## Problem Solving Using C
## KCA 102: Session 2020-21

| Unit 4 | Lecture 1 | Introduction, Initializing, defining and declaring structure |
|--------|-----------|--------------------------------------------------------------|

**Introduction to Structure:**

Structure is a user-defined datatype in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful.

**Let's Understand it with an Example:**

If I have to write a program to store Student information, which will have Student's name, age, branch, permanent address, father's name etc., which included string values, integer values etc., how can I use arrays for this problem, I will require something which can hold data of different types together.

In structure, data is stored in form of **records**.

**Defining a Structure:**

In C, we use struct keyword to define a structure. struct defines a new data type which is a collection of primary and derived datatypes.

**Syntax:**

struct [structure_tag]

{

//member variable 1

//member variable 2

//member variable 3

...

}[structure_variables];

As you can see in the syntax above, we start with the struct keyword, then it's optional to provide your structure a name, we suggest you to give it a name, then inside the curly braces, we have to mention all the member variables, which are nothing but normal C language variables of different types like int, float, array etc.

After the closing curly brace, we can specify one or more structure variables, again this is optional.

**Note: <span style="color:red">The closing curly brace in the structure type declaration must be followed by a semicolon (;).</span>**

**Example of Structure:**

struct Student

{

char name[25];

```
        int age;

        char branch[10];

        // F for female and M for male

        char gender;

        };
```

Here, struct Student declares a structure to hold the details of a student which consists of 4 data fields, namely name, age, branch and gender. These fields are called **structure elements or members**.

Each member can have different datatype, like in this case, name is an array of char type and age is of int type etc. **Student** is the name of the structure and is called as the **structure tag**.

**Declaring Structure Variables:**

It is possible to declare variables of a **structure**, either along with structure definition or after the structure is defined. **Structure** variable declaration is similar to the declaration of any normal variable of any other datatype.

Structure variables can be declared in following three ways:

1. **Declaring Structure variables separately**

```
        struct Student

        {

          char name[25];

          int age;

          char branch[10];

          //F for female and M for male

          char gender;

        };

        struct Student S1, S2;    //declaring variables of struct Student
```

2. **Declaring Structure variables with structure definition**

```
        struct Student

        {

          char name [25];

          int age;

          char branch [10];

          //F for female and M for male

          char gender;
```

} S1, S2;

Here S1 and S2 are variables of structure Student. However, this approach is not much recommended.

### 3. Structure definition using typedef:

Let's take a example to understand, how we can define a structure in C using typedef keyword.

```c
#include<stdio.h>
#include<string.h>
typedef struct employee
{
    char name[50];
    int salary;
}emp;
void main( ){
    emp e1;
    printf("\nEnter Employee record:\n");
    printf("\nEmployee name:\t");
    scanf("%s", e1.name);
    printf("\nEnter Employee salary: \t");
    scanf("%d", &e1.salary);
    printf("\nstudent name is %s", e1.name);
    printf("\nroll is %d", e1.salary);
}
```

**Structure Initialization:**

Like a variable of any other datatype, structure variable can also be initialized at compile time.

```c
struct Patient
{
    float height;
    int weight;
    int age;
};
struct Patient p1 = { 180.75 , 73, 23 };   //initialization
```
                OR

```
struct Patient p1;

p1.height = 180.75;    //initialization of each member separately

p1.weight = 73;

p1.age = 23;
```

| Unit 4 | Lecture 2 | Accessing members, Operations on individual members, Operations on structures |
|--------|-----------|-------------------------------------------------------------------------------|

**Accessing Structure Members:**

Structure members can be accessed and assigned values in a number of ways. Structure members have no meaning individually without the structure. In order to assign a value to any structure member, the member name must be linked with the **structure** variable using a dot. operator also called **period** or **member access** operator.

**For example:**

```
#include<stdio.h>

#include<string.h>

struct Student

{

  char name[25];

  int age;

  char branch[10];

  //F for female and M for male

  char gender;

};

int main()

{

  struct Student s1;

    /*

    s1 is a variable of Student type and

    age is a member of Student

  */

  s1.age = 18;

  /*

    using string function to add name
```

```c
    */
    strcpy(s1.name, "Viraaj");
    /*
        displaying the stored values
    */
    printf("Name of Student 1: %s\n", s1.name);
    printf("Age of Student 1: %d\n", s1.age);
    return 0;
}
```

**Output:**

Name of Student 1: Viraaj

Age of Student 1: 18

We can also use scanf() to give values to structure members through terminal.

```c
scanf(" %s ", s1.name);
scanf(" %d ", &s1.age);
```

**Operations on Individual Members of Structure:**

Let's understand operations on individual members of structure, Here, we have C program to add, subtract, multiply and divide complex numbers. It is a menu driven program in which a user will have to enter his/her choice to perform an operation and can perform operations as many times as required. To easily handle a complex number a structure named complex has been used, which consists of two integers, first integer is for real part of a complex number and second is for imaginary part.

