Storage Classes in C

Some terms and definitions

1. **Scope**: the scope of a variable determines over what part(s) of the program a variable is actually available for use(active).
2. **Longevity**: it refers to the period during which a variables retains a given value during execution of a program(alive)
3. **Local(internal) variables**: are those which are declared within a particular function.
4. **Global(external) variables**: are those which are declared outside any function.

Scope of a Variable

- What decides the point till a variable is visible/alive /accessible?
- Scope is how long the variable persists in memory, or when the variable's storage is allocated and deallocated
- Each variable has a scope (area in which they are visible/available)
- In general: It is visible within the block in which they are defined
- C has a concept of 'Storage classes' which are used to define the scope (visibility) and life time of variables and/or functions

What will be the output?

```
#include<stdio.h>
void fun(void);
int x=20;
main()
{
printf("%d",x);
fun();
}
void fun()
{
printf("%d",x);
} /* By moving the definition of x, you change its scope.*/
```

What will be the output?

```
#include<stdio.h>
void fun(void);
main()
{
int x=20;
printf("%d",x);
fun();
}
void fun()
{
printf("%d",x);
}
```

If a variable declared in the outer block shares the same name with one of the variables in the inner block, the variable within the outer block is hidden by the one within the inner block for the scope of the inner block

```
{           /*First Block */
int a=2;
printf(“%d”,a);
{              /*Second  Block */
int a=5;
printf(“%d”,a);
}
printf(“%d”,++a);
}
```

Why Scope ?

Problem Decomposition : there is a need to share and hide some variables

Avoid conflicts with variables of the same name used by others in other parts of the program

- »File
- »Block scope
- »Function scope

- The general syntax:

storage_specifier   data_type   variable_name;

Memory

Two kinds of locations where variables can be stored depending upon their storage class:

1. Memory
2. CPU registers

There are 4 storage classes in C:

1. Automatic storage class
2. Register storage class
3. Static storage class
4. External storage class

Automatic Storage class

- Variables declared inside main( ) function are private or local to main and no other function can have direct access to these variables
- Each local variable defined in a function comes into existence only when that function is called, and is destroyed when the function is exited
- Such variables are known as “ automatic variables” and are stored in memory
- Automatic variables are accessible only within the function they are defined
- Variables declared inside a function without storage class specification are, by default, automatic variables
- Auto variables are declared using the keyword *auto*

    eg. auto int number;

Example:

```
int main()
{ int m=1000;
  function2();
  printf("%d",m);
}
function1()
{
  int m = 10;
  printf("%d",m);
}
function2()
{  int m = 100;
   function1();
   printf("%d",m);
}
```

```
Output
10
100
1000
```

Automatic Variables

•**auto** is the default storage class for local variables

```
{ int Count;
auto int Month; }
```

•Declared inside a block, created and destroyed within a block

•Use optional keyword auto

•Same variable name can be used in other blocks or functions

•If automatic variables are not initialized they will contain garbage

Example:

```
#include<stdio.h>
int main(){
auto int i=1;
{
  auto int i=2;
  {
    auto int i=3;
    printf(“%d”,i);
 }
printf(“%d”,i);
}
printf(“%d”,i);
return 0;}
```

Register - Storage Class

•register is used to define local variables that should be stored in a register instead of memory

```
 {
register int Miles;
}
```

•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 implementations restrictions

We can not use register storage class for all the types of variables.

Following declarations are wrong for 16 bit registers:

```
register float a;
register double b;
register long c;
```

Global Variable

- **static** is the default storage class for global variables. These are alive and active throughout the entire program
- Accessible to any function in the program/**Program Scope**
- Local variable has precedence over global in the block where it is declared
- Any function can change its value
- It is visible only from the point of declaration to the end of the program

```
#include <stdio.h>
int x = 1234;                 /* program scope */
double y = 1.234567;    /* program scope */
 void function_1( )
{ printf("From function_1:\n x=%d, y=%f\n", x, y);
 }
main() {
int x = 4321;                   /* block scope 1*/
function_1();
 printf("Within the main block:\n x=%d, y=%f\n", x, y);
 /* a nested block */
 {
double y = 7.654321;     /* block scope 2 */
 function_1();
 printf("Within the nested block:\n x=%d, y=%f\n", x, y);  }
 return 0; }
```

Output:

```
From function_1: x=1234,y=1.234567
Within the main block: x=4321, y=1.234567
From function_1: x=1234, y=1.234567
Within the nested block: x=4321, y=7.654321
```

Static variables

- It is just like local variable but the difference is that it retains its previous value between the function calls.
- A static variable can only be accessed from the function in which it is declared and defined.
- The variable is not destroyed on exit from the function, instead it preserves it value and when the function is called again its previous value is available.

