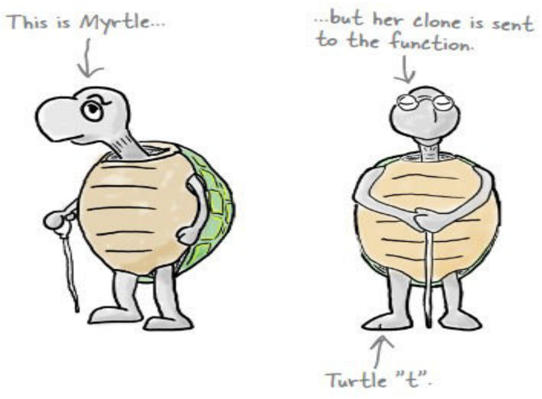
UNIT – IV
FUNCTIONS

TOPICS



Functions:

Designing structured programs
Function Basics,
User Defined Functions,
- a) Function prototype/function declaration
- b) Function definition
- c) Function call
- d) Function parameters
- e) return statement

Parameter passing mechanism
- a) call by value
- b) call by reference

Categories of functions
- Category 1: Functions with no arguments and no return value.
- Category 2: Functions with arguments and no return value.
- Category 3: Functions with arguments and return value.
- Category 4: Functions with no arguments and return value.

Local variables, global variables, static variables
Scope of variables
- a) Block scope
- b) Function scope
- c) Program scope
- d) File scope

Storage classes
- a)auto
- b)register
- c)extern
- d)static

Recursion- Recursive Functions
Inter Function Communication
- 1D array for inter function communication
  - (a)passing individual elements
    - i)passing data values
    - ii)passing addresses
  - (a)passing entire array
- 2D array for inter function communication
  - a)passing individual elements
  - b)passing a row
  - c)passing entire 2D array

Standard functions
Preprocessor Commands.

Designing Structured Programs in C

- Structured programming is a programming technique in which a larger program is divided into smaller subprograms to make it easy to understand, easy to implement and makes the code reusable, etc.
- Structured programming enables code reusability.
- **Code reusability** is a method of writing code once and using it many times.
- Using a structured programming technique, we write the code once and use it many times.
- Structured programming also makes the program easy to understand, improves the quality of the program, easy to implement and reduces time.

- In C, the structured programming can be designed using **functions** concept.
- Using functions concept, we can divide the larger program into smaller subprograms and these subprograms are implemented individually.
- Every subprogram or function in C is executed individually.

Functions

- Function is a self-contained program segment that carries out some specific, well defined task.
- A function in C is considered as a fundamental building block of the language.
- Inorder to avoid the complexity of a program while coding, debugging, and testing the program is divided into functional parts (or) subprograms.
- Thus each module is considered as a function and functions do every thing in 'C'.
- In C – language every task handled by the function. Hence C – Language is called as functional oriented language.

What is the difference between function and procedure ?

Function and procedure both are set of statements to perform some task, the difference is that, Function returns a value, where as procedure do not return a value.

Advantages of using functions:

1.Using functions we can avoid repeated code. Hence length of the code will be reduced (code reusability)
2.Easy to develop the applications as each functions treated as separate units.
3.With functions we can get application modularity .
4.easy to maintain projects.

Disadvantages :

1.Time take to jump the control between functions (considerable compared to development efforts)
2. If programmer is not expert in using recursive functions, there is a chance raising stack over flow exception.

Important facts about functions:

1. C – Program is a collections of functions
2. To compile C – Program at least one function required.
3. C-Program compiles from top to bottom but execution starts from main()
4. To compile C-Program main() function is not compulsory but to create exe file and execute main() function should be required.
5. There is **no limit on no of functions** to be defined.
6. There is **no order** of defining functions, any function can be defined any where in the program, even before the main() function also can be defined.
7. There is **no priority** of functions, all are equal, even main() also equal to other functions.
8. Any function can be called from any function, even main() function can also be called from any other function.
9. execution of functions depends on **sequence in the main()**.
10. **main() is starting point of execution**. It is controller of C-Application
11. A function calling itself is called as recursive function.
12. main() function, neither user defined nor pre-defined . It is an interface between user (programmer) and system.

Types of functions:
1. Built-in functions
2. User defined functions

1. Built-in functions:
- Built in functions are the functions that are provided by the C library. Many activities in C are carried out using library functions. These functions perform file access, mathematical computations, graphics, memory management etc.
- A library function is accessed simply by writing the function name, followed by an optional list of arguments and header file of used function should be included with the program
- Definition of built in functions are defined in a special header file. A header file can contain definition of more than one library function but the same function can not be defined in two header files.
- These are from C-Library files (header files). **Eg:** printf(), scanf(), strcpy(), strlwr(), strcmp(), strlen(), strcat(), gets(), puts(), getchar(), putchar() etc

2. User defined functions:
A user defined function has to be developed by the user at the time of writing a program. Another approach is to design a function that can be called and used whenever required. This saves both time and spaces. Every function has the following elements associated with it.

a) Function prototype/function declaration
b) Function definition
c) Function call
d) Function parameters
e) return statement

a) Function prototype/function declaration:
Every function in C programming should be declared before they are used. These type of declaration are also called function prototype. Function declaration gives compiler information about function name, type of arguments to be passed and return type.

The general format for declaring a function that accepts some arguments and returns some value as a result can be given as:

```
return_type function_name(data_type varaiable1, data_type variable2,...);
```

Here function_name is valid name for the function. return_type specifies the data type of the value that will be returned to the calling function.

b) Function definition:
A function definition describes how the function computes the value it returns. A function definition consists of a function header followed by a function body. The syntax of a function header followed by a function body.

The syntax of a function definition can be given as

```
return_type function_name(data_type variable1,data_type variable2...)
{
==========;
statements;
==========;
==========;
return(variable);
}
```

The body of the function enclosed in braces. 'C' allows the definition to be placed if the function is defined before its calling then its prototype declaration is optional. Parameter list is the list of formal parameters that the function receives. Return type is the datatype the function returns. In a function every statement should be terminated with semicolon (;)

c) Function call:

The function call statement invokes the function. When a function is invoked, the compiler jumps to the called function to execute the statements that are a part of the function. Once the called function is executed, the program control passes back to the calling function.

The function call statement has the following syntax:

```
function_name(variable1,variable2,....);
```

When the function declaration is present before the function call, the compiler can check if the correct number and type of arguments are used in the function call and the returned value, if any, is being used reasonably.

d) Function parameter:

Function parameters are may be classified under two groups, actual and formal arguments or parameters.
The parameters specified in the function call are known as actual parameters and those specified in the function declaration are known as formal parameters.

e) return statement:

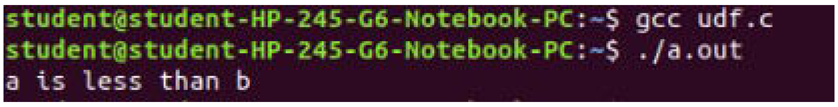
The return statement is used to terminate the execution of a function and returns control to the calling function.
A return statement may or may not return a value to the calling function. The syntax of return statement can be given as

```
return<expression>;
```

Here expression is placed in between angular brackets because specifying an expression is optional.

```c
//Example of user defined function
#include<stdio.h>
int check_relation(int a,int b);//function declaration
void main()
{
        int a=3,b=5,res;
        res=check_relation(a,b);//function call
        if(res==0)
                printf("\nEQUAL");
        if(res==1)
                printf("\n a is greater than b");
        if(res==-1)
                printf("\n a is less than b");
}
int check_relation(int a,int b)//function definition
{
        if(a==b)
                return 0;
        if(a>b)
                return 1;
        else if(a<b)
                return -1;
}
```

<u>Output:</u>



```
student@student-HP-245-G6-Notebook-PC:~$ gcc udf.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
a is less than b
```

<u>Actual arguments and Formal arguments:</u>

The arguments may be classified under two groups, actual and formal arguments.

a) <u>Actual arguments</u>:

An actual argument is a variable or an expression contained in a function call that replaces the formal parameter which is a part of the function declaration.

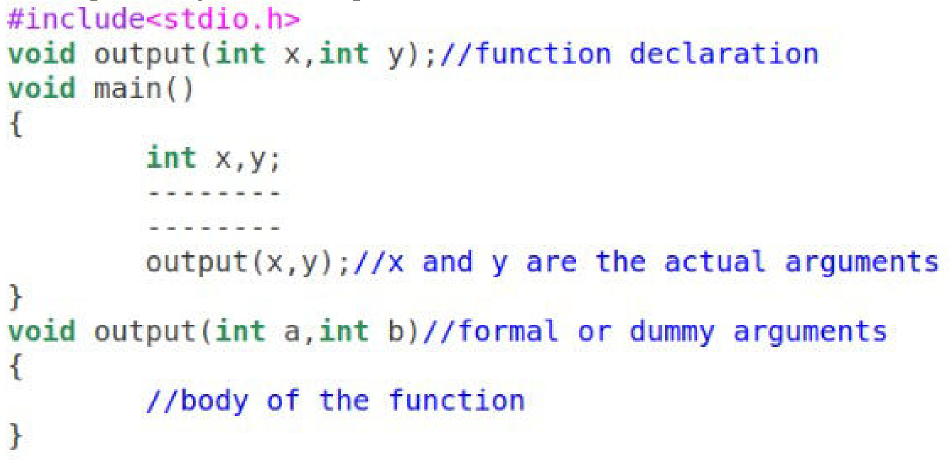
Sometimes, a function may be called by a portion of a program with some parameters and these parameters are known as the actual arguments.

