# Storage Classes in C :

→ To fully define a variable one need to mention not only its data type but also its 'storage class'. In other words, not only do all variables have a data type, they also have a storage class.

→ From C compiler's point of view, a variable name identifies some physical location within the computer where the string of bits representing the variable's value is stored.

→ There are basically two kinds of locations in a computer where such a value may be kept; Memory and CPU registers. If the variable's storage class is defined then that determines in which of these two locations the value is stored.

→ Moreover, a variable's storage class tells us;

(a) where the variable would be stored.

(b) what will be the initial value of a variable, if initial value is not specifically assigned (i.e. default initial value)

(c) what is the scope of the variable i.e. in which functions the value of the variable would be available

(d) what is the life of the variable, i.e. how long would the variable exist.

## Types of Storage Classes :

There are four storage classes in C

1) Automatic storage class
2) Register storage class
3) static storage class
4) External storage class

1) Automatic Storage class :

→ The features of a variable defined to have an automatic storage class are as under :

| | | |
|---|---|---|
| Storage | — | Memory |
| Default initial value | — | An unpredictable value, which is often called a garbage value. |
| Scope | — | Local to the block in which the variable is defined. |
| Life | — | Till the control remains within the block in which the variable is defined. |

Example :

```
/* The following program shows how an automatic storage class
   variable is declared and the fact that if the variable is not
   initialized it contains a garbage value */

void main( )
{
   auto int i, j;
   printf ("%d %d", i, j);
}
```

output : 1211 221 (Garbage value).

Example : /* scope and life of an automatic variable */

```
#include <stdio.h>
#include <conio.h>
void main( )
{
   auto int i = 1;
      {
         {
            {
```

```
        printf("%d\n", i);
      }
       printf("%d", i);
      }
    printf("%d", i);
  }
  getch();
}
```

output: 1 1 1

Example:

```
#include <stdio.h>
#include <conio.h>
void main()
{
   auto int i=1;
     {
       auto int i=2;
         {
           auto int i=3;
           printf("\n%d", i);
         }
       printf("%d", i);
     }
  printf("%d", i);
}
```

output: 3 2 1

2. Register storage class:

→ The features of a variable defined to be of register storage class are as under:

Storage — CPU register

Default initial value — Garbage value

Scope — Local to the block in which the variable is defined.

Life — Till the control remains within the blocks in which the variable is defined.

→ A value stored in a CPU register can always be accessed faster than the one that is stored in memory. Therefore, if a variable is used at many places in a program it is better to declare its storage class as register.

Example: A good example of frequently used variable is loop counter

```
# include <stdio.h>
# include <conio.h>
void main()
{
   register int i;
    clrscr();
   for (i=1; i<=10; i++)
       printf ("\n%d", i);
   getch();
}
```

output 1 2 3 4 5 6 7 8 9 10

→ Here, even though we have declared the storage class of i as register, we cannot say for sure that the value of i would be stored in a CPU register. Because the number of CPU registers are limited (14 in case of a microcomputer), and they may be busy doing some other task. In this case, the variable works as if its storage class is auto.

→ We cannot use register storage class for all types of variables. For example the following declarations are wrong:

```
register float a;
register double a;
```

register long c ;

This is because the CPU register in a microcomputer are usually 16 bit registers and therefore cannot hold a float value or a double value. which require 4 and 8 bytes respectively for storing a value However, if we use the above declarations we won't get any error messages All that would happen is the compiler would treat the variables to be of auto storage class

3. Static storage class :

The features of a variable defined to have a static storage class are as under

| | | |
|---|---|---|
| Storage | — | Memory |
| Default initial value | — | zero |
| Scope | — | Local to the block in which the variable is defined. |
| Life | — | value of the variable persists between different function calls |

```
void main()
{
   increment();
   increment();
   increment();
}
 increment()
 {
   auto int i = 1 ;
   printf ("%d\n", i) ;
   i = i+1 ;
 }
```

Output : 1
1
1

```
Void main()
{
   increment();
   increment();
   increment();
}
 increment()
 {
   static int i = 1 ;
   printf ("%d\n", i) ;
   i = i+1 ;
 }
```

Output : 1
2
3

## 4) External storage class :

The features of a variable whose storage class has been defined as external are as follows.

| | | |
|---|---|---|
| Storage | — | Memory |
| Default initial value | — | Zero |
| Scope | — | Global |
| Life | — | As long as the program's execution doesn't come to an end. |

→ External variables are declared outside all functions. yet are available to all functions that care to use them.

Example :

```
#include <stdio.h>
extern int i;
void main()
{
  printf("\n i = %d", i);
  increment();
  increment();
  decrement();
  decrement();
}
increment()
{
  i = i+1;
  printf("\non incrementing i = %d", i);
}
decrement()
{
  i = i-1;
  printf("\n on decrementing i = %d", i);
}
```

output :

```
i = 0
on incrementing i = 1
on incrementing i = 2
on decrementing i = 1
on decrementing i = 0
```