UNIT – 7
Functions

C Programming (CSC115)
BSc CSIT First Semester (TU)

Prepared by:
Dabbal Singh Mahara
ASCOL (2025)

2 FUNCTION

- A function is defined as a self-contained block of statements that performs a particular task or job.
- Every C program has one or more functions.
- The function main() is always present in every C program which is executed first and other functions are optional.

ADVANTAGES OF FUNCTIONS

- **Code Reusability:** Avoids rewriting the same code again and again. For example, if we need to compute combination using formula C(n, r)=n!/(n-r)!*r!, we need to compute factorial of n, (n-r) and r. If we try to solve this problem without using function we have to repeat the same logic of computing factorial 3 times.
- **Manageability and Logical Clarity:** Writing a specific function for a specific task, it is easier to decompose a large program into manageable chunks. This will make our program easier to write, understand, debug and test.
- **Easy to divide the work among different programmers** while working on a large project

```
int main()
{
long f1=1, f2=1, f3=1, comb;
int  n, r, i;
printf("\nEnter n and r:\t");
scanf("%d %d", &n, &r);
for(i=1;i<=n;i++)
    f1 *= i;
for(i=1;i<=n-r;i++)
    f2 *= i;
for(i=1;i<=r;i++)
    f3 *= i;
comb=f1/(f2*f3);
printf("\n The combinations is:%ld", comb);
getch();
return 0;
}
```

Problem:

Redundant code

```
// Combination Problem
#include <stdio.h>
#include <conio.h>
long factorial( int n);
int main()
{
  long f1=1,f2=1,f3=1,comb;
  int n, r;
  printf("\nEnter n and r:");
  scanf("%d %d",&n,&r);
  f1=factorial(n);
  f2=factorial(n-r);
  f3=factorial(r);
  comb=f1/(f2*f3);
  printf("\n The combination is: %ld", comb);
  getch();
  return 0;
}
```

```
long factorial(int n)
{
        long fact=1;
        int i;
        for(i=1;i<=n;i++)
            fact *= i;
        return fact;
}
```

TWO TYPES OF FUNCTIONS

- **Library functions (Built-in functions):** These functions are provided in the programming language and we can directly use them as required. However, the function's name, return type, number of arguments and types must be known in advance. For e.g. printf(), scanf(), sqrt(), getch(), etc.
- **User-defined functions:** These functions are written by the user. The user selects the name of the function, return type, number of arguments and types.
- Note: main() is a user defined function. However the name of the function is defined or fixed by the programming language.