For example:

```c
#include<stdio.h>
void output(int x,int y);//function declaration
void main()
{
        int x,y;
        --------
        --------
        output(x,y);//x and y are the actual arguments
}
```

b) <u>Formal arguments</u>:

Formal arguments are the parameters present in a function definition which may also be called as dummy arguments or the parametric variables. When the fuction is invoked, the formal parameters are replaced by the actual parameters.



```
#include<stdio.h>
void output(int x,int y);//function declaration
void main()
{
        int x,y;
        --------
        --------
        output(x,y);//x and y are the actual arguments
}
void output(int a,int b)//formal or dummy arguments
{
        //body of the function
}
```

PARAMETER PASSING TECHNIQUES IN C:

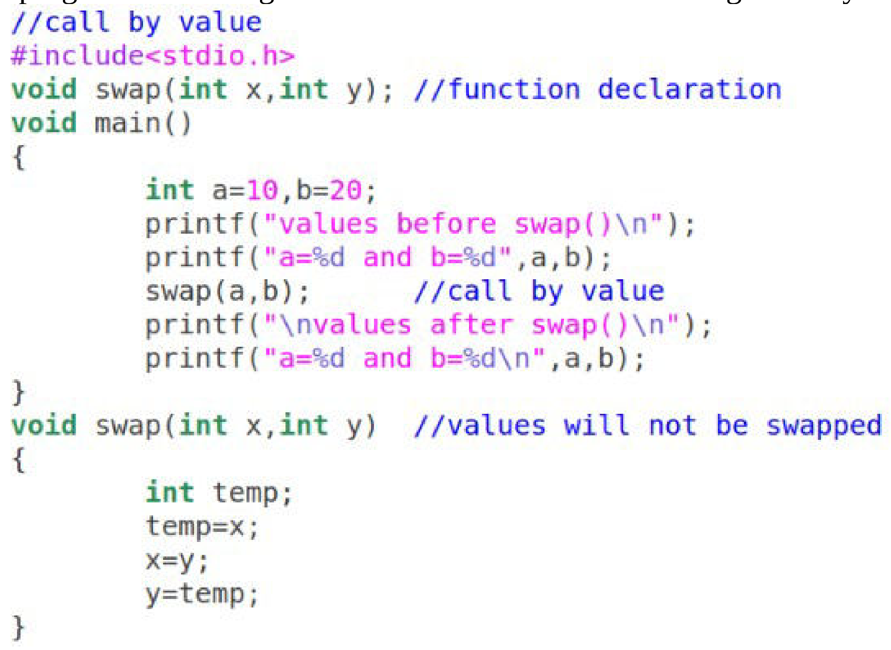
The parameter passing mechanism refers to the actions taken regarding the parameters when a function is invoked. There are two parameter passing techniques are available in C. They are

1) Call by value
2) Call by Reference

1) Call by value:

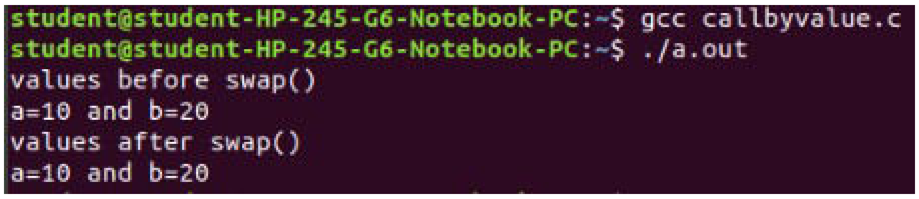
- Whenever a portion of the program invokes a function with formal arguments, control will be transferred from the main to the calling function and the value of the actual argument is copied to the function.
- Within the function, the actual value copied from the calling portion of the program may be altered (or) changed.
- When the control is transfered back from the function to the calling portion of the program, the altered values are not transfered back.
- This type of passing formal arguments to a function is technically known as "call by value".

Example: A program to exchange the contents of two variables using a call by value.



```
//call by value
#include<stdio.h>
void swap(int x,int y); //function declaration
void main()
{
        int a=10,b=20;
        printf("values before swap()\n");
        printf("a=%d and b=%d",a,b);
        swap(a,b);      //call by value
        printf("\nvalues after swap()\n");
        printf("a=%d and b=%d\n",a,b);
}
void swap(int x,int y)  //values will not be swapped
{
        int temp;
        temp=x;
        x=y;
        y=temp;
}
```

Output:



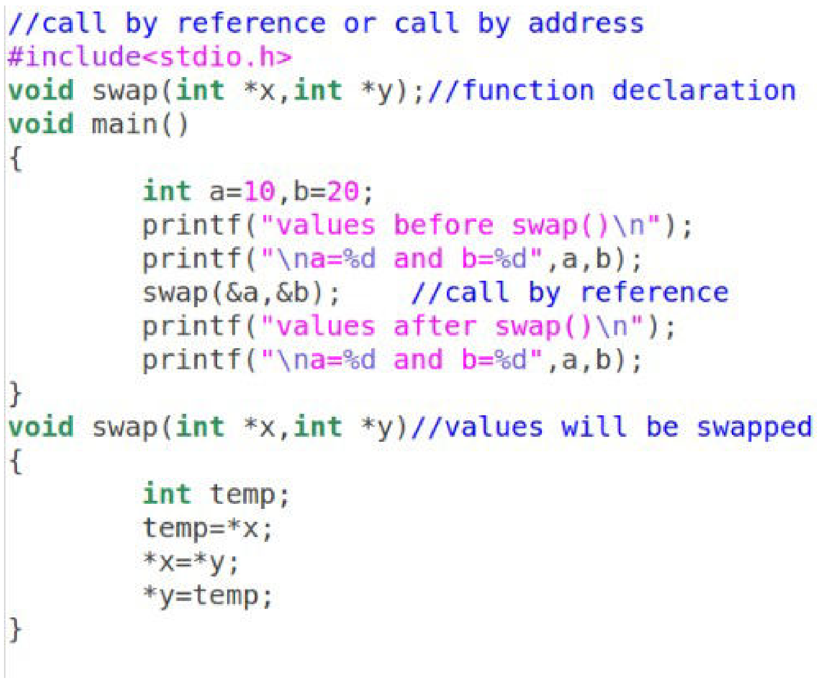
```
student@student-HP-245-G6-Notebook-PC:~$ gcc callbyvalue.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
values before swap()
a=10 and b=20
values after swap()
a=10 and b=20
```

2) Call by Reference:

- When a function is called by a portion of a program, the address of the actual arguments are copied onto the formal arguments, though they may be referred by different variable names.
- The content of the variables that are altered within the function block are returned to the calling portion of a program in the altered form itself, as the formal and the actual arguments are referencing the same memory location (or) address.
- This is technically known as call by reference (or) call by address.

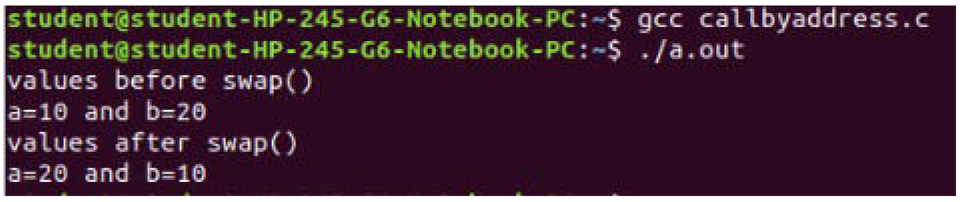
Example:

A program to exchange the contents of two variables using a call by reference



```
//call by reference or call by address
#include<stdio.h>
void swap(int *x,int *y);//function declaration
void main()
{
        int a=10,b=20;
        printf("values before swap()\n");
        printf("\na=%d and b=%d",a,b);
        swap(&a,&b);    //call by reference
        printf("values after swap()\n");
        printf("\na=%d and b=%d",a,b);
}
void swap(int *x,int *y)//values will be swapped
{
        int temp;
        temp=*x;
        *x=*y;
        *y=temp;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc callbyaddress.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
values before swap()
a=10 and b=20
values after swap()
a=20 and b=10
```

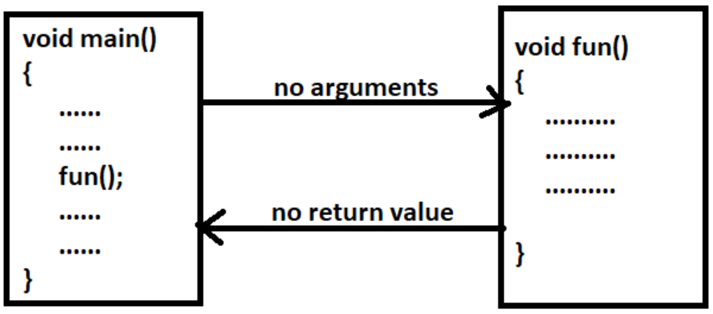
Category of functions:

A function, depending on whether arguments are present (or) not and whether a value is returned (or) not, may belong to one of the following categories.

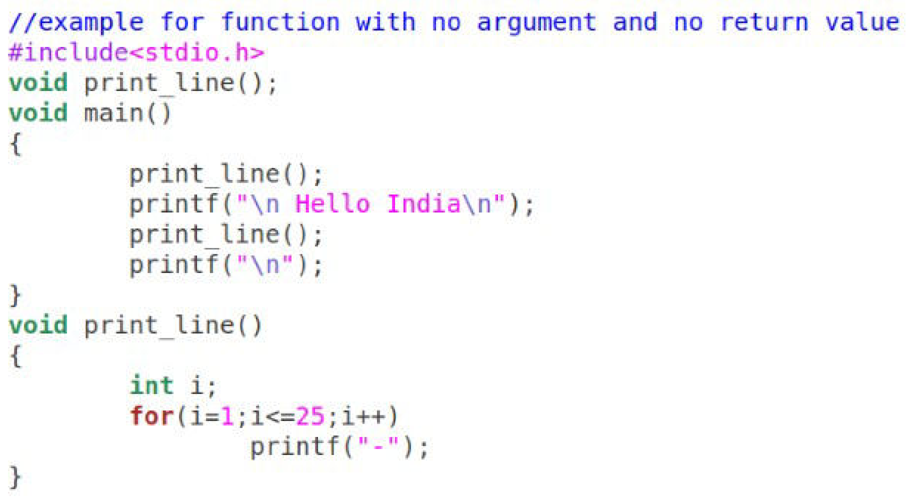
Category 1: Functions with no arguments and no return value.
Category 2: Functions with arguments and no return value.
Category 3: Functions with arguments and return value.
Category 4: Functions with no arguments and return value.

1. Functions with no arguments and no return value:

When a function has no arguments, it does not receive any data from the calling function. Similarly, when it does not return a value, the calling function does not receive any data from the called function. In effect, there is no data transfer between the calling function and the called function.

