UNIT 8: STRUCTURE AND UNION , ENUMERATION AND BIT OPERATORS

UNIT STRUCTURE

8.0 Objectives

After going through this unit you should be able to :

- understand the concept of Structure
- Define and declare a structure variable
- Implementation of arrays in structures •
- use pointer for structure variable
- understand the concept and use of Union
- Differentiate between Union and structure
- do bitwise manipulation
- understand the concept of enumerated data type

8.1 Introduction to Structure in C

1. As we know that array is collection of the elements of same type , but many time we have to store the elements of the different data types.
2. Suppose Student record is to be stored , then for storing the record we have to group together all the information such as Roll ,name, Percent which may be of different data types.
3. Ideally Structure is collection of different variables under single name.
4. Basically Structure is for storing the complicated data.
5. A structure is a convenient way of grouping several pieces of related information together.

8.1.1 Definition of Structure in C:

Structure is composition of the different variables of different data types , grouped under same name.

```
typedef struct {
        char name[64];
        char course[128];
        int age;
        int year;
  } student;
```

Some Important Definitions of Structures :

1. Each member declared in Structure is called **member.**

```
char name[64];
char course[128];
int age;
int year;
```

are some examples of members.

2. Name given to structure is called as **tag**

```
student
```

3. Structure **member** may be of **different data type** including **user defined data-type** also

```
typedef struct {
        char name[64];
        char course[128];
        book b1;
        int year;
 } student;
```

Here book is user defined data type.

C Structure is a collection of different data types which are grouped together and each element in a C structure is called member.

- If you want to access structure members in C, structure variable should be declared.
- Many structure variables can be declared for same structure and memory will be allocated for each separately.
- It is a best practice to initialize a structure to null while declaring, if we don't assign any values to structure members.

DIFFERENCE BETWEEN C VARIABLE, C ARRAY AND C STRUCTURE:

- A normal C variable can hold only one data of one data type at a time.
- An array can hold group of data of same data type.
- A structure can hold group of data of different data types and Data types can be int, char, float, double and long double etc.

Note the following points while declaring a structure type:

- The closing brace in the structure type declaration must be followed by a semicolon.
- It is important to understand that a structure type declaration does not tell the compiler to reserve any space in memory. All a structure declaration does is, it defines the 'form' of the structure.
- Usually structure type declaration appears at the top of the source code file, before any variables or functions are defined. In very large programs they are usually put in a separate header file, and the file is included (using the preprocessor directive #include) in whichever program we want to use this structure type.

C Structure:

Syntax	struct student
	{
	int a;
	char b[10];
	}

Example	a = 10; b = "Hello";

C Variable:

int	Syntax: int a; Example: a = 20;
char	Syntax: char b; Example: b='Z';

C Array:

int	**Syntax:** int a[3];
	Example:
	a[0] = 10;
	a[1] = 20;
	a[2] = 30;
	a[3] = '\0';

char	**Syntax:** char b[10]; **Example:** b="Hello";

BELOW TABLE EXPLAINS FOLLOWING CONCEPTS IN C STRUCTURE.

1. How to declare a C structure?
2. How to initialize a C structure?
3. How to access the members of a C structure?

Using normal variable	**Using pointer variable**

Syntax:
struct tag_name
{
data type var_name1;
data type var_name2;
data type var_name3;
};
	Syntax:
	struct tag_name
	{
	data type var_name1;
	data type var_name2;
	data type var_name3;
	};

Example:
struct student
{
int mark;
char name[10];
float average;
};
	Example:
	struct student
	{
	int mark;
	char name[10];
	float average;
	};

Declaring structure using normal variable: struct student report;	**Declaring structure using pointer variable:** struct student *report, rep;
Initializing structure using normal variable: struct student report = {100, "Mani", 99.5};	**Initializing structure using pointer variable:** struct student rep = {100, "Mani", 99.5}; report = &rep;
Accessing structure members using normal variable: report.mark;	**Accessing structure members using pointer variable:** report -> mark;

report.name; report.average;	report -> name; report -> average;

EXAMPLE 8.1.1 :PROGRAM FOR C STRUCTURE:
This program is used to store and access "id, name and percentage" for one student. We can also store and access these data for many students using array of structures.

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

struct student
{
        int id;
        char name[20];
        float percentage;
};

int main()
{
        struct student record = {0}; //Initializing to null

        record.id=1;
        strcpy(record.name, "Raju");
        record.percentage = 86.5;

        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
        return 0;
}
```

In C we can group some of the user defined or primitive data types together and form another compact way of storing complicated information is called as Structure. Let us see how to declare structure in c programming language –

Syntax Of Structure in C Programming :

```
struct tag

{

  data_type1 member1;

  data_type2 member2;

  data_type3 member3;

};
```

Structure Alternate Syntax :

```
struct <structure_name>
{
   structure_Element1;
```

```
    structure_Element2;
    structure_Element3;
    ...
    ...
};
```

Some Important Points Regarding Structure in C Programming :

1. **Struct** keyword is used to declare structure.
2. **Members of structure** are enclosed within opening and closing braces.
3. **Declaration** of Structure reserves **no space**.
4. It is nothing but the " **Template / Map / Shape** " of the structure .
5. Memory is created , very first time when the **variable is created** / **Instance** is created.

8..1.2 Different Ways of Declaring Structure Variable :

Way 1 : Immediately after Structure Template

```
struct date
{
    int date;
    char month[20];
    int year;
}today;

// 'today' is name of Structure variable
```

Way 2 : Declare Variables using struct Keyword

```
struct date
{
   int date;
   char month[20];
   int year;
};

struct date today;
```

where "**date**" is name of structure and "**today**" is name of variable.

Way 3 : Declaring Multiple Structure Variables

```
struct Book
{
   int pages;
   char name[20];
   int year;

}book1,book2,book3;
```

We can declare multiple variables separated by comma directly after closing curly.

8.1.3 Initialization of structure

1. When we declare a structure, memory is not allocated for un-initialized variable.
2. Let us discuss very familiar example of structure student , we can initialize structure variable in different ways –

Way 1 : Declare and Initialize

```
struct student
```

```
{
  char name[20];
  int roll;
  float marks;
}std1 = { "Pritesh",67,78.3 };
```

In the above code snippet, we have seen that structure is declared and as soon as after declaration we have initialized the structure variable.

