Storage Classes in C

C storage classes define the scope (visibility) and lifetime of variables and/or functions within a C Program. They precede the type that they modify.

We have four different storage classes in a C program −

- auto
- register
- static
- extern

The auto Storage Class

The **auto** is a default storage class for all variables that are declared inside a function or a block. The keyword "**auto**", which is optional, can be used to define local variables.

The scope and lifetime of **auto** variables are within the same block in which they are declared.

Example of auto Storage Class

The following code statements demonstrate the declaration of an automatic (auto) variable −

```
{
   int mount;
   auto int month;
}
```

The example above defines two variables with in the same storage class. 'auto' can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

Example of register Storage Class

The following code statement demonstrates the declaration of a register variable –

```
{
   register int  miles;
}
```


The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when **static** is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

Example of static Storage Class

The following example demonstrates the use of a static storage class in a C program –

</> Open Compiler

```
#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main(){
```

```
   while(count--) {
      func();
   }

   return 0;
}

/* function definition */
void func(void) {

   static int i = 5; /* local static variable */
   i++;

   printf("i is %d and count is %d\n", i, count);
}
```

Output

When the above code is compiled and executed, it produces the following result –

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then extern will be used in another file to provide the reference of defined variable or function. Just for understanding, extern is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

Example of extern Storage Class

The example of an extern storage class may contain two or more files. Here is an example demonstrating the use of an extern storage class in C language −

First File: main.c

```
#include <stdio.h>

int count;
extern void write_extern();

main(){
   count = 5;
   write_extern();
}
```

Second File: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void) {
   printf("Count is %d\n", count);
}
```

Here, **extern** is being used to declare **count** in the second file, whereas it has its definition in the first file (main.c). Now, compile these two files as follows −

```
$gcc main.c support.c
```

It will produce the executable program **a.out**. When this program is executed, it will produce the following output −

```
Count is 5
```

Use of storage classes

Storage classes are used to define the scope, visibility, lifetime, and initial (default) value of a variable.

Summary of Storage Classes

The following table provides a summary of the scope, default value, and lifetime of variables having different storage classes –

Storage Class	Name	Memory	Scope, Default Value	Lifetime
auto	Automatic	Internal Memory	Local Scope, Garbage Value	Within the same function or block in which they are declared.
register	Register	Register	Local Scope, 0	Within the same function or block in which they are declared.
static	Static	Internal Memory	Local Scope, 0	Within the program i.e., as long as program is running.
extern	External	Internal Memory	Global Scope, 0	Within the program i.e., as long as program is running.