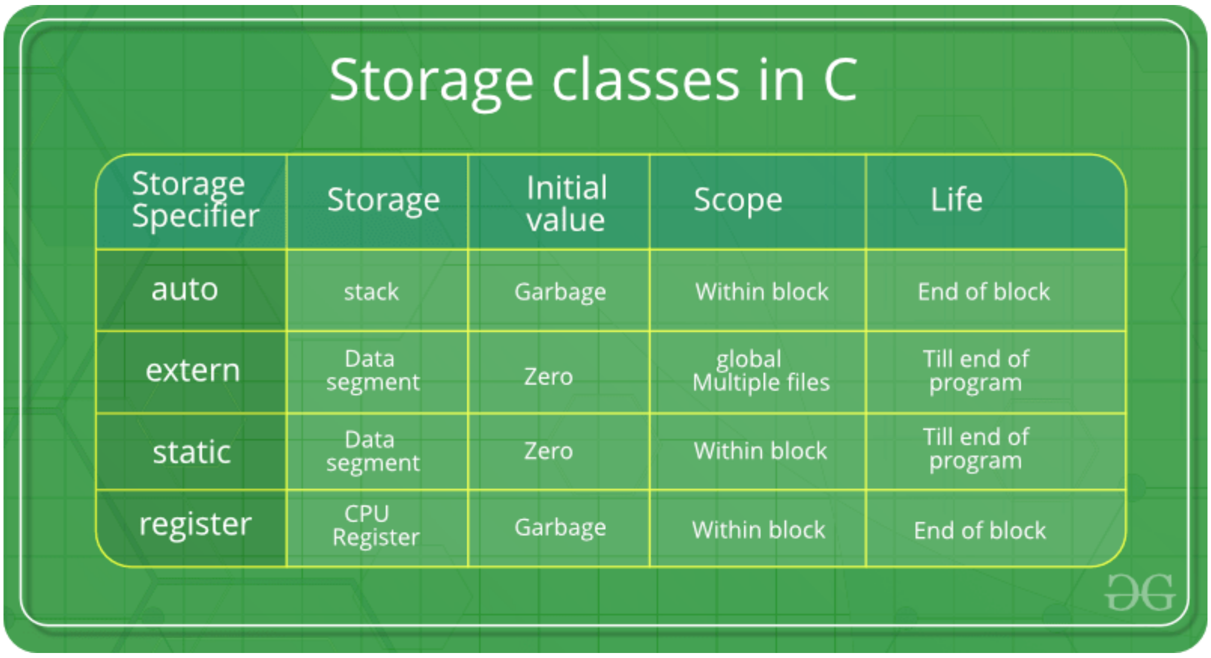
Storage Classes in C

Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.

C language uses 4 storage classes, namely:

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.



Storage classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

1. **auto**: This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. **Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope).** Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared. However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables resides. They are assigned a garbage value by default whenever they are declared.

2. extern: **Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well.** So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. Also, a normal global variable can be made extern as well by placing the ‘extern’ keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

3. static: This storage class is used to declare static variables which are popularly used while writing programs in C language. **Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope**. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

4. register: **This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program.** If a free register is not available, these are then stored in the memory only. Usually few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

To specify the storage class for a variable, the following syntax is to be followed:

Syntax:

```
storage_class data_type var_name;
```

Functions follow the same syntax as given above for variables. Have a look at the following C example for further clarification:

```
// A C program to demonstrate different storage
// classes
#include <stdio.h>

// declaring the variable which is to be made extern
// an intial value can also be initialized to x
int x;

void autoStorageClass()
{

    printf("\nDemonstrating auto class\n\n");

    // declaring an auto variable (simply
    // writing "int a=32;" works as well)
    auto int a = 32;

    // printing the auto variable 'a'
    printf("Value of the variable 'a'"
           " declared as auto: %d\n",
           a);

    printf("--------------------------------");
}
```

```c
void registerStorageClass()
{

    printf("\nDemonstrating register class\n\n");

    // declaring a register variable
    register char b = 'G';

    // printing the register variable 'b'
    printf("Value of the variable 'b'"
           " declared as register: %d\n",
           b);

    printf("--------------------------------");
}

void externStorageClass()
{

    printf("\nDemonstrating extern class\n\n");

    // telling the compiler that the variable
    // z is an extern variable and has been
    // defined elsewhere (above the main
    // function)
    extern int x;

    // printing the extern variables 'x'
    printf("Value of the variable 'x'"
           " declared as extern: %d\n",
           x);

    // value of extern variable x modified
    x = 2;

    // printing the modified values of
    // extern variables 'x'
    printf("Modified value of the variable 'x'"
           " declared as extern: %d\n",
           x);

    printf("--------------------------------");
}

void staticStorageClass()
{
    int i = 0;

    printf("\nDemonstrating static class\n\n");

    // using a static variable 'y'
    printf("Declaring 'y' as static inside the loop.\n"
           "But this declaration will occur only"
           " once as 'y' is static.\n"
           "If not, then every time the value of 'y' "
           "will be the declared value 5"
           " as in the case of variable 'p'\n");

    printf("\nLoop started:\n");

    for (i = 1; i < 5; i++) {

        // Declaring the static variable 'y'
        static int y = 5;
        // Declare a non-static variable 'p'
        int p = 10;

        // Incrementing the value of y and p by 1
        y++;
        p++;

        // printing value of y at each iteration
        printf("\nThe value of 'y', "
               "declared as static, in %d "
               "iteration is %d\n",
               i, y);

        // printing value of p at each iteration
        printf("The value of non-static variable 'p', "
               "in %d iteration is %d\n",
               i, p);
    }

    printf("\nLoop ended:\n");

    printf("--------------------------------");
}

int main()
{

    printf("A program to demonstrate"
           " Storage Classes in C\n\n");

    // To demonstrate auto Storage Class
    autoStorageClass();

    // To demonstrate register Storage Class
    registerStorageClass();

    // To demonstrate extern Storage Class
    externStorageClass();

    // To demonstrate static Storage Class
    staticStorageClass();

    // exiting
    printf("\n\nStorage Classes demonstrated");

    return 0;
}
```