FUNCTION DEFINITION

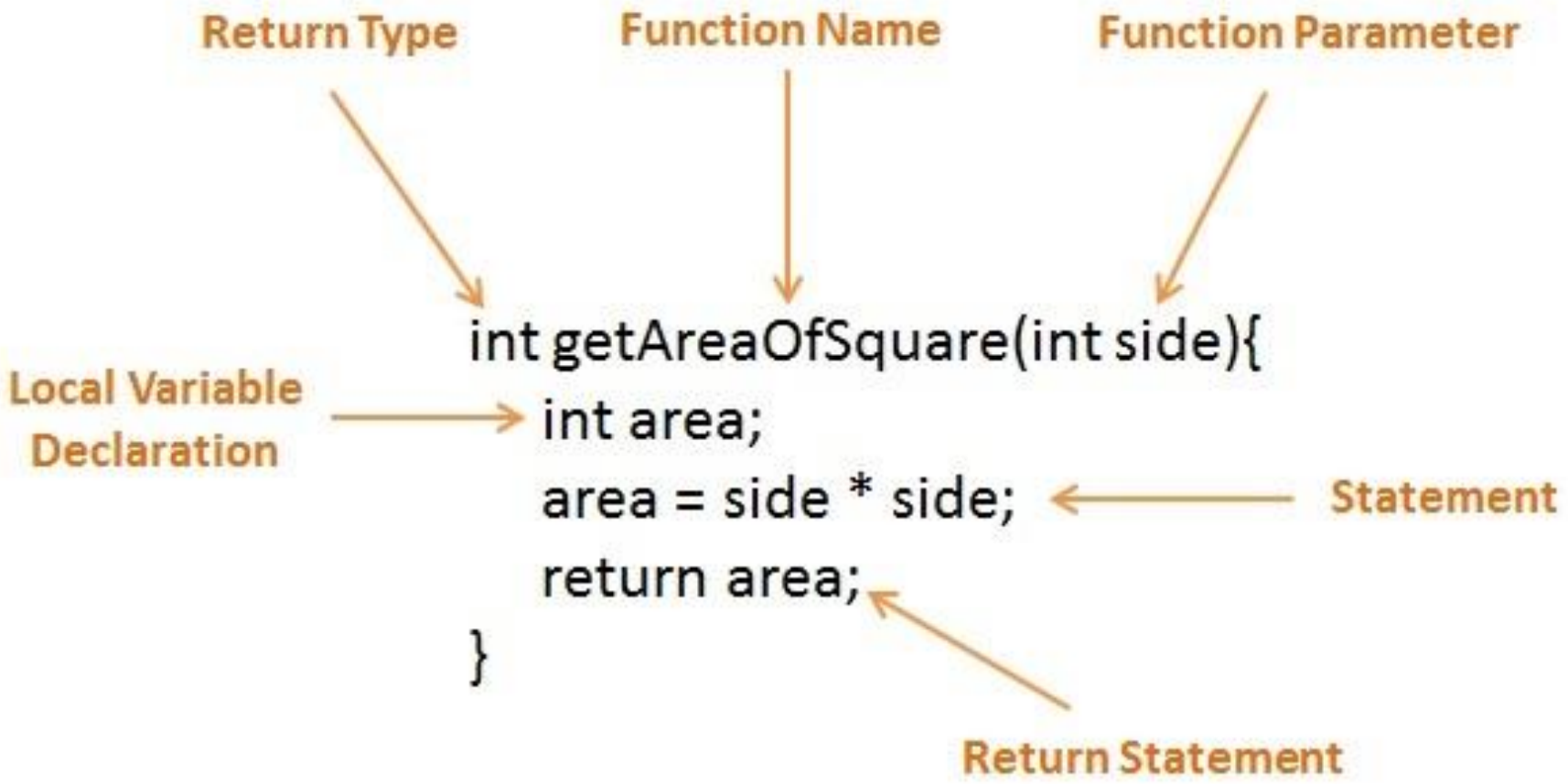
- The collection of program statements that describes the specific task to be done by the function is called a function definition.
- Function definition consists of function header (function's name, return type, and number and types of arguments) and function body (block of code or statements enclosed in parentheses).
- Syntax:

```
return_type function_name(data_type variable1, …,data_type variableN )
{
……;
……;
statements;
}
```

FUNCTION DEFINITION…

- *return_type* is optional. The default value is integer.
- Whenever *return_type* is provided, a value must be returned using return statement except for void case.
- *function_name* is a user-defined name given to the function. (Same as identifier naming)
- variable1, …,variableN are called *formal arguments* or *formal parameters* that are passed to the function. Also these are local variables for the function.

Function Definition



CALLING A FUNCTION THROUGH MAIN()

- A function can be called by specifying the function's name, followed by a list of arguments enclosed in parentheses and separated by commas.
- For example, the function getAreaOfSquare() can be called with one argument from main() function as:

```
getAreaOfSquare(s);
```

