In C language, the life time and scope of a variable is defined by its storage class.

The following are four types of storage class available in C language.

1. autos
2. register
3. extern
4. static

In this article, we will discuss the ‘static’ storage class and explain how to use static variables and static functions in C with some sample code snippets.

Before moving ahead, lets quickly understand the difference between life time and scope of a variable. A region in code where a variable can be accessed is known as its scope and the duration during which a variable remains active is known as its life time.

## I. Static Variables

### 1. Impact on Life Time

static variables are those variables whose life time remains equal to the life time of the program. Any local or global variable can be made static depending upon what the logic expects out of that variable. Lets consider the following example :

#include<stdio.h>

char\*\* func\_Str();

int main(void)

{

char \*\*ptr = NULL;

ptr = func\_Str();

printf("\n [%s] \n",\*ptr);

return 0;

}

char\*\* func\_Str()

{

char \*p = "Linux";

return &p;

}

In the code above, the function ‘func\_str()’ returns the address of the pointer ‘p’ to the calling function which uses it further to print the string ‘Linux’ to the user through ‘printf()’. Lets look at the output :

$ ./static

[Linux]

The output above is as expected. So, is everything fine here? Well, there is a hidden problem in the code. More specifically, its the return value of the function ‘func\_Str()’. The value being returned is the address of the local pointer variable ‘p’. Since ‘p’ is local to the function so as soon as the function returns, the lifetime of this variable is over and hence its memory location becomes free for any further modifications.

Lets prove this observation. Look at the code below :

#include<stdio.h>

char\*\* func1\_Str();

char\*\* func2\_Str();

int main(void)

{

char \*\*ptr1 = NULL;

char \*\*ptr2 = NULL;

ptr1 = func1\_Str();

printf("\n [%s] \n",\*ptr1);

ptr2 = func2\_Str();

printf("\n [%s] \n",\*ptr2);

printf("\n [%s] \n",\*ptr1);

return 0;

}

char\*\* func1\_Str()

{

char \*p = "Linux";

return &p;

}

char\*\* func2\_Str()

{

char \*p = "Windows";

return &p;

}

In the code above, now there are two functions ‘func1\_Str()’ and ‘func2\_Str()’. The logical problem remains the same here too. Each of these function returns the address of its local variable. In the main() function, the address returned by the func1\_Str() is used to print the string ‘Linux’ (as pointed by its local pointer variable) and the address returned by the function func2\_Str() is used to print the string ‘Windows’ (as pointed by its local pointer variable). An extra step towards the end of the main() function is done by again using the address returned by func1\_Str() to print the string ‘Linux’.

Now, lets see the output :

$ ./static

[Linux]

[Windows]

[Windows]

The output above is not as per expectations. The third print should have been ‘Linux’ instead of ‘Windows’. Well, I’d rather say that the above output was expected. Its just the correct scenario that exposed the loophole in the code.

Lets go a bit more deep to see what happened after address of local variable was returned. See the code below :

#include<stdio.h>

char\*\* func1\_Str();

char\*\* func2\_Str();

int main(void)

{

char \*\*ptr1 = NULL;

char \*\*ptr2 = NULL;

ptr1 = func1\_Str();

printf("\n [%s] :: func1\_Str() address = [%p], its returned address is [%p]\n",\*ptr1,(void\*)func1\_Str,(void\*)ptr1);

ptr2 = func2\_Str();

printf("\n [%s] :: func2\_Str()address = [%p], its returned address is [%p]\n",\*ptr2,(void\*)func2\_Str,(void\*)ptr2);

printf("\n [%s] [%p]\n",\*ptr1,(void\*)ptr1);

return 0;

}

char\*\* func1\_Str()

{

char \*p = "Linux";

return &p;

}

char\*\* func2\_Str()

{

char \*p = "Windows";

return &p;

}

The code is above is modified to print the address of the functions and the address of their respective local pointer variables. Here is the output :

$ ./static

[Linux] :: func1\_Str() address = [0x4005d5], its returned address is [0x7fff705e9378]

[Windows] :: func2\_Str()address = [0x4005e7], its returned address is [0x7fff705e9378]

[Windows] [0x7fff705e9378]

The above output makes it clear that once the lifetime of the local variable of the function ‘func1\_Str()’ gets over then same memory address is being used for the local pointer variable of the function ‘func2\_Str()’ and hence the third print is ‘Windows’ and not ‘Linux’.

