CS101: Problem Solving through C Programming

Sachchida Nand Chaurasia
Assistant Professor

Department of Computer Science
Banaras Hindu University
Varanasi
Email id: snchaurasia@bhu.ac.in, sachchidanand.mca07@gmail.com

February 22, 2021

Function basics in C I

✓ A function is a collection of C statements to do something specific.

✓ A C program consists of one or more functions. Every program must have a function called main().

Advantages of functions:

- A large problem can be divided into subproblems and then solved by using functions.
- The functions are reusable. Once you have created a function you can call it anywhere in the program without copying and pasting entire logic.
- The program becomes more maintainable because if you want to modify the program sometimes later, you need to update your code only at one place.

Types of function

1. Library function
2. User defined function

1. **Library function**

   ✓ C has many built-in library functions to perform various operations, for example: sqrt() function is used to find the square root of a number.

   ✓ Similarly, scanf() and printf() are also library functions.

   ✓ To use a library function we must first include corresponding header file using #include preprocessor directive. For scanf() and printf() corresponding header file is stdio.h, for sqrt() and other mathematical related functions, it is math.h.

Function basics in C III

```c
// Program to find the square root of a number
#include<stdio.h>
#include<math.h>
int main()
{
     float a;
     printf("Enter number: ");
     scanf("%f", &a);
     printf("Square root of %.2f is %.2f", a, sqrt(a));
     return 0;
}
```

Common mathematical functions:

Function	Description	Example
sqrt(x)	square root of x	sqrt(900.0) is 30.0
exp(x)	exponential function	exp(1.0) is 2.718282
log(x)	natural logarithm of x (base e)|log(2.718282)is1.0‘	
log10(x)	logarithm of x (base 10)	log10(1.0) is 0.0
fabs(x)	absolute value of x as a floating-point number	fabs(13.5) is 13.5
ceil(x)	rounds x to the smallest integer not less than x	ceil(9.2) is 10.0
floor(x)	rounds x to the largest integer not greater than x	floor(9.2) is 9.0
pow(x,y)	x raised to power y	pow(2, 7) is 128.0
sin(x)	trigonometric sine of x (x is in radians)	sin(0.0) is 0.0
cos(x)	trigonometric cosine of x (x in radians)	cos(0.0) is 1.0
tan(x)	trigonometric tangent of x (x in radians)	tan(0.0) is 0.0

2. **User defined function:** User created function is known as user-defined functions. To create your own functions you need to know about three things.
   - A. Function definition.
   - B. Function call.
   - C. Function declaration.

Function basics in C VI

A. Function definition:

✓ A function definition consists of the code that makes the function. A function consists of two parts function header and function body. Here is the general syntax of the function.

```
return_type function_name(type1 argument1, type2 argument2, ...)
{
     local variables;
     statement1;
     statement2;
     return (expression);
}
```

✓ A small function.

```
void my_func()
{
     printf("Hello i am my_func()");
}
```

✓ Another example:

```c
int product(int num1, int num2)
{
     int result;
     result = num1 * num2;
     return result;
}
```

B. Function call

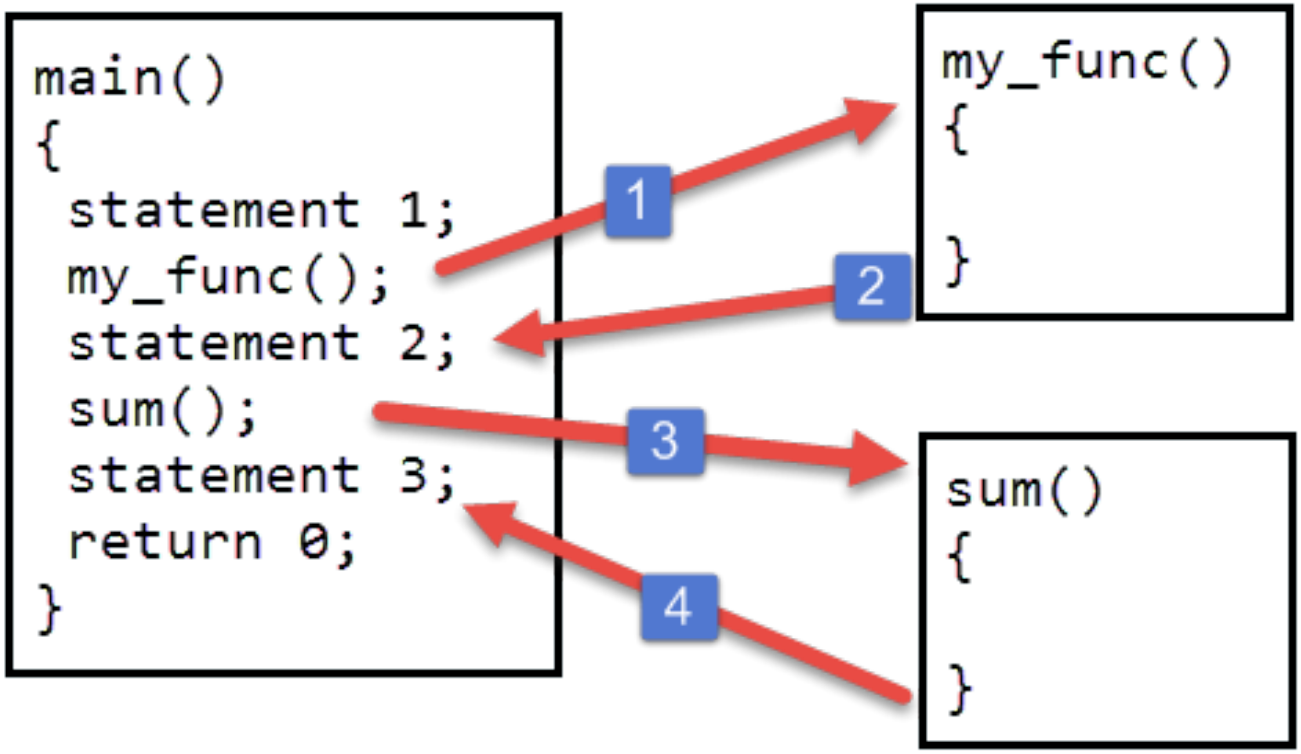
✓ After the function is defined the next step is to use the function, to use the function you must call it. To call a function you must write its name followed by arguments separated by a comma (,) inside the parentheses ().

For example, here is how we can call the product() function we created above.

```c
product(12, 10);
```

Function basics in C VIII

✓ The following figure describes what happens when you call a function.