```
std1 = { "Pritesh",67,78.3 }
```

This is the code for initializing structure variable in C programming

Way 2 : Declaring and Initializing Multiple Variables

```
struct student
{
  char name[20];
  int roll;
  float marks;
}

std1 = {"Pritesh",67,78.3};
std2 = {"Don",62,71.3};
```

In this example, we have declared two structure variables in above code. After declaration of variable we have initialized two variable.

```
std1 = {"Pritesh",67,78.3};
std2 = {"Don",62,71.3};
```

Way 3 : Initializing Single member

```
struct student

{

  int mark1;
  int mark2;
  int mark3;
} sub1={67};
```

Though there are three members of structure,only one is initialized , Then remaining two members are initialized with Zero. If there are variables of other data type then their initial values will be –

Data Type	Default value if not initialized
integer	0
float	0.00
char	NULL

Way 4 : Initializing inside main

```
struct student
{
   int mark1;
   int mark2;
   int mark3;
};
```

```
void main()
{
struct student s1 = {89,54,65};
- - - - --
- - - - --
- - - - --
};
```

When we declare a structure then memory won't be allocated for the structure. i.e only writing below declaration statement will never allocate memory

```
struct student
{
   int mark1;
   int mark2;
   int mark3;
};
```

We need to initialize structure variable to allocate some memory to the structure.

```
struct student s1 = {89,54,65};
```

8..1.4 Nested Structures

Nested structure in C is nothing but structure within structure. One structure can be declared inside other structure as we declare structure members inside a structure.The structure variables can be a normal structure variable or a pointer variable to access the data.

EXAMPLE 8.1.4.1 : structure within structure in c using normal variable

This program explains how to use structure within structure in C using normal variable. "student_college_detail' structure is declared inside "student_detail" structure in this program. Both structure variables are normal structure variables.
Please note that members of "student_college_detail" structure are accessed by 2 dot(.) operator and members of "student_detail" structure are accessed by single dot(.) operator.

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

struct student_college_detail
{
   int college_id;
   char college_name[50];
};

struct student_detail
{
   int id;
   char name[20];
   float percentage;
   // structure within structure
   struct student_college_detail clg_data;
}stu_data;

int main()
```

```
{
   struct student_detail stu_data = {1, "Raju", 90.5, 71145,
                                "Anna University"};
   printf(" Id is: %d \n", stu_data.id);
   printf(" Name is: %s \n", stu_data.name);
   printf(" Percentage is: %f \n\n", stu_data.percentage);

   printf(" College Id is: %d \n",
              stu_data.clg_data.college_id);
   printf(" College Name is: %s \n",
              stu_data.clg_data.college_name);
   return 0;
}
```

- Structure written inside another structure is called as nesting of two structures.
- Nested Structures are allowed in C Programming Language.
- We can write one Structure inside another structure as **member** of another structure.

Way 1 : Declare two separate structures

```
struct date
{
  int date;
  int month;
  int year;
};

struct Employee
  {
  char ename[20];
  int ssn;
  float salary;
  struct date doj;
}emp1;
```

Accessing Nested Elements :

1. Structure members are accessed using **dot operator**.
2. '**date**' structure is nested within Employee Structure.
3. Members of the '**date**' can be accessed using 'employee'
4. **emp1 & doj are two structure names (Variables)**

Explanation Of Nested Structure :

Accessing Month Field : emp1.doj.month
Accessing day Field : emp1.doj.day
Accessing year Field : emp1.doj.year

Way 2 : Declare Embedded structures

```
struct Employee
{
  char ename[20];
  int ssn;
  float salary;
  struct date
```

```
    {
    int date;
    int month;
    int year;
    }doj;
}emp1;
```

Accessing Nested Members :

```
Accessing Month Field :  emp1.doj.month
Accessing day Field   :  emp1.doj.day
Accessing year Field  :  emp1.doj.year
```

Example :8.1.4.2

```
#include <stdio.h>

struct Employee
{
  char ename[20];
  int ssn;
  float salary;
  struct date
      {
      int date;
      int month;
      int year;
      }doj;
}emp = {"Pritesh",1000,1000.50,{22,6,1990}};

int main(int argc, char *argv[])
{
printf("\nEmployee Name   : %s",emp.ename);
printf("\nEmployee SSN    : %d",emp.ssn);
printf("\nEmployee Salary : %f",emp.salary);
printf("\nEmployee DOJ    : %d/%d/%d", \
       emp.doj.date,emp.doj.month,emp.doj.year);

return 0;
}
Output :
Employee Name   : Pritesh
Employee SSN    : 1000
Employee Salary : 1000.500000
Employee DOJ    : 22/6/1990
```

8.1.5 Array of Structure :

Structure is used to store the information of One particular object but if we need to store such 100 objects then Array of Structure is used.

Example 8.1.5:

```
struct Bookinfo

{
```

```
    char[20] bname;
    int pages;
    int price;
}Book[100];
```

Explanation :

1. Here Book structure is used to Store the information of one Book.
2. In case if we need to store the Information of 100 books then Array of Structure is used.
3. b1[0] stores the Information of 1st Book , b1[1] stores the information of 2nd Book and So on. We can store the information of 100 books.

Accessing Pages field of Second Book :

```
Book[1].pages
```

Live Example : 8.1.5.2

```
#include <stdio.h>
struct Bookinfo
{
    char[20] bname;
    int pages;
    int price;
}book[3];
int main(int argc, char *argv[])
{
int i;
for(i=0;i<3;i++)
   {
   printf("\nEnter the Name of Book    : ");
   gets(book[i].bname);
   printf("\nEnter the Number of Pages : ");
   scanf("%d",book[i].pages);
   printf("\nEnter the Price of Book   : ");
   scanf("%f",book[i].price);
   }
printf("\n--------- Book Details ---------- ");
for(i=0;i<3;i++)
```

```
    {
    printf("\nName of Book    : %s",book[i].bname);
    printf("\nNumber of Pages : %d",book[i].pages);
    printf("\nPrice of Book   : %f",book[i].price);
    }
return 0;
}
```

Output of the Structure Example:

```
Enter the Name of Book    : ABC
Enter the Number of Pages : 100
Enter the Price of Book   : 200
Enter the Name of Book    : EFG
Enter the Number of Pages : 200
Enter the Price of Book   : 300
Enter the Name of Book    : HIJ
Enter the Number of Pages : 300
Enter the Price of Book   : 500

--------- Book Details ------------

Name of Book    : ABC
Number of Pages : 100
Price of Book   : 200
Name of Book    : EFG
Number of Pages : 200
Price of Book   : 300
Name of Book    : HIJ
Number of Pages : 300
Price of Book   : 500
```

Accessing Element in Structure Array

1. Array of Structure can be accessed using dot[.] operator.
2. Here Records of 3 Employee are Stored.
3. ‘for loop’ is used to Enter the Record of first Employee.
4. Similarly ‘for Loop’ is used to Display Record.