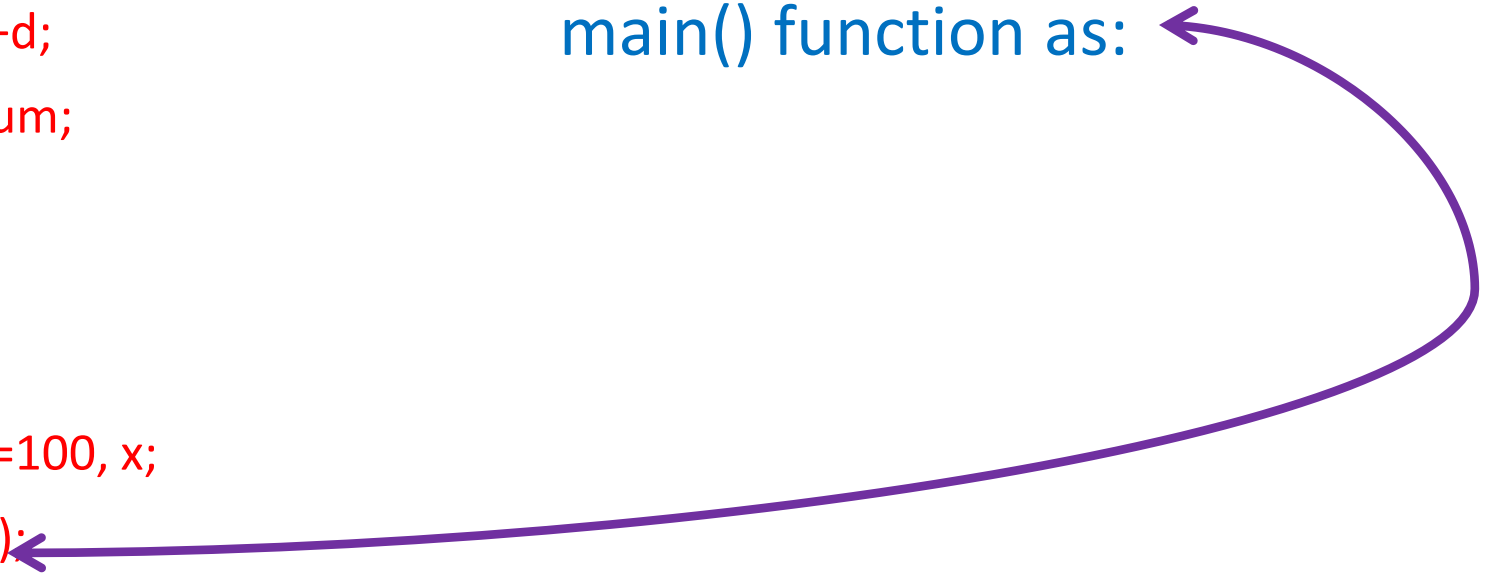
- These arguments appearing in the function call are called *actual arguments* or *actual parameters*.
- In this case, main() is called calling function and getAreaOfSquare() is called called function.

To be noted ! About Function Call

- In function call, there must be one actual argument for each formal argument. This is because the value of actual argument is transferred into the function and assigned to the corresponding formal argument.
- If a function returns a value, the returned value can be assigned to a variable of data type same as return type of the function to the calling function.
- When a function is called, the program control is passed to the function and once the function completes its task, the program control is transferred back to the calling function.

```
#include <stdio.h>
#include <conio.h>
int add(int c, int d)
{
   int sum;
   sum = c+d;
   return sum;
}
int main()
{
int a=50,b=100, x;
x=add(a, b);
printf("%d", x);
getch();
return 0;
}
```

When value is being returned by a function, capture it in main() function as:



```
   void add(int c, int d)
   {
   int sum;
   sum = c+d;
   printf("%d", sum);
   }
int main()
{
int a=50, b=100;
add(a, b);
getch();
return 0;
}
```

When value is not being returned (i.e. return type is void), print within the function, if needed.

Practice Time

Write functions to add, subtract, multiply and divide two numbers a and b.

FUNCTION PROTOTYPE OR DECLARATION

- The function prototype is a model or blueprint of the function.
- The function prototype is necessary if a function is used before function is defined.
- When a user-defined function is defined before the use, function prototype is not necessary (may be given though).
- Syntax:

  return_type function_name(data_type1, …,data_typeN);

- E.g.

  void add(int, int);

```
#include <stdio.h>
#include <conio.h>
int add(int, int);    //Function Prototype
int main()
{
  int a=12,b=5,x;
  x=add(a,b);
  printf("%d",x);
  getch();
  return 0;
}
int add(int c, int d)
  {
        int sum;
        sum=c+d;
        return sum;
  }
```

THE *RETURN* STATEMENT

The return statement serves two purposes:

1. It immediately transfers the control back to the calling function (i.e. no statements within the function body after the return statement are executed).
2. It returns the value to the calling function.

Syntax:

```
return (expression);
```

where expression, is optional and, if present, it must evaluate to a value of the data type specified in the function header for the *return_type*.