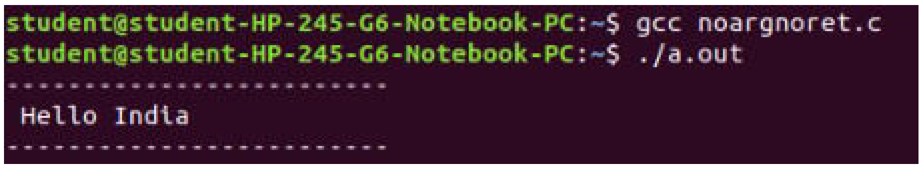


Example:



```
//example for function with no argument and no return value
#include<stdio.h>
void print_line();
void main()
{
        print_line();
        printf("\n Hello India\n");
        print_line();
        printf("\n");
}
void print_line()
{
        int i;
        for(i=1;i<=25;i++)
                printf("-");
}
```

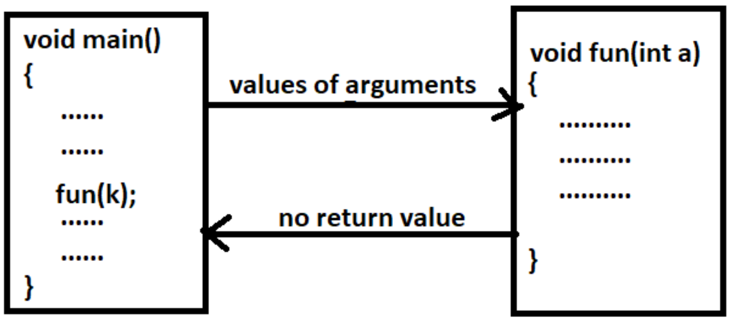
Output:



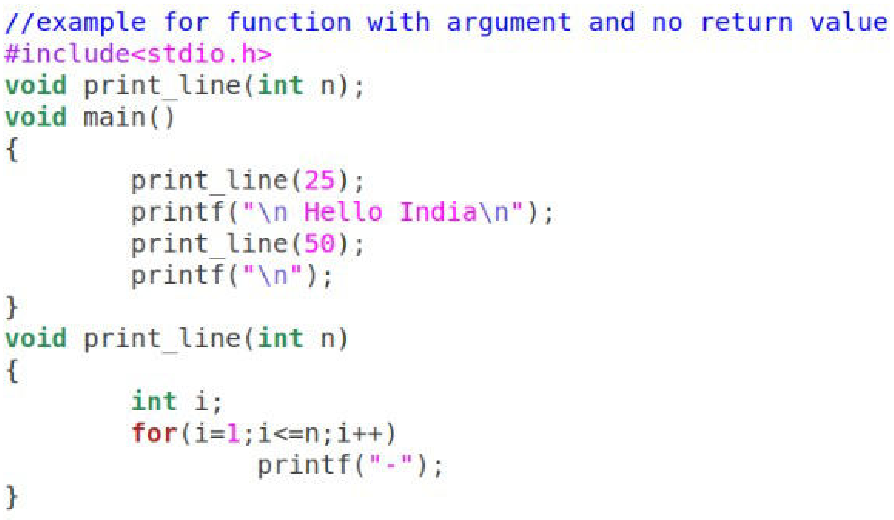
```
student@student-HP-245-G6-Notebook-PC:~$ gcc noargnoret.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
-------------------------
 Hello India
-------------------------
```

2. Functions with arguments and no return value:

In this category of the function, the main program or the calling program will send argument values but called program or the functions of program will not return any value.

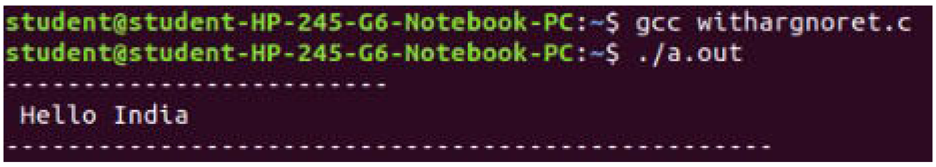


Example:



```
//example for function with argument and no return value
#include<stdio.h>
void print_line(int n);
void main()
{
        print_line(25);
        printf("\n Hello India\n");
        print_line(50);
        printf("\n");
}
void print_line(int n)
{
        int i;
        for(i=1;i<=n;i++)
                printf("-");
}
```

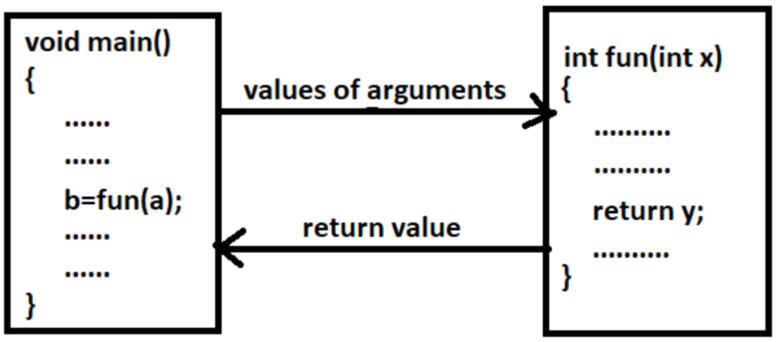
Output:



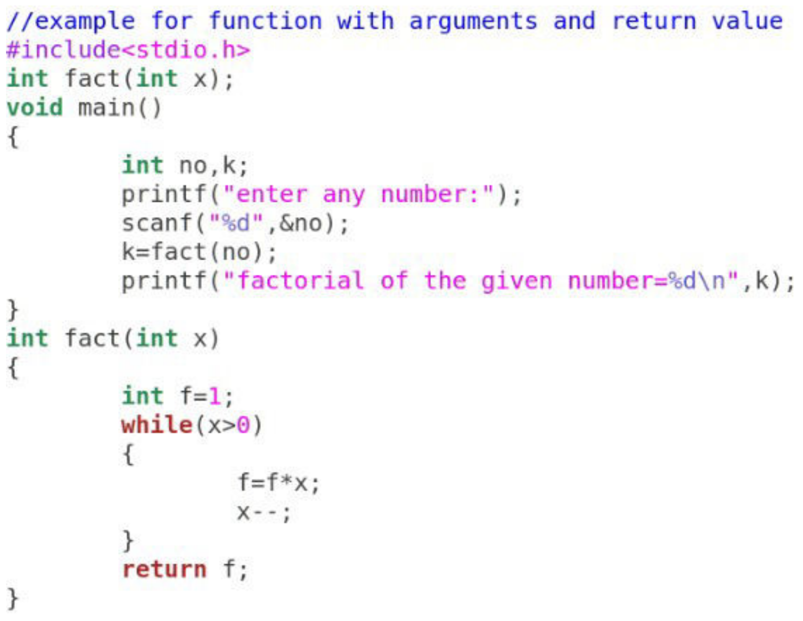
```
student@student-HP-245-G6-Notebook-PC:~$ gcc withargnoret.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
-------------------------
 Hello India
--------------------------------------------------
```

3. Functions with arguments and return value:

In this category of the function, the main program or the calling program will return argument to the called program or the functions sub-program will send back the value.

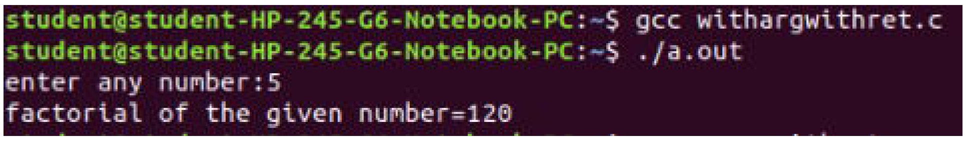


Example:



```
//example for function with arguments and return value
#include<stdio.h>
int fact(int x);
void main()
{
        int no,k;
        printf("enter any number:");
        scanf("%d",&no);
        k=fact(no);
        printf("factorial of the given number=%d\n",k);
}
int fact(int x)
{
        int f=1;
        while(x>0)
        {
                f=f*x;
                x--;
        }
        return f;
}
```

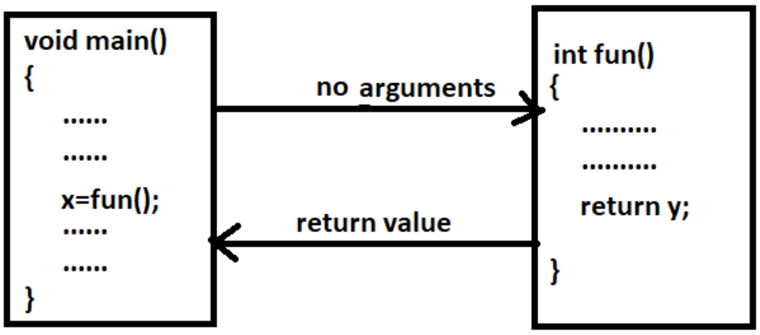
Output:



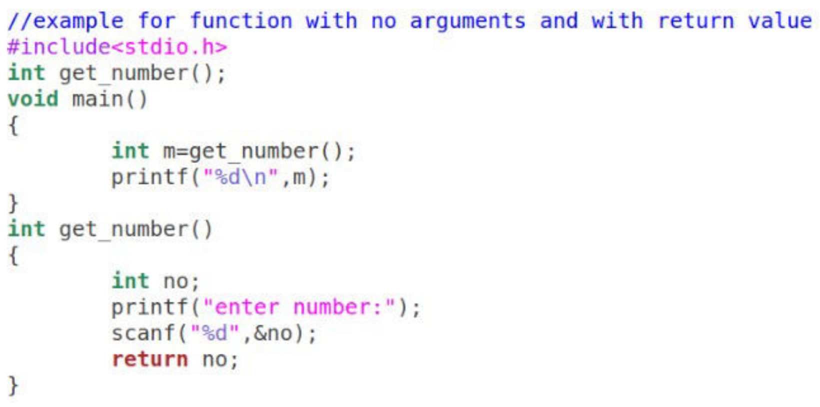
```
student@student-HP-245-G6-Notebook-PC:~$ gcc withargwithret.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
enter any number:5
factorial of the given number=120
```

4. Function with no arguments and return value:

When a function has no arguments it does not receive the data from the calling function and also function sub-program will return any value to the main program.

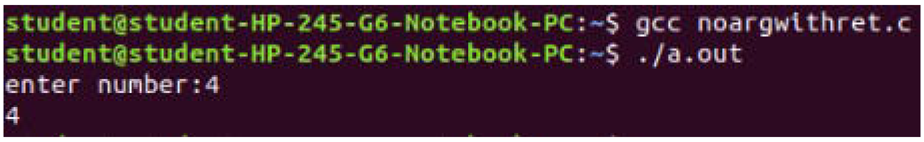


Example:



```
//example for function with no arguments and with return value
#include<stdio.h>
int get_number();
void main()
{
        int m=get_number();
        printf("%d\n",m);
}
int get_number()
{
        int no;
        printf("enter number:");
        scanf("%d",&no);
        return no;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc noargwithret.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
enter number:4
4
```

TYPES OF VARIABLES

IMPORTANT POINTS:

C has ***five* (5) kinds of variables** - divided into ***2 categories***:

- **"*Long term*" variables**: (also known as: ***compile-time* (allocated) variables**)

  1. **Global variables** --- accessible everywhere
  2. ***static* global variables** --- accessible within the ***same* C program file**
  3. ***static* local variables** --- accessible within the ***same* C function**

  Common property:

  - The **memory space** used for these **kinds** of **variables** are **allocated (reserved)** using **language constructs.**

- ***Short term* variables**: (also known as: ***run-time* variables**)

  1. **Local variables**
  2. **Parameter variables**

  Common property:

  - The **memory space** used for these **kinds** of **variables** are **allocated (reserved)** ***during*** the **execution (running)** of the program

  - They are **created** on the **system stack**

Local variables:

- These are also called normal local variables.
- These are declared in functions.
- These variables retains its values in between the function execution time. (i.e.) Memory is allocated when the function starts executing and released when the function returns.
- These are given in between the curly{} braces of any function.
- These variables are lost once they are used in a function. These are needed no longer.