Live Example: 8.1.5.2

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

struct Employee
{
   int ssn;
   char ename[20];
   char dept[20];
}emp[3];

//-----------------------------------------------
void main()
{
int i,sum;

//Enter the Employee Details
for(i=0;i<3;i++)
   {
   printf("nEnter the Employee Details : ");
   scanf("%d %s %s",&emp[i].ssn,emp[i].ename,emp[i].dept);
   }
//Print Employee Details
for(i=0;i<3;i++)
      {
      printf("nEmployee SSN   : %d",emp[i].ssn);
      printf("nEmployee Name  : %d",emp[i].ename);
      printf("nEmployee Dept  : %d",emp[i].dept);
      }
getch();
}
```

Output :

```
Enter the Employee Details : 1 Ram Testing
```

Employee SSN : 1

Employee Name : Ram

Employee Dept : Testing

8.1.6 Self Referential Structures

Structures can have members which are of the type the same structure itself in which they are included, This is possible with pointers and the phenomenon is called as self referential structures**.**

A self referential structure is a structure which includes a member as pointer to the present structure type. The general format of self referential structure is

struct parent

{ memeber1;

memeber2;

_____;

_____;

struct parent *name; };

The structure of type parent is contains a member, which is pointing to another structure of the same type i.e. parent type and name refers to the name of the pointer variable.

Here, name is a pointer which points to a structure type and is also an element of the same structure.

Example 8.1.6

struct element

{

char name{20};

int num;

struct element * value;

}

Element is of structure type variable.

This structure contains three members

– a 20 elements character array called **name**

- An integer element called **num**

- a pointer to another structure which is same type called **value.**

Hence it is self referential structure. These structure are mainly used in applications where there is need to arrange data in ordered manner.

8.1.7 Passing Structure to Function in C Programming

1. Structure can be passed to **function as a Parameter**.
2. function can also Structure as **return type**.
3. Structure can be passed as follow

Example :

```
#include<stdio.h>
#include<conio.h>
//---------------------------------------
struct Example
{
  int num1;
  int num2;
}s[3];
//---------------------------------------
void accept(struct Example *sptr)
{
  printf("\nEnter num1 : ");
  scanf("%d",&sptr->num1);
  printf("\nEnter num2 : ");
  scanf("%d",&sptr->num2);
}
//---------------------------------------
void print(struct Example *sptr)
{
  printf("\nNum1 : %d",sptr->num1);
  printf("\nNum2 : %d",sptr->num2);
}
//---------------------------------------
void main()
{
int i;
clrscr();
for(i=0;i<3;i++)
accept(&s[i]);
```

```
for(i=0;i<3;i++)
print(&s[i]);


getch();
}
```

Output :

```
Enter num1 : 10
Enter num2 : 20
Enter num1 : 30
Enter num2 : 40
Enter num1 : 50
Enter num2 : 60
Num1 : 10
Num2 : 20
Num1 : 30
Num2 : 40
Num1 : 50
Num2 : 60
```

In C, structure can be passed to functions by two methods:

1. Passing by value (passing actual value as argument)
2. Passing by reference (passing address of an argument)

Passing structure by value

A structure variable can be passed to the function as an argument as a normal variable.

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

C program to create a structure student, containing name and roll and display the information.

```
#include <stdio.h>
struct student
{
   char name[50];
   int roll;
};
```

```
void display(struct student stu);
// function prototype should be below to the structure declaration otherwise compiler shows error

int main()
{
   struct student stud;
   printf("Enter student's name: ");
   scanf("%s", &stud.name);
   printf("Enter roll number:");
   scanf("%d", &stud.roll);
   display(stud);   // passing structure variable stud as argument
   return 0;
}
void display(struct student stu){
  printf("Output\nName: %s",stu.name);
  printf("\nRoll: %d",stu.roll);
}
```

Output

```
Enter student's name: Kevin Amla
Enter roll number: 149
Output
Name: Kevin Amla
Roll: 149
```

Passing structure by reference

The memory address of a structure variable is passed to function while passing it by reference.

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

C program to add two distances (feet-inch system) and display the result without the return statement.

```
#include <stdio.h>

struct distance
{
   int feet;
```

```
    float inch;
};
void add(struct distance d1,struct distance d2, struct distance *d3);

int main()
{
    struct distance dist1, dist2, dist3;

    printf("First distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist1.feet);
    printf("Enter inch: ");
    scanf("%f", &dist1.inch);

    printf("Second distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist2.feet);
    printf("Enter inch: ");
    scanf("%f", &dist2.inch);

    add(dist1, dist2, &dist3);

    //passing structure variables dist1 and dist2 by value whereas passing structure variable dist3 by reference
    printf("\nSum of distances = %d\'-%.1f\"", dist3.feet, dist3.inch);

    return 0;
}
void add(struct distance d1,struct distance d2, struct distance *d3)
{
     //Adding distances d1 and d2 and storing it in d3
     d3->feet = d1.feet + d2.feet;
     d3->inch = d1.inch + d2.inch;

     if (d3->inch >= 12) {     /* if inch is greater or equal to 12, converting it to feet. */
```

```
        d3->inch -= 12;
        ++d3->feet;
    }
}
```

Output

```
First distance
Enter feet: 12
Enter inch: 6.8
Second distance
Enter feet: 5
Enter inch: 7.5

Sum of distances = 18'-2.3"
```

In this program, structure variables dist1 and dist2 are passed by value to the addfunction (because value of dist1 and dist2 does not need to be displayed in main function).

But, dist3 is passed by reference ,i.e, address of dist3 (&dist3) is passed as an argument.

Due to this, the structure pointer variable d3 inside the add function points to the address of dist3 from the calling main function. So, any change made to the d3 variable is seen in dist3 variable in main function.

As a result, the correct sum is displayed in the output.

Function as return type as Structure