Note: When nothing is to be returned, the *return_type* in the function definition is specified as void. For void *return_type*, return statement is optional.

```
/*Program using function to find the greatest number among three numbers*/
#include <stdio.h>
#include <conio.h>
int greater(int, int);
int main()
{
int a, b, c, d, e;
printf("\n Enter three numbers:");
scanf("%d %d %d",&a,&b,&c);
d=greater(a, b);
e=greater(d, c);
printf("\n The greatest number is:%d", e);
getch();
return 0;
}

int greater(int x, int y)
{
    if(x>y)
                return x;
    else
                return y;
}
```

19 FUNCTION PARAMETERS

- Function parameters are the means for communication between the calling and the called functions.
- Two types: formal parameters and actual parameters.
- Formal parameters are given in the function definition while actual parameters are given in the function call.
- The name of formal and actual parameters need not be same but data types and the number of parameters must match.

TYPES OF FUNCTIONS

1) Functions with no arguments and no return values
2) Functions with arguments but no return values
3) Functions with arguments and return values
4) Functions with no arguments but return type

FUNCTIONS WITH NO ARGUMENTS AND NO RETURN VALUES

- When a function has no arguments, it does not receive any data from the calling function.
- When a function does not return a value, the calling function does not receive any data from the called function.
- Thus, there is no data transfer between the calling function and the called function.
- Syntax:

```
void function_name()
{
    /* body of function */
}
```

//EXAMPLE

```
#include <stdio.h>
#include <conio.h>
void add()
{
int a,b,sum;
printf("\nEnter two numbers:");
scanf("%d %d",&a,&b);
sum=a+b;
printf("\nThe sum is:%d",sum);
}

int main()
   {
   add();
   getch();
    return 0;
   }
```

FUNCTIONS WITH ARGUMENTS BUT NO RETURN VALUE

- This type of function has arguments and receives the data from the calling function.
- But after the function completes its task, it does not return any values to the calling function.
- Syntax:

```
void function_name(argument_list)
{
/* body of function */
}
```

//EXAMPLE

```
#include <stdio.h>
#include <conio.h>
void add(int a, int b)
{
int sum;
sum=a + b;
printf("\n The sum is:%d", sum);
}

int main()
{
    int a,b;
    printf("\n Enter two numbers:");
    scanf("%d %d",&a,&b);
    add(a, b);
    getch();
    return 0;
}
```

FUNCTIONS WITH ARGUMENTS AND RETURN VALUES

- This type of function has arguments and receives the data from the calling function.
- However, after the task of the function is complete, it returns the result to the calling function via return statement.
- So, there is data transfer between called function and calling function using return values and arguments.
- Syntax: *return_type function_name(argument_list)*

```
{
/* body of function */
}
```

//EXAMPLE

```
#include <stdio.h>
#include <conio.h>
int add(int a, int b)
{
int sum;
sum=a + b;
return sum;
}
int main()
    {
    int a, b, x;
    printf("\n Enter two numbers:");
    scanf("%d %d", &a, &b);
    x=add(a, b);
    printf("\n The sum is:%d", x);
    getch();
   return 0;
    }
```

Question

- Write a function to solve a quadratic equation $ax^2+bx+c=0$. The inputs to the function are the values a, b and c and the outputs of the function should be stored in variable names p and q appropriately declared.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
void quad(float, float, float);
int main()
{
float a, b, c;
printf("\n Enter values a, b and c of the quadratic
   equation:");
scanf("%f %f %f", &a, &b, &c);
quad(a, b, c);
getch();
}
```

```
void quad(float a, float b, float c)
{
float p, q;
float d;
d = b*b-4*a*c;

if(d<0)
    {
    printf("\n Imaginary Roots.");
    d = sqrt(fabs(d));    //compute absolute value of discriminant
    p = -b/(2*a);
    q = d/(2*a);
    printf("\nRoot1 = %.2f +i %.2f",p,q);
    printf("\nRoot2 = %.2f -i %.2f",p,q);
    }
else
    {
    printf("\nRoots are real.");
    d = sqrt(d);
    p = (-b+d)/(2*a);
    q = (-b-d)/(2*a);
    printf("\nRoot1 = %.2f \t Root2 = %.2f",p,q);
    }
}
```

FUNCTION WITH RETURN VALUE AND NO ARGUMENTS

- These types of functions have no arguments and but have return value.
- These functions define the data within the function as per requirement.
- After manipulating these local data, the result is returned to the calling function.

//EXAMPLE

