Chapter 8: Functions

What are we studying in this chapter?

- ♦ Library functions, User defined functions, function prototype
- ♦ Argument passing: Call by value, Call by address
- ♦ Categories of functions, Functions and program structure
- ♦ Location of functions

8.1 Introduction

In this chapter, we discuss a very important concept in C called *functions.* Functions are the main building blocks of all C programs. So, we should learn functions very well. Before we discuss something about functions, let us see "*What is a function*? *What are the different types of functions*?"

Definition: A large program can be divided into manageable pieces called modules where each module does a specific task. Thus, each module also called a *function* is self-contained small program called program segment that carry out some specific, well-defined task. Functions act like building blocks using which any desired activity can be performed by combining one or more functions. The functions can be classified into two categories as shown below:

$$\begin{tikzcd}
 & \text{Library functions/Predefined functions} \\
\text{Functions} \arrow[ur] \arrow[dr] & \\
 & \textbf{U}\text{ser }\textbf{D}\text{efined }\textbf{F}\text{unctions (UDFs)}
\end{tikzcd}$$

(also called Programmer Defined Functions)

8.2 Library/pre-defined functions

Now, let us see formally *what are library functions and what are the various types of libraries that are available as part of C compiler.*

Definition: C library that comes with C compiler has a collection of various functions which perform standard and pre-defined tasks. These functions written by designers of C compilers are called *library functions.* The implementation details of these functions are not known to the programmers. The library functions are also called *pre-defined* or *built-in* or *in-built* or *standard library functions.* The library functions are ready to be used in our programs. For example,

- ♦ pow(x,y) - computes x^y (read as *x* to the power of *y*)
- ♦ sqrt(x) - computes square root of *x*
- ♦ printf() - Used to print the data on the screen
- ♦ scanf() - Used to read the data from the keyboard

8.2 Functions

Note: The built-in functions that does various activities are shown in APPENDIX B

Example 8.1: Program to demonstrate the usage of library function

PROGRAM / **TRACING**

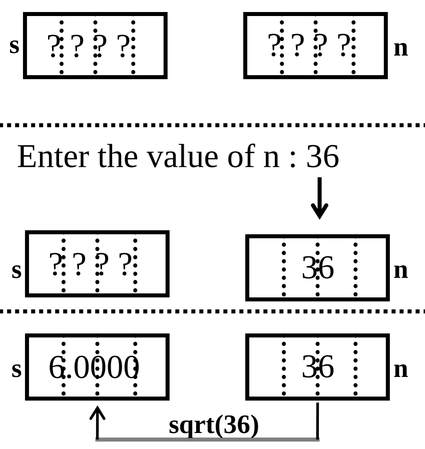
```
#include <stdio.h>
#include <math.h>

void main()        (1)      // Execution starts from main
{
        float s, n;
```

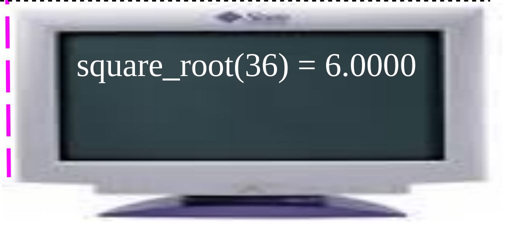
```
(2)     printf("Enter the value of n:");
        scanf("%f",&n);
```

```
Enter the value of n : 36
```

```
(3)     s = sqrt(n);
```

```
(4)     printf("square_root(%d) = %f\n", n, s);

}
```





```
square_root(36) = 6.0000
```

The various *built in functions* or *library functions* that are used in the above program are shown below:

- *scanf()* : Used to read the data from the keyboard.
- *printf()* : Used to print the result on the screen
- *sqrt()* : Used to find square root of a given number

Note: The declaration of *scanf()* and *printf()* functions are available in the header file "stdio.h" and the declaration of *sqrt()* function is available in header file "math.h"
The user defined functions are:

- *main()* : *It is a user defined function.* It controls and co-ordinates all the above activities by executing each statement in the sequence provided:

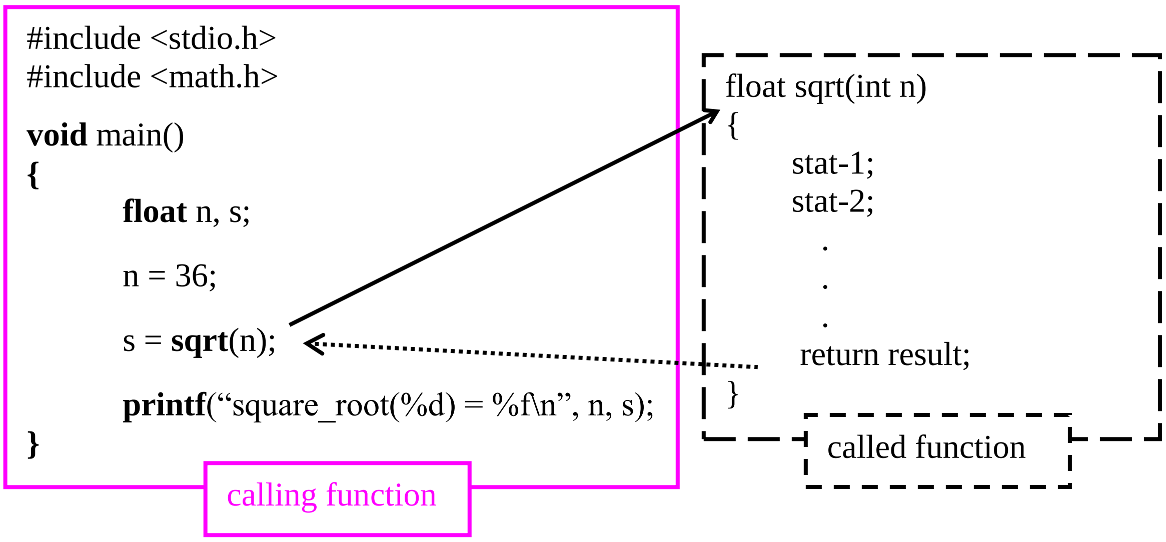
(1) Execution starts from the function main()

(2) The value 36 entered through keyboard is copied into *n*.

3 Invoke the function **sqrt()** and obtain the square root of *n* and store it in the variable *s*.

4 Display the square root of 36 i.e., 6, using the function **printf()**.

Now, let us see *what will happen when a function **sqrt()** is invoked.* For the explanation purpose consider the program shown below:



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

void main()
{
        float n, s;

        n = 36;

        s = sqrt(n);

        printf("square_root(%d) = %f\n", n, s);
}
```

```
float sqrt(int n)
{
        stat-1;
        stat-2;
          .
          .
          .
        return result;
}
```

Here, we have not written the function *sqrt()* which is shown using dotted box. This is because, it is a library function written by the compiler developers. So, we do not know what statements they have used to compute square-root. But, we know that by invoking the function sqrt(), we can get the result. Some points to remember are:

- Using the statement:

  s = sqrt(n);

  we can invoke the function. We say that the function sqrt() is *invoked* or *called*. So, the function sqrt() is often called *"called function".*
- Who is calling the function sqrt()? It is clear from the above program that the function *main()* is calling. So, the function *main()* is called *"calling function".*
- When the function sqrt() is called, control is transferred from main() to sqrt() as shown using solid arrow. Each statement in the function sqrt() is executed and *square root* of given number is computed.
- After executing the last statement i.e., *return result*, the control will be transferred to the function *main()* along with the result. This is shown by drawing dotted line from sqrt() to main().
- The result obtained from *sqrt()* is copied into variable *s*.
- The result is displayed on the screen.

Now, we can easily define a <u>*called function*</u> and a <u>*calling function*</u>.

Definition: A function that is invoked by writing the name of the function is called "called function" and the invoking function is called "calling function".

For example, if *main()* function invokes *sqrt()* function, the function *main()* is *calling function* and function *sqrt()* is *called function*.

8.3 User-defined functions

Now, let us see *what are <u>U</u>ser <u>D</u>efined <u>F</u>unctions (UDFs)?*

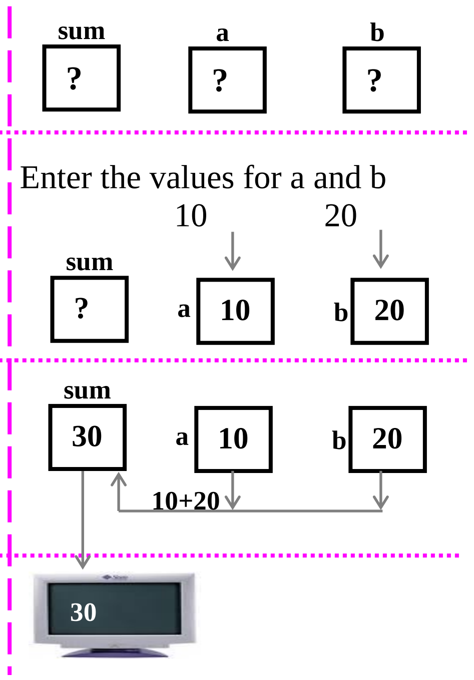
Definition: The functions written by the programmer/user to do the specific tasks are called *user defined functions (UDFs).* They are also called *programmer defined functions (PDFs).*

Let us write a user-defined function to add two numbers. For this, consider the following program to add two numbers:

Example 8.2: Program to add two numbers