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

struct complex
{
 int real, img;
};

int main()
{
 int choice, x, y, z;
 struct complex a, b, c;

 while(1)
 {
  printf("Press 1 to add two complex numbers.\n");
  printf("Press 2 to subtract two complex numbers.\n");
  printf("Press 3 to multiply two complex numbers.\n");
  printf("Press 4 to divide two complex numbers.\n");
```

```c
    printf("Press 5 to exit.\n");
    printf("Enter your choice\n");
    scanf("%d", &choice);

    if (choice == 5)
     exit(0);

    if (choice >= 1 && choice <= 4)
    {
     printf("Enter a and b where a + ib is the first complex number.");
     printf("\na = ");
     scanf("%d", &a.real);
     printf("b = ");
     scanf("%d", &a.img);
     printf("Enter c and d where c + id is the second complex number.");
     printf("\nc = ");
     scanf("%d", &b.real);
     printf("d = ");
     scanf("%d", &b.img);
    }
    if (choice == 1)
    {
     c.real = a.real + b.real;
     c.img = a.img + b.img;

     if (c.img >= 0)
       printf("Sum of the complex numbers = %d + %di", c.real, c.img);
     else
       printf("Sum of the complex numbers = %d %di", c.real, c.img);
    }
    else if (choice == 2)
    {
     c.real = a.real - b.real;
     c.img = a.img - b.img;

     if (c.img >= 0)
       printf("Difference of the complex numbers = %d + %di", c.real, c.img);
     else
       printf("Difference of the complex numbers = %d %di", c.real, c.img);
    }
    else if (choice == 3)
    {
     c.real = a.real*b.real - a.img*b.img;
     c.img = a.img*b.real + a.real*b.img;

     if (c.img >= 0)
       printf("Multiplication of the complex numbers = %d + %di", c.real, c.img);
     else
       printf("Multiplication of the complex numbers = %d %di", c.real, c.img);
    }
    else if (choice == 4)
```

```c
 {
  if (b.real == 0 && b.img == 0)
   printf("Division by 0 + 0i isn't allowed.");
  else
  {
   x = a.real*b.real + a.img*b.img;
   y = a.img*b.real - a.real*b.img;
   z = b.real*b.real + b.img*b.img;

   if (x%z == 0 && y%z == 0)
   {
    if (y/z >= 0)
     printf("Division of the complex numbers = %d + %di", x/z, y/z);
    else
     printf("Division of the complex numbers = %d %di", x/z, y/z);
   }
   else if (x%z == 0 && y%z != 0)
   {
    if (y/z >= 0)
     printf("Division of two complex numbers = %d + %d/%di", x/z, y, z);
    else
     printf("Division of two complex numbers = %d %d/%di", x/z, y, z);
   }
   else if (x%z != 0 && y%z == 0)
   {
    if (y/z >= 0)
     printf("Division of two complex numbers = %d/%d + %di", x, z, y/z);
    else
     printf("Division of two complex numbers = %d %d/%di", x, z, y/z);
   }
   else
   {
    if (y/z >= 0)
     printf("Division of two complex numbers = %d/%d + %d/%di",x, z, y, z);
    else
     printf("Division of two complex numbers = %d/%d %d/%di", x, z, y, z);
   }
  }
 }
 else
  printf("Invalid choice.");

 printf("\nPress any key to enter choice again...\n");
 }
}
```

**Output:**

**Operations on Structure:**

1. Structures may be copied by assignment statement
2. & before a structure variable is the address of the structure
3. Can be passed to functions and can be returned by functions
   a. One can pass an entire structure
   b. One can pass components
   c. One can pass a pointer to a structure
4. Structures may not be compared

| Unit 4 | Lecture 3 | Structure within structure, Array of structure |
|--------|-----------|------------------------------------------------|

**Structure within Structure (Nested Structure):**

Nested structure in C is nothing but structure within structure. One structure can be declared inside other structure. We can take any data type for declaring structure members like int, float, char etc. In the same way we can also take object of one structure as member in another structure.

**For Example:**