```
#include <stdio.h>
#include <conio.h>
int add( )
{
   int a, b, sum;
    printf("\n Enter two numbers:");
    scanf("%d %d", &a, &b);
    sum=a + b;
    return sum;
}
int main()
    {
     int x;
     x=add();
    printf("\n The sum is:%d", x);
    getch();
    return 0;
    }
```

TYPES OF FUNCTION CALLS

- Function calls are divided in two ways according to how arguments are passed in the function.

1. Pass arguments by value (Function call by Value)
2. Pass arguments by address (Function call by Reference)

PASSING BY VALUE

- When values of actual arguments are passed to the function as arguments, it is known as passing by value.
- Here, the value of each actual argument is copied into corresponding formal argument of the function definition.
- Note: The contents of the actual arguments in the calling function are not changed, even if they are changed in the called function.

```
#include <stdio.h>
#include <conio.h>
void swap(int, int);
int main()
{
int a=50, b=100;
printf("\n Before swap function call: a=%d and b=%d", a, b);
swap(a,b);
printf("\n After swap function call: a=%d and b=%d", a, b);
getch();
}

void swap(int x, int y)
   {
            int temp;
            temp=x;
            x=y;
            y=temp;
            printf("\n Values within swap: x=%d and y=%d", x, y);
   }
```

35 PASSING BY ADDRESS

- When addresses of actual arguments are passed to the function as arguments (instead of values of actual arguments), it is known as passing by address.
- Here, the address of each actual argument is copied into corresponding formal argument of the function definition.
- In this case, the formal arguments must be of type pointers.
- Note: The values contained in addresses of the actual arguments in the calling function are changed, if they are changed in the called function.

WHAT IS A POINTER???

- A pointer is a variable that stores the memory address of a variable
- Pointer naming is same as variable naming and it is declared in the same way like other variables but is always preceded by * (asterisk) operator.
- E.g.

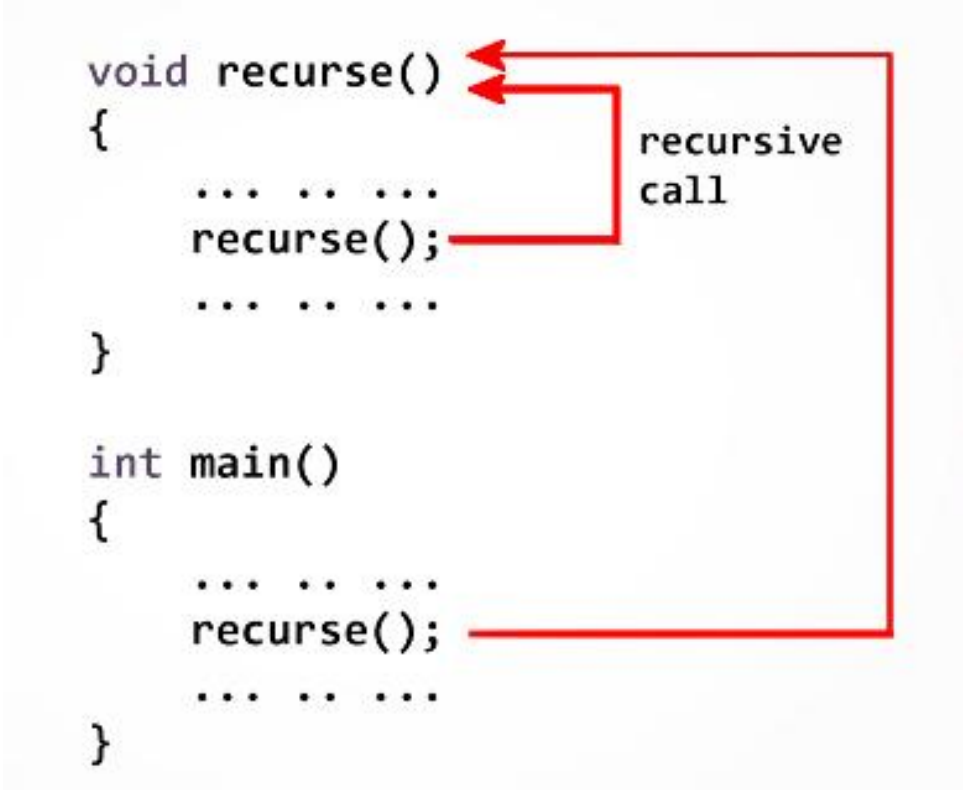
```
int  b, *a;//pointer declaration
a=&b;                /* address of b is assigned to
pointer variable a */
```