```
#include <stdio.h>

void main()
{
        int     sum, a, b;

        printf("Enter the values for a and b\n");
        scanf("%d %d",&a, &b);

        sum = a + b;

        printf("%d", sum);
}
```

<u>TRACING</u>



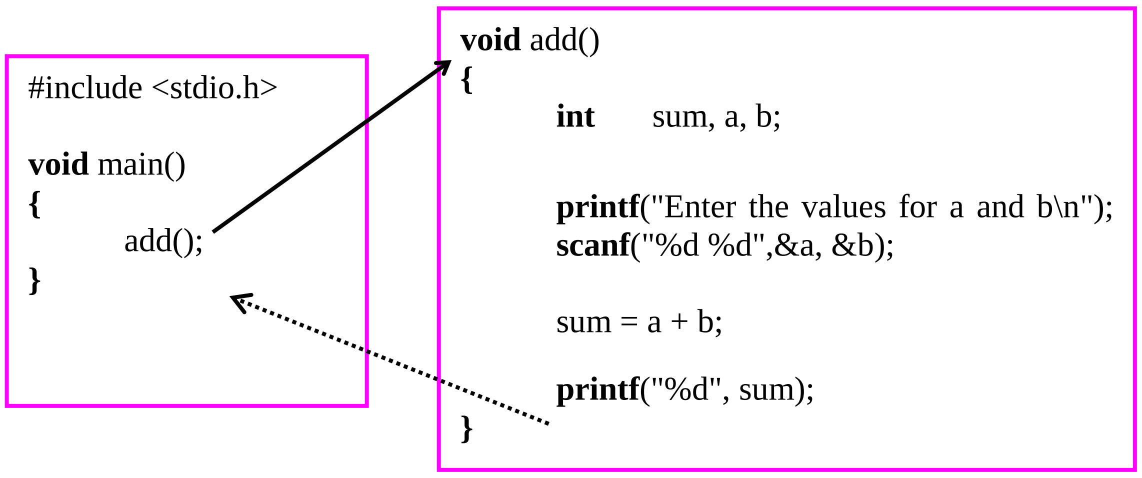
```
Enter the values for a and b
           10          20
```

Now, let us see *what does the above program do.* It is very simple. It accepts two numbers, adds them and displays the result. Now, *"Can we write a function to add* *two numbers?"* Yes, we can!!! The above program itself is a function whose name is *main()*. If we want to write our own function, then just change the name of the function *main()* to *add()* for better readability as shown below:

```
#include <stdio.h>

void add()      // Note: Here, main is changed to add
{
        int     a, b, sum;

        printf("Enter the values for a and b\n");
        scanf("%d %d",&a, &b);

        sum = a + b;

        printf("%d", sum);
}
```

But, the above program is not complete. We know that execution always starts from the function *main()* and so, we have to write one function *main()* which invoke the function *add()* as shown below:

Example 8.3: Program to add two numbers using the function



```
#include <stdio.h>

void main()
{
        add();
}
```

```
void add()
{
        int     sum, a, b;

        printf("Enter the values for a and b\n");
        scanf("%d %d",&a, &b);

        sum = a + b;

