collection of one or more declaration of variables of same data type or dissimilar data types, grouped together as a single entity.

Syntax:

```
union
{
    type1  member1;
    type2  member2;
    ....   ......
};
```

Example:

```
union
{
    char  name[20];
    int   marks;
    float double;
};
```

1. tagged union.

In the union definition, the keyword union can be followed by an identifier.

- This identifier is called tag name.
- The union definition associated with tag name is called tagged/named union.

Syntax:

```
union tag-name
{
    type1  member1;
    type2  member2;
    ....   ......
};
union tag_name V1, V2, ... Vn;
```

Example:

```
union student
{
    char name[10];
    int  marks;
    double average;
};
union student cse;
```

2. tagless union

In the union definition, the keyword union is not followed by an identifier.

- That means, there is no tag associated with the union.
- The union definition without tag name is called tagless union
- Since, there is no name associated with keyword union, it is

also called name less union

Syntax

```
union
{
   type1 member1;
   type2 member2;
   ....  ....
} v1, v2, ... vn;
```

Example:

```
union
{
   char name[10];
   int marks;
   float average;
} cse;
```

2. Type_defined union

In the union definition, the keyword union is not followed by an identifier.

- It is a type of tagless union. But, it is preceded by a keyword typedef.
- The union definition with keyword typedef is called type-defined union.
- The type_defined union must be followed by an identifier ending with semicolon.

Syntax:

```
typedef union
{
   type1 member1;
   type2 member2;
   ....  .....
} TYPE_ID;
TYPE_ID v1, v2, ... vn;
```

Example:

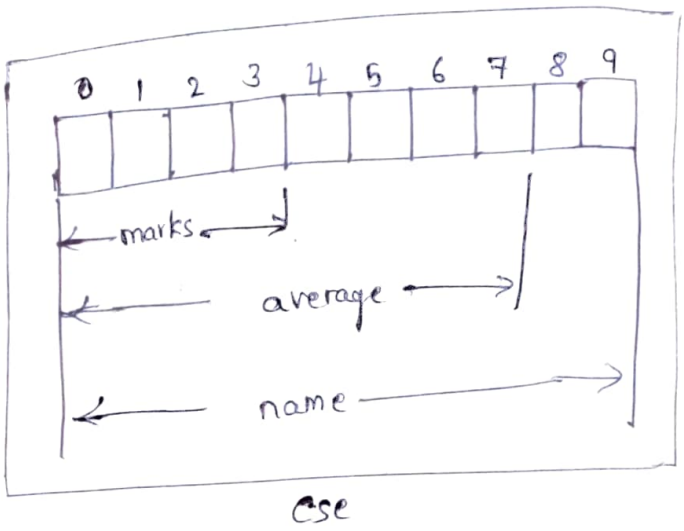
```
typedef union
{
   char name[10];
   int marks;
   float average;
} STUDENT;
STUDENT cse;
```

memory is allocated for union.

- A block of memory is allocated.
- The memory allocated by the compiler is large enough to hold the largest member of the union.
- So, the size of block is size of the largest member of the union.
- All the members share the same set of memory locations.
- At any point of time only one member can be accessed and change of one member affects the other member.

Example:

```
union student
{
    char name[10];
    int marks;
    double average;
};
union student cse;
```



5. Write a c program to sort student details in increasing order of average marks using structure.

```
#include <stdio.h>
typedef struct
{
    char name[10];
    int marks[3];
    float average;
} STUDENT;
```

```
Void sort_student_info (STUDENT a[], int n)
{
    int i, j;
    STUDENT temp;
    for (j=1; j<n; j++)
    {
        for (i=0; i<n-j; i++)
        {
            if (a[i].average > a[i+1].average)
            {
                temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
            }
        }
    }
Void print_student_info (STUDENT a[], int n)
{
    int i, j;
    Print("Name marks1 marks2 marks3 average\n");
    for (i=0; i<n; i++)
    {
        Printf("%s", a[i].name);
        for (j=0; j<3; j++)
            Printf("%d", a[i].marks[j]);
        Printf("%f", &a[i].average);
    }
}
```

```
Void read_student_info(STUDENT a[], int n)
{
    int i, j;
    Printf("Name mark1 mark2 mark3 average\n");
    for(i=0; i<n; i++)
    {
        Scanf("%s", a[i].name);
            for(j=0; j<3; j++)
                Scanf("%d", &a[i].marks[j]);
            Scanf("%f", &a[i].average);
    }
}
Void main()
{
    STUDENT a[10];
    int     n;
    Printf("Enter no. of students:");
    Scanf("%d", &n);
    read_student_info(a,n);
    Sort_student_info(a,n);
    print_student_info(a,n);
}
```

6 What is nested structure. How to initialize and access members of nested structure.

Ans A structure inside a structure is called nested structure. As we declare variables inside a structure, a structure can also be declared inside a structure. So, a structure whose member itself is a structure is a nested structure.

