

FUNCTIONS

Topics Covered

Functions: Declaration, Definition, Call and return, Call by value, Call by Reference, Scope of variables, Storage classes, Recursive functions, Recursion vs. Iteration.

Function

- Function is a self contained block of statements that perform a coherent task of some kind.
- Every C program can be a thought of the collection of functions.
- `main()` is also a function.

Types of Functions

- Library functions
 - These are the in-built functions of 'C' library.
 - e.g. `printf()` ; is a function which is used to print an output. It is declared in 'stdio.h' file .
- User defined functions.
 - Programmers can create their own functions in C to perform specific task

Terms

- Actual arguments:- The arguments of calling function are actual arguments.
- Formal arguments:- Arguments of called function are formal arguments.
- Argument list:- Means variable name enclosed within the parenthesis. They must be separated by comma
- Return value:- It is the outcome of the function. The result obtained by the function is sent back to the calling function through the return statement. Function can return only a single value.

Need of Function

- Writing functions avoids rewriting of the same code again and again in the program.
- Function makes program structured.
- Using functions, large programs can be reduced to smaller ones. It is easy to debug and find out the errors in it.

Syntax

Function Declaration/Prototype

```
ret_type func_name (data_type par1,data_type par2...data_type parn);
```

Function Definition

```
ret_type func_name (data_type par1,data_type par2...data_type parn)  
{  
}
```

Function Call

```
func_name (par1, par2... parn);
```

Function declaration/prototype

- A prototype statement helps the compiler to check the return type and arguments type of the function.
- A prototype function consist of the functions return type, name and argument list.
- Example:
`void sum(int x, int y);`


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

```
void functionName()  
{
```

```
    ... ..  
    ... ..
```

```
}
```

```
int main()  
{
```

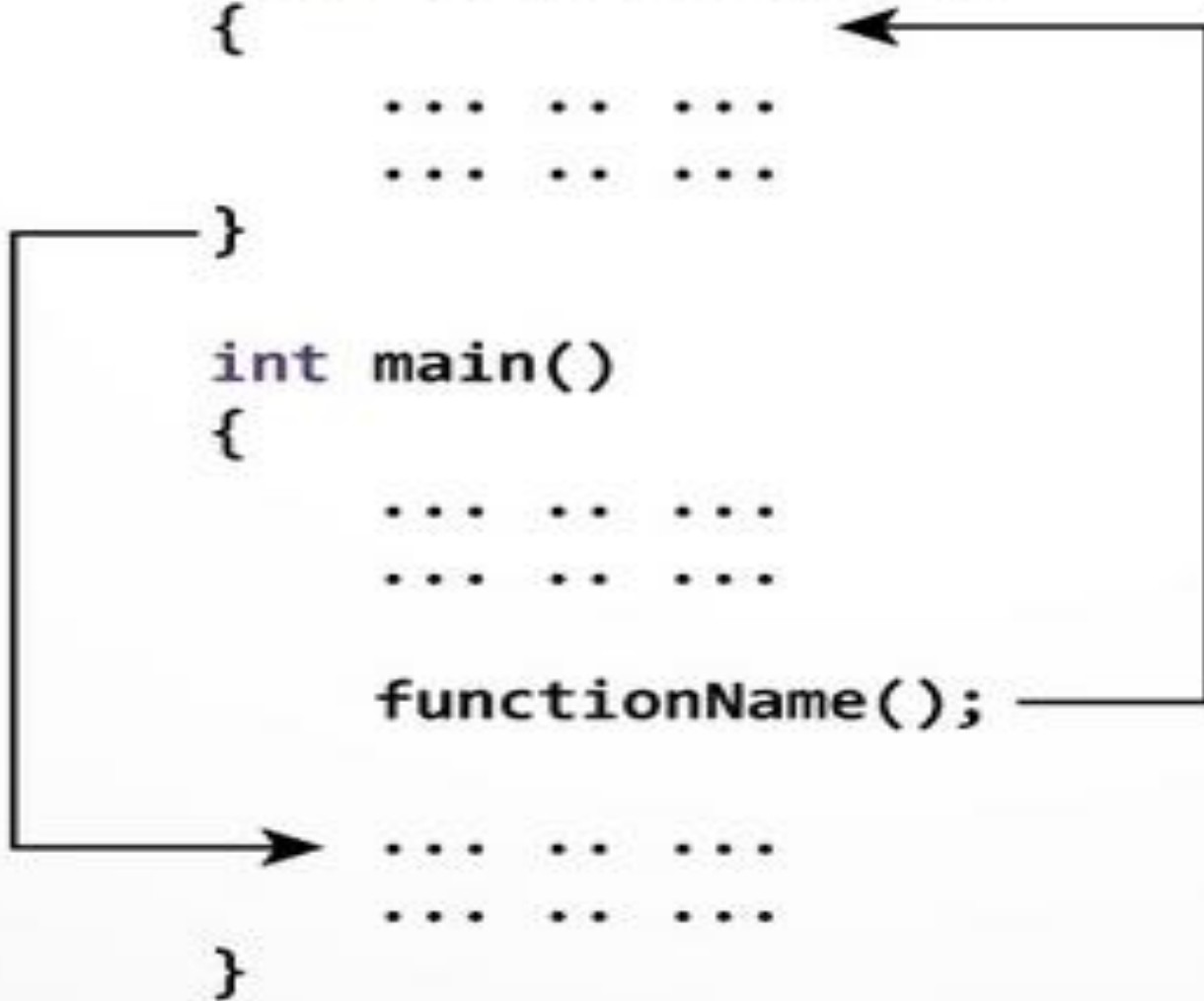
```
    ... ..  
    ... ..
```

```
    functionName();
```

```
    ... ..  
    ... ..
```

```
}
```