```
struct student

{

char[30] name;

int age;

struct address  {

        char[50] locality;

        char[50] city;

        int pincode;
```

```
}; };
```

**Let's take a program to understand it:**

```c
#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, \"Ravi", 90.5, 029,"KIET"};
    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;
}
```

Output:

**Id is: 1**
**Name is: Ravi**
**Percentage is: 90.50000**

**College id is: 029**

**College name is: KIET**

**Array of Structure:**

A structure is simple to define if there are only one or two element, but in case there are too many objects needed for a structure, for ex. A structure designed to show the data of each student of the class, and then in that case, the way will be introduced. A structure declaration using array to define object is given below.

**For Example:**

```
struct student {

        char name[15];

        int roll_no;

        char gender;

};

student data[50];
```

**Initializing Array of Structures:**

It is to be noted that only static or external variable can be initialized.

**For Example:**

```
struct employee {

        int enum;

        char gender;

        int sal;

};

Employee data[3]={{146,'m',100},{200,'f',5000},{250,'m',10000}};
```

If in case, the structure of object or any element of struct are not initialized, the compiler will automatically assigned zero to the fields of that particular record.

**For Example:**

**employee data [3] = { { 146,'m'} ,{ 200, 'f' },{250 ,'m'}};**

**Compiler will assign:**

**data [0].sal=0; data [1].sal=0; data [2].sal=0;**

**Let's Understand with a Program:**

```
#include<stdio.h>

void main ()

{
```

```
int i;
  struct student
{
  long int rollno;
  char gender;
  float height;
  float weight;
};
  struct student data [3] = { {121,'m',5.7,59.8},{122,'f',6.0,65.2},{123,'m',
6.0, 7.5} };
  clrscr ();
  printf ("the initialized contents are:\n");
for ( i=0; i< =2; i++)
  {
        printf ("%d/n   ** Record is  \n ", data [i].rollno);
        printf ("%c\n",  data [i] .gender);
        printf ("%f\n",  data [i].height);
        printf ("%f\n",  data [i]. weight);
  }
}
```

**Output:**

```
The initialized contents are:
121
** Record is
m
5.700000
59.799999
122
** Record is
f
6.00000
65.19997
123
** Record is
m
6.000000
7.500000
```

| Unit 4 | Lecture 4 | Pointers to structure |
|--------|-----------|-----------------------|

**Pointers to Structure:**

It is also possible to create structure pointers. We can also create a pointer, pointing to structure elements.
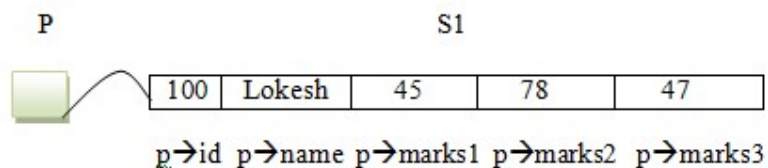
For this we require "**->**" operator.

The arrow operator (**->**) introduced in second version of C language, it is a shortcut operator to access members of a structure through pointer.

If user-defined type is "struct student", then its pointer type is "struct student*"

**For example:**

> struct student *p;      // p is a pointer variable to structure
>
> p=&s1;      //making pointer 'p' to s1



**Point to be Remember:**

1. Now the expression "*p" accesses the total memory of "s1"
2. The expression "(*p).id" accesses the "id" in s1.
3. The expression "(*p).id" is equal to "p -> id"
4. The expression "(*p).name" accesses the "name" in s1,  "(*p).name" is equal to "p -> name"

**Let's take an example to understand it:**

```
#include <stdio.h>

#include <string.h>

 struct Books

{

  char  title[50];

  char  author[50];

  char  subject[100];

  int   book_id;