~~Syntax~~

```
typedef struct
{
    int mark1;
    int mark2;
    int mark3;
} MARKS;
```

~~Example~~

```
typedef struct
{
    char name[10];
    MARKS m;
    float average;
} STUDENT;
STUDENT a;
```

Initialize the members of a nested structure

- The variable in the declaration must be followed by '=' sign and followed by data items.
- The data items that are to be initialized must be separated by commas.
- The data items that are to be initialized must be enclosed within braces.

~~Syntax~~ Example:

```
typedef struct
{
    int mark1;
    int mark2;
    int mark3;
} MARKS;

typedef struct
{
    char name[10];
    MARKS m;
    float average;
} STUDENT;
STUDENT a = { "ram",
              { 25, 24, 23 },
              98.5
            };
```

Access the members of a nested structures

The data stored in each member can be accessed using dot operator as shown below:

Example

```
typedef struct
{
    char name[10];
    MARKS m;
    float average;
} STUDENT
```

```
typedef struct
{
    int mark1;
    int mark2;
    int mark3;
} MARKS;
```

```
a.name       // ram
a.m.mark1    // 25
a.m.mark2    // 24
a.m.mark3    // 23
a.average    // 98.5
```

7. Define Data structure. Explain the classification of data structure with example.

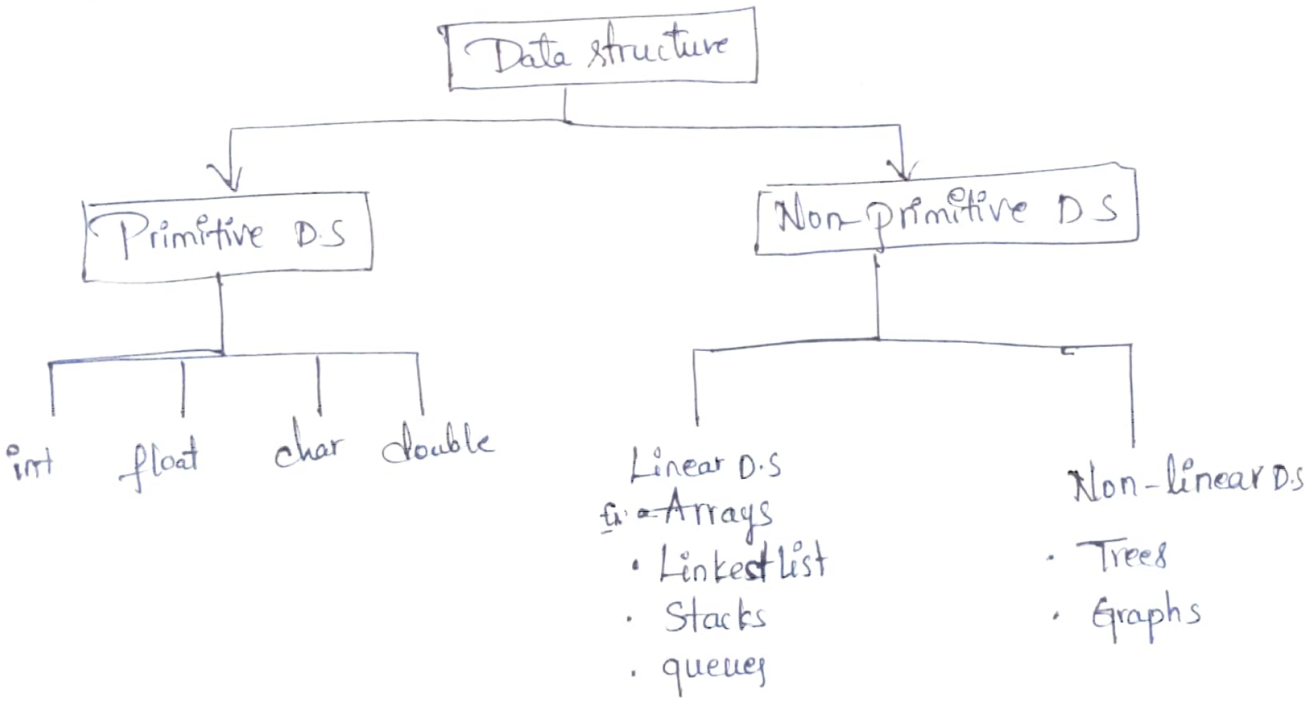
Sol^n Data structure is a logical or mathematical model of storing and organizing data in a particular way in a computer required for designing and implementing efficient algorithms and program development.

(or)

Data structure is a way of organizing the data along with relationship among data.