Function declaration:

The calling function needs some information about the called function. When function definition comes before the calling function then function declaration is not needed. For example:

```c
#include<stdio.h>
// function definition
int sum(int x, int y)
{
     int s;
     s = x + y;
     return s;
}

int main()
{
     // function call
     printf("sum = %d", sum(10, 10));
     return 0;
}
```

Function basics in C X

✓ Notice that the definition of function sum() comes before the calling function i.e main(), that's why function declaration is not needed.
Generally function definition comes after main() function. In this case, the function declaration is needed.
Function declaration consists of function header with a semicolon (;) at the end.

✓ Here are function declarations of function my_func() and sum().

```
1 void my_func(void);
2
3 int product(int x, int y);
4
5 //Names of arguments in a function declaration is optional so,
6 int product(int x, int y)
7 can be written as:
8
9 int product(int , int )
```

✓ Another important point I want to mention is that the name of the arguments defined in the function declaration needs not to be the same as defined in the function definition.

Function basics in C XI

```
1 int sum(int abc, int xyx)  // Function declaration
2
3 int sum(int x, int y)  // Function definition
4 {
5      int s;
6      s = x + y;
7      return s;
8 }
```

A function declaration is generally placed below preprocessor directives.

The following program demonstrates everything we have learned so far.

Function basics in C XII

```c
#include<stdio.h>

// function declaration
int sum(int x, int y);

int main()
{
     // function call
     printf("sum = %d", sum(10, 10));


     return 0;
}

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

Function basics in C XIII

✓ The following program prints the largest number using a function.

```c
#include<stdio.h>
// function declaration
int max(int x, int y);
int main()
{
     // function call
     max(100, 12);
     max(10, 120);
     max(20, 20);
        return 0;
}

// function definition
int max(int x, int y)
{
     if(x > y)
     {
          printf("%d > %d\n", x, y );
```

Function basics in C XIV

```
19        }
20        else if(x < y)
21        {
22              printf("%d < %d\n", x, y );
23        }
24        else
25        {
26              printf("%d == %d\n", x, y );
27        }
28  }
```

The return statement in C

The return statement is used to return some value or simply pass the control to the calling function. The return statement can be used in the following two ways.

```
return;
return expression;
```

✓ The following program demonstrates the use of the first form of the return statement.

```
#include<stdio.h>
void eligible_or_not(int x);
int main()
{
     int n;
     printf("Enter age: ");
     scanf("%d", &n);
     eligible_or_not(n);


     return 0;
}
```

Function basics in C XVI

```
13 void eligible_or_not(int age)
14 {
15      if(age >= 18)
16      {
17           printf("You are eligible to vote");
18           return;
19      }
20      else if(age == 17 )
21      {
22           printf("Try next year");
23           return;
24      }
25      else
26      {
27           printf("You are not yet ready");
28           return;
29      }
30 }
```

✓ The following program computes the factorial of a number using a function.

Function basics in C XVII

```c
#include<stdio.h>
int factorial(int x); // declaration of factorial function
int main()
{
     int n;
     printf("Calculate factorial: \n\n");
     printf("Enter number : ");
     scanf("%d", &n);

     if(n < 0)
     {
          printf("\nFactorial is only defined for positive numbers");
     }
     else
     {
          printf("\n%d! = %d", n, factorial(n)); // calling factorial function
     }


     return 0;
}
```

Function basics in C XVIII

```
22
23 // definition of factorial function
24 int factorial(int n)
25 {
26      if(n == 0)
27      {
28           return 1;
29      }
30
31      int f = 1, i;
32
33      for(i = n; i > 0; i-- )
34      {
35           f = f * i;
36      }
37      return f;
38 }
```

Actual and Formal arguments in C

Actual arguments: Arguments which are mentioned in the function call is known as the actual argument. For example:

func1(12, 23);

Here 12 and 23 are actual arguments.

Actual arguments can be constant, variables, expressions etc.

1. func1(a, b); // here actual arguments are variable
2. func1(a + b, b + a); // here actual arguments are expression

Formal Arguments: Arguments which are mentioned in the definition of the function is called formal arguments. Formal arguments are very similar to local variables inside the function. Just like local variables, formal arguments are destroyed when the function ends.

```
1 int factorial(int n)
2 {
3     // write logic here
4 }
```

✓ Things to remember about actual and formal arguments.

Function basics in C XX

1. Order, number, and type of the actual arguments in the function call must match with formal arguments of the function.
2. If there is type mismatch between actual and formal arguments then the compiler will try to convert the type of actual arguments to formal arguments if it is legal, Otherwise, a garbage value will be passed to the formal argument.
3. Changes made in the formal argument do not affect the actual arguments.

✓ The following program demonstrates this behaviour.

```
1  #include<stdio.h>
2  void func_1(int);
3
4  int main()
5  {
6       int x = 10;
7
8       printf("Before function call\n");
9       printf("x = %d\n", x);
10
11      func_1(x);
12
```

Function basics in C XXI

```
13      printf("After function call\n");
14      printf("x = %d\n", x);
15
16      // signal to operating system program ran fine
17      return 0;
18  }
19
20  void func_1(int a)
21  {
22      a += 1;
23      a++;
24      printf("\na = %d\n\n", a);
25  }
```

Local, Global and Static variables in C

Local Variables: The variables which are declared inside the function, compound statement (or block) are called Local variables.

```
1 void function_1()
2 {
3     int a, b; // you can use a and b within braces only
4 }
5
6 void function_2()
7 {
8     printf("%d\n", a); // ERROR, function_2() doesn't know any variable a
9 }
```

✓ Consider the following code:

Function basics in C XXIII

```c
#include<stdio.h>
int main()
{
     int a = 100;
     {
          /* variable a declared in this block is completely different from
     variable declared outside. */
          int a = 10;
          printf("Inner a = %d\n", a);
     }
     printf("Outer a = %d\n", a);
     return 0;
}
```

✓ You can use the same variable names in a different function and they will not conflict with each other. For example:

Function basics in C XXIV

```
1 void function_1()
2 {
3       int a = 1, b = 2;
4 }
5
6 void function_2()
7 {
8       int a = 10, b = 20;
9 }
```

Global Variables: The variables declared outside any function are called global variables.

- ✓ They are not limited to any function.
- ✓ Any function can access and modify global variables.
- ✓ Global variables are automatically initialized to 0 at the time of declaration.
- ✓ Global variables are generally written before main() function.

