

## Chapter-09

### User Defined Functions

#### Functions:

A large program can be divided into a series of individual related program called modules. These modules are called functions.

#### Types of Functions:

1. Library function/Built-in function/pre-defined
2. User Defined Functions

##### 1. Library Function:

Library functions are those functions which are already defined in C library, example printf(), scanf(), strcat() etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

Library functions are also known as built in function or predefined function.

##### 2. User Defined Functions:

The function which is written by the user are called user-defined function. It reduces the complexity of a big program and optimizes the code.

#### Benefits of Using Functions:

- It provides modularity to your program's structure.
- It makes your code reusable. You just have to call the function by its name to use it, wherever required.
- In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
- It makes the program more readable and easy to understand.

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

```
int main()
```

```
{
```

```
    add();
```

```
    return 0;
```

```
}
```

```
void add()
```

```
{
```

```
    int a,b,c;
```

```
    scanf("%d%d",&a,&b);
```

```
    c=a+b;
```

```
    printf("%d",c);
```

```
}
```

In the above example add() is a user defined function .

#### Elements of User-defined Function:

1. Function declaration/prototype
2. Function Definition
3. Function Call

### 1. Function declaration/prototype:

Function prototype is a declaration of function without its body to give compiler information about user-defined function.

#### Syntax:

```
return_type function_name (parameters [optional] );
```

#### Example:

```
void add( );
```

### 2. Function Definition:

Function definition means defining body of the function. In this the user defines the functionality (task) of the user defined function.

#### Syntax:

```
return_type function_name (parameters [optional] )  
{  
    body of the function;  
}
```

#### Example:

```
void add( )  
{  
    int a=4,b=5,c;  
    c=a+b;  
    printf("%d",c);  
}
```

### 3. Function Call:

In order to use any pre-defined or user-defined function it has to be invoked (called). This is called function call.

#### Syntax:

```
Function_name( );
```

#### Example:

```
add( );
```

### Categories of Functions:

1. Function with no parameter and no return value
2. Function with no parameter and return value
3. Function with parameter and no return value
4. Function with parameter and return value

### 1. Function with no parameter and no return value:

- In this category, there is no data transfer between the calling function and called function, but there is flow of control from calling function to the called function.
- When no parameters are passed, the called function can not receive any values from the calling function. when the function does not return any value, then the calling function cannot receive any value from the called function.

**Example:**

```
#include<stdio.h>
void sum(); //function declaration/prototype
int main()
{
sum(); // function call
return 0;
}

void sum() //function definition(called function)
{
int first=10,second=20,r;
r= first + second;
printf(“%d+%d=%d”,first,second,r);
}
```

**Output:**

10+20=30

**2. Function with no parameter and return value:**

In this category, there is no data transfer between the calling function to the called function, but there is data transfer from the called function to the calling function.

**Example:**

```
#include <stdio.h>
int sum(); //function declaration/prototype

int main()
{
int n;
n=sum();
printf("Addition=%d",n);
return 0;
}

int sum()
{
int first,second,r;
printf("Enter 2 values\n");
scanf("%d%d",&first,&second);
r=first+second;
return r;
}
```

**Output:**

Enter 2 values

12

32

Addition=44

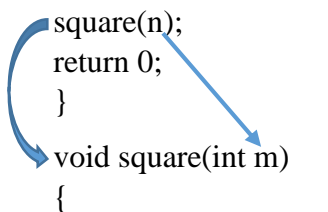
**3. Function with parameter and no return value:**

In this category, there is data transfer between the calling function to the called function, but there is no data transfer from the called function to the calling function.

**Example:**

```
#include <stdio.h>
void square(int); //function declaration/prototype
int main()
{
    int n;
    printf("Enter number:\n");
    scanf("%d",&n);
    square(n);
    return 0;
}

void square(int m)
{
    int r;
    r=m*m;
    printf("square of %d =%d",m,r);
}
```

A blue curved arrow points from the `square(n);` line in the `main` function to the `void square(int m)` function definition. A straight blue arrow points from the `n` in `square(n);` to the `m` in the function definition's parameter list.**Output:**

Enter number:

6

square of 6 =36

**4. Function with parameter and return value:**

- In this category, there is data transfer between the calling function and the called function.
- When parameters are passed, the called function receive values from the calling function and When functions returns a value, then the calling function receives value from the called function.