};

void printBook( struct Books *book );
```

```c
int main( )
{
   struct Books Book1;      //Declare Book1 of type Book
   struct Books Book2;      // Declare Book2 of type Book
 //Initializing Book1
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "vinay");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;
 //Initializing Book2
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "naveen");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;
 //passing address of structure to function as pointer
   printBook( &Book1 );
   printBook( &Book2 );
 return 0;
}
void printBook( struct Books *book )
{
   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
   printf( "Book book_id : %d\n", book->book_id);
}
```

**Output:**

**Book title: C Programming**
**Book author: vinay**
**Book subject: C Programming Tutorial**
**Book book_id: 6495407**
**Book title: Telecom Billing**
**Book author: naveen**

**Book subject: Telecom Billing Tutorial**

**Book book_id: 6495700**

| Unit 4 | Lecture 5 | Introduction, Declaring union, Usage of unions, Operations on union |
|--------|-----------|---------------------------------------------------------------------|

**Introduction to Union:**

Unions are conceptually similar to structures. The syntax of union is also similar to that of structure. The only difference is in terms of storage. In structure each member has its own storage location, whereas all members of union use a single shared memory location which is equal to the size of its largest data member.
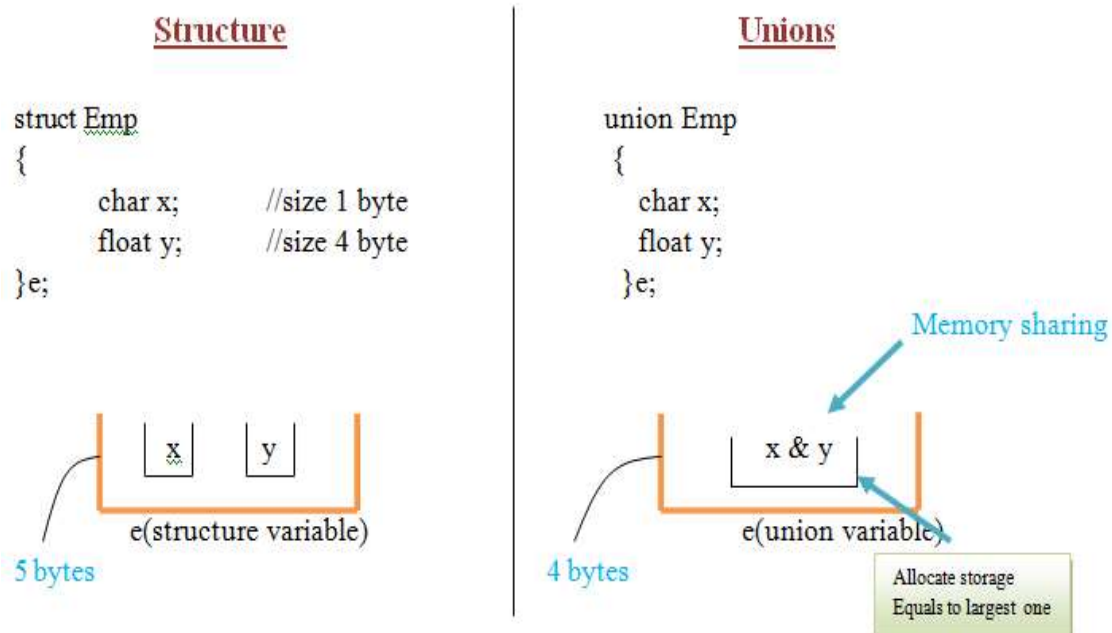


This implies that although a union may contain many members of different types, it cannot handle all the members at same time.

**Declaration of Union:**

A **union** is declared using the union keyword.

       **For Example:**

```
union item
{
    int m;
    float x;
    char c;
} It1;
```

This declares a variable It1 of type union item. This union contains three members each with a different data type. However only one of them can be used at a time. This is due to the fact that only

one location is allocated for all the union variables, irrespective of their size. The compiler allocates the storage that is large enough to hold the largest variable type in the union.

In the union declared above the member x requires **4 bytes** which is largest amongst the members for a 16-bit machine. Other members of union will share the same memory address.

**Accessing a Union Member:**

Syntax for accessing any union member is similar to accessing structure members,

```
union test
{
    int a;
    float b;
    char c;
}t;
```

**To access members of union t:**

```
t.a;
t.b;
t.c;
```

**Usage of Union:**

**Major advantages of using unions are**:

1. Efficient use of memory as it does not demand memory space for its all members rather it require memory space for its largest member only.
2. Same memory space can be interpreted differently for different members of the union.
3. Mostly used in system programing

**Disadvantages:**

if one of the member variable is updated the it will be reflected in the remaining variables.

| Unit 4 | Lecture 6 | Enumerated data types |
|--------|-----------|------------------------|

**Enumerated Data Types:**

Enumeration data type used to symbolize a list of constants with valid meaningful names, so that, it makes the program easy to read and modify. Enumeration has advantage of generating the values automatically for given list of names. Keyword enum is used to define enumerated data type.

**Syntax:**

