I B.Tech CSE - PPSC

STORAGE CLASSES

Variables in C differ in behavior from those in most other Languages. A variable in C can have any one of the four storage classes.

1. Automatic variables.
2. External variables.
3. Static variables.
4. Register variables.

SCOPE AND LIFETIME OF VARIABLES:

SCOPE: The scope of a variable determines over what part(s) of the program a variable is actually available for use.

LIFETIME: Lifetime refers to the period during which a variable retains a given value during execution of a program (alive).

A variable may also be broadly categorized depending on the place of their declaration as internal (local) or external (global). Internal variables are those which are declared within a particular function, while external variables are declared outside of any function.

1. AUTOMATIC VARIABLES:

Automatic variables are declared inside a function in which they are to be utilized. They are created when the function is called and destroyed automatically when the function is exited; hence the name automatic variables are also referred to as local or internal variables.

A variable declared inside a function without storage class specification is, by default, an automatic variable. We may also use the keyword auto to declare automatic variables explicitly.

```
void  main( )
{
   int num;
      -
      -
}
```

```
void main( )
 {
   auto int num;
      -
      -
 }
```

The value of automatic variable can not be changed accidentally by what happens in some other function in the program. This assures that we may declare and use the same variable name in different functions in the same program without causing any confusion to the compiler.

```
#include <stdio.h>
void main( )
{
   int m=1000;

   function2();

   printf("%d\n",m);

   getch();
}

function1( )
{
   int m=10;

   printf("%d\n",m);
}

function2( )
{
   int m=100;

   function1( );
   printf("%d\n",m);
}
```

Output
10
100
1000

A program with two sub programs function1( ) and function2( ) is shown. m is an automatic variable and it is declared at the beginning of the each function. m is initialized to 10, 100, 1000 in function1, function2 and main( ) respectively.

When executed, main( ) calls function2, which in turn calls function1. When main is active m=1000, but when f2 is called the main's m is temporarily put on the shelf and the new local m=100 becomes active. Similarly when function1 is called both the previous values of m are put on the shelf and the latest value of m=10 becomes active. As soon as function1 is finished, function2 takes over again. As soon it is done, main takes over the output clearly. Shows that the value assigned to m is one function does not effect its value in the other functions, and the local value of m is destroyed when its leaves a function.

2. EXTERNAL VARIABLES:

Variables that are both alive and active through out the entire program are known as external variables. They are also known as global variables. Unlike local variables, global variables can be accessed by any function in the program. External variables are declared outside a function.

Ex:

```
int a;
int b;
void main( );
{
   -
   -
}
function1( )
{
   -
   -
}
function2( )
{
   -
   -
}
```

The variables a and b are available for use in all the three functions. In case a local variable and global variable have the same name, the local variable will have precedence over the global one in the function where it is declared.

Once a variable has been declared as global, any function can use it and change its value. Then subsequent functions can reference only that new value.

One other aspect of a global variable is that it is visible only from the point of declaration to the end of the program. Consider a program segment as shown below.

```
void  main( )
 {
   y=5;
   -
   -
 }

 int y;

function1( )
{
    y=y+1;
}
```

As far as main() is concerned, y is not defined. So, the compiler will issue an error message. Unlike local variables global variables are initialized to zero by default.

In the above program segment, the main() can not access the variable y as it has been declared after the main() function. Declaring the variable with the storage class extern can solve this problem.

```
void main( )
{
  extern int y;
  -
  -
}

function1( )
{
  extern int y;
  -
  -
}
 int y;
```

Although the variable y has been defined after both the functions, the external declaration of y inside the functions informs the compiler that y is an integer type defined some where else in the program.