```
#include <stdio.h>
#include <conio.h>
void swap(int *, int *);
int main()
{
int a=50, b=100;
printf("\n Before swap function call: a=%d and b=%d", a, b);
swap(&a, &b);
printf("\n After swap function call: a=%d and b=%d", a, b);
getch();
}

void swap(int *x, int *y)
   {
            int temp;
            temp=*x;
            *x=*y;
            *y=temp;
            printf("\n Values within swap: x=%d and y=%d", *x, *y);
   }
```

Recursive function

When a function calls itself directly or indirectly, it is called recursive function and the process of calling itself is called recursion.



39 RECURSIVE FUNCTION

- To solve a problem using recursive method, two conditions must be satisfied:

1) Problem should be written or defined in terms of its previous result.
2) Problem statement must include a terminating condition, otherwise the function will never terminate. This means that there must be an if statement somewhere in the recursive function to force the function to return without the recursive call being executed.

//Factorial of a number using recursion

```
#include <stdio.h>
#include <conio.h>
long int factorial(int n)
{
if(n==1)
  return 1;
else
  return (n*factorial(n-1));
}
int main()
{
  int number;
  long int x;
  printf("Enter a number whose factorial is needed:\t");
  scanf("%d", &number);
  x=factorial(number);
  printf("\n The factorial is:%ld", x);
  getch();
 return 0;
}
```

41 PROBLEM

- Write a program to find $\mathbf{a^b}$ using recursion.

Storage Class

Variables in C are categorized into four different storage classes according to the *scope* and *lifetime* of variables:

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

Note:

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

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

Note:

- Variables can also be broadly categorized, depending on the place of their declaration, as *internal* (local) or *external* (global).

- *Local* variables are those which are declared within a particular function, while *global* variables are declared outside of any function.

44 LOCAL VARIABLE (AUTOMATIC OR INTERNAL VARIABLE)

- Automatic variables are always declared inside a function or block in which they are to be used.
- They are *created* when the function is called and *destroyed* automatically when the function is exited, hence the name is automatic.
- The keyword *auto* is used for storage class specification although it is optional.
- Initial value: garbage
- Scope: local to the block where the variable is defined
- Lifetime: till the control remains within the block where variable is defined

```
#include <stdio.h>
#include <conio.h>
void function1()
{
auto int m=20;
printf("\n m=%d", m);
}
int main()
  {
        auto int m=10;
        printf("m=%d", m);
        function1();
        getch();
         return 0;
  }
```

Note: Use of keyword “auto” is optional

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int m=10;
        printf("m=%d", m);
        {
                int m=20;
                printf("\nm=%d", m);
        }
        printf("\nm=%d", m);
        getch();
        return 0;
}
//Two variables can have same name in different scope
```

47 GLOBAL VARIABLE (EXTERNAL VARIABLE)

- The global variables are declared outside any block or function.
- These variables are both alive and active throughout the entire program.
- Unlike local variables, global variables can be accessed by any function in the program.
- Initial value: zero
- Scope: global (i.e. throughout the program (multifile too))
- Lifetime: throughout program's execution

```
#include <stdio.h>
#include <conio.h>
int a;
int main()
{
printf("%d", a);
getch();
return 0;
}
//Global Variable
```

```
#include <stdio.h>
#include <conio.h>
int main()
{
int a;
printf("%d", a);
getch();
return 0;
}
//Local Variable
```

```
#include <stdio.h>
#include <conio.h>
int a=100;
void function()
{
a=200;
printf("%d\n", a++);
}
int main()
{
printf("%d\n", a);
function();
printf("%d", a);
getch();
return 0;
}
```

- **Description**

  Here, once a variable is declared as global, any function can use it and change its value. Then subsequent functions can reference only that new value. As the global variable a is recognized by main() as well as function(), so values are printed as 100, 200 and 201.