Output:

```
A program to demonstrate Storage Classes in C

Demonstrating auto class

Value of the variable ‘a’ declared as auto: 32
________________________
Demonstrating register class

Value of the variable ‘b’ declared as register: 71
________________________
Demonstrating extern class

Value of the variable ‘x’ declared as extern: 0
Modified value of the variable ‘x’ declared as extern: 2
________________________
Demonstrating static class

Declaring ‘y’ as static inside the loop.
But this declaration will occur only once as ‘y’ is static.
If not, then every time the value of ‘y’ will be the declared value 5 as in the case of variable
‘p’

Loop started:

The value of ‘y’, declared as static, in 1 iteration is 6
The value of non-static variable ‘p’, in 1 iteration is 11

The value of ‘y’, declared as static, in 2 iteration is 7
The value of non-static variable ‘p’, in 2 iteration is 11

The value of ‘y’, declared as static, in 3 iteration is 8
The value of non-static variable ‘p’, in 3 iteration is 11

The value of ‘y’, declared as static, in 4 iteration is 9
The value of non-static variable ‘p’, in 4 iteration is 11

Loop ended:
________________________

Storage Classes demonstrated
```

Auto storage class

The variables defined using auto storage class are called as local variables. Auto stands for automatic storage class. A variable is in auto storage class by default if it is not explicitly specified.

The scope of an auto variable is limited with the particular block only. Once the control goes out of the block, the access is destroyed. This means only the block in which the auto variable is declared can access it.

A keyword auto is used to define an auto storage class. By default, an auto variable contains a garbage value.

```
Example, auto int age;
```

The program below defines a function with has two local variables

```
int add(void) {
   int a=13;
   auto int b=48;
return a+b;}
```

We take another program which shows the scope level "visibility level" for auto variables in each block code which are independently to each other:

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

OUTPUT:

```
 3 2 1
```

Extern storage class

Extern stands for external storage class. Extern storage class is used when we have global functions or variables which are shared between two or more files.

Keyword **extern** is used to declaring a global variable or function in another file to provide the reference of variable or function which have been already defined in the original file.

The variables defined using an extern keyword are called as global variables. These variables are accessible throughout the program. Notice that the extern variable cannot be initialized it has already been defined in the original file

```
Example, extern void display();
```

First File: main.c

```
#include <stdio.h>
extern i;
main() {
```

```
printf("value of the external integer is = %d\n", i);
return 0;}
```

Second File: original.c

```
#include <stdio.h>
i=48;
```

Result:

```
value of the external integer is = 48
```

Static storage class

The static variables are used within function/ file as local static variables. They can also be used as a global variable

- Static local variable is a local variable that retains and stores its value between function calls or block and remains visible only to the function or block in which it is defined.
- Static global variables are global variables visible **only to the file in which it is declared.**

```
Example: static int count = 10;
```

Keep in mind that static variable has a default initial value zero and is initialized only once in its lifetime.

```
#include <stdio.h> /* function declaration */
void next(void);
static int counter = 7; /* global variable */
main() {
 while(counter<10) {
      next();
      counter++;   }
return 0;}
void next( void ) {    /* function definition */
   static int iteration = 13; /* local static variable */
   iteration ++;
   printf("iteration=%d and counter= %d\n", iteration, counter);}
```

Result:

```
iteration=14 and counter= 7
iteration=15 and counter= 8
iteration=16 and counter= 9
```

Global variables are accessible throughout the file whereas static variables are accessible only to the particular part of a code.

The lifespan of a static variable is in the entire program code. A variable which is declared or initialized using static keyword always contains zero as a default value.

Register storage class

You can use the register storage class when you want to store local variables within functions or blocks in CPU registers instead of RAM to have quick access to these variables. For example, "counters" are a good candidate to be stored in the register.

```
Example: register int age;
```

The keyword **register** is used to declare a register storage class. The variables declared using register storage class has lifespan throughout the program.

It is similar to the auto storage class. The variable is limited to the particular block. The only difference is that the variables declared using register storage class are stored inside CPU registers instead of a memory. Register has faster access than that of the main memory.

The variables declared using register storage class has no default value. These variables are often declared at the beginning of a program.

```
#include <stdio.h> /* function declaration */
main() {
```

```
{register int  weight;
int *ptr=&weight ;/*it produces an error when the compilation occurs ,we
cannot get a memory location when dealing with CPU register*/}
}
```

OUTPUT:

```
error: address of register variable 'weight' requested
```

The next table summarizes the principal features of each storage class which are commonly used in C programming

Storage Class	Declaration	Storage	Default Initial Value	Scope	Lifetime
auto	Inside a function/block	Memory	Unpredictable	Within the function/block	Within the function/block
register	Inside a function/block	CPU Registers	Garbage	Within the function/block	Within the function/block
extern	Outside all functions	Memory	Zero	Entire the file and other files where the variable is declared as extern	program runtime
Static (local)	Inside a function/block	Memory	Zero	Within the function/block	program runtime
Static (global)	Outside all functions	Memory	Zero	Global	program runtime

----

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
   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).

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

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.

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.

```
#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);
}
```

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.

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, where as 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 produces the following result −

```
count is 5
```