For example:

```
void funct(int i,int j)
{
        int k,m;//local variables
        ---
        ---
        //body of the function
}
```

The integer variables are defined within a function block of the funct(). Local variables are referred only the particular part of a block (or) a function.

Global variables:

- Global variables are variables defined outside the main function block.
- These variables are referred by the same data type and by the same name through out the program in both the calling portion of a program and in the function block.

Example:

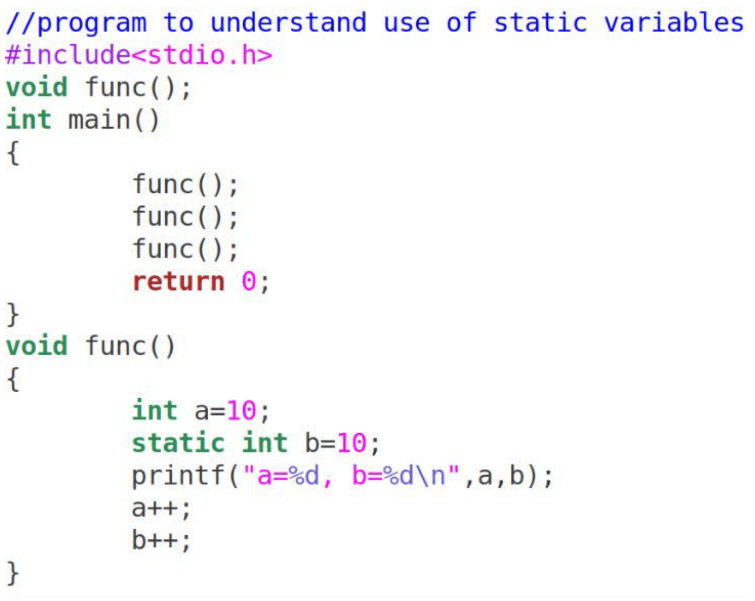
```
int x;//global declarations
void function1();
void main()
{
        x=10;
        ---
        ---
        function1();
}
function1()
{
        int sum;
        sum=x++;
        ---
        ---
}
```

static variable

static variables are declared by writing keyword **static** in front of the declaration. If a static variable is not initialized then it is automatically initialized to 0.

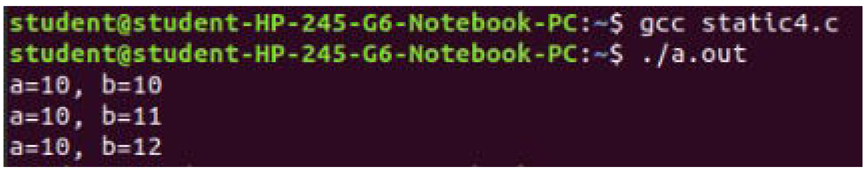
static datatype var_name;

A static variable is initialized only once and its value is retained between function calls.
Program:



```c
//program to understand use of static variables
#include<stdio.h>
void func();
int main()
{
        func();
        func();
        func();
        return 0;
}
void func()
{
        int a=10;
        static int b=10;
        printf("a=%d, b=%d\n",a,b);
        a++;
        b++;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc static4.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
a=10, b=10
a=10, b=11
a=10, b=12
```

Here b is a static variable. First time when the function is called b is initialized to 10. Inside the function, value of b becomes 11. This value is retained and when next time the function is called, value of b is 11 and the initialization is neglected. Similarly when third time function is called, value of b is 12. Note that the variable a, which is not static is initialized on each call and its value is not retained.

SCOPE OF VARIABLES

In C, all constants and variables have a defined scope. By scope we mean the accessibility and visibility of the variables at different points in the program. A variable or a constant in C has four types of scope:

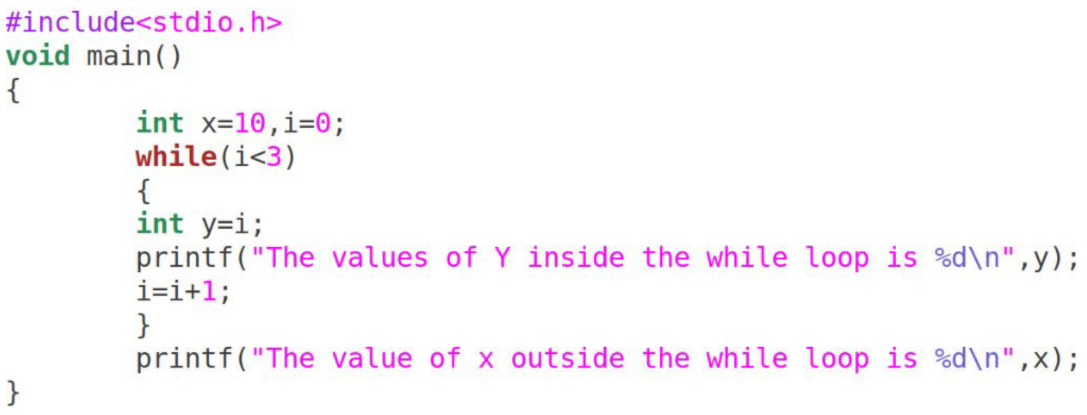
1. Block scope
2. Function scope
3. Program scope
4. File scope

1. Block scope:

We know that a statement block is a group of statements enclosed within opening and closing braces{}. If a variable is declared within a statement block then, as soon as the control exists that block, the variable will cease to exist. Such as variable, also known as a local variable, is said to have a *block scope.*

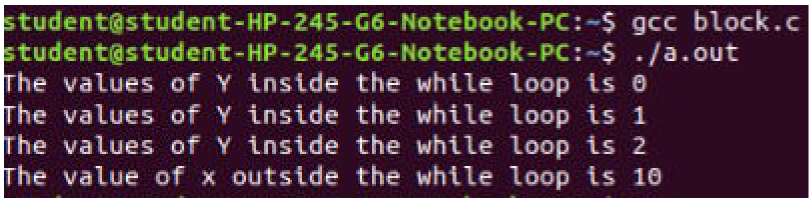
For example, if we declare an integer x inside a function, then that variable is unknown to the rest of the program(i.e., outside that function).

Example:



```
#include<stdio.h>
void main()
{
        int x=10,i=0;
        while(i<3)
        {
        int y=i;
        printf("The values of Y inside the while loop is %d\n",y);
        i=i+1;
        }
        printf("The value of x outside the while loop is %d\n",x);
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc block.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
The values of Y inside the while loop is 0
The values of Y inside the while loop is 1
The values of Y inside the while loop is 2
The value of x outside the while loop is 10
```

2. Function scope:

Function scope indicates that a variable is active and visible from the beginning to the end of a function. In C, only the goto label has function scope. In other words function scope is applicable only with goto label names. This means that the programmer cannot have the same label name inside a function.

```
void main()
{
        ...
        loop://a goto label has function scope
        ...
        goto loop;//the goto statement
}
```

3. Program scope:

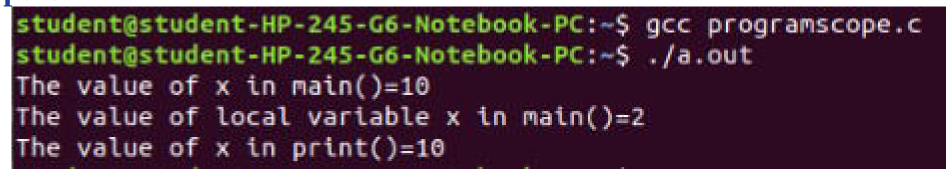
Local variables(also known as internal variables) are automatically created when they are declared in the function and are usable only within that function. The local variables are unknown to other functions in the program. Such variables cease to exist after the last statement in the function in which they are declared and re-created each time the function is called.

However, if we want this function to access some variables which are not passed to them as arguments, then declare those variables outside any function blocks. Such variables are commonly known as *global variables* and can be accessed from any point in the program.

Example:

```
#include<stdio.h>
int x=10;
void print();
void main()
{
        printf("The value of x in main()=%d\n",x);
        int x=2;
        printf("The value of local variable x in main()=%d\n",x);
        print();
}
void print()
{
        printf("The value of x in print()=%d\n",x);
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc programscope.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
The value of x in main()=10
The value of local variable x in main()=2
The value of x in print()=10
```

4. File scope:

When a global variable is accessible until the end of the file, the variable is said to have file scope. To allow a variable to have file scope, declare that variable with the static keyword before specifying its data type,as shown:

```
static int x=10;
```

A global static variable can be used anywhere from the file in which it is declared but it is not accessible by any other files. Such variables are useful when the programmer writes his/her own header files.

Storage classes in C

In addition to data type, each variable has one more attribute known as storage class. The proper use of storage classes makes our program efficient and fast. In larger multifile programs, the knowledge of storage classes is indispensable. We can specify a storage class while declaring a variable.

storage_class datatype variable_name;

There are four types of storage classes:

1. Automatic
2. register
3. External
4. static

So we may write declaration statements like this-

```
auto int x,y;
static float d;
register int z;
```

When the storage class specifier is not present in the declaration, compiler assumes a default storage class based on the place of declaration.

A storage class decides about these four aspects of a variable-

(1) **Lifetime** - Time between the creation and destruction of a variable.
(2) **Scope** - Locations where the variable is available for use.
(3) **Initial value** - Default value taken by an uninitialized variable.
(4) **Place of storag**e - Place in memory where the storage is allocated for the variable

1. Automatic variables:

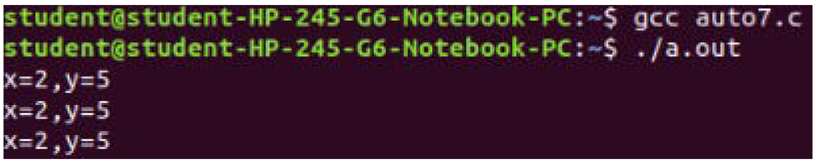
- All the variables declared inside a block/ function without any storage class specifier are called automatic variables.
- We may also use the keyword auto to declare automatic variables, although this is generally notbdone.
- The uninitialized automatic variables initially contain garbage value.
- The scope of these varaibles is inside the function or block in which they are declared and they cant be used in any other function/block.
- They are named automatic since storage for them is reserved automatically each time when the control enters the function/block and are released automatically when the function/block terminates.
- The features of a variable defined to have on automatic storage class are

Storage	Primary memory of computer
Default initial value	Garbage value
Scope	Local to the block in which the variable is defined
Life time	Till the control remains within the block in which the variable is defined

Example 1:

```c
//program to understand automatic variables
#include<stdio.h>
void func();
void main()
{
        func();
        func();
        func();
}
void func()
{
        int x=2,y=5;
        printf("x=%d,y=%d\n",x,y);
        x++;
        y++;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc auto7.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
x=2,y=5
x=2,y=5
x=2,y=5
```

Here when the function func() is called first time, the variables x and y are created and initialized, and when the control returns to main(), these variables are destroyed.

