CP MODULE 5(Part 2of3)

Scope rules Storage classes.

STORAGE CLASSES IN C

- Storage class refers to the permanence of a variable, and its scope within the program.
- Storage class defines the **life-time and scope(visibility) of a variable**.
    - Scope of variable is, the portion of the program where the variable can be accessed.

There are four different storage-class specifications in C:

	keywords
Automatic	**auto**
External	**extern**
Static	**static**
Register	**register**

Example declarations

```
auto int  a, b, c;
extern float rootl, root2;
static  int count = 0;
extern char star;
```

Here a, b and c are automatic integer variables, and the second declaration establishes rootl and root2 as external floating-point variables. The third declaration states that count is a static integer variable whose initial value is 0, and the last declaration shows that star is an external character-type variable.

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

AUTOMATIC VARIABLES

- Automatic variables are always declared **inside a function**.
- If no storage class is specified when we declare a variable inside a function, it is considered as automatic variable.

E.g.

```
main()
{
        int a;
        int b;
}
```

This is same as:-

```
main()
{
        auto int a;
        auto int b;
}
```

a and b are automatic variables.

- Automatic variables are **local** to the function in which they are declared.
    - An automatic variable does not retain its value when control is transferred out of its defining function.

E.g

```
void show()
{
int a=5;          //This variable a AUTOMATIC.  cannot be accessed from outside this function
printf("a=%d",a);
}
main()
{
        show();
        printf("a=%d",a); //This is ERROR. Because a is local inside function show()
                          // a is not declared inside this block(function)// a is not global(external)
}
```

- Formal arguments in function definition header are also automatic variable.

E.g.

```
void add(int a ,int b)              // here a and b are formal arguments. They are automatic variables.
{

}
```

- *Scope of automatic variable*: inside the block.
- *Life time of automatic variable* : ends at the end of the block
- *Storage of automatic variable:* inside memory(stack)
- If not initialized then the *initial value* of automatic variable is garbage value.

```
main()
{
int a;
printf("a=%d",a);
}
```

OUTPUT

a=52241111

Here initial value is not assigned to a. So its initial value will be garbage value.

- An automatic variable <u>does not retain its value once control is transferred out of its defining function</u>.
  - So any value assigned to an automatic variable within a function will be lost once the function is exited.
- Automatic variables can be declared within a single compound statement. (block enclosed within { and }
  - It can be accessed only inside that block

E.g

```
#include<stdio.h>
main()
{
int a=3;                    // auto// local to main function
int b=5;                    // auto// local to main function
printf("\na=%d b=%d",a,b);          //a=3 b=5

        if(a>0)
        {
        int b;              // b is declared in this block.Soscope of this b is only inside this if block
                            //  when curly brace } comes its life ends
        b=2;
        printf("\na=%d b=%d",a,b);   //a=3 b=2 // a is not in this block.But a is in this function
                                                //with value 3
        }
        for(;a<4;a++)
```

```
        {
        int b;          //b is declared in this block  scope of this b is only inside this for block
                        //  when curly brace } comes its life ends
        b=10;
        printf("\na=%d b=%d",a,b);   //a=3 b=10//// a is not in this block.But a is in this function
        }

        printf("\na=%d b=%d",a,b);           //a=3 b=5 in this main function
        a=a+10;
        b=b+1;
        printf("\na=%d b=%d",a,b);           //a=13 b=6
}
```

Here in main() two variables a and b are declared. Their scope is inside the main() function block.
In **if** block another b is declared. The value of that b is available inside that if block
In **for** block also another b is declared. The value of that b is available only inside that for block.

