Storage Classes are used to describe about 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:

**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.

---

**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. For more information on how extern variables work, have a look at this link.

(Opening that extern link)

# Understanding "extern" keyword in C

I'm sure that this post will be as interesting and informative to C virgins (i.e. beginners) as it will be to those who are well versed in C. So let me start by saying that extern keyword applies to C variables (data objects) and C functions. Basically extern keyword extends the visibility of the C variables and C functions. Probably that's is the reason why it was named as extern.

Though (almost) everyone knows the meaning of declaration and definition of a variable/function yet for the sake of completeness of this post, I would like to clarify them. Declaration of a variable/function simply declares that the variable/function exists somewhere in the program but the memory is not allocated for them. But the declaration of a variable/function serves an important role. And that is the type of the variable/function. Therefore, when a variable is declared, the program knows the data type of that variable. In case of function declaration, the program knows what are the arguments to that functions, their data types, the order of arguments and the return type of the function. So that's all about the declaration. Coming to the definition, when we define a variable/function, apart from the role of the declaration, it also allocates memory for that variable/function. Therefore, we can think of definition as a superset of the declaration. (or declaration as a subset of definition). From this explanation, it should be obvious that a variable/function can be declared any number of times but it can be defined only once. (Remember the basic principle that you can't have two locations of the same variable/function). So that's all about declaration and definition.

Same is the case with the definition of a C function (Definition of a C function means writing the body of the function). Therefore whenever we define a C function, an extern is present there in the beginning of the function definition. Since the declaration can be done any number of times and definition can be done only once, we can notice that declaration of a function can be added in several C/H files or in a single C/H file several times. But we notice the actual definition of the function only once (i.e. in one file only). And as the extern extends the visibility to the whole program, the functions can be used (called) anywhere in any of the files of the whole program provided the declaration of the function is known. (By knowing the declaration of the function, C compiler knows that the definition of the function exists and it goes ahead to compile the program). So that's all about extern with C functions.

Now let us then take the second and final case i.e. use of extern with C variables. I feel that it more interesting and informative than the previous case where extern is present by default with C functions. So let me ask the question, how would you declare a C variable without defining it? Many of you would see it trivial but it's an important question to understand extern with C variables. The answer goes as follows.

```
    extern int var;
```

Here, an integer type variable called var has been declared (remember no definition i.e. no memory allocation for var so far). And we can do this declaration as many times as needed. (remember that declaration can be done any number of times) So far so good. 🙂

Now how would you define a variable? Now I agree that it is the most trivial question in programming and the answer is as follows.

```
int var;
```

Here, an integer type variable called var has been declared as well as defined. (remember that definition is the superset of declaration). Here the memory for var is also allocated. Now here comes the surprise, when we declared/defined a C function, we saw that an extern was present by default. While defining a function, we can prepend it with extern without any issues. But it is not the case with C variables. If we put the presence of extern in a variable as default then the memory for them will not be allocated ever, they will be declared only. Therefore, we put extern explicitly for C variables when we want to declare them without defining them. Also, as the extern extends the visibility to the whole program, by using the extern keyword with a variable we can use the variables anywhere in the program provided we know the declaration of them and the variable is defined somewhere.

Now let us try to understand extern with examples.

Example 1:

```
int var;
int main(void)
{
   var = 10;
   return 0;
}
```

Analysis: This program is compiled successfully. Here var is defined (and declared implicitly) globally.

Example 2:

```
extern int var;
int main(void)
{
  return 0;
}
```

Analysis: This program is compiled successfully. Here var is declared only. Notice var is never used so no problems arise.

Example 3:

```c
extern int var;
int main(void)
{
  var = 10;
  return 0;
}
```

Analysis: Supposing that somefile.h has the definition of var. This program will be compiled successfully.

Example 5:

```c
extern int var = 0;
int main(void)
{
 var = 10;
 return 0;
}
```

Analysis: Guess this program will work? Well, here comes another surprise from C standards. They say that..if a variable is only declared and an initializer is also provided with that declaration, then the memory for that variable will be allocated i.e. that variable will be considered as defined. Therefore, as per the C standard, this program will compile successfully and work.

So that was a preliminary look at “extern” keyword in C.

