

Storage Classes in C

Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage classes in C

- Automatic
- External
- Static
- Register

| Storage Classes | Storage Place | Default Value | Scope | Lifetime |
|-----------------|------------------------------|---------------|-------------------------|--|
| auto | RAM (Stack) | Garbage Value | Local (within block) | Within function (end of block) |
| extern | RAM (Data segment) | Zero | Global (multiple files) | Till the end of the main program Maybe declared anywhere in the program |
| static | RAM (Data segment) | Zero | Local (within block) | Till the end of the main program, Retains value between multiple functions call |
| register | CPU general purpose Register | Garbage Value | Local (within block) | Within the function (within block) |

➤ Automatic

- Automatic variables are allocated memory automatically at runtime
- This is the default storage class for all the variables declared inside a function or a block.
- The visibility of the automatic variables is limited to the block in which they are defined
- The scope of the automatic variables is limited to the block in which they are defined
- Every local variable is automatic in C by default.
- Also it 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.
- The automatic variables are initialized to garbage by default.

Above program have three functions each with different scopes; each scope holds different local variables.

The variable a, b and c of main function are inaccessible through myAdd or myMull functions.

The variables x, y and z of myMull function are inaccessible through myAdd or main functions, even if myAdd function has the same variables names

```
#include "stdio.h"

int myMull(int x, int y)
{
    int z;
    z = x * y;
    return z;
}

int myAdd(int x, int y)
{
    int z;
    z = x + y;
    return z;
}

void main()
{
    int a = 5, b = 6;
    printf("a + b = %d\r\n", myAdd(a,b));
    printf("a * b = %d\r\n", myMull(a,b));
}
```

myMull Scope
Local variables:
x,y,z

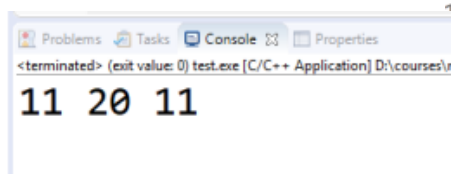
myAdd Scope
Local variables:
x,y,z

main Scope
Local variables:
a,b,c

EX

The visibility of the automatic variables is limited to the block in which they are defined

The scope of the automatic variables is limited to the block in which they are defined



```
1 /*
2  * main.c
3  *
4  * Created on: Sep 5, 2020
5  * Author: kkkhalil
6  */
7
8 #include <stdio.h>
9 int main()
10 {
11     int a = 10;
12     printf("%d ", ++a);
13     {
14         int a = 20;
15         printf("%d ", a);
16     }
17     printf("%d ", a);
18     return 0 ;
19 }
20
```

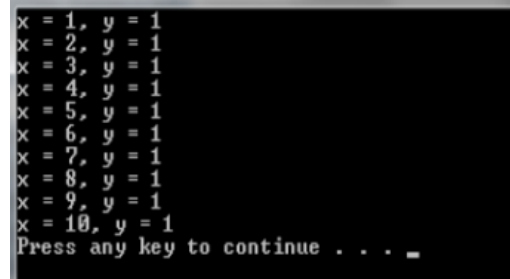
➤ Static

- The variables defined as static specifier can hold their value between the multiple function calls as it is stored on Data Memory.
- Static local variables are visible only to the function or the block in which they are defined
- A same static variable can be declared many times but can be assigned at only one time.
- Default initial value of the static integral variable is 0 otherwise null.
- The visibility of the static global variable is limited to the file in which it has declared.
- The keyword used to define static variable is static

```
#include "stdio.h"

void myprint()
{
    static int x = 0;
    int y = 0;
    x++;
    y++;
    printf("x = %d, y = %d\r\n", x, y);
}

void main()
{
    int i;
    for(i=0;i<10;i++)
        myprint();
}
```

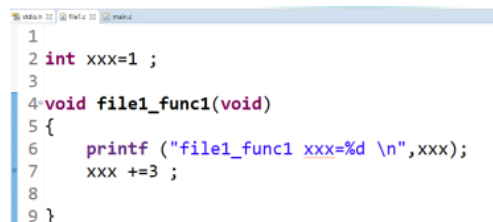


```
x = 1, y = 1
x = 2, y = 1
x = 3, y = 1
x = 4, y = 1
x = 5, y = 1
x = 6, y = 1
x = 7, y = 1
x = 8, y = 1
x = 9, y = 1
x = 10, y = 1
Press any key to continue . . . _
```

Static variables are defined by the modifier static. Static variables are initialized once in the program life. For example if the variable (x) is defined inside a function, the variable is initialized only at first function call, further function calls do not perform the initialization step, this means that if the variable is modified by any call the modification result remains for the next call. Following example illustrate this idea

EX

The visibility of the static global variable is limited to the file in which it has declared