E.g.

```
main()
{
int a=10;       // a is local inside main()
        if(a>0)
        {
        int b=3;        //b is local inside this if block
        }
printf("\n a=%d",a);   //a=10
printf("\n b=%d",b);   //ERROR. Because b is not declared inside main function() block
                       // b is declared only inside if block
                       //b is not global
}
```

EXTERNAL VARIABLES(global variable)

- External variables are called *global variables*.
- External variables are defined outside all functions
- They are not defined inside any function

E.g

```
#include<stdio.h>
```

```
extern int a=3,b;
int c;
void increment()
{a=a+1;                  // a is global//a=3+1
printf("\na=%d",a);     //a=4
}
main()
{
printf("\na=%d",a);     //a=3
increment();
printf("\na=%d",a);     //a=4
}
```

Here a ,b and c are external variables, since they are defined outside all functions. Storage class extern is not compulsory for external variables.

- Since variables are defined outside the function, the extern keyword is optional.
    - By default all variables defined outside all functions will be extern if no other storage class is specified.

- An external variable definition will automatically allocate the required storage space for the external variables within the computer's memory.
- The assignment of initial values can be included within an external variable definition if desired.
    - If not initialized its initial value will be 0(zero).
    - the initial values must be expressed as constants rather than expressions.
    - These initial values will be assigned only once, at the beginning of the program.

- *Scope of external variable*: whole program-
    - Also can be accessed from different files
- *Life time of external variable* : ends when program terminates
- *Storage:* inside memory(data segment)
- If not initialized then the *initial value* of *external* variable is 0.

- *External variables* can be accessed from any function in the program.
- *External variables* **retain their assigned values** within this scope.
    - Therefore an external variable can be assigned a value within one function, and this value can be used (by accessing the external variable) within another function.
    - If its value is modified it is also seen by all functions
- Any variable declared outside all function are external variables..

- *If external variable definition comes after a function definition that uses external variable*, then that function must include the **external variable declaration** of that variable.

External variable definition	External variable **declaration**
• *does not need extern* keyword. • It can contain initial values.	• must contain ***extern*** keyword. • External variable declaration should not contain initial value. • The name of the external variable and its data type must be same as the corresponding external variable definition.

External variable definition before all function definitions	External variable definition after show function definitions
int a=5; *//external variable definition* *void show()* { printf("\n a=%d",a); //a=5//a is global } main() { printf("\n a=%d",a); //a=5 show(); }	*void show()* { **extern int a;** *//external variable declaration* printf("\n a=%d",a); //a=5// a is global } **int a=5;** *//external variable definition* main() { printf("\n a=%d",a); //a=5 show(); } ///a is an extern variable but it is defined after show() function definition. //show() function prints the value of **a**. So it should declare the external variable **a** before accessing it.

Extern variables defined in one file can be accessed from another file

The external variable declared or defined in one file can be accessed by another file. We have to include the filename of first file in second file using #include.

sample.h

```
extern int g=5;
```

COMPILE THIS FILE and run

Fle1.c

```
#include<stdio.h>
#include"sample.h"
main()
{
printf("global variable g=%d",g);
g=g+3;
printf("\nglobal variable g=%d",g);
}
```

OUTPUT
global variable g=5
global variable g=8

STATIC VARIABLES

- Static variables are defined within individual functions
- Scope is same as automatic variables;
- i.e., they are local to the functions in which they are defined.
- The initial values must be expressed as constants, not expressions.

Static variables	Automatic variables
• Must begin with static keyword. • static variables retain their values throughout the life of the program even if a function is exited and then re-entered at a later time. • Static variables in a function are **initialized only once** when the function is called first time. • Static variable **does not lose** its value when control *exit from the function*. • Cannot be accessed outside of their defining function.	• Keyword auto is optional. • Automatic variables does not retain their values throughout the life of the program. • Automatic variables in a function are initialized ***every*** time the function is called. • Automatic variables **loses**its value when control exit from the function. • Cannot be accessed outside of their defining function.