I’m sure that you want to have some take away from the reading of this post. And I would not disappoint you. 🙂
In short, we can say

1. A declaration can be done any number of times but definition only once.
2. “extern” keyword is used to extend the visibility of variables/functions().
3. Since functions are visible throughout the program by default. The use of extern is not needed in function declaration/definition. Its use is redundant.
4. When extern is used with a variable, it’s only declared not defined.
5. As an exception, when an extern variable is declared with initialization, it is taken as the definition of the variable as well.

**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.

# Static Variables in C

Static variables have a property of preserving their value even after they are out of their scope!Hence, static variables preserve their previous value in their previous scope and are not initialized again in the new scope.
Syntax:

```
static data_type var_name = var_value;
```

Following are some interesting facts about static variables in C.

**1)** A static int variable remains in memory while the program is running. A normal or auto variable is destroyed when a function call where the variable was declared is over.

For example, we can use static int to count number of times a function is called, but an auto variable can't be sued for this purpose.

For example below program prints "1 2"

```c
#include<stdio.h>
int fun()
{
  static int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

But below program prints 1 1

```c
#include<stdio.h>
int fun()
{
  int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

**2)** Static variables are allocated memory in data segment, not stack segment (see memory layout for this)

**3)** Static variables (like global variables) are initialized as 0 if not initialized explicitly. For example in the below program, value of x is printed as 0, while value of y is something garbage. See this for more details.

```c
#include <stdio.h>
int main()
{
    static int x;
    int y;
    printf("%d \n %d", x, y);
}
```

Output:

```
0
[some_garbage_value]
```

In C, static variables can only be initialized using constant literals. For example, following program fails in compilation.

```
#include<stdio.h>
int initializer(void)
{
    return 50;
}

int main()
{
    static int i = initializer();
    printf(" value of i = %d", i);
    getchar();
    return 0;
}
```

If we change the program to following, then it works without any error.

```
#include<stdio.h>
int main()
{
    static int i = 50;
    printf(" value of i = %d", i);
    getchar();
    return 0;
}
```

The reason for this is simple: All objects with static storage duration must be initialized (set to their initial values) before execution of main() starts. So a value which is not known at translation time cannot be used for initialization of static variables.

In C, functions are global by default. The "*static*" keyword before a function name makes it static. For example, below function *fun()* is static.

```
static int fun(void)
{
  printf("I am a static function ");
}
```

Unlike global functions in C, access to static functions is restricted to the file where they are declared. Therefore, when we want to restrict access to functions, we make them static. Another reason for making functions static can be reuse of the same function name in other files.

For example, if we store following program in one file *file1.c*

```c
/* Inside file1.c */
static void fun1(void)
{
  puts("fun1 called");
}
```

And store following program in another file *file2.c*

```c
/* Iinside file2.c  */
int main(void)
{
  fun1();
  getchar();
  return 0;
}
```

Now, if we compile the above code with command "*gcc file2.c file1.c*", we get the error *"undefined reference to `fun1'"*. This is because *fun1()* is declared *static* in *file1.c* and cannot be used in *file2.c*.

---

**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 var_data_type var_name;
```

# Understanding “register” keyword in C

Registers are faster than memory to access, so the variables which are most frequently used in a C program can be put in registers using *register* keyword. The keyword *register* hints to compiler that a given variable can be put in a register. It’s compiler’s choice to put it in a register or not. Generally, compilers themselves do optimizations and put the variables in register.

1) If you use & operator with a register variable then compiler may give an error or warning (depending upon the compiler you are using), because when we say a variable is a register, it may be stored in a register instead of memory and accessing address of a register is invalid. Try below program.

```
int main()
{
  register int i = 10;
  int *a = &i;
  printf("%d", *a);
  getchar();
  return 0;
}
```

2) *register* keyword can be used with pointer variables. Obviously, a register can have address of a memory location. There would not be any problem with the below program.

```
int main()
{
  int i = 10;
  register int *a = &i;
  printf("%d", *a);
  getchar();
  return 0;
}
```

3) Register is a storage class, and C doesn’t allow multiple storage class specifiers for a variable. So, *register* can not be used with *static* . Try below program.

```
int main()
{
  int i = 10;
  register static int *a = &i;
  printf("%d", *a);
  getchar();
  return 0;
}
```