Note: Care should be taken while using global variables. Only those variables should be declared as global which are to be shared among different functions when it is inconvenient to pass them as parameters.

- Question: Since global variables are declared outside of any function, can you declare a global variable anywhere in the program and use it among many different functions?

- Ans: No. A global variable is visible only from the point of declaration to the end of the program.

- Solution: declare variable using storage class *extern* in all functions

```
#include <stdio.h>
#include <conio.h>
void function()
{
y=y+1;
printf("%d\n", y);
}
int main()
{
y=100;
function();
getch();
return 0;
}
int y;
//Error
```

```
#include <stdio.h>
#include <conio.h>
void function()
{
y=y+1;
printf("%d\n", y);
}
int y;
int main()
{
y=100;
function();
getch();
return 0;
}
//Error
```

```
#include <stdio.h>
#include <conio.h>
void function()
{
extern int y;
y=y+1;
printf("%d", y);
}
void main()
{
extern int y;
y=100;
clrscr();
function();
getch();
}
int y;
```

54 MULTIFILE PROGRAM

- In real-life programming environment, more than one source files are compiled separately and linked later to form an executable code.
- Multiple source files can share a variable if it is declared as an external (global) variable appropriately as shown in coming slide:

//file1.c saved in C:\TC\BIN

```
#include <stdio.h>
#include <conio.h>
#include "file2.c"
void fun2();
int m;
void fun1()
{
   m=1;
   printf("\nThis is fun1 in file1.c where m=%d",m);
}
int main()
{
printf("\nThis is main where m=%d",m);
fun1();
fun2();
getch();
return 0;
}
```

//file2.c saved in C:\TC\BIN

```
extern int m;

void fun2()

{

m=2;

printf("\n This is fun2 in file2.c where m=%d", m);

}
```

/*A multifile global variable should be declared without the *extern* keyword in one (and only one) of the files*/

STATIC VARIABLE

- As the name suggests, the value of static variables persists till the end of the program.
- A variable is declared static using the keyword *static*:

```
static int x;
```

- A static variable may be either an internal type or an external type, depending on the place of declaration.
- Initial value: zero (initialized only once)
- Scope: local (within the function) or global (within the program (single file))
- Lifetime: throughout the program for local as well as global static variables

Static Variable…

Q.1) What is the difference between internal *static* variable and *auto* variable?

Ans: The lifetime of internal static variables is throughout the program. So they retain values between function calls. Also internal static variables are initialized only once.

Q.2) What is the difference between external *static* variable and *extern* variable?

Ans: The static external variable is available only within the file where it is defined whereas external variable can be accessed by other files as well.

```
#include <stdio.h>
#include <conio.h>
void stat()
{
int x=0;
x=x+1;
printf("x=%d\n", x);
}
int main()
{
int i;
for(i=1;i<=3;i++)
    stat();
getch();
return 0;
}
```

```
#include <stdio.h>
#include <conio.h>
void stat()
{
static int x=0;
x=x+1;
printf("x=%d\n", x);
}
int main()
{
int i;
for(i=1;i<=3;i++)
    stat();
getch();
return 0;
}
```

REGISTER VARIABLE

- Normally, variables are stored in memory.
- However, we can instruct the compiler that a variable should be kept in one of the CPU's registers, instead of keeping in the memory.
- Use: A register access is much faster than a memory access. So keeping frequently accessed variables (e.g. loop control variables) in the register will help faster execution of the program.
- A register variable is declared using *register* keyword:
- Example: *register int i;*

REGISTER VARIABLE…

- Register variables are always declared inside a function or block.
- They are allocated space upon entry to a block; and the storage is freed when the block is exited.
- Initial value: garbage
- Scope: only in that function or block
- Lifetime: until end of function or block

```
#include <stdio.h>
#include <conio.h>
int main()
{
register int i;
for(i=1;i<100;i++)
  printf(" %d\t", i);
getch();
return 0;
}
```


Thank You