```c
#include<stdio.h>
void func_1();
void func_2();
int a, b = 10;  // declaring and initializing global variables

int main()
{
     printf("Global a = %d\n", a);
     printf("Global b = %d\n\n", b);

     func_1();
     func_2();
     return 0;
}
```

Function basics in C XXVI

```
15
16 void func_1()
17 {
18      printf("From func_1() Global a = %d\n", a);
19      printf("From func_1() Global b = %d\n\n", b);
20 }
21
22 void func_2()
23 {
24      int a = 5;
25      printf("Inside func_2() a = %d\n", a);
26 }
```

Static variables: A Static variable is able to retain its value between different function calls.

✓ The static variable is only initialized once, if it is not initialized, then it is automatically initialized to 0.

Here is how to declare a static variable.

```
#include<stdio.h>
void func_1();
int a, b = 10;
int main()
{
     func_1();
     func_1();
     func_1();

     return 0;
}

void func_1()
{
     int a = 1;
```

Function basics in C XXVIII

```
16      static int b = 100;
17      printf("a = %d\n", a);
18      printf("b = %d\n\n", b);
19      a++;
20      b++;
21  }
```

Call by Value and Call by Reference in C I

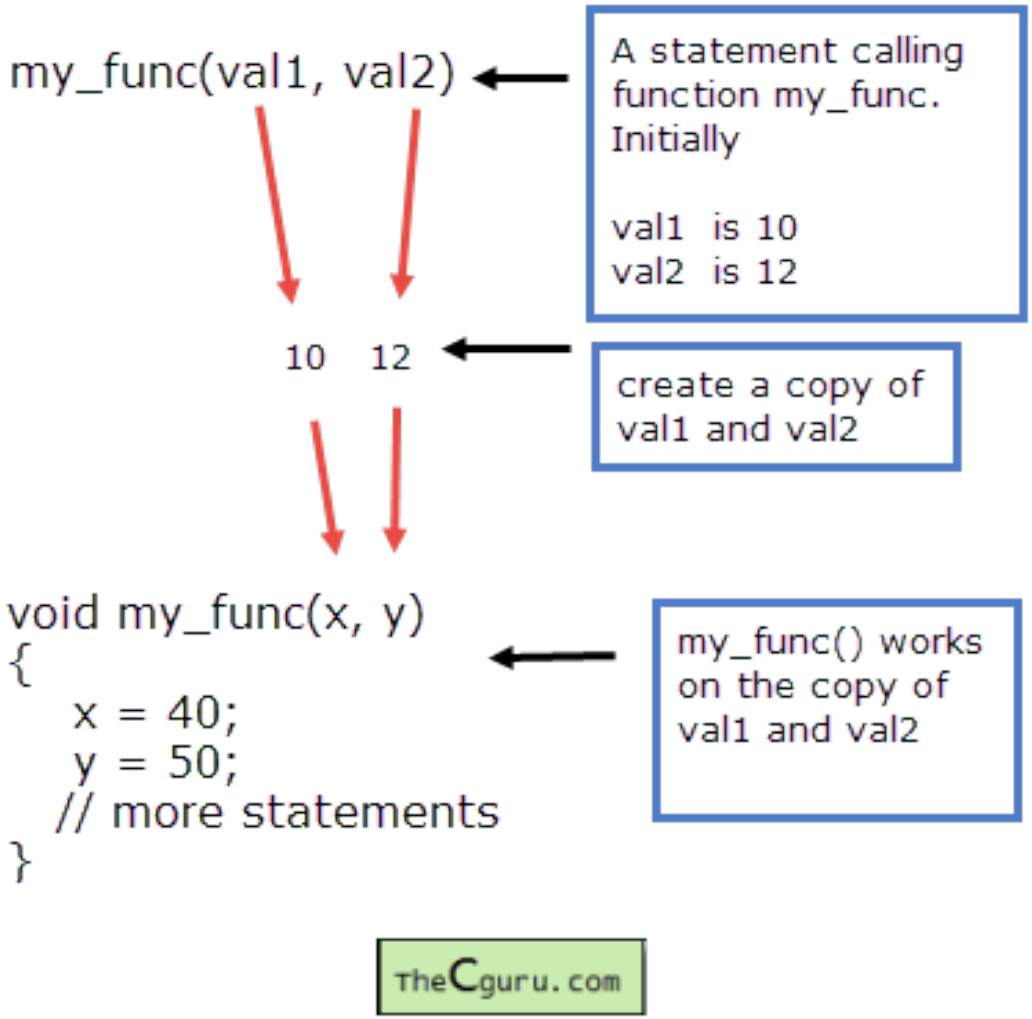
C provides two ways of passing arguments to a function.

1. Call by value or Pass by value.
2. Call by reference.

Call by Value:

✓ In this method a copy of each of the actual arguments is made first then these values are assigned to the corresponding formal arguments.

Call by Value and Call by Reference in C II



Call by Value and Call by Reference in C III

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

int main()
{
     int x = 10, y = 20;

     printf("Initial value of x = %d\n", x);
     printf("Initial value of y = %d\n", y);

     printf("\nCalling the function\n");

     try_to_change(x, y);

     printf("\nValues after function call\n\n");

     printf("Final value of x = %d\n", x);
     printf("Final value of y = %d\n", y);

     // signal to operating system program ran fine
     return 0;
```

Call by Value and Call by Reference in C IV

```
22 }
23
24 void try_to_change(int x, int y)
25 {
26      x = x + 10;
27      y = y + 10;
28
29      printf("\nValue of x (inside function) = %d\n", x);
30      printf("Value of y (inside function) = %d\n", y);
31 }
```

Call by reference:

✓ In this method addresses of the actual arguments are copied and then assigned to the corresponding formal arguments.

✓ Now formal and actual arguments both points to the same data (because they contain the same address).

✓ As a result, any changes made by called function also affect the actual arguments.

```c
#include<stdio.h>
void try_to_change(int *, int *);

int main()
{
     int x = 10, y = 20;
     printf("Initial value of x = %d\n", x);
     printf("Initial value of y = %d\n", y);

     printf("\nCalling the function\n");
     try_to_change(&x, &y);

     printf("\nValues after function call\n\n");

```

Call by Value and Call by Reference in C VI

```
15      printf("Final value of x = %d\n", x);
16      printf("Final value of y = %d\n", y);
17
18
19      return 0;
20  }
21  void try_to_change(int *x, int *y)
22  {
23      (*x)++;
24      (*y)++;
25      printf("\nValue of x (inside function) = %d\n", *x);
26      printf("Value of y (inside function) = %d\n", *y);
27  }
```

Returning more than one value from function in C I

✓ A single return statement can only return one value from a function.