How does function
work?



Example

```
#include<stdio.h>
void sum(int, int);    // function declaration/prototype
int main()
{
    int a, b;
    printf("enter the two no");
    scanf("%d %d", &a, &b);
    sum(a,b);          // function calling
}

void sum( int x, int y)    // function definition
{
    int c;
    c=x+y;
    printf ("sum is %d", c);
}
```

Example

```
#include<stdio.h>
int sum(int, int);
int main()
{
    int a=10,b=20, c;
    c=sum(a,b);
    printf("sum is %d", c);
}
```

/*actual arguments

```
int sum(int x, int y)
{
    int s;
    s=x+y;
    return(s);
}
```

/*formal arguments

/*return value

Categories of functions

- A function with no parameter and no return value
- A function no parameter and with return value
- A function with parameter and no return value
- A function with parameter and with return value

A function with no parameter and no return value

```
#include<stdio.h>
void print();           //func declaration
int main()
{
    printf ("no parameter and no return value");
    print();            //func calling
}
void print()            //func definition
{
    for(int i=1;i<=30;i++)
    {
        printf("*");
    }
    printf("\n");
}
```

A function with no parameter and no return value (Contd...)

- There is no data transfer between calling and called function
- The function is only executed and nothing is obtained
- Such functions may be used to print some messages, etc.

A function with parameters and no return value

```
#include<stdio.h>
void mul(int, int);
int main()
{
    int a=10,b=20;

    mul(a,b);                //actual arguments

}
void mul(int x, int y)       //formal arguments
{
    int s;
    s=x*y;
    printf ("mul is %d", s);
}
```

A function with no parameters and with return value

```
#include<stdio.h>
int sum();
int main()
{

    int c=sum();
    printf("sum is %d", c);
}
int sum()
{
    int x=10,y=30;
    return(x+y);    //return value
}
```


A function with parameter and with return value

```
#include<stdio.h>
int max(int, int);
int main()
{
    int a=10,b=20,c;
    c=max(a,b);
    printf ("greatest no is %d", c);
}
int max(int x, int y)
{
    if(x>y)
        return(x);
    else
        return(y);
}
```

- Discuss few more programs of functions....

Argument passing techniques

- Call By Value
- Call By Reference

Call By Value

- It is a default mechanism for argument passing.
- When an argument is passed by value then the copy of argument is made known as formal arguments which is stored at separate memory location
- Any changes made in the formal argument are not reflected back to actual argument, rather they remain local to the block which are lost once the control is returned back to calling program

Example

```
void swap(int,int);
int main()
{
int a=10,b=20;
printf("before function calling a =%d b= %d", a, b);
swap(a,b);
printf("after function calling a= %d b=%d", a, b);
return 0;
}
void swap(int x, int y)
{
int z;
z=x;
x=y;
y=z;
printf("Value of x is %d and y is %d ", x, y);
}
```

Output:

before function calling a=10 b=20
value of x is 20 and y is 10
after function calling a=10 b=20

Call By Reference

- In this instead of passing value, address are passed.
- Here formal arguments are pointers to the actual arguments
- Hence change made in the argument are permanent.
- The address of arguments is passed by preceding the address operator(&) with the name of the variable whose value you want to modify.
- The formal arguments are processed by asterisk (*) which acts as a pointer variable to store the addresses of the actual arguments

Example

```
void swap(int *,int *);
int main()
{
int a=10 ,b=25;
printf("before function calling a =%d b= %d", a, b);
swap(&a, &b);
printf("after function calling a= %d b= %d", a, b);
return 0;
}
void swap(int *x, int *y)
{
int z;
z=*x;
*x=*y;
*y=z;
printf("Value of x is %d and y is %d ", *x, *y);
}
```

Output:

before function calling a= 10 b= 25
value of x is 25 and y is 10
after function calling a=25 b= 10

- Call by value: This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- Call by reference: This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Scope of variables

- A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed.