```
#include <stdio.h>
typedef struct complex
{
   float real;
   float imag;
} complex;
complex add(complex n1,complex n2);

int main()
{
   complex n1, n2, temp;
```

```
    printf("For 1st complex number \n");
    printf("Enter real and imaginary part respectively:\n");
    scanf("%f %f", &n1.real, &n1.imag);

    printf("\nFor 2nd complex number \n");
    printf("Enter real and imaginary part respectively:\n");
    scanf("%f %f", &n2.real, &n2.imag);


    temp = add(n1, n2);
    printf("Sum = %.1f + %.1fi", temp.real, temp.imag);

    return 0;
}

complex add(complex n1, complex n2)
{
    complex temp;

    temp.real = n1.real + n2.real;
    temp.imag = n1.imag + n2.imag;

    return(temp);
}
```

Output

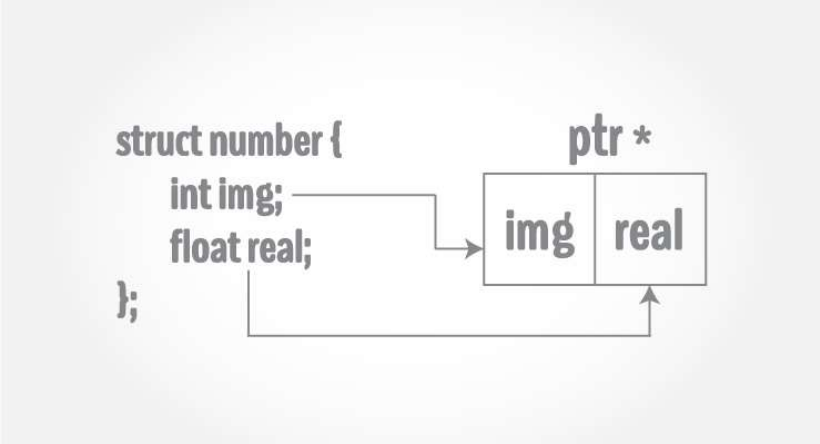
```
For 1st complex number
Enter real and imaginary part respectively: 2.3
4.5

For 2nd complex number
Enter real and imaginary part respectively: 3.4
5
Sum = 5.7 + 9.5i
```

In this program, structures n1 and n2 are passed as an argument of function add().

This function computes the sum and returns the structure variable temp to the main() function.

Structures and pointers



Structures can be created and accessed using pointers. A pointer variable of a structure can be created as below:

```
struct name {
   member1;
   member2;
   .
   .
};


int main()
{
   struct name *ptr;
}
```

Here, the pointer variable of type **struct name** is created.

Accessing structure's member through pointer

A structure's member can be accesssed through pointer in two ways:

1. Referencing pointer to another address to access memory
2. Using dynamic memory allocation

1. Referencing pointer to another address to access the memory

Consider an example to access structure's member through pointer.

```
#include <stdio.h>

typedef struct person

{

  int age;

  float weight;

};



int main()

{

   struct person *personPtr, person1;

   personPtr = &person1;          // Referencing pointer to memory address of person1

   printf("Enter integer: ");

   scanf("%d",&(*personPtr).age);



   printf("Enter number: ");

   scanf("%f",&(*personPtr).weight);



   printf("Displaying: ");
```

```
    printf("%d%f",(*personPtr).age,(*personPtr).weight);



    return 0;

}
```

In this example, the pointer variable of type `struct person` is referenced to the address of `person1`. Then, only the structure member through pointer can can accessed.

Using -> operator to access structure pointer member

Structure pointer member can also be accessed using `->` operator.

(*personPtr).age is same as personPtr->age

(*personPtr).weight is same as personPtr->weight

2. Accessing structure member through pointer using dynamic memory allocation

To access structure member using pointers, memory can be allocated dynamically using malloc() function defined under `"stdlib.h"` header file.

Syntax to use malloc()

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

Example to use structure's member through pointer using malloc() function.

```
#include <stdio.h>

#include <stdlib.h>

struct person {

  int age;

  float weight;

  char name[30];

};



int main()
```

```
{
  struct person *ptr;
  int i, num;

  printf("Enter number of persons: ");
  scanf("%d", &num);

  ptr = (struct person*) malloc(num * sizeof(struct person));
  // Above statement allocates the memory for n structures with pointer personPtr pointing to base address */

  for(i = 0; i < num; ++i)
  {
     printf("Enter name, age and weight of the person respectively:\n");
     scanf("%s%d%f", &(ptr+i)->name, &(ptr+i)->age, &(ptr+i)->weight);
  }

  printf("Displaying Infromation:\n");
  for(i = 0; i < num; ++i)
     printf("%s\t%d\t%.2f\n", (ptr+i)->name, (ptr+i)->age, (ptr+i)->weight);

  return 0;
}
```

Output

Enter number of persons: 2

Enter name, age and weight of the person respectively:

Adam

2

3.2

Enter name, age and weight of the person respectively:

Eve

6

2.3

Displaying Information:

Adam 2 3.20

Eve 6 2.30

Union

Union, like structures, contain members whose individual data types may differ from one another. However, the members that compose a union all share the same storage area within the computer's memory, whereas each member within a structure is assigned its own unique storage area. Thus, unions are used to conserve memory. In general terms, the composition of a union may be defined as

```
union tag{
 member1;
member 2;
- - -
member m
};
```

Where union is a required keyword and the other terms have the same meaning as in a structure definition. Individual union variables can then be declared as storage-class union tag variable1, variable2, -----, variable n; where storage-class is an optional storage class specifier, union is a required keyword, tag is the name that appeared in the union definition and variable 1, variable 2, variable n are union variables of type tag.

The two declarations may be combined, just as we did in the case of structure. Thus, we can write

```
 Storage-class union tag{
          member1;
         member 2;
          - - -
         member m
         }variable 1, varibale2, . . . ., variable n;
```

The tag is optional in this type of declaration.
Let us take a 'C' program which contains the following union declaration:

```
union code{
char color [5];
int size ;
}purse, belt;
```

Here we have two union variables, purse and belt, of type code.

Each variable can represent either a 5–character string (color) or an integer quantity (size) of any one time.

A union may be a member of a structure, and a structure may be a member of a union.
An individual union member can be accessed in the same manner as an individual structure members, using the operators (→) and.
Thus if variable is a union variable, then varibale.member refers to a member of the union. Similarly, if ptr is a pointer variable that points to a union, then ptr→ member refers to a member of that union.
Let us consider the following C program:

```
 # include <stdio.h>
main()
union code{
char color;
 int size;
}; struct
 { char company [10];
float cost ;
union code detail;
}purse, belt;
printf("%d\n", sizeof (union code));
purse.detail.color='B';
printf("%c%d\n", purse.detail.color,purse.detail.size);
purse.detail.size=20;
 printf("%c%d\n", purse. detail.color,purse.detail.size);
 }
```

The output is as follows:

```
2
 B- 23190
@ 20
```

The first line indicates that the union is allocated 2 bytes of memory to accommodate an integer quantity. In line two, the first data item [B] is meaningful, but the second is not. In line three, the first data item is meaningless, but the second data item [20] is meaningful. A union variable can be initialized, provided its storage class is either external or static. Only one member of a union can be assigned a value at any one time. Unions are processed in the same manner, and with the same restrictions as structures. Thus, individual union members can be processed as though they were ordinary variables of the same data type and pointers to unions can be passed to or from functions.

Difference between structure and union

Differences between structure and union in C are presented in the following table. Structure and union are different in some ways yet they are conceptually same and have following similarities too:
Both are container data types and can contain objects of any type, including other structures and unions or arrays as their members.
Both structures and unions support only assignment = and sizeof operators.
In particular, structures and unions cannot appear as operands of the equality ==, inequality !=,
or *typecast* operators. The two structures or unions in the assignment must have the same members and member types.

Both structure and union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable; that is, the entire structure or union is copied into the corresponding parameter.

structure	**union**
Keyword struct defines a structure.	Keyword union defines a union.
Example structure declaration: struct s_tag { int ival; float fval; char *cptr; }s;	Example union declaration: union u_tag { int ival; float fval; char *cptr; }u;
Within a structure all members gets memory allocated and members have addresses that increase as the declarators are read left-to-right. That is, the members of a structure all begin at different offsets from the base of the structure. The offset of a particular member corresponds to the order of its declaration; the first member is at offset 0. The total size of a structure is the sum of the size of all the members or more because of appropriate alignment.	For a union compiler allocates the memory for the largest of all members and in a union all members have offset zero from the base, the container is big enough to hold the WIDEST member, and the alignment is appropriate for all of the types in the union. When the storage space allocated to the union contains a smaller member, the extra space between the end of the smaller member and the end of the allocated memory remains unaltered.
Within a structure all members gets memory allocated; therefore any member can be retrieved at any time.	While retrieving data from a union the type that is being retrieved must be the type most recently stored. It is the programmer's responsibility to keep track of which type is currently stored in a union; the results are implementation-dependent if something is stored as one type and extracted as another.
One or more members of a structure can be initialized at once.	A union may only be initialized with a value of the type of its first member; thus union u described above (during example declaration) can only be initialized with an integer value.

The typedef Statement

C provides a capability that enables you to assign an alternate name to a data type.This is done with a statement known as typedef.The statement

typedef int Counter;

defines the name Counter to be equivalent to the C data type int.Variables can subsequently be declared to be of type Counter, as in the following statement:

Counter j, n;

The C compiler actually treats the declaration of the variables j and n, shown in the preceding code, as normal integer variables.The main advantage of the use of the typedef in this case is in the added readability that it lends to the definition of the variables. It is clear from the definition of j and n what the intended purpose of these variables is in the program.
In many instances, a typedef statement can be equivalently substituted by the appropriate #define statement. For example, you could have instead used the statement

#define Counter int

to achieve the same results as the preceding statement. However, because the typedef is handled by the C compiler proper, and not by the preprocessor, the typedef statement provides more flexibility than does the #define when it comes to assigning names to derived data types.

For example, the following typedef statement:
typedef char Linebuf [81];
defines a type called Linebuf, which is an array of 81 characters. Subsequently declaring variables to be of type Linebuf, as in
Linebuf text, inputLine;
has the effect of defining the variables text and inputLine to be arrays containing 81 characters.This is equivalent to the following declaration:
char text[81], inputLine[81];

Note that, in this case, Linebuf could *not* have been equivalently defined with a #define preprocessor statement.

The following typedef defines a type name StringPtr to be a char pointer:
typedef char *StringPtr;
Variables subsequently declared to be of type StringPtr, as in
StringPtr buffer;
are treated as character pointers by the C compiler.

Enumerated Data Types

An enumerated data type definition is initiated by the keyword enum. Immediately following this keyword is the name of the enumerated data type, followed by a list of identifiers (enclosed in a set of curly braces) that define the permissible values that can be assigned to the type. For example, the statement
enum primaryColor { red, yellow, blue };
defines a data type primaryColor.Variables declared to be of this data type can be assigned the values red, yellow, and blue inside the program, *and no other values*. An attempt to assign another value to such a variable causes some compilers to issue an error message. Other compilers simply don't check.

To declare a variable to be of type enum primaryColor, you again use the keyword enum, followed by the enumerated type name, followed by the variable list. So the statement

enum primaryColor myColor, gregsColor;

defines the two variables myColor and gregsColor to be of type primaryColor.The only permissible values that can be assigned to these variables are the names red, yellow, and blue. So statements such as
myColor = red;
and

```
if ( gregsColor == yellow )
...
```
are valid.

As another example of an enumerated data type definition, the following defines the type enum month, with permissible values that can be assigned to a variable of this type being the months of the year:

```
enum month { january, february, march, april, may, june,
july, august, september, october, november, december };
```

The C compiler actually treats enumeration identifiers as integer constants. Beginning with the first name in the list, the compiler assigns sequential integer values to these names, starting with 0. If your program contains these two lines:

```
enum month thisMonth;
...
thisMonth = february;
```

the value 1 is assigned to thisMonth (and not the name february) because it is the second identifier listed inside the enumeration list.

If you want to have a specific integer value associated with an enumeration identifier, the integer can be assigned to the identifier when the data type is defined. Enumeration identifiers that subsequently appear in the list are assigned sequential integer values beginning with the specified integer value plus 1. For example, in the definition

```
enum direction { up, down, left = 10, right };
```

an enumerated data type direction is defined with the values up, down, left, and right.The compiler assigns the value 0 to up because it appears first in the list; 1 to down because it appears next; 10 to left because it is explicitly assigned this value; and 11 to right because it appears immediately after left in the list.

Example

Program 14.1 **Using Enumerated Data Types**