Program 1) Global variable declaration before the functions

```
#include<stdio.h>
int num;                    /* Global variable Declaration */

void main()
 {
     num=2;

     printf("num=%d\n",num);

     fun1();
     printf("num=%d\n",num);

      fun2();
      printf("num=%d\n",num);

      getch();
 }

void fun1()
{
    num=num+5;
}

  void fun2()
{
    num=num+10;
}
```

OUTPUT

```
num = 2
num = 7
num = 17
```

Program 2) Global variable declaration after the functions

```
#include<stdio.h>
 void  main()
 {
      extern int num;

       num=2;                                          OUTPUT
       printf("num=%d\n",num);                          num=  2
       fun1();                                         num=  7
       printf("num=%d\n",num);                         num=17
       fun2();
        printf("num=%d\n",num);

        getch();
 }

void fun1()
{
     extern int num;

     num=num+5;
}

void fun2()
 {
      extern int num;

      num=num+10;
 }

 int num;                    /* Global variable Declaration */
```

3. STATIC VARIABLES:

As the name suggests, the value of static variables persists until the end of the program. A variable can be declared static using the key word static like

```
static int x;
static float y;
```

A static variable may be either an internal type or an external type, depending on the place of declaration. Internal static variables are those which are declared inside a function. The scope of internal static variables extends up to the end of the function, in which they are defined. Therefore, internal static variables are similar to auto variables, except that they remains in existence (alive) throughout the remainder of the program. Therefore internal static variables can be used to retain values between function calls.

Example for static variable

```
#include <stdio.h>
void fun( )
{
   static int x=0;
   x=x+1;
   printf("x=%d\n",x);
}

void main( )
{
   int i;

   for(i=1; i<=3; i++)
        fun( );

   getch();
}
```

Example for Auto variable

```
#include <stdio.h>
void afun( )
{
      int x=0;
      x=x+1;
      printf("x=%d\n",x);
 }

void  main( )
{
     int i;

     for(i=1; i<=3; i++)
          afun( );

     getch();
}
```

out put

x=1
x=2
x=3

out put

x=1
x=1
x=1

A static variable is initialized only once, when the program is compiled it is never initialized again.

During the first call to fun( ) function x is incremented to 1. Because x is static, this value persists and therefore, the next call adds another 1 to x giving it a value of 2. The value of x becomes 3 when the third call is made.

4. REGISTER VARIABLES:

We can tell the compiler that a variable should be kept in one of the machine's registers, instead of keeping in the memory. Since a register access is much faster than a memory access, keeping the frequently accessed variables in the register will lead to the faster execution.

```
register int count;
```

Most compilers allow only int or char variables to be placed in the register

MULTIPLE PROGRAMS

So far we have been assuming that all the functions are defined in one file. However in real life programming environment, we may use more than one-source files which may be compiled separately and linked later to form an executable object code.

Multiple source files can share a variable provided it is declared as an external variable appropriately. Variables that are shared by two or more files are global variables and therefore we must declare them accordingly in one file and then explicitly define them with extern in other files.

/*program to find factorial of a given number using multiple programmes*/

/* program file containing main() function --- mul1.c */

```
#include<stdio.h>
#include <conio.h>
#include "mul2.c"

void main()
{
   int n,fn;

   clrscr();
   printf("enter n");
   scanf("%d",&n);

   fn=factorial(n);     /* Function Call */

   printf("the factorial %d is %d\n",n,fn);

   getch();
 }
```

/* program file containing function factorial ---- mul2.c */

```
int factorial(x)
int x;
{
   int i,f;

   f=1;
   for(i=1;i<=x;i++)
      f=f*i;

   return(f);
}
```

PREPROCESSOR COMMANDS

As the name suggests the preprocessor commands are the instructions that are executed before the source code passes through the compiler. The program that processes the preprocessor command lines or directives is called a preprocessor.

Preprocessor directives are placed in the source program before the main() line. Preprocessor directives begins with the symbol # and do not require a semicolon at the end.

Directive	Function
#define	Defines a macro substitution
#undef	undefines a macro
#include	specifies the files to be included
#ifdef	test for a macro definition
#endif	specifies end of #if
#if	test a compile time condition
#else	specifies alternatives when #if test fails

Examples

```
#define   COUNT      100
#define   TRUE       1
#define   FALSE      0
#define   PI         3.1415
#define   AREA       5*12.46
#define   SIZE       sizeof(int)*4
#define   M          5
#define   N          M+1
#define   SQUARE(x)  ((x)*(x))
```