Create a function to return sum, difference and product of two numbers passed to it.

```c
#include<stdio.h>
void return_more_than_one(int a, int b, int *sum, int *diff, int *prod);

int main()
{
     int x = 40, y = 10, sum, diff, prod;

     return_more_than_one(x, y, &sum, &diff, &prod);

     printf("%d + %d = %d\n",x, y, sum);
     printf("%d - %d = %d\n",x, y, diff);
     printf("%d * %d = %d\n",x, y, prod);

     // signal to operating system program ran fine
     return 0;
}

void return_more_than_one(int a, int b, int *sum, int *diff, int *prod)
{
```

Returning more than one value from function in C II

```
20      *sum = a+b;
21      *diff = a-b;
22      *prod = a*b;
23  }
```

Returning a Pointer from a Function in C I

✓ A function can return data of types int , float, char etc.

✓ Similarly, a function can return a pointer to data. The syntax of a function returning a pointer is as follows.

Syntax: type *function_name(type1, type2, ...);

```
1 int *func(int, int); // this function returns a pointer to int
2
3 double *func(int, int); // this function returns a pointer to double
```

The following program demonstrates how to return a pointer from a function.

```
1  #include<stdio.h>
2  int *return_pointer(int *, int); // this function returns a pointer of type int
3
4  int main()
5  {
6       int i, *ptr;
7       int arr[] =
8       {
9            11, 22, 33, 44, 55
10      }
11      ;
```

Returning a Pointer from a Function in C II

```
12      i = 4;
13
14      printf("Address of arr = %u\n", arr);
15
16      ptr = return_pointer(arr, i);
17
18      printf("\nAfter incrementing arr by 4 \n\n");
19
20      printf("Address of ptr = %u\n\n" , ptr);
21      printf("Value at %u is %d\n", ptr, *ptr);
22
23      // signal to operating system program ran fine
24      return 0;
25  }
26
27  int *return_pointer(int *p, int n)
28  {
29      p = p + n;
30      return p;
31  }
```

Returning a Pointer from a Function in C III

Never return a pointer to local variable from a function.

```
#include<stdio.h>
int *abc(); // this function returns a pointer of type int

int main()
{
     int *ptr;
     ptr = abc();
     return 0;
}

int *abc()
{
     int x = 100, *p;
     p = &x;
     return p;
}
```

Returning a Pointer from a Function in C IV

✓ In the function abc() we are returning a pointer to the local variable. Recall that a local variable exists only inside the function and as soon as function ends the variable x cease to exists, so the pointer to it is only valid inside the function abc().

✓ Even though the address returned by the abc() is assigned to ptr inside main(), the variable to which ptr points is no longer available. On dereference the ptr you will get some garbage value.

Note: Sometimes you may even get the correct answer i.e 100, but you must never rely on this behaviour.

Passing 1-D Array to a Function in C I

```
int my_arr[5] = [11,44,66,90,101];
```

1st way:

```
void function(int a[]) // here the size of the array is omitted
{
     // statements;
}
```

2nd way:

```
void function(int a[5]) // here the size of the array is specified
{
     // statements;
}
```

3rd way:

```
void function(int *a)
{
     // statements;
}
```

Passing 1-D Array to a Function in C II

✓ Essentially in all the three cases base type of a is a pointer to int or (int *), we are just using three different ways to represent them.

```
#include<stdio.h>
void new_array(int a[]);
int main()
{
     int my_arr[] =
     {
          1,4,9,16,23
     }
     , i;
     printf("Original array: \n\n");
     for(i = 0; i < 5; i++)
     {
          printf("%d ", my_arr[i]);
     }
     my_func(my_arr);
     printf("\n\nModified array : \n\n");
     for(i = 0; i < 5; i++)
     {
          printf("%d ", my_arr[i]);
```

Passing 1-D Array to a Function in C III

```
20      }
21      return 0;
22  }
23  void my_func(int a[5])
24  {
25      int i;
26      // increment original elements by 5
27      for(i = 0; i < 5; i++)
28      {
29          a[i] = a[i] + 5;
30      }
31  }
```

Passing 1-D Array to a Function in C IV

✓ We know that my_arr is a pointer to the first element of the array. So we can pass my_arr to the function my_func() without using & operator.

✓ In line 15, the my_func() is called with an actual argument of my_arr which is then assigned to a.

✓ Again note that we are passing the address of my_arr to a, that means we are using call by reference instead of call by value.

✓ So now both my_arr and a points to the same array.

✓ Inside the function, we are using for loop to increment every element of the array by 5.

✓ Since we are operating on the original array all the changes made here effect the original array.

Passing 2-D Array to a Function in C I

Just like a 1-D array, when a 2-D array is passed to a function, the changes made by function effect the original array.

```
int two_d[2][3] =
{

     {
          99,44,11
     }
     ,

     {
          4,66,9
     }

}
;
```

1st way:

Passing 2-D Array to a Function in C II

```
void function(int a[][3])
{
     // statements;
}
```

2nd way:

```
void function(int a[2][3])
{
     // statements;
}
```

3rd way:

```
void function(int (*a)[3])
{
     // statements;
}
```

Passing 2-D Array to a Function in C III

```
#include<stdio.h>
void change_twod(int (*a)[3]);
int main()
{
     int i,j, two_d[2][3] =
     {

          {
               99,44,11
          }
          ,

          {
               4,66,9
          }

     }
     ;
     printf("Original array: \n\n");
     for(i = 0; i < 2; i++)
     {
```

Passing 2-D Array to a Function in C IV

```
22          for(j = 0; j < 3; j++)
23          {
24               printf("%3d ", two_d[i][j]);
25          }
26          printf("\n");
27     }
28     change_twod(two_d);
29     printf("\n\nModified array : \n\n");
30
31     for(i = 0; i < 2; i++)
32     {
33          for(j = 0; j < 3; j++)
34          {
35               printf("%3d ", two_d[i][j]);
36          }
37          printf("\n");
38     }
39     return 0;
40 }
41
42 void change_twod(int (*arr)[3])
```

Passing 2-D Array to a Function in C V

```
43 {
44      int i, j;
45      printf("\n\nIncrementing every element by 5\n");
46      // increment original elements by 6
47      for(i = 0; i < 2; i++)
48      {
49           for(j = 0; j < 3; j++)
50           {
51                arr[i][j] = arr[i][j] + 5;
52           }
53      }
54 }
```