```
enum type_name
{
value1, value2... valueN
```

```
};
```

Here, type_name is the name of enumerated data type or tag. And value1, value2 …valueN are values of type type_name.

By default, value1 will be equal to 0, value2 will be 1 and so on but, the programmer can change the default value.

**Let's take an example to understand it:**

```
#include <stdio.h>

enum week{ sunday, monday, tuesday, wednesday, thursday, friday, saturday};

int main(){

        enum week today;

         today=wednesday;

        printf("%d th day",today+1);

        return 0;

   }
```

**Output:**

**4th day**

**Explicit initialization of enumerators:**

At the declaration of enumeration, we can give explicit values to the names and these values can be any integers. Even for two names, we can give the same value.

**Example:**

```
enum month

{

jan=1, feb=2, mar=3, apr, may, june, july, aug, sep, oct, nov, dec;

};
```

Here names "jan", "feb", "mar", are assigned with values 1,2,3 respectively and remaining names are assigned with next succeeding number from the last value, that is 4, 5, 6… for the apr, may, june, july,….

**Enumeration variables:**

Using enumerator tag name, we can specify variables of its type. This makes the convenient and increases the readability of the program, the enumerator variables are nothing but unsigned int types.

**For Example:**

```
enum  Color
```

```
{

Red=15, Green=26, Blue=13, White=5, Yellow=4, Grey=17;

};

void main()

{

enum Color wallcolor, floorcolor;

wallcolor=White;

floorcolor=Grey;

……………

……………

}
```

| Unit 4 | Lecture 7 | Storage classes: Introduction, Types- automatic, register, static and external. |
|--------|-----------|--------------------------------------------------------------------------------|

**Introduction to Storage Classes:**

A storage class represents the visibility and a location of a variable. It tells from what part of code we can access a variable. A storage class is used to describe the following things:

- The variable scope.

- The location where the variable will be stored.

- The initialized value of a variable.

- A lifetime of a variable.

- Who can access a variable?

Thus a storage class is used to represent the information about a variable.

**NOTE:** A variable is not only associated with a data type, its value but also a storage class.

**Types of Storage Classes:**

**There are total four types of standard storage classes. The table below represents the storage classes in 'C'.**

| Storage class | Purpose |
|---------------|---------|
| **auto** | **It is a default storage class.** |
| **extern** | **It is a global variable.** |

| static | It is a local variable which is capable of returning a value even when control is transferred to the function call. |
|---|---|
| register | It is a variable which is stored inside a Register. |

**Automatic Variables: auto**

**Scope:** Variable defined with auto storage class are local to the function block inside which they are defined.

**Default Initial Value:** Any random value i.e garbage value.

**Lifetime:** Till the end of the function/method block where the variable is defined.

A variable declared inside a function without any storage class specification, is by default an **automatic variable**. They are created when a function is called and are destroyed **automatically** when the function's execution is completed. Automatic variables can also be called **local variables** because they are local to a function. By default, they are assigned **garbage value** by the compiler.

**For Example:**

```
#include<stdio.h>

  void main()
{
  int detail;
  // or
  auto int details;   //Both are same
}
```

**External or Global variable:**

**Scope:** Global i.e everywhere in the program. These variables are not bound by any function; they are available everywhere.

**Default initial value:** 0(zero).

**Lifetime:** Till the program doesn't finish its execution, you can access global variables.

A variable that is declared outside any function is a **Global Variable**. Global variables remain available throughout the program execution. By default, initial value of the Global variable is 0(zero). One important thing to remember about global variable is that their values can be changed by any function in the program.

**For Example:**