There are two places where variables can be declared in C programming language –

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.

Local versus global

```
#include<stdio.h>
int g = 20;
int main () {
printf("g = %d\n",g); //global
int g = 10; //local
printf("g = %d\n",g); //local
}
```

Storage Classes in C

Storage Classes are used to describe the features of a variable.

These features basically include:

- (a) Where the variable would be **stored**.
- (b) What will be the **initial value**
- (c) What is the **scope** of the variable
- (d) How **long** would the variable **exist**.

Four Storage classes

- 1) auto
- 2) register
- 3) static
- 4) extern

auto

auto	
Storage	memory
Initial value	Garbage value
Scope	Local
Life	Till within the scope

auto – example 1

```
main()
{
    auto int i, j;
    printf(“%d, %d”, i, j);
}
```

Garbage value will be printed.

auto- example 2

```
main( ){  
    auto int i = 1 ;  
    {  
        {  
            printf ( "\n%d ", i ) ;  
        }  
        printf ( "%d ", i ) ;  
    }  
    printf ( "%d", i ) ;  
}
```

auto- example 3

```
main( ){  
    auto int i = 1 ;  
    {  
        auto int i = 2 ;  
        {  
            auto int i = 3 ;  
            printf ( "\n%d ", i ) ;  
        }  
        printf ( "%d ", i ) ;  
    }  
    printf ( "%d", i ) ;  
}
```


register – fast but limited in number

REGISTER	
Storage	CPU Registers
Initial value	Garbage value
Scope	Local
Life	Till within the scope

register – example 1

```
main( ){  
    register int i ;  
    for ( i = 1 ; i <= 10 ; i++ )  
        printf ( "\n%d", i ) ;  
}
```

static— value persists and shared among functions

STATIC	
Storage	memory
Initial value	zero
Scope	Local
Life	value persists among different functions

```

main( )
{
    increment( ) ;
    increment( ) ;
    increment( ) ;
}

increment( )
{
    auto int i = 1 ;
    printf ( "%d\n", i ) ;
    i = i + 1 ;
}

```

```

main( )
{
    increment( ) ;
    increment( ) ;
    increment( ) ;
}

increment( )
{
    static int i = 1 ;
    printf ( "%d\n", i ) ;
    i = i + 1 ;
}

```

The output of the above programs would be:

1
1
1

1
2
3

extern – global variables

EXTERN	
Storage	memory
Initial value	zero
Scope	global
Life	as long as the program

extern – example 1

```
int i ;
```

```
increment( ) {i++; printf ( "i = %d\n", i ) ; }
```

```
decrement( ){i--; printf ( "i = %d\n", i ) ;}
```

```
main( ){  
printf ( "\n i = %d", i ) ;  
increment( ) ; increment( ) ;  
decrement( ) ; decrement( ) ;  
}
```

extern – 1 variable and 2 files

//file 1.c

```
#include<stdio.h>
int a;
void fun()
{
    a=a+2;
    printf("%d",a);
}
```

//file2.c

```
#include<stdio.h>
#include"file1.c"
int main()
{
    extern int a;
    a=7;
    fun();
}
```

```
#include<stdio.h>
main(){
    extern int i;
    printf("%d",i);
}
```

//Error – undefined integer i

Recursion

- Recursion is the process of repeating items in a self-similar way.
- In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

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

Recursion (Contd...)

- The C programming language supports recursion, i.e., a function to call itself.
- But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.
- Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Example

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

```
int fact(int);
```

```
int main(){
```

```
int n=5;
```

```
printf("Factorial = %d\n",fact(n));
```

```
}
```

```
int fact(int n){
```