So, now we see what that the root of the problem is the life time of the pointer variables. This is where the ‘static’ storage class comes to rescue. As already discussed the static storage class makes the lifetime of a variable equal to that of the program. So, lets make the local pointer variables as static and then see the output :

#include<stdio.h>

char\*\* func1\_Str();

char\*\* func2\_Str();

int main(void)

{

char \*\*ptr1 = NULL;

char \*\*ptr2 = NULL;

ptr1 = func1\_Str();

printf("\n [%s] :: func1\_Str() address = [%p], its returned address is [%p]\n",\*ptr1,(void\*)func1\_Str,(void\*)ptr1);

ptr2 = func2\_Str();

printf("\n [%s] :: func2\_Str()address = [%p], its returned address is [%p]\n",\*ptr2,(void\*)func2\_Str,(void\*)ptr2);

printf("\n [%s] [%p]\n",\*ptr1,(void\*)ptr1);

return 0;

}

char\*\* func1\_Str()

{

static char \*p = "Linux";

return &p;

}

char\*\* func2\_Str()

{

static char \*p = "Windows";

return &p;

}

Note that in code above, the pointers were made static. Here is the output :

$ ./static

[Linux] :: func1\_Str() address = [0x4005d5], its returned address is [0x601028]

[Windows] :: func2\_Str()address = [0x4005e0], its returned address is [0x601020]

[Linux] [0x601028]

So we see that after making the variables as static, the lifetime of the variables becomes equal to that of the program.

On a related note, if you are not familiar with C pointers, this [C pointer series](http://www.thegeekstuff.com/2012/06/c-constant-pointers/) of articles should give you a jump start.

### 2. Impact on Scope

In case where code is spread over multiple files, the static storage type can be used to limit the scope of a variable to a particular file. For example, if we have a variable ‘count’ in one file and we want to have another variable with same name in some other file, then in that case one of the variable has to be made static. The following example illustrates it :

Here we use two files (static.c and static\_1.c)

//static.c

#include<stdio.h>

int count = 1;

int main(void)

{

printf("\n count = [%d]\n",count);

return 0;

}

// static\_1.c

#include<stdio.h>

int count = 4;

int func(void)

{

printf("\n count = [%d]\n",count);

return 0;

}

Now, when both the files are compiled and linked together to form a single executable, here is the error that is thrown by gcc :

$ gcc -Wall static.c static\_1.c -o static

/tmp/ccwO66em.o:(.data+0x0): multiple definition of `count'

/tmp/ccGwx5t4.o:(.data+0x0): first defined here

collect2: ld returned 1 exit status

$

So we see that gcc complains of multiple declarations of the variable ‘count’.

As a corrective measure, this time one of the ‘count’ variable is made static :

//static.c

#include<stdio.h>

static int count = 1;

int main(void)

{

printf("\n count = [%d]\n",count);

return 0;

}

// static\_1.c

#include<stdio.h>

int count = 4;

int func(void)

{

printf("\n count = [%d]\n",count);

return 0;

}

Now, if both the files are compiled and linked together :

$ gcc -Wall static.c static\_1.c -o static

So we see that no error is thrown this time because static limited the scope of the variable ‘count’ in file static.c to the file itself.

## II. Static Functions

By default any function that is defined in a C file is extern. This means that the function can be used in any other source file of the same code/project (which gets compiled as separate translational unit). Now, if there is a situation where the access to a function is to be limited to the file in which it is defined or if a function with same name is desired in some other file of the same code/project then the functions in C can be made static.

Extending the same example that was used in previous section, suppose we have two files :

//static.c

#include<stdio.h>

void func();

int main(void)

{

func();

return 0;

}

void funcNew()

{

printf("\n Hi, I am a normal function\n");

}

// static\_1.c

#include<stdio.h>

void funcNew();

int func(void)

{

funcNew();

return 0;

}

If we compile, link and run the code above :

$ gcc -Wall static.c static\_1.c -o static

$ ./static

Hi, I am a normal function

So we see that the function funcNew() was defined in one file and successfully got called from the other. Now, if the file static\_1.c wants to have its own funcNew(), ie :

// static\_1.c

#include<stdio.h>

void funcNew();

int func(void)

{

funcNew();

return 0;

}

void funcNew()

{

printf("\n Hi, I am a normal function\n");

}

Now, if both the files are compiled and linked together :

$gcc -Wall static.c static\_1.c -o static

/tmp/ccqI0jsP.o: In function `funcNew':

static\_1.c:(.text+0x15): multiple definition of `funcNew'

/tmp/ccUO2XFS.o:static.c:(.text+0x15): first defined here

collect2: ld returned 1 exit status

$

So we see that the compiler complains of multiple definitions of the function funcNew(). So, we make the funcNew() in static\_1.c as static :

// static\_1.c

#include<stdio.h>

static void funcNew();

int func(void)

{

funcNew();

return 0;

}

static void funcNew()

{

printf("\n Hi, I am also a normal function\n");

}

Now, if we compile, then we see that the compiler never complains :

$ gcc -Wall static.c static\_1.c -o static

$ ./static

Hi, I am also a normal function

$

Similarly, if static.c wants that its funcNew() should be accessible from within static.c only then in that case funcNew() in static.c can be made static.

# **Static functions in C**

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

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

And store following program in another file file2.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.