```
// Program to print the number of days in a month
#include <stdio.h>
int main (void)
{
enum month { january = 1, february, march, april, may, june,
july, august, september, october, november, december };
enum month aMonth;
int days;
printf (“Enter month number: “);
scanf (“%i”, &aMonth);
switch (aMonth ) {
case january:
case march:
case may:
case july:
case august:
case october:
case december:
days = 31;
break;
```

```
case april:
case june:
case september:
case november:
days = 30;
break;
case february:
days = 28;
break;
default:
printf (“bad month number\n”);
days = 0;
break;
}
if ( days != 0 )
printf (“Number of days is %i\n”, days);
if ( amonth == february )
printf (“...or 29 if it’s a leap year\n”);
return 0;
}
```

Output
Enter month number: **5**
Number of days is 31
Output (Rerun)
Enter month number: **2**
Number of days is 28
...or 29 if it’s a leap year

Enumeration identifiers can share the same value. For example, in
enum switch { no=0, off=0, yes=1, on=1 };
assigning either the value no or off to an enum switch variable assigns it the value 0;
assigning either yes or on assigns it the value 1.
Explicitly assigning an integer value to an enumerated data type variable can be done with the type cast operator. So if monthValue is an integer variable that has the value 6, for example, the expression
thisMonth = (enum month) (monthValue - 1);
is permissible and assigns the value 5 to thisMonth.

Enumerated type definitions behave like structure and variable definitions as far as their scope is concerned: Defining an enumerated data type within a block limits the scope of that definition to the block. On the other hand, defining an enumerated data type at the beginning of the program, outside of any function, makes the definition global to the file.

BIT FIELDS

The C language was basically developed with systems programming applications in mind. Pointers are the perfect case in point because they give the programmer an enormous amount of control over and access into the computer’s memory. Along these same lines, systems programmers frequently must get in and “twiddle with the bits” of particular computer words. C provides a host of operators specifically designed for performing operations on individual bits.
On most
computer systems, a byte consists of eight smaller units called *bits*.A bit can assume either

of two values: 1 or 0. So a byte stored at address 1000 in a computer's memory, for example, might be conceptualized as a string of eight binary digits as shown:

01100100

The rightmost bit of a byte is known as the *least significant* or *low-order* bit, whereas the leftmost bit is known as the *most significant* or *high-order* bit. If you treat the string of bits as an integer, the rightmost bit of the preceding byte represents 2^0 or 1, the bit immediately to its left represents 2^1 or 2, the next bit 2^2 or 4, and so on.Therefore, the preceding binary number represents the value $2^2 + 2^5 + 2^6 = 4 + 32 + 64 = 100$ decimal.

The representation of negative numbers is handled slightly differently. Most computers represent such numbers using a so-called "twos complement" notation. Using this notation, the leftmost bit represents the *sign* bit. If this bit is 1, the number is negative; otherwise, the bit is 0 and the number is positive.The remaining bits represent the value of the number. In twos complement notation, the value –1 is represented by all bits being equal to 1:

11111111

A convenient way to convert a negative number from decimal to binary is to first add 1 to the value, express the absolute value of the result in binary, and then "complement" all the bits; that is, change all 1s to 0s and 0s to 1s. So, for example, to convert -5 to binary, 1 is added, which gives -4; 4 expressed in binary is 00000100, and complementing the bits produces 11111011.

To convert a negative number from binary back to decimal, first complement all of the bits, convert the result to decimal, change the sign of the result, and then subtract 1. Given this discussion about twos complement representation, the largest positive number that can be stored into *n* bits is 2^{n-1}–1. So in eight bits, you can store a value up to 2^7 – 1, or 127. Similarly, the smallest negative number that can be stored into *n* bits is $–2^{n-1}$, which in an eight-bit byte comes to –128.

On most of today's processors, integers occupy four contiguous bytes, or 32 bits, in the computer's memory.The largest positive value that can, therefore, be stored into such an integer is 2^{31}–1 or 2,147,483,647, whereas the smallest negative number that can be stored is –2,147,483,648.

Bit Operators

Now that you have learned some preliminaries, it's time to discuss the various bit operators that are available.The operators that C provides for manipulating bits are presented in Table

Table **Bit Operators**

Symbol	Operation
&	Bitwise AND
|	Bitwise Inclusive-OR
^	Bitwise Exclusive-OR
~	Ones complement
<<	Left shift
>>	Right shift

All the operators listed in Table , with the exception of the ones complement operator ~, are binary operators and as such take two operands. Bit operations can be performed on any type of integer value in C—be it short, long, long long, and signed or unsigned—and on characters, but cannot be performed on floating-point values.

The Bitwise AND Operator

When two values are *ANDed* in C, the binary representations of the values are compared bit by bit. Each corresponding bit that is a 1 in the first value *and* a 1 in the second value produces a 1 in the corresponding bit position of the result; anything else produces a 0.

If *b1* and *b2* represent corresponding bits of the two operands, then the following table, called a *truth table*, shows the result of *b1* ANDed with *b2* for all possible values of *b1* and *b2*.

b1	b2	b1 & b2
0	0	0
0	1	0
1	0	0
1	1	1

So, for example, if w1 and w2 are defined as short ints, and w1 is set equal to 25 and w2 is set equal to 77, then the C statement
w3 = w1 & w2;
assigns the value 9 to w3.
This can be more easily seen by treating the values of w1, w2, and w3 as binary numbers. Assume that you are dealing with a short int size of 16 bits.