        printf("%d", sum);
}
```

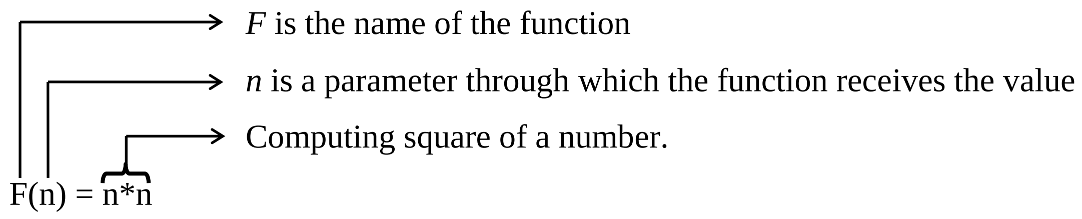
The sequence of operations that are performed when above program is executed are:

- Execution starts from function *main()*
- The function add() is invoked
- Control is transferred from function *main()* to the function *add()* as shown using thick arrow mark.
- The function *add()* accepts two numbers, adds those numbers and print the result

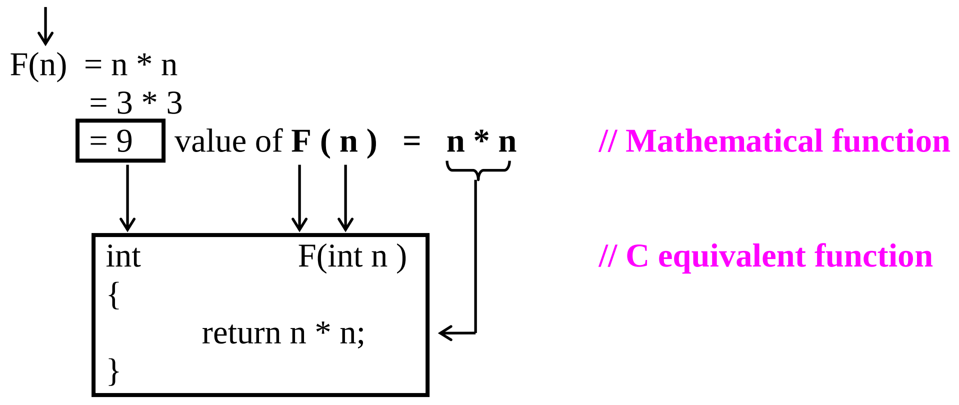
- Then, control is transferred from function *add()* to function *main()* as shown using dotted arrow.
- Since there are no statements to be executed in the function main, the program is terminated and control returns to the operating system.

8.3.1 Find square of a number

Now, let us write a user-defined function to find the square of a given number. In mathematics, we can compute square of a given number using the following function.


F is the name of the function
n is a parameter through which the function receives the value
Computing square of a number.
F(n) = n*n

This mathematical function can be evaluated and converted into its equivalent C function as shown below:
Let n = 3



```
int          F(int n )
{
        return n * n;
}
```

Note: For better readability and understanding purpose we can replace the function name F by the name SQUARE and change the type from **int** to **double.** The resulting function is shown below:

Example 8.4: Main program to compute square of *n*. **Note:** sizeof(double) = 8

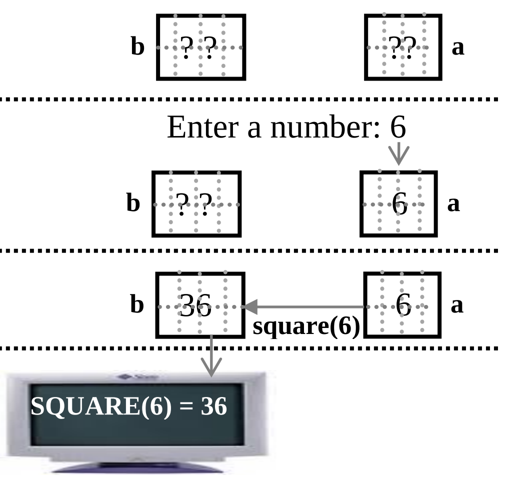
```
double SQUARE (double  n)       // accept n as the parameter
{
        return   n*n;           // return n²
}
```

The *main* function that computes and prints square of a given number is shown below:

Example 8.5: Main program to compute square of *n*. **Note:** sizeof(double) = 8

```
#include <stdio.h>

/* Include: Example 8.4: To compute square of a number  */

void main()
{
        double      a, b;

        printf("Enter a number :");
        scanf("%lf",&a);

        b = SQUARE(a);