Classification of data structure :

DS are classified into primitive & Non-primitive data structure



(1) Primitive Data Structure

- These are the fundamental standard data types.
- These are used to represent single values

(2) Non-primitive data structure:

- These are derived from primitive data types.
- These are used to store group of values.

Eg: Arrays, stacks Etc

- Non-primitive data structure are further classified into linear and Non-linear D.S

(a) linear D.S: A data structure is said to be linear, if its elements are stored in sequential order

Ex: Arrays, stacks, queues, linked list.

(b) Non-linear D.S: A D.S is said to be non-linear if the data is not arranged in a sequential order.

Ex Trees, graphs.

- Elements are stored on hierarchical relationship among the data Ex: Trees
- Graphs: used to represent data that has relationship between pair of elements Ex: Graphs, routes, networks.

8 Explain the different operations performed on data structure

Data appearing in our data structures are processed by means of certain operations.

4 major operations

① ~~Traits~~ Traversing: accessing each record (element) exactly once so that certain items may be processed.

② Searching: Finding the location of the record with a given key value or finding the location of all records which satisfy one/more conditions

③ Insertion: Adding a new record into the data structure.

④ Deletion: removing a record from the structure

→ Something 2 or more operations may be used in a given situation. Ex: we want to delete the element the element with given key i.e., first search for location and delete.

Other operations like:

① Sorting: arranging the records in ascending/descending order

② Merging: Combining two data structures

⑨ Write a C program to insert and delete an item in a specified position of an array.

```
#include <stdio.h>
#include <stdlib.h>
// Delete item at the specified position.
int delete_at_pos(int a[], int n, int pos) {
    if (pos<0 || pos>=n) {
        printf("Invalid position\n");
        return n;
    }
    for (int i = pos; i<n-1; i++) {
        a[i] = a[i+1];
    }
    return n-1;
}
```

```
// Insert item at the specified position
int insert_at_pos (int item, int a[], int n, int pos)
{
  if (Pos<0 || Pos>n)
  { Printf ("Invalid Position\n");
    return n;
  }
  for (int i=n-1; i>=Pos; i--)
  { a[i+1] = a[i];
  }
  a[Pos] = item;
  return n+1;
}
// Display array elements
void print_array (int a[], int n)
{ for (int i=0; i<n; i++)
  { Printf ("%d", a[i]);
  }
  Printf("\n");
}
int main () {
  int choice, item, pos, a[10], n=0;
  for (;;)
  { Printf ("1: Insert 2: Delete 3: Display 4: Exit\n");
    Printf ("Enter your choice:");
    Scanf ("%d", &choice);
    Switch (choice) {
    {
```

```
            Printf ("Enter the item: ");
            Scanf ("%d", &item);
            Printf ("Enter the position: ");
            Scanf ("%d", &pos);
            n = insert_at_pos(item, a, n, pos);
        case 2:
            Printf ("Enter the position: ");
            Scanf ("%d", &pos);
            n = delete_at_pos(a, n, pos);
            break;
        case 3:
            Printf ("Array elements: ");
            print_array(a, n);
            break;
        case 4:
            Exit(0);
        default:
            Printf ("Invalid choice\n");
    }
  }
  return 0;
}
```

n. Explain the dynamic memory allocation functions supported by c with syntax and Examples.

10. explain the dynamic memory allocation functions supported by c with syntax and examples

* Memory management is important task in computer Programming.

There two types of memory management

1. Static memory management.

2. Dynamic Memory management.

Static Memory Management: The allocation & deallocation of memory is done during Compilation time of the Program.

EX: int a [10];

Dynamic Memory Management: - The memory allocation & deallocation is performed during run-time of the Program. Thus, When Program is getting executed at that time memory is managed. This is the efficient method compared to static memory management.

* There are four functions used in DMA:

malloc (): allocates a block of memory.

- It is used to allocate memory space as Per requirements.
- Function allocates memory & return a pointer of type void * to the start that memory block. * If function fails, it returns NULL. It is necessary to verify Pointer returned is not NULL.
- This function, does not initialize the memory allocated during execution. It carries garbage values.

Syntax:
Ptr = (datatype *) malloc (size);

Ptr- Pointer variable of type datatype.
datatype - any c datatype or user defined datatype.
Size- no. of bytes required.

EX:
int * Ptr ;
Ptr = (in t*) malloc(10) ;

Calloc (): allocate multiple blocks of memory.

* It is similar to malloc, but it initializes the allocated memory to Zero.

Syntax:

Ptr = (datatype*) calloc(n, size)

n - no. of blocks to be allocated.

Ex:
Ptr = calloc (20, sizeof (int));

This function computes memory required for 20x2 bytes
(for int) = 40 bytes of memory. Block is allocated.

Realloc (): used to modify the size of allocated block by malloc(), calloc() to new size.