```
w1 0000000000011001        25
w2 0000000001001101     & 77
-------------------------------      --------
w3 0000000000001001         9
```

Bitwise ANDing is frequently used for masking operations.That is, this operator can be used to easily set specific bits of a data item to 0. For example, the statement
w3 = w1 & 3;
assigns to w3 the value of w1 bitwise ANDed with the constant 3.This has the effect of setting all of the bits in w3, other than the rightmost two bits, to 0, and of preserving the rightmost two bits from w1.
As with all binary arithmetic operators in C, the binary bit operators can also be used as assignment operators by tacking on an equal sign. So the statement
word &= 15;
performs the same function as
word = word & 15;
and has the effect of setting all but the rightmost four bits of word to 0.
When using constants in performing bitwise operations, it is usually more convenient to express the constants in either octal or hexadecimal notation.

Example Program 12.1: **The Bitwise AND Operator**

```
// Program to demonstrate the bitwise AND operator
#include <stdio.h>
int main (void)
{
unsigned int word1 = 077u, word2 = 0150u, word3 = 0210u;
printf ("%o ", word1 & word2);
printf ("%o ", word1 & word1);
printf ("%o ", word1 & word2 & word3);
printf ("%o\n", word1 & 1);
return 0;
}
```

Output

```
50 77 10 1
```

Program explanation : if an integer constant has a leading 0, it represents an octal (base 8) constant in C.Therefore, the three unsigned ints, word1, word2, and word3, are given initial *octal* values of 077, 0150, and 0210, respectively.
The first printf call displays octal 50 as the result of bitwise ANDing word1 with word2.The following depicts how this value was calculated:

```
word1 ... 000 111 111          077
word2 ... 001 101 000       & 0150
----------------------------     ---------
...          000 101 000          050
```

Only the rightmost nine bits of the previous values are shown because all bits to the left are 0.

The second printf call results in the display of octal 77, which is the result of ANDing word1 with itself. By definition, any quantity *x*, when ANDed with itself, produces *x*.
The third printf call displays the result of ANDing word1, word2, and word3 together. The operation of bitwise ANDing is such that it makes no difference whether an expression such as a & b & c is evaluated as (a & b) & c or as a & (b & c), but for the record, evaluation proceeds from left to right. It is left as an exercise to you to verify that the displayed result of octal 10 is the correct result of ANDing word1 with word2 with word3.
The final printf call has the effect of extracting the rightmost bit of word1.This is actually another way of testing if an integer is even or odd because that rightmost bit of any odd integer is 1 and of any even integer is 0.Therefore when the if statement

```
if ( word1 & 1 )
...
```

is executed, the expression is true if word1 is odd (because the result of the AND operation is 1) and false if it is even (because the result of the AND operation is 0). (*Note:* On machines that use a ones complement representation for numbers, this does not work for negative integers.)

The Bitwise Inclusive-OR Operator

When two values are bitwise Inclusive-ORed in C, the binary representation of the two values are once again compared bit by bit.This time, each bit that is a 1 in the first value *or* a 1 in the second value produces a 1 in the corresponding bit of the result. The truth table for the Inclusive-OR operator is shown next.

b1	b2	b1 | b2
0	0	0
0	1	1
1	0	1
1	1	1

So, if w1 is an unsigned int equal to octal 0431 and w2 is an unsigned int equal to octal 0152, then a bitwise Inclusive-OR of w1 and w2 produces a result of octal 0573 as shown:

```
w1 ... 100 011 001      0431
w2 ... 001 101 010    | 0152
-----------------------  ---------
...    101 111 011      0573
```

Bitwise Inclusive-ORing, frequently called just bitwise ORing, is used to set some specified bits of a word to 1. For example, the statement

```
w1 = w1 | 07;
```

sets the three rightmost bits of w1 to 1, regardless of the state of these bits before the operation was performed. Of course, you could have used a special assignment operator in the statement, as follows:

```
w1 |= 07;
```

The Bitwise Exclusive-OR Operator

The bitwise Exclusive-OR operator, which is often called the XOR operator, works as follows: For corresponding bits of the two operands, if either bit is a 1—but not both—the corresponding bit of the result is a 1; otherwise it is a 0.The truth table for this operator is as shown.

b1	b2	b1 ^ b2
0	0	0
0	1	1
1	0	1

1 1 0

If w1 and w2 were set equal to octal 0536 and octal 0266, respectively, then the result of w1 Exclusive-ORed with w2 would be octal 0750, as illustrated:

```
w1 ... 101 011 110      0536
w2 ... 010 110 110    ^ 0266
-------------------------  ---------
...    111 101 000      0750
```

One interesting property of the Exclusive-OR operator is that any value ExclusiveORed with itself produces 0.
Another interesting application of the Exclusive-OR operator is that it can be used to effectively exchange two values without the need for an extra memory location. You know that you would normally interchange two integers called i1 and i2 with a
sequence of statements such as

```
temp = i1;
i1 = i2;
i2 = temp;
```

Using the Exclusive-OR operator, you can exchange values without the need of the temporary storage location:

```
i1 ^= i2;
i2 ^= i1;
i1 ^= i2;
```

It is left as an exercise to you to verify that the previous statements do in fact succeed in interchanging the values of i1 and i2.

The Ones Complement Operator

The ones complement operator is a unary operator, and its effect is to simply "flip" the bits of its operand. Each bit of the operand that is a 1 is changed to a 0, and each bit that is a 0 is changed to a 1.The truth table is shown next simply for the sake of completeness.

```
b1      ~b1
---------------
0        1
1        0
```

If w1 is a short int that is 16 bits long, and is set equal to octal 0122457, then taking the ones complement of this value produces a result of octal 0055320:

```
w1  1 010 010 100 101 111          0122457
`w1 0 101 101 011 010 000      0055320
```

The ones complement operator (~) should not be confused with the arithmetic minus operator (–) or with the logical negation operator (!). So if w1 is defined as an int, and set equal to 0, then –w1 still results in 0. If you apply the ones complement operator to w1, you end up with w1 being set to all ones, which is –1 when treated as a signed value in twos complement notation. Finally, applying the logical negation operator to w1 produces the result true (1) because w1 is false (0).

The ones complement operator is useful when you don't know the precise bit size of the quantity that you are dealing with in an operation. Its use can help make a program more portable—in other words, less dependent on the particular computer on which the program is running and, therefore, easier to get running on a different machine. For example, to set the low-order bit of an int called w1 to 0, you can AND w1 with an int consisting of all 1s except for a single 0 in the rightmost bit. So a statement in C such as

```
w1 &= 0xFFFFFFFE;
```

works fine on machines in which an integer is represented by 32 bits.
If you replace the preceding statement with

```
w1 &= ~1;
```

w1 gets ANDed with the correct value on any machine because the ones complement of

1 is calculated and consists of as many leftmost one bits as are necessary to fill the size of an int (31 leftmost bits on a 32-bit integer system).

Program summarizes the various bitwise operators presented thus far. Before proceeding, however, it is important to mention the precedences of the various operators. The AND, OR, and Exclusive-OR operators each have lower precedence than any of the arithmetic or relational operators, but higher precedence than the logical AND and logical OR operators. The bitwise AND is higher in precedence than the bitwise Exclusive-OR, which in turn is higher in precedence than the bitwise OR. The unary ones complement operator has higher precedence than *any* binary operator.