static	auto
#include<stdio.h> void increase() { **static** int s=1; //static variable printf("\n s=%d",s); s=s+1; } **main()** { increase(); increase(); increase(); }	#include<stdio.h> void increase() { int s=1; //automatic variable printf("\n s=%d",s); s=s+1; } **main()** { increase(); increase(); increase(); }
OUTPUT s=1 s=2	OUTPUT s=1 s=1

s=3	s=1
● Here when first time increase() is called, static variable s is **initialized** to 1. It prints s as 1.Then s is incremented by 1. ○ So s becomes 2. ● Second time when increase() is called(s is not initialized again). Here it reads s as 2. Itprints s as 2. Then s is incremented by 1. ○ So s becomes 3. ○ (static variable retains its value even when the function is called again) ● Third time when increase() is called(s is not initialized again). Here it reads s as 3. It prints s as 3. Then s is incremented by 1. So s becomes 4.	● Here when first time increase() is called, auto variable s is **initialized** to 1. It prints s as 1.Then s is incremented by 1. So s becomes2. ● When second time increase() is called, auto variable s is **initialized** to 1. It prints s as 1.Then s is incremented by 1. So s becomes2. ● When third time increase() is called, auto variable s is **initialized** to 1. It prints s as 1.Then s is incremented by 1. So s becomes2. ●

- *Scope of static variable*: inside the block.
- *Life time of static variable* : ends when the program terminates
- *Storage of static variable:* inside memory(data segment)
- If not initialized then the *initial value* of static variable is 0(zero).

REGISTER VARIABLES

- Registers are special **storage areas within the REGISTERS of computer's central processing unit**.
- The actual *arithmetic and logical operations* that comprise a program are carried out within these registers.
- Variable declaration should be preceded by register keyword.

register	auto
register variables are **local** to the function in which they are declared The address operator(&) ***cannot*** be applied to register variables.	Auto variables are **local** to the function in which they are declared. The address operator (&) can be applied to register variables.
main () { register int a=2; printf("%x",&a); // ERROR //because address of register variable	main () { int a=2; printf("%x",&a); //prints the address of a

//cannot be accessed. So & cannot be used }	}
• *Scope of register variable*: inside the block. • *Life time of register variable* : ends at the end of the block • *Storage of register variable:* inside **CPU register** • If not initialized then the *initial value* of register variable is garbage value.	• *Scope of automatic variable*: inside the block. • *Life time of automatic variable* : ends at the end of the block • *Storage of automatic variable:* inside **memory**(stack) • If not initialized then the *initial value* of automatic variable is garbage value.
A *program* that makes use of register variables should **run faster** than the corresponding program without register variables. It may also be somewhat *smaller in size*	

- If the *requested register space is not available* register variables will be treated as automatic variables.
  Suppose a C program contains the variable declaration
  register int a, b, c;
  This declaration specifies that the variables a, b and c will be integer variables with storage class register. Hence, the values of a, b and c will be stored within the registers of the computer's central processing unit rather than in memory, if the register space is available.
  If the register space is not available, then the variables will be treated as integer variables with storage class automatic. Then,
  register int a, b, c;
  is equivalent to the declaration
  auto int a, b, c;
  or simply
  int a, b, c;

- Usually, **only integer variables** are assigned the **register storage class**.
  - some compilers allow the register storage class to be associated with other types of variables having the same word size (e.g., short or unsigned integers).
  - Moreover, pointers to such variables may also be permitted.

- The ***register is the only storage class specifier that can be used*** part of a **formal argument declaration** within a function, or as a part of an **argument type specification** within a *function prototype*.

Initialization of static extern, auto and register variables

The static initialized to 0.
The extern initialized to 0
The auto initialized to garbage values
The register initialized to garbage values

```
#include<stdio.h>

int x;              //external variable
main()
{
static int z;       //static variable
int y;              //automatic variable
register int r;     //external register variable

printf("\nstatic initialized  x=%d",x);
printf("\nextern initialized  z=%d",z);
printf("\nauto initialized   y=%d",y);
printf("\nregister initialized   r=%d",r);

}
```

OUTPUT
static initialized x=0
extern initialized z=0
auto initialized y=Garbage value
register initialized r=Garbage value