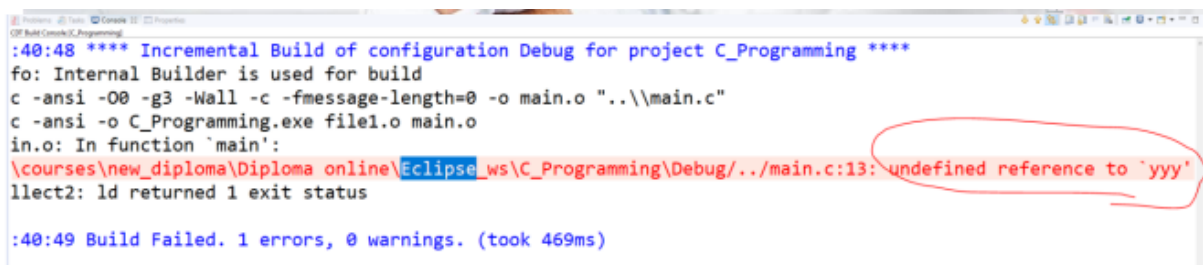
```
1
2 int xxx=1 ;
3
4 void file1_func1(void)
5 {
6     printf ("file1_func1 xxx=%d \n",xxx);
7     xxx +=3 ;
8
9 }
```

```
1 /*
2 // * main.c
3 // *
4 // * Created on: Aug 20, 2020
5 // * Author: Keroles Shenouda
6 */
7
8 #include "stdio.h"
9 static int xxx ;
10
11 void file1_func1(void);
12
13 int main ()
14 {
15     printf ("main.c xxx=%d \n",xxx);
16     file1_func1 () ;
17     printf ("main.c xxx=%d \n",xxx);
18     xxx = 8 ;
19     file1_func1 () ;
20     printf ("main.c xxx=%d \n",xxx);
21
22     return 0 ;
23 }
```

➤ Extern

- The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
- The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
- The default initial value of external integral type is 0 otherwise null.
- We can only initialize the extern variable globally
- An external variable can be declared many times but can be initialized at only once.
- If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program, If it is not, then the compiler will show an error

```
1 /*
2 // * main.c
3 // *
4 // * Created on: Aug 20, 2020
5 // * Author: Keroles Shenouda
6 */
7
8 #include "stdio.h"
9 extern int yyy ;
10
11 int main ()
12 {
13     yyy = 1 ;
14
15     return 0 ;
16 }
```



```
:40:48 **** Incremental Build of configuration Debug for project C_Programming ****
fo: Internal Builder is used for build
c -ansi -O0 -g3 -Wall -c -fmessage-length=0 -o main.o "../main.c"
c -ansi -o C_Programming.exe file1.o main.o
in.o: In function `main':
\courses\new_diploma\Diploma online\Eclipse_ws\C_Programming\Debug/../main.c:13: undefined reference to `yyy'
llect2: ld returned 1 exit status

:40:49 Build Failed. 1 errors, 0 warnings. (took 469ms)
```

➤ Register

- The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.
- We can not dereference the register variables, i.e., we can not use &operator for the register variable.
- The access time of the register variables is faster than the automatic variables.
- The initial default value of the register local variables is 0
- The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler's choice whether or not; the variables can be stored in the register.
- We can store pointers into the register, i.e., a register can store the address of a variable.
- Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

Example 1

```
#include <stdio.h>
int main()
{
    register int a; // variable a is allocated memory in the CPU register. The initial default value of a is 0.
    printf("%d",a);
}
```

Output:

0

```
#include <stdio.h>
int main()
{
    register int a = 1; // vari
    printf("%d", &a);
}
```

Output

Show output from: Build

```
1>----- Build started: Project: Diploma_online, Configuration: Debug Win32 -----
1>Compiling...
1>main.c
1>c:\users\public\documents\vector canape 15\extras\diploma_online\main.c(7) : error C2103: '&' on register variable
1>Build log was saved at "file:///c:/Users/Public/Documents/Vector CANape 15/Extras/Diploma_online/Debug/BuildLog.htm"
1>Diploma_online - 1 error(s), 0 warning(s)
===== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped =====
```

| Class | Scoop | Lifetime | Storage | Note |
|----------|-----------------|----------|--------------|--|
| auto | Block | Block | Stack | <ul style="list-style-type: none"> ✓variables are always local (automatic) and are stored on the stack. ✓we don't need to write it, because all the variables default is auto. |
| register | Block | Block | CPU register | <ul style="list-style-type: none"> ✓keep the variable in a register if at all possible. Otherwise it is stored on the stack. ✓We cannot get the address of such variable. |
| static | Local Global | Program | Data Segment | <ul style="list-style-type: none"> it Visible only within the scope of the variable: ✓using static variable to reduce the coupling between modules ✓ the value of the static variable will not be destroyed even if the function run ended, and still use this value in memory. |
| extern | All files | Program | Data Segment | <ul style="list-style-type: none"> ✓Meaning "foreign" ... ✓tell the compiler: there is this variable, it may not exist in the current file, but it will certainly be present in one source file in the project. ✓most commonly used state when the project is more than one file, ✓call the total of variables and functions of another file. ✓don't create another instance of it or there will be a name collision at link time. |

INLINE ASSEMBLY

The inline assembler lets us embed assembly-language instructions in our C and C++ source programs without extra assembly and link steps.