```
/* Program to illustrate bitwise operators */
#include <stdio.h>
int main (void)
{
unsigned int w1 = 0525u, w2 = 0707u, w3 = 0122u;
printf ("%o %o %o\n", w1 & w2, w1 | w2, w1 ^ w2);
printf ("%o %o %o\n", ~w1, ~w2, ~w3);
printf ("%o %o %o\n", w1 ^ w1, w1 & ~w2, w1 | w2 | w3);
printf ("%o %o\n", w1 | w2 & w3, w1 | w2 & ~w3);
printf ("%o %o\n", ~(~w1 & ~w2), ~(~w1 | ~w2));
w1 ^= w2;
w2 ^= w1;
w1 ^= w2;
printf ("w1 = %o, w2 = %o\n", w1, w2);
return 0;
}
```

Program 12.2 **Output**

```
505 727 222
37777777252 37777777070 37777777655
0 20 727
527 725
727 505
w1 = 707, w2 = 525
```

You should work out each of the operations from Program 12.2 with a paper and pencil to verify that you understand how the results were obtained.The program was run on a computer that uses 32 bits to represent an int.

In the fourth printf call, it is important to remember that the bitwise AND operator has higher precedence than the bitwise OR, because this fact influences the resulting value of the expression.

The fifth printf call illustrates DeMorgan's rule, namely that ~(~a & ~b) is equal to a | b and that ~(~a | ~b) is equal to a & b.

The Left Shift Operator

When a left shift operation is performed on a value, the bits contained in the value are literally shifted to the left. Associated with this operation is the number of places (bits) that the value is to be shifted. Bits that are shifted out through the high-order bit of the data item are lost, and 0s are always shifted in through the low-order bit of the value. So if w1 is equal to 3, then the expression

```
w1 = w1 << 1;
```

which can also be expressed as

```
w1 <<= 1;
```

results in 3 being shifted one place to the left, which results in 6 being assigned to w1:

```
w1 ...          000 011                    03
w1 << 1 ... 000 110              06
```

The operand on the left of the << operator is the value to be shifted, whereas the operand on the right is the number of bit positions by which the value is to be shifted.
If you shift w1 one more place to the left, you end up with octal 014 as the value of w1:

w1 ...	000 110	06
w1 << 1 ..	001 100	014

Left shifting actually has the effect of multiplying the value that is shifted by two. In fact, some C compilers automatically perform multiplication by a power of two by left shifting the value the appropriate number of places because shifting is a much faster operation than multiplication on most computers.

The Right Shift Operator

As implied from its name, the right shift operator >> shifts the bits of a value to the right. Bits shifted out of the low-order bit of the value are lost. Right shifting an unsigned value always results in 0s being shifted in on the left; that is, through the high-order bits. What is shifted in on the left for signed values depends on the sign of the value that is being shifted and also on how this operation is implemented on your computer system. If the sign bit is 0 (meaning the value is positive), 0s are shifted in regardless of which machine you are running. However, if the sign bit is 1, on some machines 1s are shifted in, and on others 0s are shifted in.This former type of operation is known as an *arithmetic* right shift, whereas the latter is known as a *logical* right shift. Never make any assumptions about whether a system implements an arithmetic or a logical right shift.A program that shifts signed values right might work correctly on one system but fail on another due to this type of assumption.
If w1 is an unsigned int, which is represented in 32 bits, and w1 is set equal to hexadecimal F777EE22, then shifting w1 one place to the right with the statement
w1 >>= 1;
sets w1 equal to hexadecimal 7BBBF711.

w1	1111 0111 0111 0111 1110 1110 0010 0010	F777EE22
w1 >> 1	0111 1011 1011 1011 1111 0111 0001 0001	7BBBF711

If w1 were declared to be a (signed) int, the same result would be produced on some computers; on others, the result would be FBBBF711 if the operation were performed as an arithmetic right shift.

Summary

- Structure is composition of the different variables of different data types , grouped under same name.
- Name given to structure is called as **tag.**
- Structure **member** may be of **different data type** including **user defined data-type** also.
- When we declare a structure, memory is not allocated for un-initialized variable.
- One structure can be declared inside other structure as we declare structure members inside a structure. This is known as nested structure.
- Structure can be passed to **function as a Parameter**.
- Structures can be created and accessed using pointers.
- Union, like structures, contain members whose individual data types may differ from one another. However, the members that compose a union all share the same storage area within the computer's memory, whereas each member within a structure is assigned its own unique storage area.
- C provides a capability that enables you to assign an alternate name to a data type.This is done with a statement known as typedef..
- An enumerated data type definition is initiated by the keyword enum. Immediately following this keyword is the name of the enumerated data type, followed by a list of identifiers (enclosed in a set of curly braces) that define the permissible values that can be assigned to the type.
- Bit operations can be performed on any type of integer value in C—be it short, long, long long, and signed or unsigned—and on characters, but cannot be performed on floating-point values.

Questions for Exercises

1. What is Structure? Explain in detail.

2. Explain the advantages of structure type over the array the variable.

3. Explain about structure and arrays.

4. What is Union? Explain in detail.

5. What are differences between Structure and Unions?

6. What distinguishes an array from a structure?

7. What is a self referential structure and where can it be used?

8. Consider the following structure,

```
struct
{
int a;
int *p;
}*p1;
```

How can the content pointed by member pointer p be accessed via structure variable p1?

9. What will be the result when the following code is executed?

```
struct stud_type
{
int rollno;
char name[15];
int age;
};
union person
{
char surname[10];
struct stud_type s1;
}ex;
printf("Size = %d", sizeof (ex));
```

10. Write notes on bitwise operators.

11. Explain typedef and enumerated data type.

Books for reference:

- ***The C programming Language,*** By Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall , 2004
- ***Programming in C,*** By Stephen G. Kochan, Fourth Edition, 2001
- ***Let Us C,*** By Yashavant P. Kanetkar, Fifth Edition, 2012
- ***Programming with C***, Second Edition, Byron Gottfried, Tata McGraw Hill, 2003.
- ***C. The Complete Reference***, Fourth Edition, Herbert Schildt, Tata McGraw Hill, 2002.
- ***The C Primer***, Leslie Hancock, Morris Krieger, Mc Gravy Hill, 2013.