Important

- Static modifier has different effects upon Local and Global variables:
    - » Static local variables
    - » Static global variables

Static local variable

When static modifier is applied to local variable:

- The compiler creates permanent storage for it. The key difference between this type and global variable is that static local variable remains KNOWN only to the block in which it is declared.

- A static local variable RETAINS its value between the function calls.

Initialization

- If a static local variable is not initialized inside a function by default it is initialized to “ zero”.

  static int array[10];

- If a static local variable is initialized, this value is assigned only once at program start up, next time when the block of code is re-entered it retains it last value. (initialization does not take place again)

Static global variable

- Applying the specifier static to a global variable, instructs the compiler to create a global variable KNOWN ONLY to that file in which it is declared

- That is even though the variable is global in file, routines in other files have no knowledge of it and can not alter its content directly, keeping it FREE from side effects

Initialization

- Global and Static variables are initialized only at the start of the program
- Local variables (excluding static local variable) are initialized each time the block in which they are declared is entered
- Local variables that are not initialized have unknown values before the first assignment is made to them
- Un initialized global and static local variables are automatically initialized to zero

Example

```
main( )
{increment( );
 increment( );
 increment( );
}
    increment( )
    {      int i=1;
           printf(“%d\n”,i);
           i=i+1;
    }
```

```
Output:
1
1
1
```

Example

```
main( )
{increment( );
 increment( );
 increment( );
}
   increment( )
        {static int i=1;
        printf("%d\n",i);
        i=i+1;
   }
```

```
Output:
1
2
3
```

Extern storage class

- These variables are declared outside any function
- These variables are active and alive throughout the entire program
- Also known as global variables and default value is zero
- Unlike local variables they are accessed by any function in the program
- In case local variable and global variable have the same name, the local variable will have precedence over the global one
- The keyword ***extern*** used to declare these variables
- It is visible only from the point of declaration to the end of the program

Difference in extern and global

- Lifetime of both is entire program.
- Scope of global is “limited to the function or block in which it is declared”- although retains values between function calls.
- Scope of extern is global. Extern variables accessible in other files also.

```
 extern  int y;
  void Fun(void);
main() <------------------  extern  int y;
{
y=5;
printf(“%d”,y);
Fun();
}
int y;<-------------------  extern  int y;
Fun(){
y=y+1;
Printf(“y+1=%d”,y);}
```

```
#include<stdio.h>
int a=1,b=2,c=3;                          /* global variables*/
int f(void) ;

int main(void)
{ printf(“%3d\n”,f());                    /*12 is printed*/
printf(“%3d%3d%3d\n”,a,b,c);      /* 4 2 3 is printed*/
return 0;
}
int f(void)
{ int b,c;                                        /*b,c are local */
a=b=c=4;                                  /* global b,c are masked*/
return (a+b+c);
}
```

In file file1.c

```
#include<stdio.h>
int a=1,b=2,c=3;                /* external variables*/
int f(void);
int main(void)
{ printf("%3d\n",f());
printf("%3d%3d%3d\n",a,b,c);
return 0; }
```

In file file2.c

```
int f(void)
{ extern int a;             /* look for it elsewhere*/
  int b,c;
a=b=c=4;
return (a+b+c);}
```

Example

```
int i;
main( )
{printf(“\n i=%d\n”,i);
increment( );
 increment( );
 decrement( );
 decrement( );
}
```

```
increment( )
{i=i+ 1;
printf(“\n On Incrementing i= %d”,i);
}
decrement( )
{i=i- 1;
printf(“\n On decrementing i= %d”,i);
}
```

```
Output:
i=0
On Incrementing i=1
On Incrementing i=2
On decrementing i =1
On decrementing i =0
```