When the function func() is called for the second time, again these variables are created and initialized, and are destroyed after execution of the function.

So automatic variables come into existence each time the function is executed and are destroyed when the function terminates.

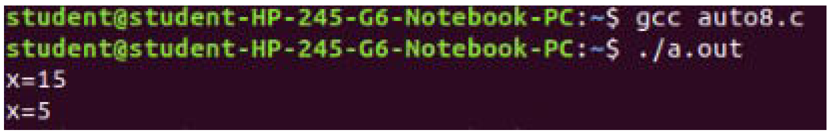
Since automatic variables are known inside a function or block only, so we can have variables of same name in different functions or blocks without any conflict.

For example in the following program the variable x is used in different blocks (here blocks consist of function body) without any conflict.

Example 2:

```c
//program to understand automatic variables
#include<stdio.h>
int func();
void main()
{
        int x=5;
        printf("x=%d\n",func());
}
int func()
{
        int x=15;
        printf("x=%d\n",x);
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc auto8.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
x=15
x=5
```

Here the variable x declared inside main() is different from the variable x declared inside the function func().

2. Register variables:

- Register storage class can be applied only to local variables.
- The scope, lifetime and initial value of register variables are same as that of automatic variables.The only difference between the two is in the **place where they are stored.**
- Automatic variables are stored in **memory** while register variables are stored in **CPU registers**.
- Registers are small storage units present in the processor.
- The **variables stored in registers can be accessed much faster** than the variables stored in memory. So the variables that are frequently used can be assigned register storage class for **faster processing**.
- For example the variables used as loop counters may be declared as register variables since they are frequently used.
- Register storage class declaration is applicable only to **int**,**char**, and **pointer** data type.

Ex:

register int a;

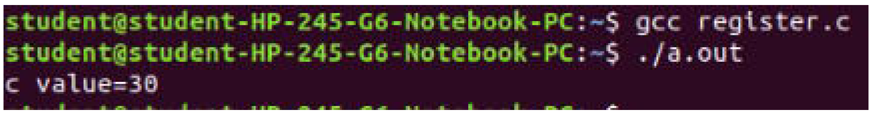
The feature of a variable define to have a register storage class

Storage:	CPU register
Default register value	Garbage value
Scope	Local to the block in which the variable is defined
Lifetime	Till the control remains within the block in which the variable is defined

Example 1:

```c
#include<stdio.h>
void main()
{
        register int a,b,c;
        a=10;
        b=20;
        c=a+b;
        printf("c value=%d\n",c);
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc register.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
c value=30
```

Example 2:

```
//sample program for register storage class
#include<stdio.h>
void main()
{
        register int k;
        for(k=1;k<=15;k++)
        {
                printf("%d\t",k);
        }
}
```

Output:

```
gcc register1.c
./a.out
1      2      3      4      5      6      7      8      9      10      11      12
13     14     15
```

- register variables don't have **memory addresses**. So we can't apply **address operator(&)** to them.
- There are limited number of registers in the processor, hence we can declare only few **variables as register**.
- If many variables are declared as register and the CPU registers are not available then compiler will **treat them as automatic variables**.
- The **register** storage class specifier **can be applied to formal arguments** of a function while the other three storage class specifiers can't be used in this way.

3. External variables:

- The variables that have to be **used by many functions** and different files can be declared as external variables.
- The **initial value** of an uninitialized external variable is **0**.
- The declaration of an external variable declares the type and name of the variable, while the **definition reserves storage for the variable** as well as behaves as a declaration.
- The keyword **extern** is **specified in declaration** but **not in definition.**
- For example the definition of an external variable salary will be written as-
  float salary;
- Its declaration will be written as-
  extern float salary;

Definition of an external variable:

(a) Definition creates the variable, so **memory is allocated at the time of definition.**
(b) There can be **only one definition**.
(c) The variable can be initialized with the definition and initializer should be constant.
(d) The keyword **extern** is **not specified in the definition.**
(e) The definition can be **written only outside functions.**

Declaration of an external variable:

(a) **The declaration does not create the variable**, it only **refers to a variable that has already been created somewhere**, so memory is not allocated at the time of declaration.

(b) There can be many declarations.
(c) The variable **cannot be initialized at the time of declaration**.
(d) The keyword **extern is always specified in the declaration.**
(e) The declaration can be **placed inside functions also**.

The features of the variable defined to have an external storage classes are

Storage:	Primary memory
Default initial value	0
Scope	global
Life time	As long as the program execution does not come to an end

Consider this program

```
#include<stdio.h>
int x=8;
void main()
{
        .....
}
void func1()
{
        .....
}
void func2()
{
        .....
}
```

In this program the variable x will be available to all the functions, since an external variable is active from the point of its definition till the end of a program.

Till now we had written our program in a single file. When the program is large, it is written in different files and these files are compiled seperately and linked together afterwards to form an executable program. Now we'll consider a multifile program, which is written in 3 files viz. first.c, second.c and third.c

First.c	Second.c	Third.c
`int x=8;`	`extern int x;`	`void func4()`
`void main()`	`void func2()`	`{`
`{`	`{`	`        ....`
`        ....`	`        ....`	`}`
`}`	`}`	`void func5()`
`void func1()`	`void func3()`	`{`
`{`	`{`	`        ....`
`        ....`	`        ....`	`}`
`}`	`}`	

Here in the **first.c** , **an external variable x is defined and initialized**. This variable can be used both in main() and func1() and it can be accessible to other files.

In the **file second.c, to access this variable** then we used the **declaration** in this file as

extern int x;

Suppose our program consists of many files and in file **first.c**, we have defined many variables that may be needed by other files also.

We can put an **extern** declaration for each variable in every file that needs it.

Another better and practical approach is to collect all extern declarations in a header file and include that header file in the files, which require access to those variables.

4. static variables:

As the name suggests, the value of static variables will be retained until the end of the program. The features of a variable defined to have a static storage class are

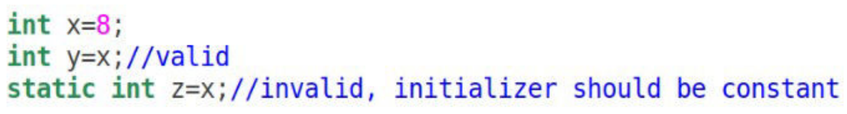
Storage	memory
Default initial value	0
Scope	Local to the block in which the variable defined
Life time	Until program ends

A static variable may be any one of the two types.
a) local static variables b) global static variables

a) Local static variables:

- The scope of a local static variable is same as that of an automatic variable i.e, it can be used only **inside the function** or block in which it is defined.
- The lifetime of a static variable is more than that of an automatic varaible.
- A static variable is created at the compilation time and it remains alive till the end of a program.
- It is not created and destroyed each time the control enters a “function/block”.
- Hence a static variable is created only once and its value is retained between function calls.
- If it has been initialized, then the initialization value is placed in it only **once at the time of creation**. **It is not initialized each time the function is called.**

If a static variable is not explicitly initialized then by default it takes initial value zero.

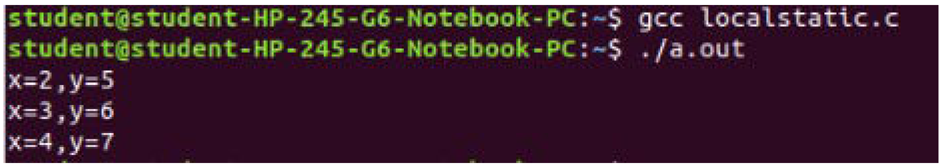


```
int x=8;
int y=x;//valid
static int z=x;//invalid, initializer should be constant
```

Example:

```c
// program to understand the use of local static variable
#include<stdio.h>
void func();
void main()
{
        func();
        func();
        func();
}
void func()
{
        static int x=2,y=5;
        printf("x=%d,y=%d\n",x,y);
        x++,y++;
}
```

Output:



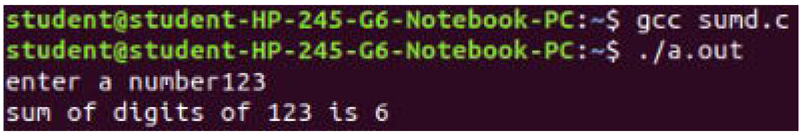
```
student@student-HP-245-G6-Notebook-PC:~$ gcc localstatic.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
x=2,y=5
x=3,y=6
x=4,y=7
```

Note:
The effect of initialization is seen only in the first call. In subsequent calls initialization is not performed and variables x and y contain values left by the previous function call.

The next program uses a recursive function to find out the sum of digits of a number. The variable sum taken inside function sumd() should be taken as static.

```c
// program to find out the sum of digits of a number recursively
#include<stdio.h>
int sumd(int num);
void main()
{
        int num;
        printf("enter a number");
        scanf("%d",&num);
        printf("sum of digits of %d is %d\n",num,sumd(num));
}
int sumd(int num)
{
        static int sum=0;
        if(num>0)
        {
                sum=sum+(num%10);
                sumd(num/10);
        }
        else
                return sum;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc sumd.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
enter a number123
sum of digits of 123 is 6
```

b) global static variable:

- It is declared outside of all functions and is available to all the functions in that program.
- If a local variable is declared as static then it remains same throughout the program.
- In case of global varaibles, the static specifier is not used to extend the lifetime since global variables have already a lifetime equal to the life of program.
- Here the static specifier is used for information hiding.
- If an external variable is defined as static, then it cant be used by other files of the program. We can make an external variable private to a file by making it static.

First.c

```
int x=8;
static int y=10;
void main()
{
        ....
}
void func1()
{
        ....
}
```

Second.c

```
extern int x;
void func2()
{
        ....
}
void func3()
{
        ....
}
```

Third.c

```
void func4()
{
        ....
}
void func5()
{
        ....
}
```

Here the variable y is defined as a static external variable, so it can be used only in the file **first.c** We cant use it in other files by putting extern declaration for it.

Storage classes in Functions:

- The storage class specifiers extern and static can be used with function definitions.
- The definition of a function without any storage specifier is equivalent to its definition with keyword extern i.e. by default the definition of a function is considered external.
- If a function is external then it can be used by all the files of the program and if it is static then it can be used only in the file where it is defined.

First.c

```
void main()
{
        .....
}
float func1(int)
{
        .....
}
```

Second.c

```
extern float func1(int);
void func2()
{
        .....
}
static int func3(int)
{
        .....
}
```

Third.c

```
void func4()
{
        ....
}
void func5()
{
        ....
}
```

- Here the function func1() is defined in file **first.c** .
- Its declaration is put in file **second.c,** so it can be used in this file also.
- The function func3() in file **second.c** is defined as static so it can't be used by any other file.
- Generally declarations of all functions are collected in a header file and that header file is included in other files.