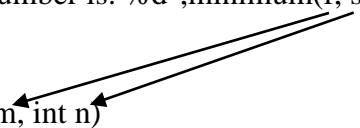
**Example:**

```
#include <stdio.h>
int minimum(int,int); //function declaration/prototype
int main()
{
    int f,s;
    printf("Enter 2 number:\n");
    scanf("%d%d",&f,&s);
}
```

```

printf("smallest number is: %d",minimum(f, s));
return 0;
}
int minimum(int m, int n)
{
if(m<n)
{
return m;
}
else
{
return n;
}
}

```



### Output:

Enter 2 number:  
11  
32  
smallest number is: 11

### Actual Parameters :

The arguments that are passed in a function call are called actual arguments. These arguments are defined in the calling function.

Eg. sum(a,b); here a and b is actual parameter

### Formal Parameters :

These are the variables or expressions referenced in the parameter list of function definition. The datatype of the receiving value must be defined.

Eg. void sum(int n1,int n2)  
{  
statements;  
}

Here n1 and n2 are formal parameters.

### Difference between actual and formal parameter:

Actual Parameter	Formal Parameter
▪ It is used during function call	▪ It is used during function definition
▪ Actual parameters are used to send values to formal parameters	▪ Formal parameters are always used to receive values
▪ Actual parameter can be any variable, value, constant or expressions.	▪ Formal parameters can only be variables.

### Passing parameter to a function:

1. Call by value(pass by value)
2. Call by reference(pass by reference)

#### 1. Call by value(pass by value):

- When a function is called with actual parameter, the value of actual parameters are copied into the formal parameters.
- If the values of the formal parameters change in the function, the value of the actual parameters are not changed. This way of passing parameter is called pass by value or call by value.

#### Example:

**program to swap 2 number using pass by value.**

```
#include <stdio.h>
void swap(int, int);

int main()
{
    int x, y;

    printf("Enter the value of x and y\n");
    scanf("%d%d",&x,&y);

    printf("Before Swapping\nx = %d\ny = %d\n", x, y);

    swap(x, y);

    printf("After Swapping\nx = %d\ny = %d\n", x, y);

    return 0;
}

void swap(int a, int b)
{
    int temp;

    temp = b;
    b = a;
    a = temp;
    printf("Values of a and b is %d %d\n",a,b);
}
```

#### Output:

```
Enter the value of x and y
23
87
Before Swapping
x = 23
y = 87
```

Values of a and b is 87 23

After Swapping

x = 23

y = 87

**Note: in above program value of formal parameter a and b are swapped in the function but, the values of actual parameters x and y after swap remain same.**

## 2. Call by reference(pass by reference):

In pass by reference, function is called with the address of actual parameter, the formal parameter receives address of actual parameter. Any changes done on formal parameter will affect the actual parameter.

### Example:

**Program to swap 2 number using pass by reference.**

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

```
void swap (int*, int*);
```

```
int main()
```

```
{
```

```
    int a, b;
```

```
    printf("\nEnter value of a & b: ");
```

```
    scanf("%d %d", &a, &b);
```

```
    printf("\nBefore Swapping:");
```

```
    printf("\na = %d\nb = %d\n", a, b);
```

```
    swap(&a, &b);
```

```
    printf("After Swapping:\n");
```

```
    printf("a = %d\nb = %d", a, b);
```

```
    return 0;
```

```
}
```

```
void swap (int *x, int *y)
```

```
{
```

```
    int temp;
```

```
    temp = *x;
```

```
    *x = *y;
```

```
    *y = temp;
```

```
}
```

### **Output:**

Enter value of a & b:

12

43

Before Swapping

a = 12

b = 43

After Swapping:

a = 43

b = 12

### **Difference between call by value & call by reference:**

<b>call by value</b>	<b>call by reference</b>
In call by value, value is passed as a parameter.	In call by reference, address is passed as a parameter.
Any changes made in formal parameter will not affect actual parameter	Any changes made in formal parameter will affect actual parameter
Separate memory location is allocated for actual and formal parameter	Same memory location is allocated for actual and formal parameter

### **Passing Array to a Function:**

Array can be passed as an actual parameter in following ways

- i. Passing individual array element
- ii. Passing whole array

#### **1. Passing individual array element:**

While passing an individual element of an array we will have to pass array with its index number.