* If allocated memory space is not Sufficient, then additional memory can be taken using realloc() .

* Can also be used to reduce the size of already allocated memory.

Syntax:

Ptr = (datatype*) realloc(ptr, size) ;

Ex:

```
char *ptr;
ptr = malloc(100);
ptr = realloc(ptr,1000);
```

free():

The free() function in C is used to free or deallocate the dynamically allocated memory and helps in reducing memory wastage.

The C free() function cannot be used to free the statically allocated memory (e.g., local variables) or memory allocated on the stack.

It can only be used to deallocate the heap memory previously allocated using malloc(), calloc() and realloc() functions.

Syntax:

void free(void *ptr)

Example:

Free(str);

11.Write a c program to add two polynomials.

```
#include <stdio.h>

typedef struct {
   int c;
   int px;
} POLY;

void print_polynomial(POLY a[], int n)
{
   int i;

   for (i = 0; i < n; ++i)
   {
      if (a[i].c >0)

         printf("+ %dx^%d", a[i].c,a[i].px);
      else
         printf("%dx^%d ", a[i].c, a[i].px);

      printf("/n");
   }
}


void read_polynomial(POLY a[], int n)
{
   int i;

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

      scanf("%d %d", &a[i].c, &a[i].px);
   }
}

int add_2_polynomials(POLY p[], int m, POLY q[], int n, POLY r[])
{
   int i = 0, j = 0, k = 0,sum;

   while (i < m && j < n)
   {
      if(p[i].px == q[i].px)
      {
         sum = p[i].c +q[j].c;
```

```
            if(sum != 0)
            {
               r[k].c = sum, r[k].px = p[i].px;
               k++;
            }
            i++,j++;

         }
         else if (p[i].px > q[j].px)
         {
            r[k].c = p[i].c, r[k].px = p[i].px;
            i++,j++;

         }
         else
         {
            r[k].c = q[j].c, r[k].px = q[j].px;
            j++,k++;

         }
      }
      while(i < m)
      {
         r[k].c = p[i].c, r[k].px = p[i].px;
         i++,j++;
      }
      while(j < n)
      {
         r[k].c = q[j].c, r[k].px = q[j].px;
         j++,k++;
      }
      return k;
}
void main() {
      POLY a[20], b[20], c[40];
      int m, n,x;

      printf("Enter the number of terms in the 1st polynomial: ");
      scanf("%d", &m);

      printf("Enter the terms of the 1st polynomial:\n");
      read_polynomial(a, m);

      printf("Enter the number of terms in the 2nd polynomial: ");
      scanf("%d", &n);
```

```
    printf("Enter the terms of the 2nd polynomial:\n");
    read_polynomial(b, n);

    x = add_2_polynomials(a, m, b, n, c);

    printf("Polynomial 1: ");
    print_polynomial(a, m);

    printf("Polynomial 2: ");
    print_polynomial(b, n);

    printf("Polynomial 3 (Sum of Polynomial 1 and Polynomial 2): ");
    print_polynomial(c, x);
}
```

12.Define sparse matrix. Express the following matrix in triple form and find its transpose.

$$A=\begin{bmatrix} 15 & 0 & 0 & 22 \\ 0 & 11 & 3 & 0 \\ 0 & 0 & 0 & -6 \\ 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 \end{bmatrix}$$

A sparse matrix is a matrix in which the majority of its elements are zero. To represent a sparse matrix in triple form, we use three arrays:

Row array (rows[]): Stores the row index of the non-zero elements.

Column array (cols[]): Stores the column index of the non-zero elements.

Value array (vals[]): Stores the non-zero values.

```
#include <stdio.h>

void transposeSparse(int rows[], int cols[], int vals[], int n, int tRows[], int tCols[], int tVals[]) {
    for (int i = 0; i < n; ++i) {
        tRows[i] = cols[i];
        tCols[i] = rows[i];
        tVals[i] = vals[i];
    }
}
```

```
int main() {
    // Example sparse matrix in triple form
    int rows[] = {0, 0, 1, 1, 2, 4, 5};
    int cols[] = {0, 3, 1, 2, 3, 0, 2};
    int vals[] = {15, 22, 11, 3, -6, 91, 28};
    int n = sizeof(rows) / sizeof(rows[0]);

    // Arrays for the transpose
    int tRows[n], tCols[n], tVals[n];

    // Obtain the transpose of the sparse matrix
    transposeSparse(rows, cols, vals, n, tRows, tCols, tVals);

    // Print the transpose
    printf("Transpose of the sparse matrix in triple form:\n");
    for (int i = 0; i < n; ++i) {
        printf("(%d, %d, %d)\n", tRows[i], tCols[i], tVals[i]);
    }

    return 0;
}
```