Linkage:

There are 3 types of linkages in C-

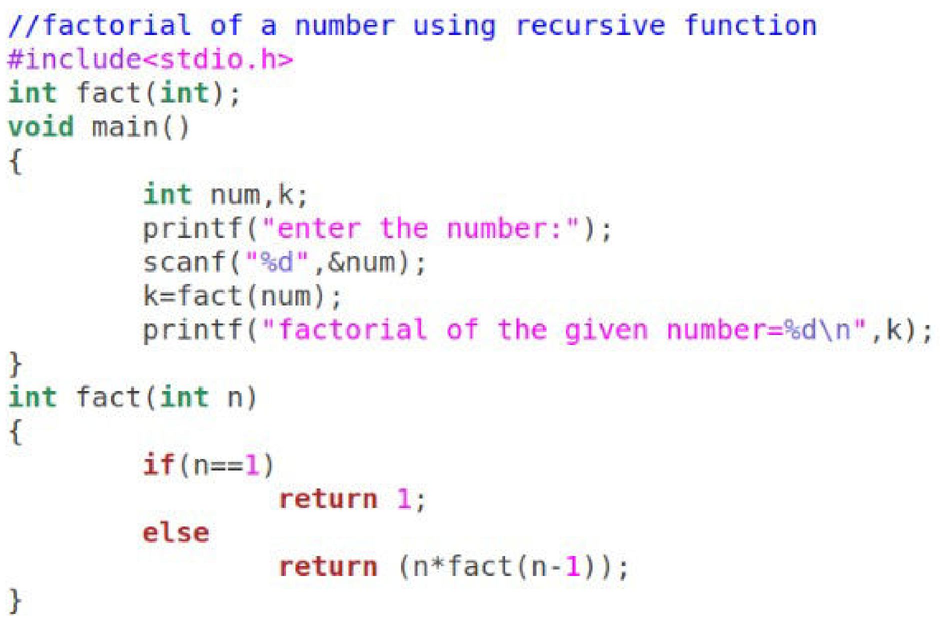
(a) External linkage
(b) Internal linkage
(c) No linkage

- Local variables have no linkage, so their scope is only within the block where they are declared.
- Global variables and functions have external linkage, so they can be used in any file of the program.
- static global variables and static functions have internal linkage, so their scope is only in the file where they are declared.

RECURSION

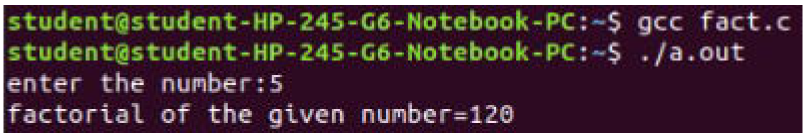
- **Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.**
- The process is used for repetitive computations in which each action is stated in terms of a previous result.
- Many iterative (i.e, repetitive) problems can be written in this form.
- In order to solve a problem recursively two conditions must be satisfied.
- First, the problem must be written in a recursive form, and second, the problem statement must include a **stopping condition**.
- When writing recursive functions, we must have a conditional statement somewhere in the function to force the function to return without the recursive call being executed otherwise the function will never return and program goes into infinite loop.

  Example:

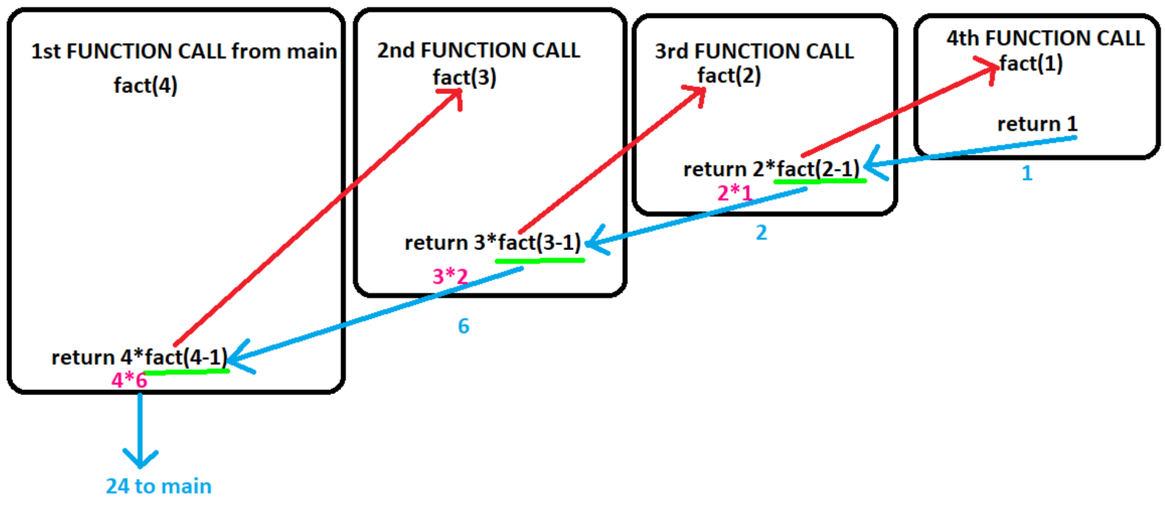


```
//factorial of a number using recursive function
#include<stdio.h>
int fact(int);
void main()
{
        int num,k;
        printf("enter the number:");
        scanf("%d",&num);
        k=fact(num);
        printf("factorial of the given number=%d\n",k);
}
int fact(int n)
{
        if(n==1)
                return 1;
        else
                return (n*fact(n-1));
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc fact.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
enter the number:5
factorial of the given number=120
```



Types of Recursion:

Any recursive function can be characterized based on the following:

1. Whether the function calls itself directly or indirectly(*direct or indirect recursion*).
2. Whether any operation is pending at each recursive call(*tail-recursive* or not).
3. The structure of the calling pattern(*linear or tree recursive*).

1. Direct recursion:

A function is said to be *directly* recursive if it explicitly calls itself.

For example:

```
int func(int n)
{
        if(n==0)
                return n;
        return(func(n-1));
}
```

Here, the function func() calls itself for all positive values of n, so it is said to be a directly recursive function.

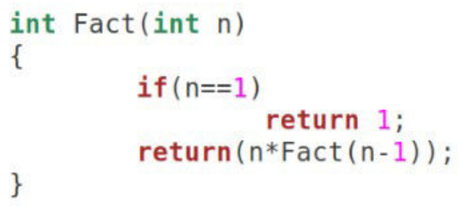
2. Indirect recursion:

A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it. Look at the functions given below. These two functions are indirectly recursive as they both call each other.

```
int Func1(int n)
{
        if(n==0)
                return n;
        return Func2(n);
}
int Func2(int x)
{
        return Func1(x-1);
}
```

3. Tail Recursion:

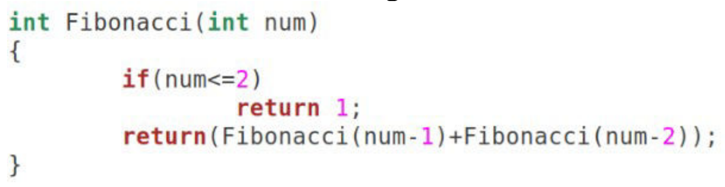
A recursive function is said to be *tail recursive*, if no operations are pending to be performed when the recursive function returns to its caller. When the called function returns, the returned value is immediately returned from the calling function. Tail recursive functions are highly desirable because they are much more efficient to use as the amount of information that has to be stored on the system stack is independent of the number of recursive calls.



```
int Fact(int n)
{
        if(n==1)
                return 1;
        return(n*Fact(n-1));
}
```

4. Linear and Tree recursion:

Recursive functions can also be characterised depending on the way in which the recursion grows in a linear fashion or forming a tree structure.



```
int Fibonacci(int num)
{
        if(num<=2)
                return 1;
        return(Fibonacci(num-1)+Fibonacci(num-2));
}
```

Difference between Recursion and Iteration:

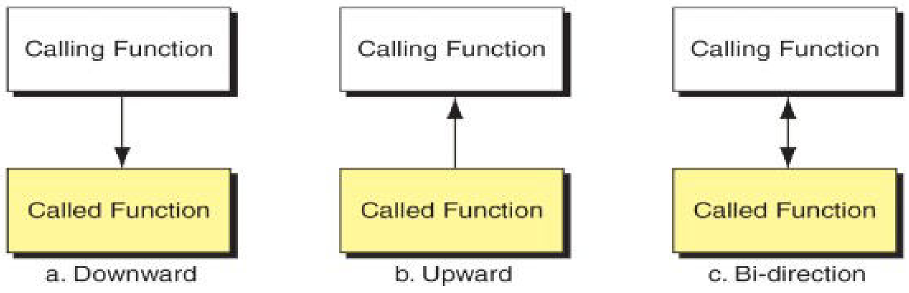
RECURSION	ITERATIONS
1. Recursive function is a function that is partially defined by itself.	1. Iterative instructions are loop based repetitions of a process.
2. Recursion uses **selection** structure	2. Iteration uses **repetition** structure
3. Infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on some condition.(base case)	3. An infinite loop occurs with iteration if the loop condition test never becomes false.
4. Recursion terminates when a base case is recognized	4. Iteration terminates when the loop-condition fails
5. Recursion is usually **slower** than iteration due to overhead of maintaining stack.	5. Iteration does not use stack so it's **faster** than recursion
6. Recursion uses **more** memory than iteration.	6. Iteration consume **less** memory
7. Infinite recursion can crash the system.	7. Infinite looping uses CPU cycles repeatedly.
8. Recursion makes code **smaller.**	8. Iteration makes code **longer**

INTER-FUNCTION COMMUNICATION

- A **function** is a self-contained block or sub program that perform a special task when it is called.
- Whenever a function is called to perform a specific task, then the called function performs that task and the result is returns back to the calling function.
- The data flow between the calling and called functions to perform a specific task is known as **inter function communication**.

Different methods for transferring data between calling and called function.

The data flow between the calling and called functions can be divided into three strategies:
(i) Downward flow
(ii) Upward flow
(iii) Bi-directional flow.



(i) Downward flow:-

- The calling function sends data to the called function is represented as downward flow.
- No data flows in opposite directions.
- It is only one way communication.
- The data items are passed from calling to called function and called function may change the values, but the original values in calling function remains unchanged.
- This is also known as call by value mechanisnm in 'C' language.