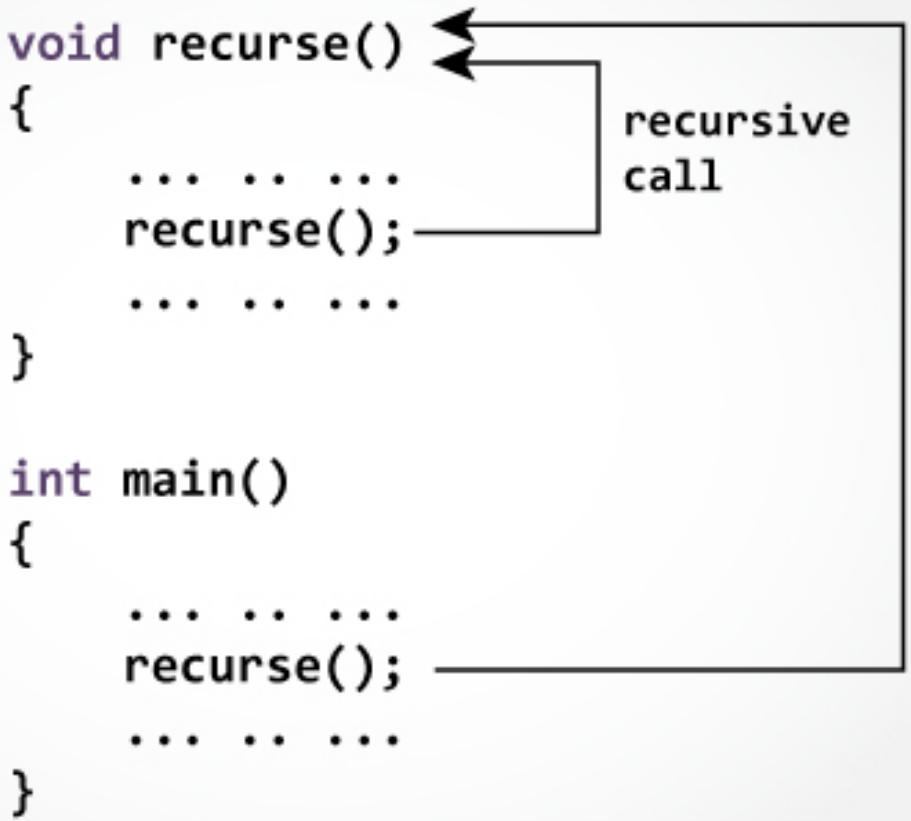
Recursive Function in C I

✓ In C, a function can call itself. This process is known as recursion.

```
int main()
{
     recursive();
     ...
     return 0;
}

void recursive()
{
     statement 1;
     ...
     recursive();
}
```

Recursive Function in C II

How does recursion work?



Recursive Function in C III

✓ As you can guess this process will keep repeating indefinitely. So, in a recursive function, there must be a terminating condition to stop the recursion. This condition is known as the base condition.

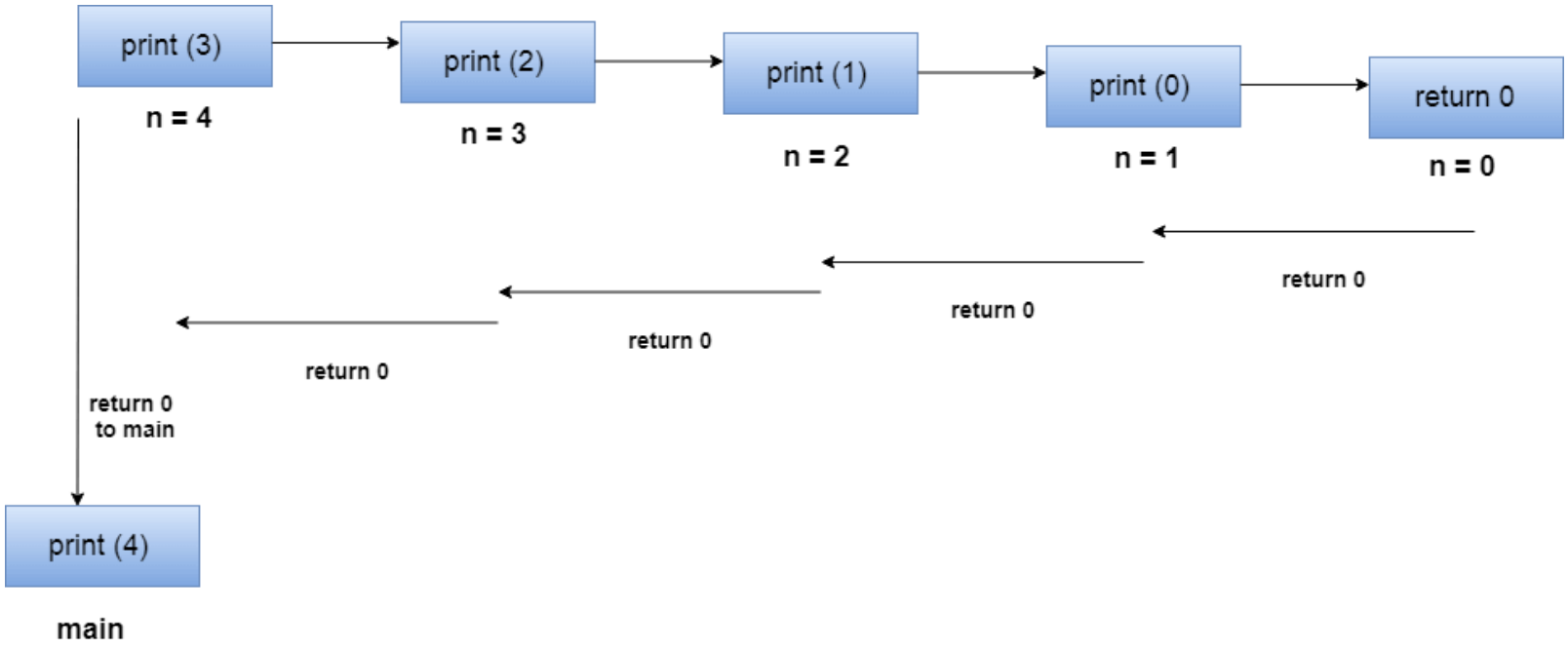
```
int main()
{
     callme();
}

void callme()
{
     if(base_condition)
     {
          // terminating condition
     }
     statement 1;
     ...
     callme();
```

Understand the memory allocation of the recursive functions.