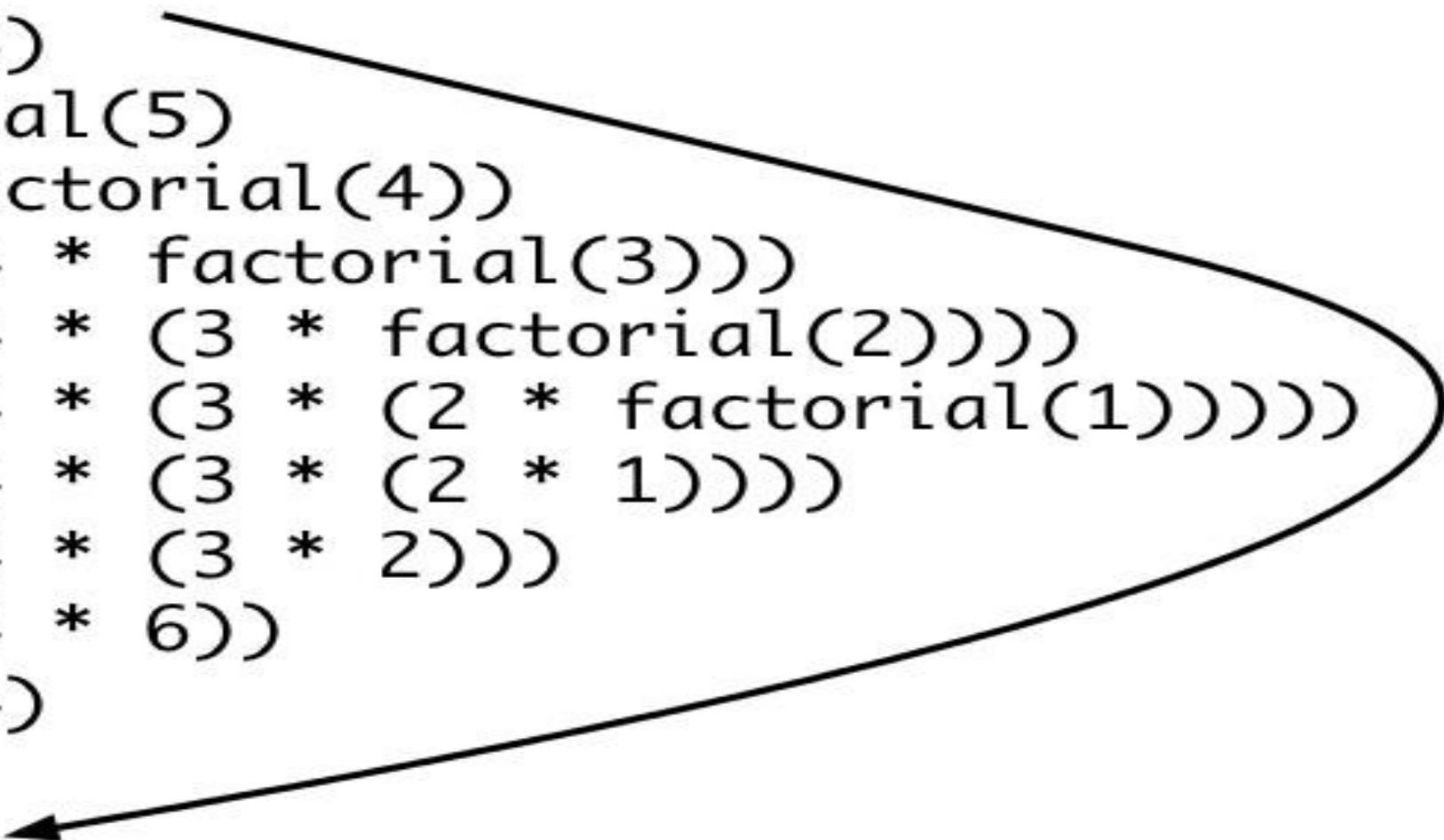
```
if(n>1) return n*fact(n-1); //calling itself with n-1
```

```
else return 1;
```

```
}
```

Recursive

```
factorial(6)
6 * factorial(5)
6 * (5 * factorial(4))
6 * (5 * (4 * factorial(3)))
6 * (5 * (4 * (3 * factorial(2))))
6 * (5 * (4 * (3 * (2 * factorial(1)))))
6 * (5 * (4 * (3 * (2 * 1))))
6 * (5 * (4 * (3 * 2)))
6 * (5 * (4 * 6))
6 * (5 * 24)
6 * 120
720
```

A large, hand-drawn style arrow originates from the top right of the recursive definition and points towards the final result, 720.

Factorial(5)



return 5 * Factorial(4) = 120



return 4 * Factorial(3) = 24



return 3 * Factorial(2) = 6



return 2 * Factorial(1) = 2



1

- Discuss Dev C++ Debugging with the help of program

Recursion programs to try

- Finding summation of n numbers
- Fibonacci series in which $a_0=0$, $a_1=1$, $a(n) = a(n-2) + a(n-1)$ i.e. any term is sum of previous two terms. So series = 0,1,1,2,3,5,8,13...
- Tower of Hanoi game <https://www.youtube.com/watch?v=7AUG7zXe-KU>

Iteration versus recursion

Iteration involves repetition of same structure. Recursion involves repetition through calling the function itself with different inputs.

Iteration needs less memory and time. Recursion consumes more memory and is slower but makes the program simpler.

Thank You