(ii) Upward flow:-

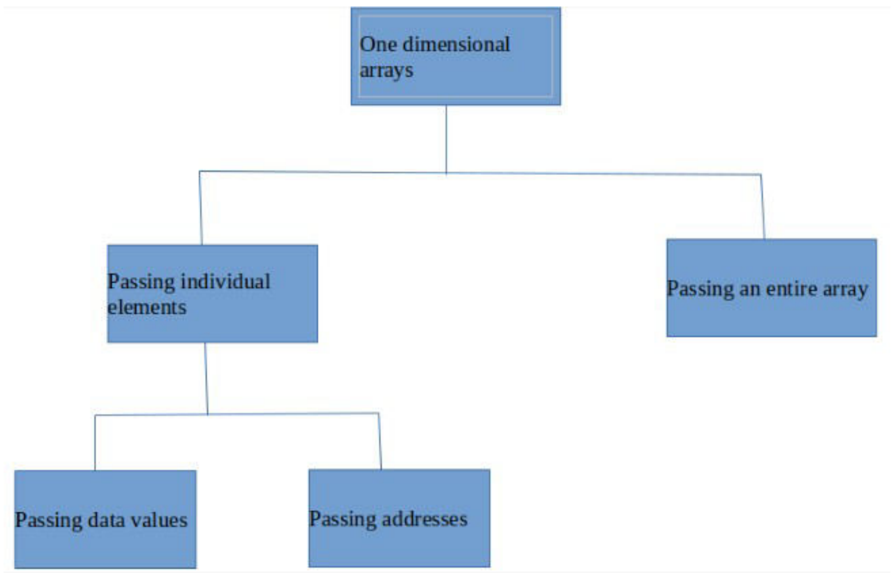
- The called function sends data to the calling function is represented as upward flow.
- Here the called function does not receive any data prior from calling function.
- It is also one way communication only.
- Since the data items are passed from only called to calling function, so that it may read data from keyboard and then passed it to calling function.

(iii) Bidirectional flow:-

- The calling function sends data to the called function, after performing task the called function sends back the result to called function is represented as bidirectional flow. Here the data items are passed in two ways.
- This is also known as call by value mechanisnm in 'C' language.

One dimensional array for inter-function communication:

- Similar to variables of other data types, we can also pass an **array** to a function.
- While in some situations, we may want to pass individual elements of the array, and in other situations we may want to pass the entire array.



1. PASSING INDIVIDUAL ELEMENTS:

The individual elements of an array can be passed to a function either by passing their addresses or their data values.

i) Passing data values:

The individual elements can be passed in the same manner as we pass variables of any other data type. The condition is just that the data type of the array element must match the type of the function parameter.

Program:

```
//(1)passing individual elements (i)passing data values
#include<stdio.h>
void func(int num);
void main()
{
        int arr[5]={1,2,3,4,5};
        func(arr[3]);
}
void func(int num)
{
        printf("%d\n",num);
}
```

Output:

```
4
```

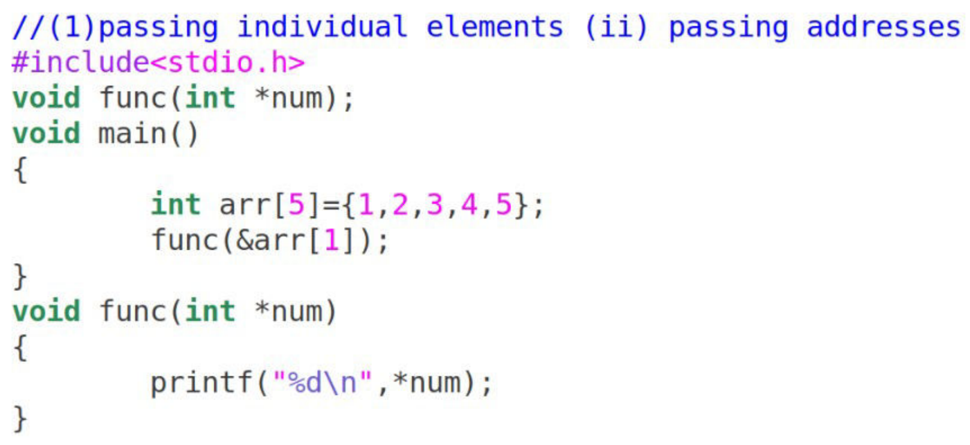
In this example, only one element of the array is passed to the called function. This is done by using the index expression.. So arr[3] actually evaluates to a single integer value.

ii) Passing addresses:

Similar to ordinary variables, we can pass the address of an individual array element by preceding the **address operator(&)** to the element's indexed reference. Therefore to pass the address of the fourth element of the array to the called function , we will write **&arr[3]**.

However, in the called function, the value of the array element must be accessed using the **indirection(*) operator.**

Program:



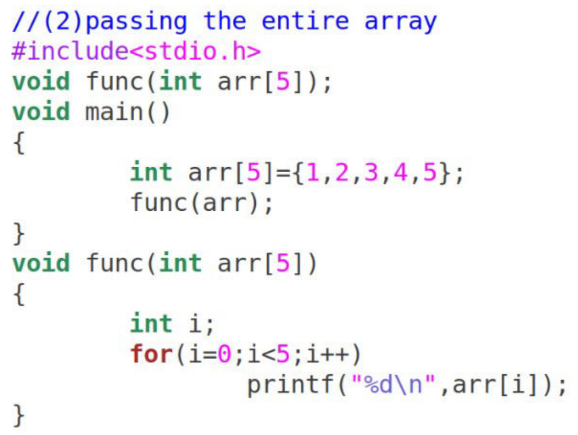
```
//(1)passing individual elements (ii) passing addresses
#include<stdio.h>
void func(int *num);
void main()
{
        int arr[5]={1,2,3,4,5};
        func(&arr[1]);
}
void func(int *num)
{
        printf("%d\n",*num);
}
```

Output:
2

(2) PASSING THE ENTIRE ARRAY:

In C, **the array name refers to the first byte of the array in memory.** When we need to pass an entire array to a function, we can simply pass the name of the array.

The program illustrates the code which passes the entire array to the called function.



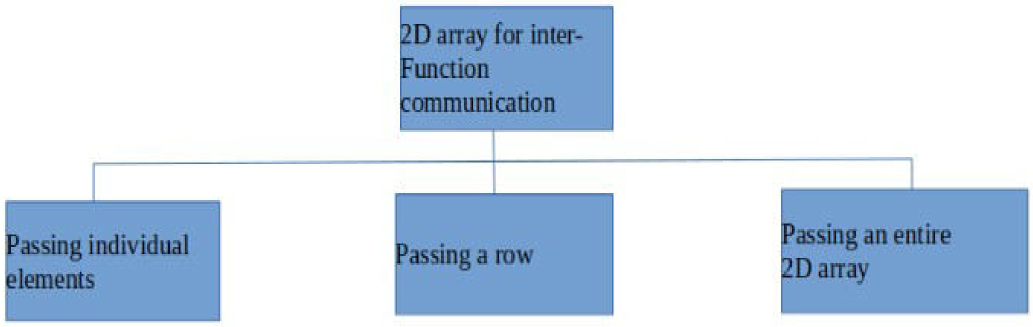
```
//(2)passing the entire array
#include<stdio.h>
void func(int arr[5]);
void main()
{
        int arr[5]={1,2,3,4,5};
        func(arr);
}
void func(int arr[5])
{
        int i;
        for(i=0;i<5;i++)
                printf("%d\n",arr[i]);
}
```

Output:
1
2
3
4
5

Two dimensional array for inter function communication:

There are 3 ways of passing parts of the 2D array to a function.

- First we can pass individual elements of the array.
- Second, we can pass a single row of the 2D array.
- Third, we can pass the entire 2D array to the function.


2D array for inter-Function communication
Passing individual elements
Passing a row
Passing an entire 2D array

1. PASSING INDIVIDUAL ELEMENTS:

The individual elements can be passed in the same manner as we passing the elements of a 1D array.

```
//(1)passing individual elements in 2D array
#include<stdio.h>
void func(int num);
void main()
{
        int a[3][3]={{1,2,3},
                     {4,5,6},
                     {7,8,9}};
        func(a[2][0]);
}
void func(int num)
{
        printf("%d\n",num);
}
```

Output:

```
7
```

(2) PASSING A ROW:

A row of 2D array can be accessed by indexing the array name with the row number. When we send a single row of a two-dimensional array, then the called function receives a one-dimensional array.

Program:

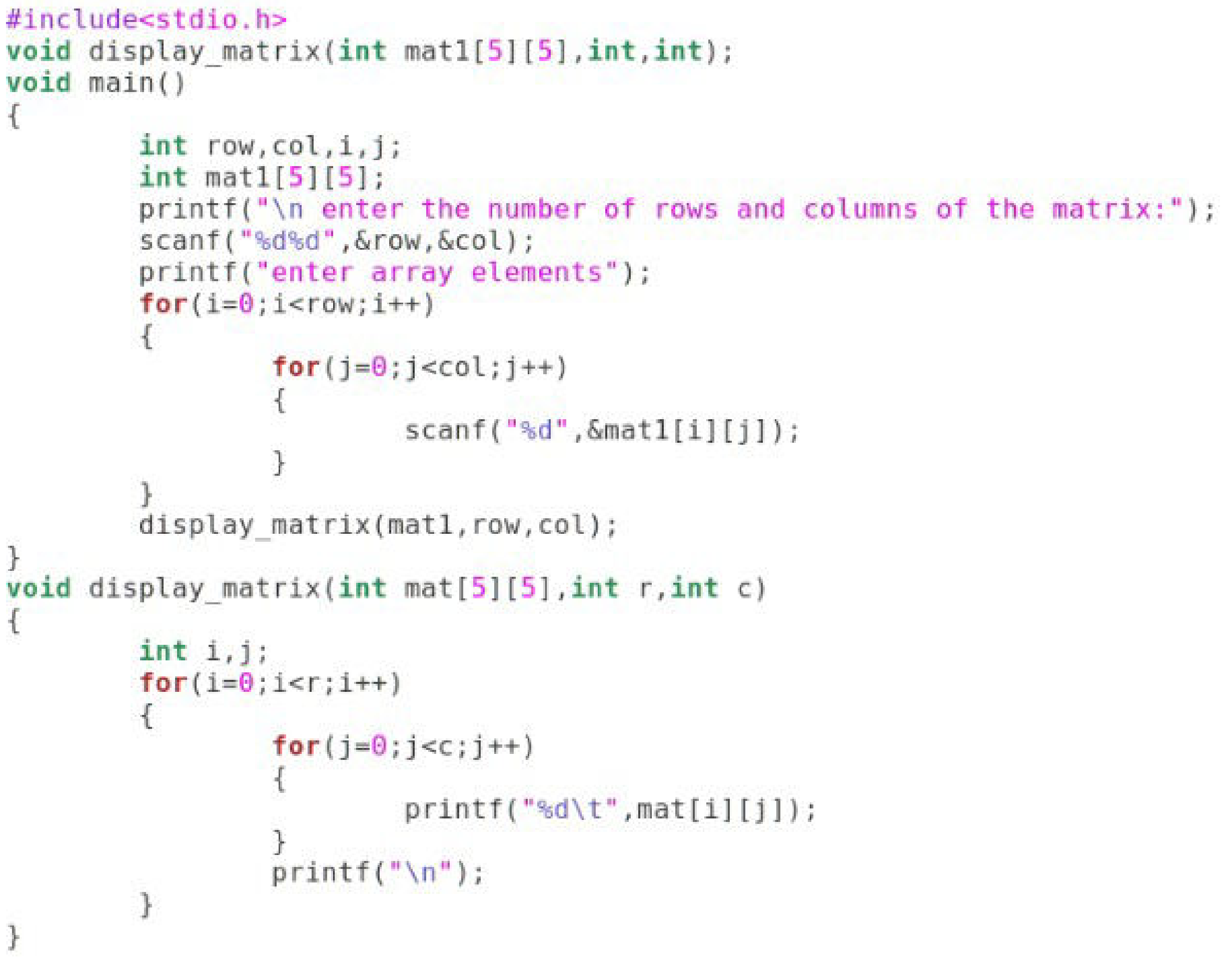
```
//(2)passing a row with 2D array
#include<stdio.h>
void func(int arr[]);
void main()
{
        int arr[2][3]={{1,2,3},
                       {4,5,6}};
        func(arr[1]);
}
void func(int arr[])
{
        for(int i=0;i<3;i++)
                printf("%d\n",arr[i]);
}
```