Recursive Function in C IV

```c
int display (int n)
{
     if(n == 0)
     return 0; // terminating condition
     else
     {
          printf("%d",n);
          return display(n-1); // recursive call
     }
}
```

Recursive Function in C V



Stack tracing for recursive function call

Recursive Function in C VI

The recursive function works in two phases:

Winding phase. Unwinding phase.

Winding phase: In Winding phase, the recursive function keeps calling itself. This phase ends when the base condition is reached.

Unwinding phase: When the base condition is reached, unwinding phase starts and control returns back to the original call.

Recursive Function in C VII

Example 1:

```c
#include<stdio.h>
void rec(int );
int main()
{
      rec(1);
     return 0;
}
void rec(int n)
{
     printf("Winding phase: Level = %d\n", n);
     if(n<3)
     {
          rec(n+1);
     }
     printf("Unwinding phase: Level = %d\n", n);
}
```

Example 2: calculates the factorial of a number using recursion.

```c
#include<stdio.h>
int factorial(int n);
int main()
{
     int n;
     printf("Enter a number: ");
     scanf("%d", &n);
     printf("%d! = %d", n, factorial(n));
     return 0;
}
int factorial(int n)
{
     if(n == 0) // base condition
     return 1;
     else
     return n * factorial(n-1);
}
```

Recursive Function in C IX

How it works:

Let's say we want to calculate factorial of 5.

main() calls factorial(5)

since 5 != 0 - factorial(5) calls factorial(4)

since 4 != 0 - factorial(4) calls factorial(3)

since 3 != 0 - factorial(3) calls factorial(2)

since 2 != 0 - factorial(2) calls factorial(1)

since 1 != 0 - factorial(1) calls factorial(0)

✓ When factorial() is called with n = 0, if condition becomes true and recursion stops and control returns to factorial(1). From now on every called function will return a value to the previous function in reverse order of function calls.

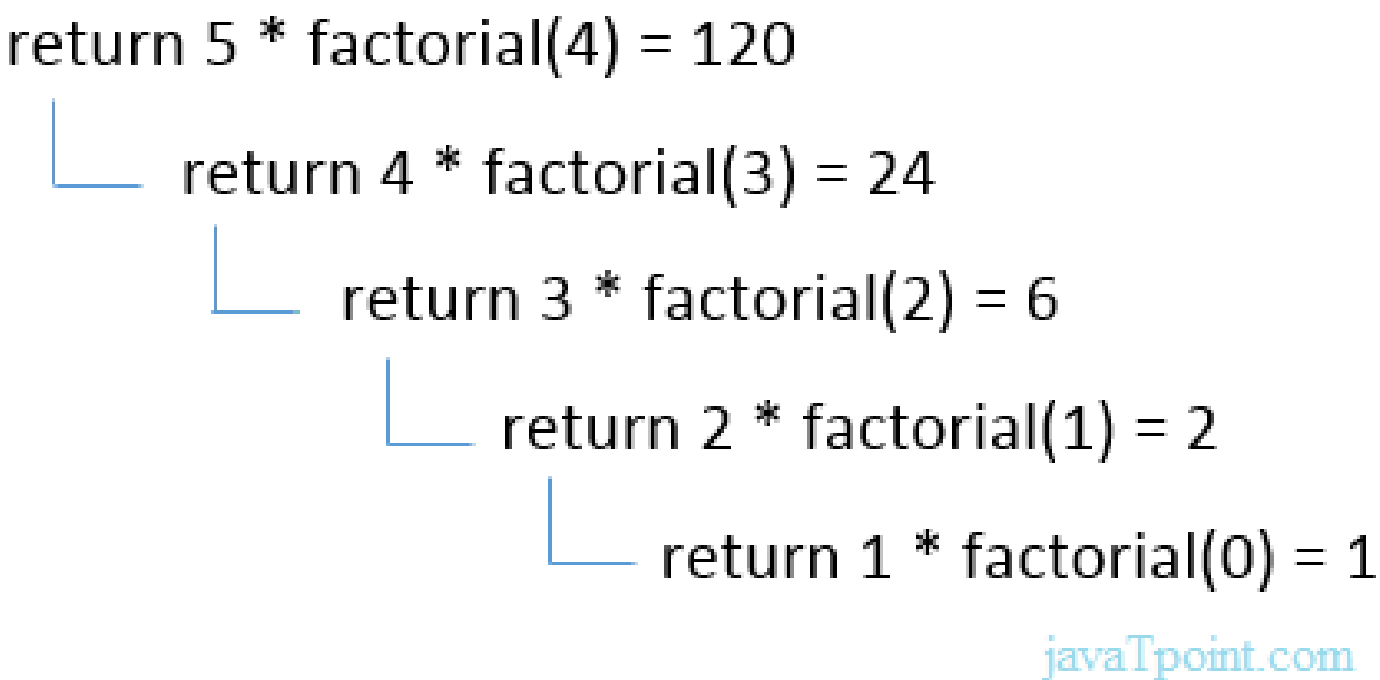
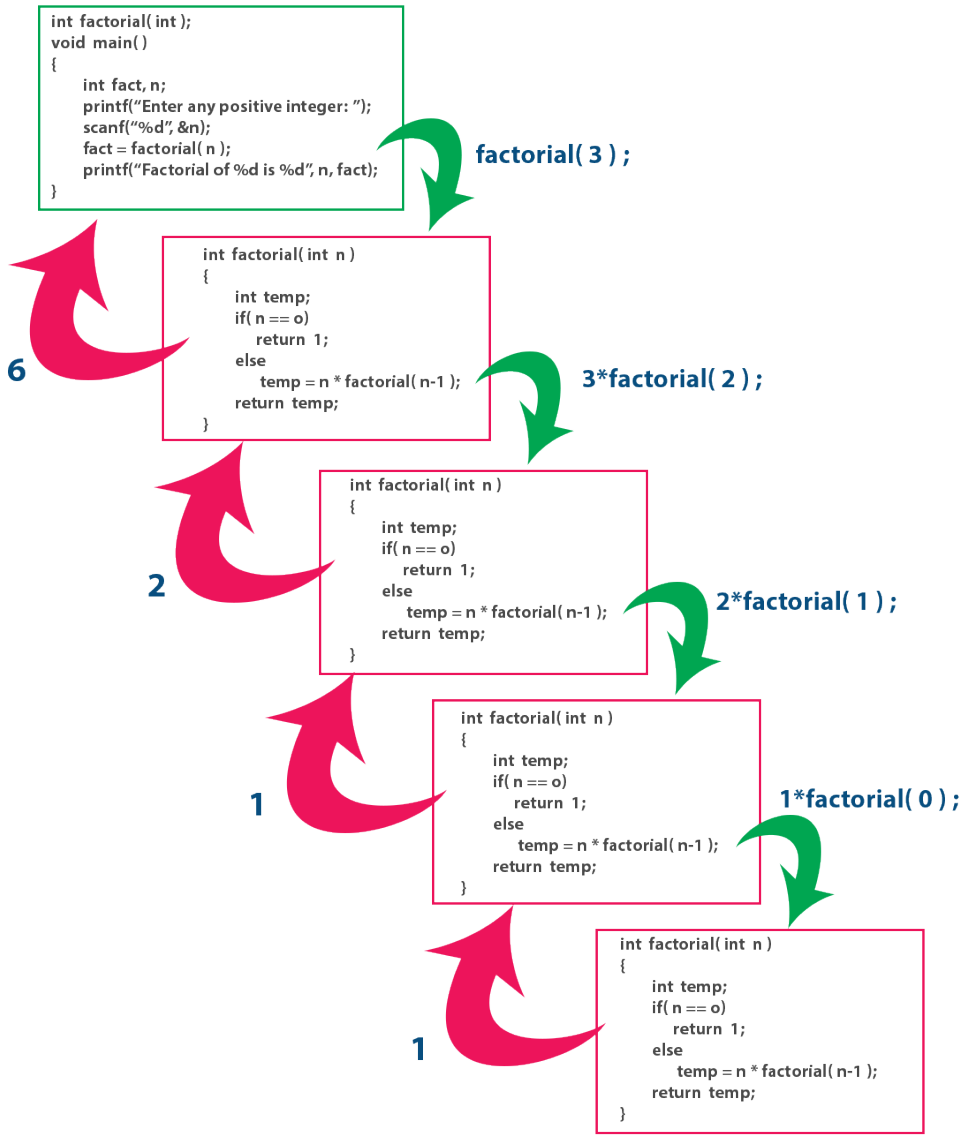
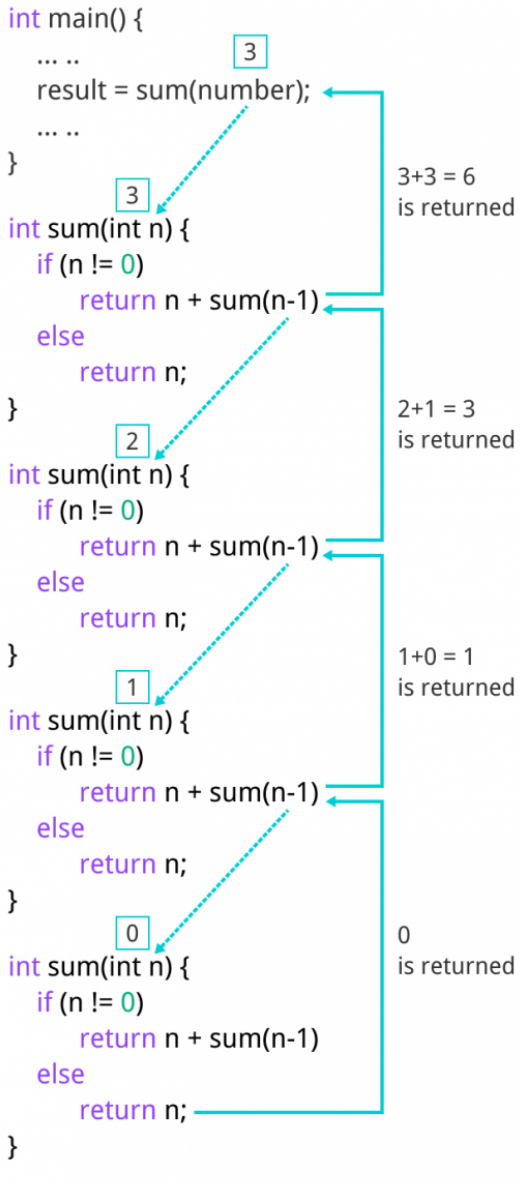