##### **Example:**

```
#include <stdio.h>
void data(int m, int n)
{
    printf("m: %d\n",m);
    printf("n: %d\n",n);
}
int main()
{
    int a[5]={2,3,4,5,6};
    data(a[2],a[4]); // passing two elements 4 and 6
    return 0;
}
```

##### **Output:**

m: 4  
n: 6

#### **2. Passing whole array as a parameter:**

While passing entire array as a parameter we will have to pass only name of the array.

##### **Example:**

```
#include <stdio.h>
void data(int m[5])
{
    int i;
    for(i=0;i<=4;i++)
    {
        printf("%d\t",m[i]);
    }
}
```



```
int main()
{
int a[5]={2,3,4,5,6};
data(a); //‘a’ is name of the array i.e. we are passing entire array
return 0;
}
```

### Output:

2    3    4    5    6

### **Recursion:**

- A function that calls itself is known as a recursive function. And, this technique is known as recursion.
- The recursion continues until some condition is met to prevent it.
- To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call, and other doesn't.

### **Example:**

```
void data()
{
data(); /* function calls itself */
}
int main()
{
data();
}
```

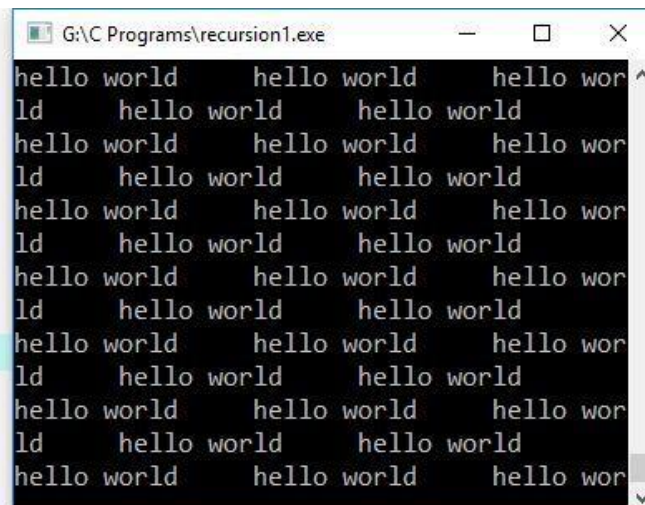
### **Properties:**

A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have –

- **Base criteria** – There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- **Progressive approach** – The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

### **1. Program for recursion without any base criteria (condition):**

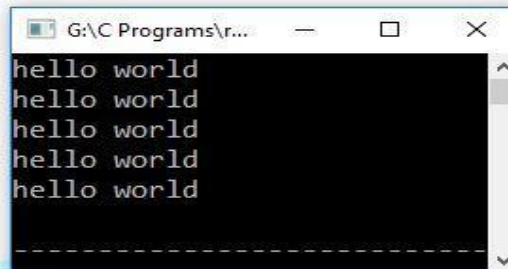
```
#include<stdio.h>
int main()
{
display();
return 0;
}
void display()
{
printf("hello world\t");
display();
}
```



- In the above program we calling display() function inside the body of display() function. This is called recursion.
- But we have not used any condition in above program so in this the display() function will be called infinite times. That's why base condition is necessary in recursion.