Output:

```
4
5
6
```

(3) PASSING THE ENTIRE ARRAY:

To pass a 2D array to a function, we use the array name as the actual parameter. (The same we did in case of a 1D array). However, the parameter in the called function must indicate that the array has 2 dimensions.



```c
#include<stdio.h>
void display_matrix(int mat1[5][5],int,int);
void main()
{
        int row,col,i,j;
        int mat1[5][5];
        printf("\n enter the number of rows and columns of the matrix:");
        scanf("%d%d",&row,&col);
        printf("enter array elements");
        for(i=0;i<row;i++)
        {
                for(j=0;j<col;j++)
                {
                        scanf("%d",&mat1[i][j]);
                }
        }
        display_matrix(mat1,row,col);
}
void display_matrix(int mat[5][5],int r,int c)
{
        int i,j;
        for(i=0;i<r;i++)
        {
                for(j=0;j<c;j++)
                {
                        printf("%d\t",mat[i][j]);
                }
                printf("\n");
        }
}
```

Program:

Output:

```
Enter the number of rows and columns of the matrix= 2 2
Enter array elements
1       2       3       4
1       2
3       4
```

Standard Functions in C

- The standard functions are built-in functions.
- In C programming language, the standard functions are declared in header files and defined in .dll files.
- In simple words, the standard functions can be defined as "the ready made functions defined by the system to make coding more easy".
- The standard functions are also called as **library functions** or **pre-defined functions**.
- In C when we use standard functions, we must include the respective header file using **#include** statement.
- For example, the function **printf()** is defined in header file **stdio.h** (Standard Input Output header file).
- When we use **printf()** in our program, we must include **stdio.h** header file using **#include<stdio.h>** statement.

C Programming Language provides the following header files with standard functions.

Header File	**Purpose**	**Example Functions**
stdio.h	Provides functions to perform standard I/O operations	printf(), scanf()
conio.h	Provides functions to perform console I/O operations	clrscr(), getch()
math.h	Provides functions to perform mathematical operations	sqrt(), pow()
string.h	Provides functions to handle string data values	strlen(), strcpy()
stdlib.h	Provides functions to perform general functions/td>	calloc(), malloc()
time.h	Provides functions to perform operations on time and date	time(), localtime()
ctype.h	Provides functions to perform - testing and mapping of character data values	isalpha(), islower()
setjmp.h	Provides functions that are used in function calls	setjump(), longjump()
signal.h	Provides functions to handle signals during program execution	signal(), raise()
assert.h	Provides Macro that is used to verify assumptions made by the program	assert()
locale.h	Defines the location specific settings such as date formats and currency symbols	setlocale()
stdarg.h	Used to get the arguments in a function if the arguments are not specified by the function	va_start(), va_end(), va_arg()
errno.h	Provides macros to handle the system calls	Error, errno
graphics.h	Provides functions to draw graphics.	circle(), rectangle()
float.h	Provides constants related to floating point data values	
stddef.h	Defines various variable types	
limits.h	Defines the maximum and minimum values of various variable types like char, int and long	

Preprocessor Commands in C

- In C programming language, preprocessor directive is a step performed before the actual source code compilation.
- It is not part of the compilation.
- Preprocessor directives in C programming language are used to define and replace tokens in the text and also used to insert the contents of other files into the source file.
- When we try to compile a program, preprocessor commands are executed first and then the program gets compiled.
- Every preprocessor command begins with # symbol. We can also create preprocessor commands with parameters.

  Following are the preprocessor commands in C programming language...

#define

#define is used to create symbolic constants (known as macros) in C programming language. This preprocessor command can also be used with parameterized macros.

#undef

#undef is used to destroy a macro that was already created using #define.

#ifdef

#ifdef returns TRUE if the macro is defined and returns FALSE if the macro is not defined.

#ifndef

#ifndef returns TRUE if the specified macro is not defined otherwise returns FALSE.

#if

#if uses the value of specified macro for conditional compilation.

#else

#else is an alternative for #if.

#elif

#elif is a #else followed by #if in one statement.

#endif

#endif is used terminate preprocessor conditional macro.

#include

#include is used to insert specific header file into C program.

#error

#error is used to print error message on stderr.

#pragma

#pragma is used to issue a special command to the compiler.

In C programming language, there are some pre-defined macros and they are as follows...

1. **__ DATE __ :** The current date as characters in "MMM DD YYYY" format.
2. **__ TIME __ :** The current time as characters in "HH : MM : SS" format.
3. **__ FILE __ :** This contains the current file name.
4. **__ LINE __ :** This contains the current line number.
5. **__ STDC __ :** Defines 1 when compiler compiles with ANSI Standards.

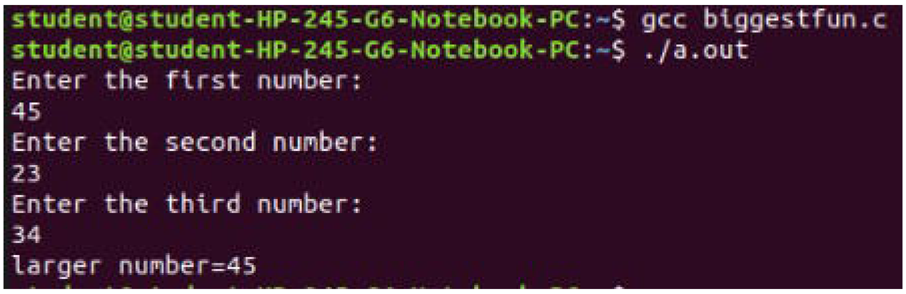
PROGRAMMING EXAMPLES

1. Write a program to find biggest of three integers using functions?

Program:

```c
//program to find biggest of three integers using function
#include<stdio.h>
int greater(int a,int b,int c);
void main()
{
        int num1,num2,num3,large;
        printf("Enter the first number:\n");
        scanf("%d",&num1);
        printf("Enter the second number:\n");
        scanf("%d",&num2);
        printf("Enter the third number:\n");
        scanf("%d",&num3);
        large=greater(num1,num2,num3);
        printf("larger number=%d\n",large);
}
int greater(int a,int b,int c)
{
        if(a>b && a>c)
                return a;
        if(b>a && b>c)
                return b;
        else
                return c;
}
```

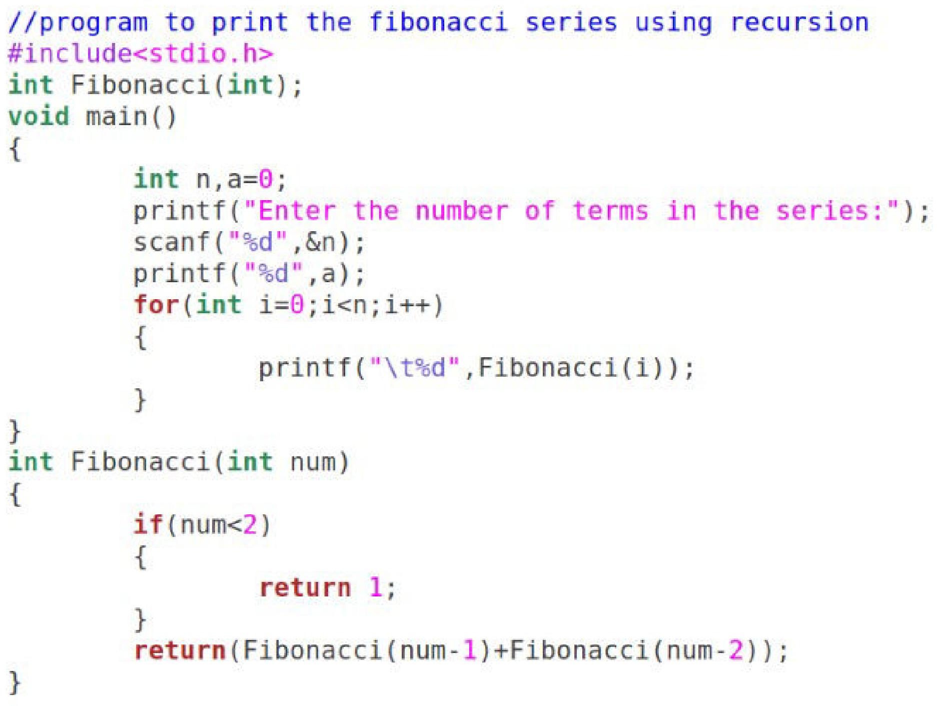
Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc biggestfun.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Enter the first number:
45
Enter the second number:
23
Enter the third number:
34
larger number=45
```

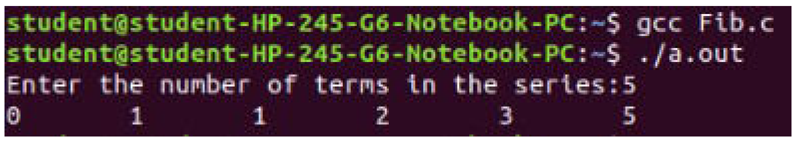
Write a C program to print Fibonacci series using recursion?

<u>Program:</u>



```c
//program to print the fibonacci series using recursion
#include<stdio.h>
int Fibonacci(int);
void main()
{
        int n,a=0;
        printf("Enter the number of terms in the series:");
        scanf("%d",&n);
        printf("%d",a);
        for(int i=0;i<n;i++)
        {
                printf("\t%d",Fibonacci(i));
        }
}
int Fibonacci(int num)
{
        if(num<2)
        {
                return 1;
        }
        return(Fibonacci(num-1)+Fibonacci(num-2));
}
```

<u>Output:</u>



```
student@student-HP-245-G6-Notebook-PC:~$ gcc Fib.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Enter the number of terms in the series:5
0       1       1       2       3       5
```