Fig: Recursion

Recursive Function in C XI



Recursive Function in C XII



Example 3: calculates the power of a number using recursion.

```c
#include<stdio.h>
int power(int base, int exp);
int main()
{
     int base, exp;

     printf("Enter base: ");
     scanf("%d", &base);

     printf("Enter exponent: ");
     scanf("%d", &exp);

     printf("%d ^ %d = %d", base, exp, power(base, exp));


     return 0;
}

int power(int base, int exp)
{
```

Recursive Function in C XIV

```
21      if(exp == 0) // base condition
22      {
23           return 1;
24      }
25
26      else
27      {
28           return base * power(base, exp - 1);
29
30      }
31 }
```

C Program to convert a decimal number to binary, octal and hexadecimal using recursion

```c
#include<stdio.h> // include stdio.h library
void convert_to_x_base(int, int);

int main(void)
{
     int num, choice, base;

     while(1)
     {
          printf("Select conversion: \n\n");
          printf("1. Decimal to binary. \n");
          printf("2. Decimal to octal. \n");
          printf("3. Decimal to hexadecimal. \n");
          printf("4. Exit. \n");

          printf("\nEnter your choice: ");
          scanf("%d", &choice);

          switch(choice)
```

Recursive Function in C XVI

```
20          {
21              case 1:
22              base = 2;
23              break;
24              case 2:
25              base = 8;
26              break;
27              case 3:
28              base = 16;
29              break;
30              case 4:
31              printf("Exiting ...");
32              exit(1);
33              default:
34              printf("Invalid choice.\n\n");
35              continue;
36          }
37
38          printf("Enter a number: ");
39          scanf("%d", &num);
40
```