## 2. Program for recursion with base criteria (condition):

```
#include<stdio.h>
int main()
{
    display();
    return 0;
}
void display()
{
    static int i=1;
    printf("hello world\n");
    i++;
    if(i<=5)
    {
        display();
    }
}
```

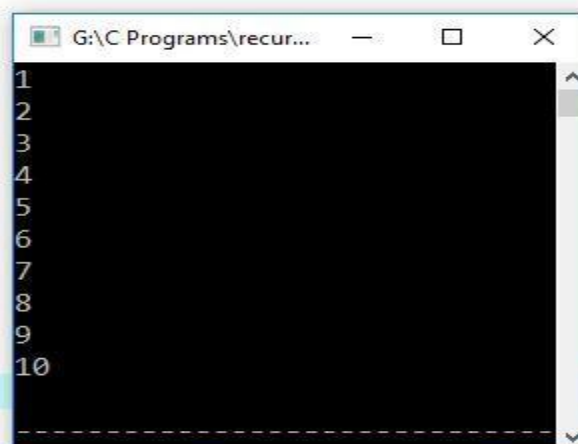


```
G:\C Programs\r...
hello world
hello world
hello world
hello world
hello world
```

- In the above program display() function is called inside display() function but with a base condition i.e.  $i \leq 5$ . It means as long as the value of  $i$  is 5 or less than 5 display() function will be called .
- But as the value of  $i$  becomes greater than 5, condition will get false and display() function will no more get called and the program will also not go in infinite loop.

## 3. Program to print 1 to 10 using recursion.

```
#include<stdio.h>
int main()
{
    Number();
    return 0;
}
void Number()
{
    static int i=1;
    printf("%d\n",i);
    i++;
    if(i<=10)
    {
        Number();
    }
}
```



```
G:\C Programs\recur...
1
2
3
4
5
6
7
8
9
10
```

## 4. Program to display factorial of a number using recursion.

### Logic:

Suppose we want factorial of 4 using recursion then logic will be

```

n*fact(n-1)
4*fact(3)
3*fact(2)
2*fact(1)
1*fact(0)

```

```

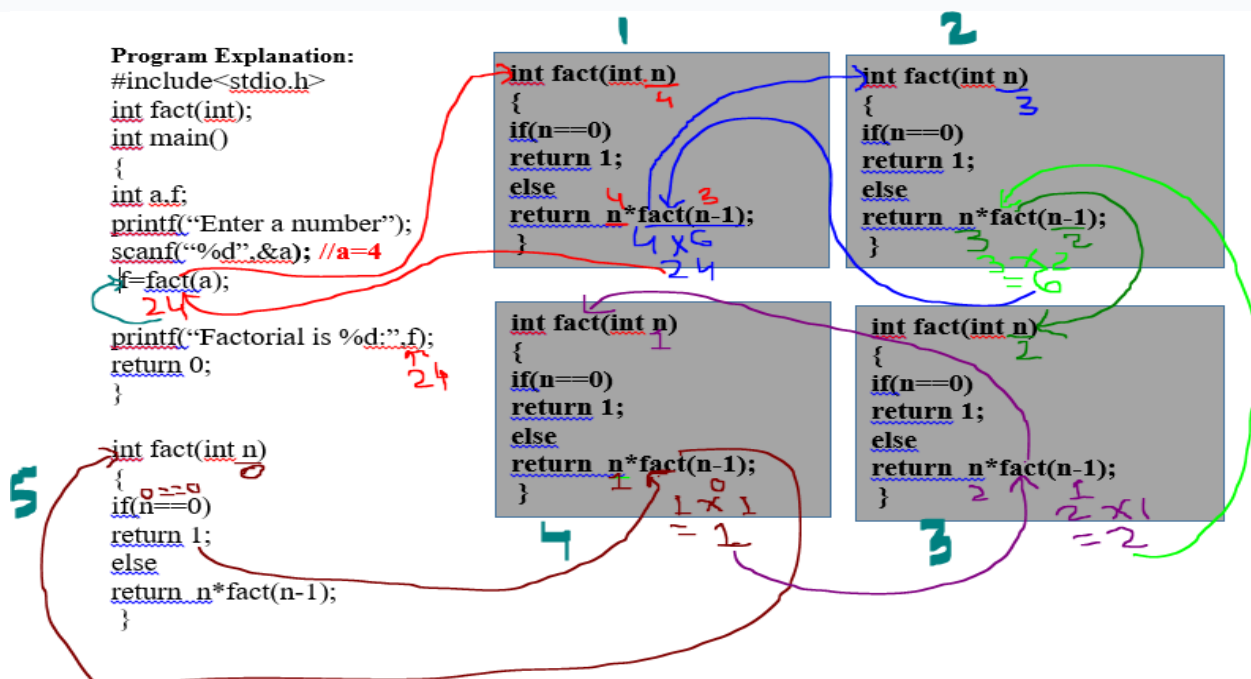
#include<stdio.h>
int fact(int);
int main()
{
    int a,f;
    printf("Enter a number");
    scanf("%d",&a);
    f=fact(a);
    printf("Factorial is %d:",f);
    return 0;
}
int fact(int n)
{
    if(n==0)
    return 1;
    else
    return n*fact(n-1);
}