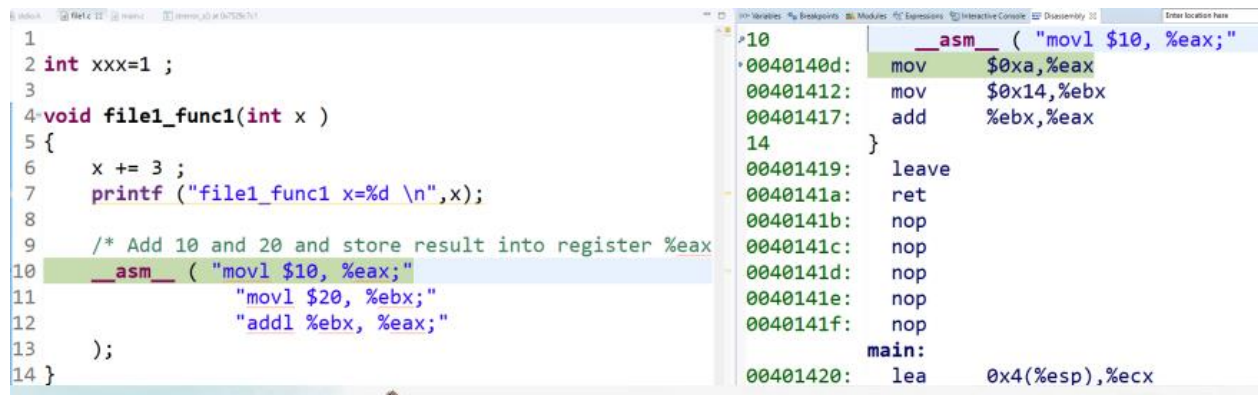
Inline assembly code can use any C or C++ variable or function name that is in scope

Using GCC

```
__asm__("movl %edx, %eax\n\t" "addl $2, %eax\n\t");
```

Using VC++

```
__asm { mov eax, edx add eax, 2 }
```



The screenshot shows a debugger window with two panes. The left pane displays C code, and the right pane shows the corresponding assembly instructions.

C Code (Left Pane):

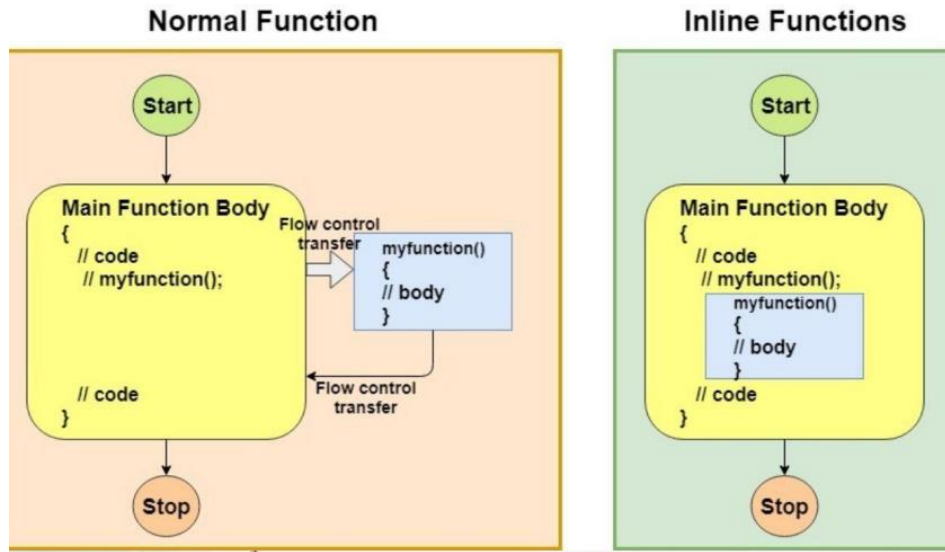
```
1  
2 int xxx=1 ;  
3  
4 void file1_func1(int x )  
5 {  
6     x += 3 ;  
7     printf ("file1_func1 x=%d \n",x);  
8  
9     /* Add 10 and 20 and store result into register %eax  
10    __asm__ ( "movl $10, %eax;"  
11              "movl $20, %ebx;"  
12              "addl %ebx, %eax;"  
13    );  
14 }
```

Assembly (Right Pane):

```
10  
0040140d: mov     $0xa,%eax  
00401412: mov     $0x14,%ebx  
00401417: add     %ebx,%eax  
14  
00401419: leave  
0040141a: ret  
0040141b: nop  
0040141c: nop  
0040141d: nop  
0040141e: nop  
0040141f: nop  
main:  
00401420: lea     0x4(%esp),%ecx
```

INLINE Function

Inline Function are those function whose definitions are small and be substituted at the place where its function call is happened. Function substitution is totally compiler choice



EX

```
4 // * Created on: Aug 20, 2020
5 // * Author: Keroles Shenouda
6 */
7
8 #include "stdio.h"
9
10 void file1_func1(int x);
11
12 /* Inline function in C */
13 inline int example(int x)
14 {
15     x += 3 ;
16     return x;
17 }
18
19 int main ()
20 {
21     int y = 7 ;
22     y=example (y);
23     printf ("%d",y);
24     return 0 ;
25 }
```