```c
#include<stdio.h>

int number;    // global variable

void main()
{
   number = 10;

   printf("I am in main function. My value is %d\n", number);

   fun1();    //function calling, discussed in next topic

   fun2();    //function calling, discussed in next topic
}


/* This is function 1 */

fun1()
{
   number = 20;

   printf("I am in function fun1. My value is %d", number);
}
/* This is function 1 */

fun2()
{
   printf("\nI am in function fun2. My value is %d", number);
}
```

I am in function main. My value is 10

I am in function fun1. My value is 20

I am in function fun2. My value is 20

Here the global variable number is available to all three functions and thus, if one function changes the value of the variable, it gets changed in every function.

**Note:** Declaring the storage class as global or external for all the variables in a program can waste a lot of memory space because these variables have a lifetime till the end of the program. Thus, variables, which are not needed till the end of the program, will still occupy the memory and thus, memory will be wasted.
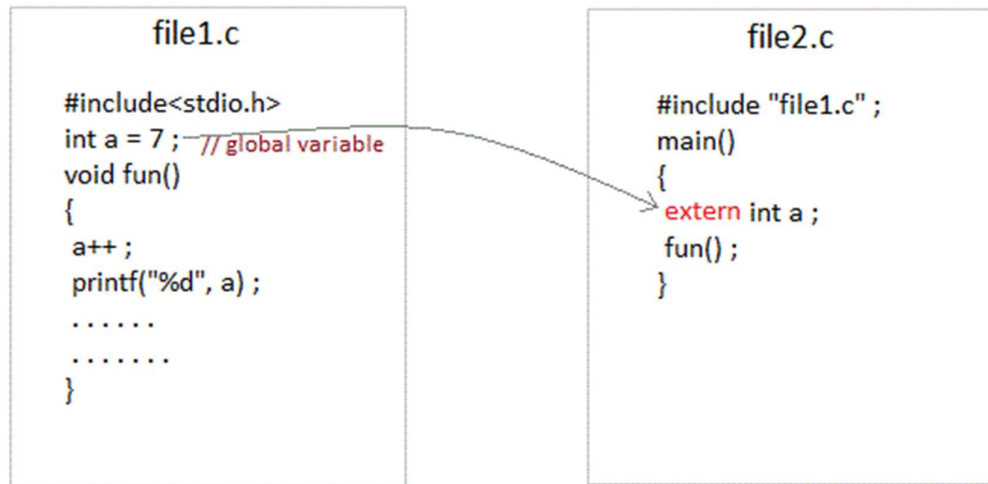
**extern keyword:**

The extern keyword is used with a variable to inform the compiler that this variable is declared somewhere else. The extern declaration does not allocate storage for variables.



global variable from one file can be used in other using **extern** keyword.

**Static Variables:**

**Scope:** Local to the block in which the variable is defined

**Default initial value:** 0(Zero).

**Lifetime:** Till the whole program doesn't finish its execution.

A static variable tells the compiler to persist/save the variable until the end of program. Instead of creating and destroying a variable every time when it comes into and goes out of scope, static variable is initialized only once and remains into existence till the end of the program. A static variable can either be internal or external depending upon the place of declaration. Scope of **internal static** variable remains inside the function in which it is defined. **External static** variables remain restricted to scope of file in which they are declared.

They are assigned **0 (zero)** as default value by the compiler.

**For Example:**

```
#include<stdio.h>

void test();   //Function declaration (discussed in next topic)

 int main()

{

   test();

   test();

   test();

}
```

```
void test()

{

   static int a = 0;      //a static variable

   a = a + 1;

   printf("%d\t",a);

}



1 2 3
```

**Register Variable:**

**Scope:** Local to the function in which it is declared.

**Default initial value:** Any random value i.e garbage value

**Lifetime:** Till the end of function/method block, in which the variable is defined.

Register variables inform the compiler to store the variable in CPU register instead of memory. Register variables have faster accessibility than a normal variable. Generally, the frequently used variables are kept in registers. But only a few variables can be placed inside registers. One application of register storage class can be in using loops, where the variable gets used a number of times in the program, in a very short span of time.

**NOTE:** We can never get the address of such variables.

**Syntax:**

register int number;

**Note:** Even though we have declared the storage class of our variable number as register, we cannot surely say that the value of the variable would be stored in a register. This is because the number of registers in a CPU are limited. Also, CPU registers are meant to do a lot of important work. Thus, sometimes they may not be free. In such scenario, the variable works as if its storage class is auto.

**Point to Remember for Storage Class:**

To improve the speed of execution of the program and to carefully use the memory space occupied by the variables, following points should be kept in mind while using storage classes:

- We should use static storage class only when we want the value of the variable to remain same every time we call it using different function calls.

- We should use register storage class only for those variables that are used in our program very often. CPU registers are limited and thus should be used carefully.

- We should use external or global storage class only for those variables that are being used by almost all the functions in the program.

- If we do not have the purpose of any of the above mentioned storage classes, then we should use the automatic storage class.