```

### Output:

Enter a number: 4  
Factorial is 24

### Explanation of above program



## Storage Classes:

Storage class is used to define the lifetime and visibility of a variable and/or function. Lifetime refers to the period during which the variable remains active and visibility refers to the module of a program in which the variable is accessible.

### Few storage classes are as follows:

- automatic (local)
- external (global)
- register
- static

#### a. automatic(local variable):

- Local variables are the variable which are defined within a function. These variables are also called as automatic variable.
- As and when control goes into the function memory is allocated for these variables and as control comes out from the function, memory is de-allocated for these variables.
- These variables are declared using auto keyword but it is not mandatory to put auto keyword before the variable name.

#### Syntax:

```
auto data_type variable_name;  
OR  
data_type Variable_name;
```

#### Example:

```
void sum()  
{  
    auto int a=10;  
    printf("a=%d",a);  
}
```

Here, a is local variable, it can also be written without auto keyword i.e. int a=10 and it can be used only inside sum() function.

#### b. External variable:

- Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used.
- “extern” variables are used to access variable of a program into any other program. External Variables are declared using “extern” keyword.

```
//file name: fun1.c  
#include <stdio.h>  
extern int a=10;
```

```
//file name: fun2.c  
#include <stdio.h>  
#include "fun1.c"  
extern int a; //it means 'a' is declared somewhere else  
int main()  
{  
    printf("a=%d",a);  
    return 0;  
}
```

**Variable 'a' is defined in "fun1.c" file with value 10. Suppose we want to access this variable 'a' inside "fun2.c" file, so in "fun2.c" include "fun1.c" file and write "extern int a" it indicates that the variable 'a' is defined in some other file and we want to access it.**

### **Global Variables:**

- Variables that are alive and active throughout the program are known as global variable or external variable.
- Global variables can be accessed by any function in the program. Global variables are declared outside of all the function.

### **Example:**

```
#include<stdio.h>
void demo();
int p=12; // p is a global variable and it is accessible everywhere
int main()
{
    printf("value of a in main is %d",p);
    demo();
    return 0;
}

void demo()
{
    printf("\nvalue of a in demo is %d",p);
}
```

### **Output:**

value of a in main is 12  
value of a in demo is 12

### **c. Static variable:**

- Static variables are initialized only once.
- Static variables can be defined inside or outside the function. The default value of static variable is 0.
- The static variable are alive till the execution of the program.

### **Example:**

```
#include<stdio.h>
void demo();
int main()
{
    demo();
    demo();
    demo();
    return 0;
}

void demo()
{
    static int i=1;
    printf("%d ",i);
```

```
i++;  
}
```

**Output:**

1 2 3

**d. Register variable:**

This storage class declares register variables which have the same functionality as that of the auto variables.

The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. Usually, a few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program.

An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers. It is declared using “register” keyword.

**Syntax:**

```
register data_type variable_name;
```

**Example:**

```
#include <stdio.h>  
int main()  
{  
    register char b = 'G';  
    printf(" value of register variable b is:%c",b);  
    return 0;  
}
```

**Output:**

value of register variable b is: G

**Note:** We can not get address of register variables.

```
#include <stdio.h>  
int main()  
{  
    register int a;  
    printf("Trying to get address of a=%d",&a);  
    return 0;  
}
```

Compile Log ✓ Debug Find Results ✖ Close

p\notes by deepa\fun2.c

\notes by deepa\fun2.c

Message

In function 'main':

[Error] address of register variable 'a' requested

\*\*\*