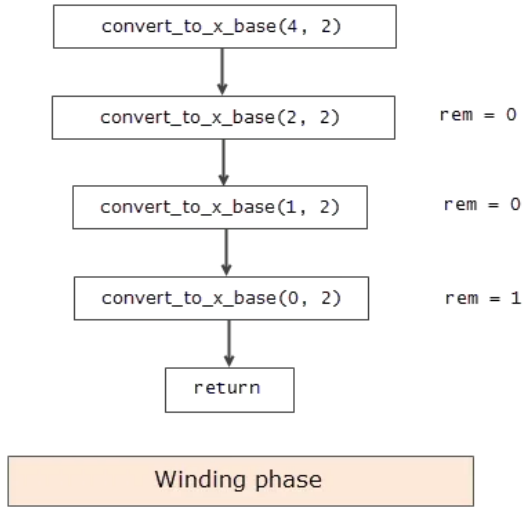
Recursive Function in C XVII

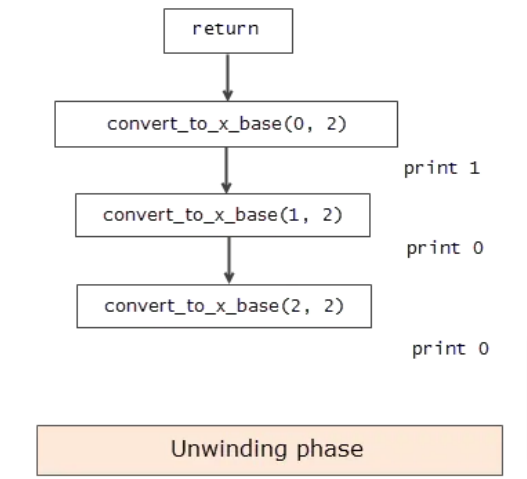
```
41            printf("Result = ");
42
43            convert_to_x_base(num, base);
44
45            printf("\n\n");
46        }
47        return 0; // return 0 to operating system
48    }
49    void convert_to_x_base(int num, int base)
50    {
51        int rem;
52        // base condition
53        if (num == 0)
54        {
55            return;
56        }
57        else
58        {
59            rem = num % base; // get the rightmost digit
60            convert_to_x_base(num/base, base);  // recursive call
61            if(base == 16 && rem >= 10)
```

Recursive Function in C XVIII

```
62              {
63                   printf("%c", rem+55);
64              }
65              else
66              {
67                   printf("%d", rem);
68              }
69         }
70   }
```

Recursive Function in C XIX





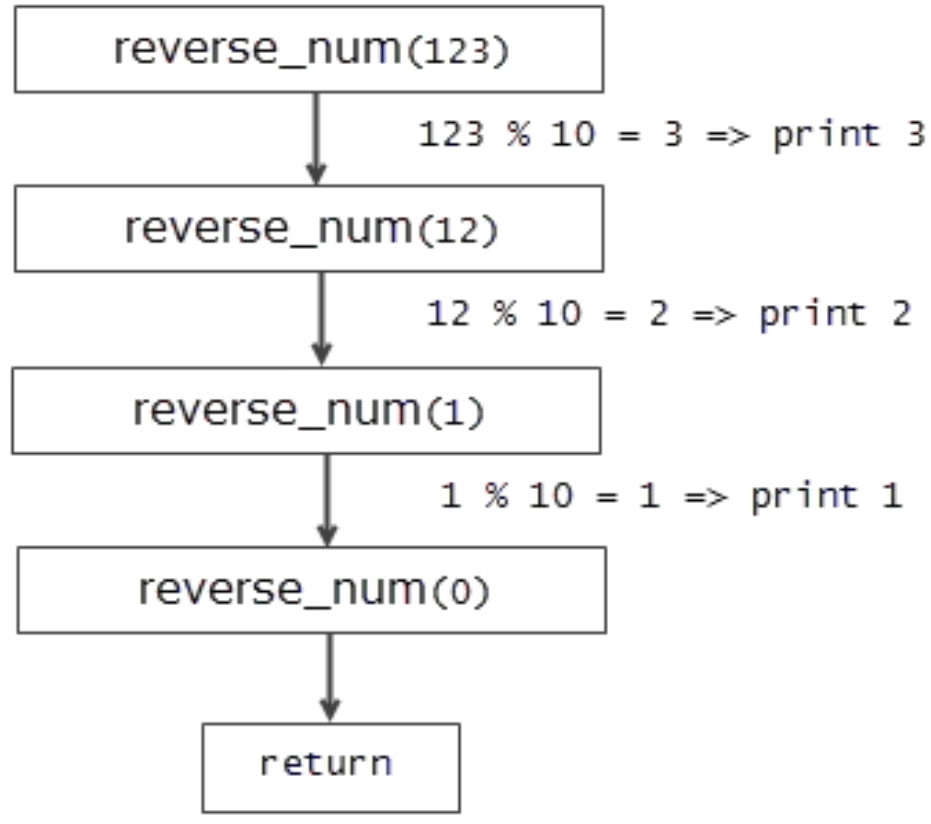
C Program to reverse the digits of a number using recursion

```c
#include<stdio.h> // include stdio.h library
void reverse_num(int num);
int main(void)
{
     int num;
     printf("Enter a number: ");
     scanf("%d", &num);
     reverse_num(num);

     return 0; // return 0 to operating system
}
void reverse_num(int num)
{
     int rem;

     // base condition
     if (num == 0)
     {
          return;
     }
```

Recursive Function in C XXI

```
21
22      else
23      {
24           rem = num % 10; // get the rightmost digit
25           printf("%d", rem);
26           reverse_num(num/10);  // recursive call
27      }
28 }
```

Recursive Function in C XXII



Evaluation of reverse_num(123)

Static function I

1. Limit the scope/ visibility of a function in a C program.
2. Reuse of the same function name in other source files

✓ A static function in C is a function that has a scope that is limited to its object file. This means that the static function is only visible in its object file.

✓ A function can be declared as static function by placing the static keyword before the function name.

✓ In C language programming , static function is also used to avoid ambiguity. If we have the same function name in different source files in an application.

An example that demonstrates this is given as follows –

There are two files first_file.c and second_file.c. The contents of these files are given as follows –

Contents of first_file.c

```
1 static void staticFunc(void)
2 {
3         printf("Inside the static function staticFunc() ");
4 }
```

Static function II

Contents of second_file.c

```
1 int main()
2 {
3         staticFunc();
4         return 0;
5 }
```

✓ Now, if the above code is compiled then an error is obtained i.e "undefined reference to staticFunc()". This happens as the function staticFunc() is a static function and it is only visible in its object file.

✓ A program that demonstrates static functions in C is given as follows–

Static function III

```c
#include <stdio.h>
static void staticFunc(void)
{
        printf("Inside the static function staticFunc() ");
}
int main()
{
        staticFunc();
        return 0;
}
```

https://overiq.com/c-examples/