        printf("SQUARE(%lf) = %lf\n", a, b);
}
```

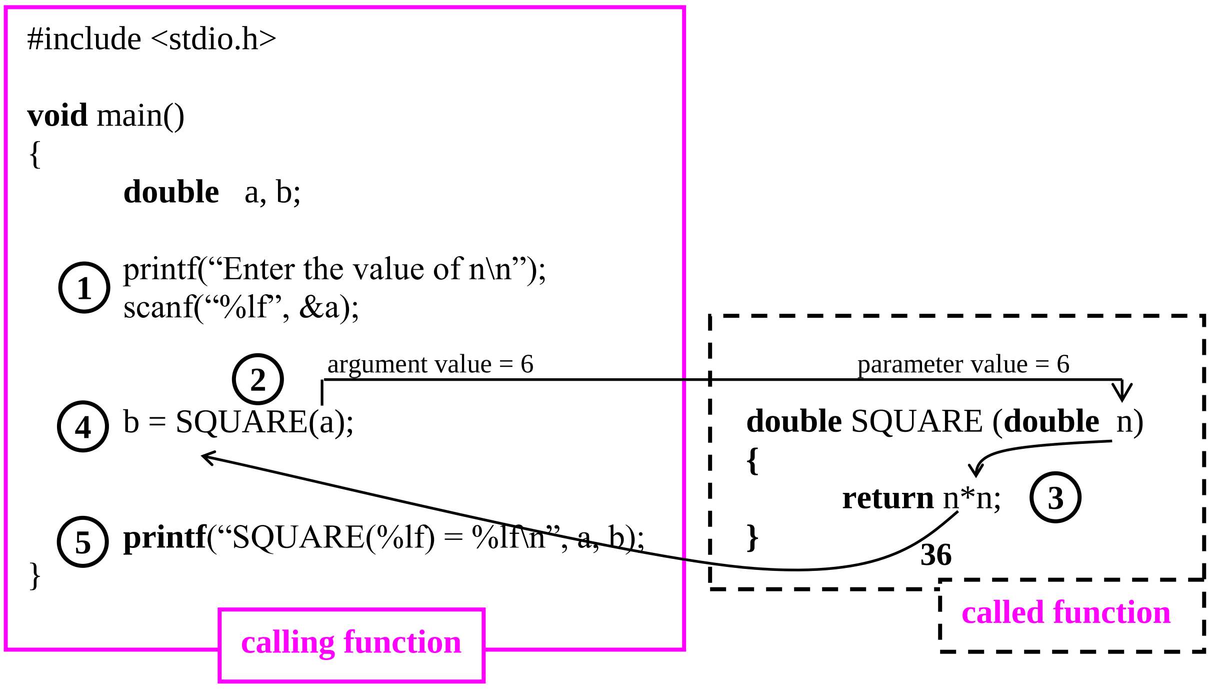
Tracing



```
SQUARE(6) = 36
```

Now, let us understand how the function SQUARE is invoked from the function *main* and how the result is returned back to the function. For example, the various activities that are carried out while executing the following program are shown below:

Example 8.6: Program to demonstrate control flow



```
#include <stdio.h>

void main()
{
        double   a, b;

   ①   printf("Enter the value of n\n");
        scanf("%lf", &a);

   ④   b = SQUARE(a);

   ⑤   printf("SQUARE(%lf) = %lf\n", a, b);
}
```

```
double SQUARE (double  n)
{
        return n*n;   ③
}
```

1 When *printf()* and *scanf()* are executed, the user enters the value for variable *a*. Let *a* = 6

2 When the statement:

```
b = SQUARE(a);
```

is executed, the function SQUARE is invoked with argument *a* whose value is 6. Since the function *main*() calls the function SQUARE(), the *main function is called "calling function"* whereas the *function SQUARE() is called "Called function"* The argument value *6* is received by the parameter *n*. So, parameter value *n* is 6

3 The parameter value of n = 6 is used in expression n*n and compute square of 6 whose value is 36. The result 36 is the value of SQUARE(n) which in turn will be the value of SQUARE(a)

4 The value of SQUARE(a) i.e., 36 is copied into *b*. Thus, variable *b* holds the square of *a*.

5 SQUARE(6) = 36 is displayed on the screen and function will be terminated and control is returned to operating system.

8.3.2 Find cube of a number

Now, let us "Design a function to find the cube of given number" The following mathematical function computes and returns the cube of a number:

$$\boxed{F(n) = n * n * n}$$

The above function can be converted into equivalent C function (The design is exactly similar to the way we found square of a number in the previous section) as shown below:

Example 8.7: C function definition to find cube of a number

```
double CUBE (double  n)    // accept n as the parameter
{
        return n*n*n;          // return n^3
}
```

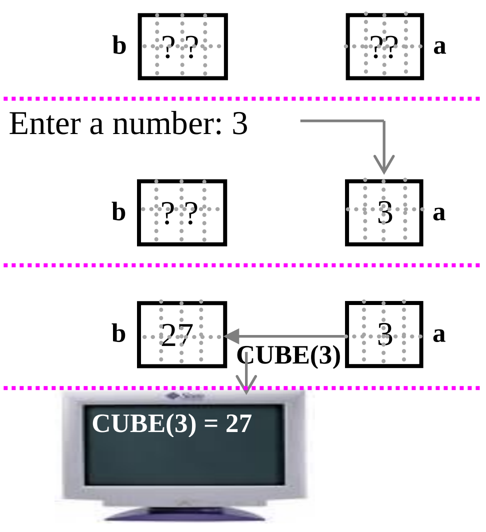
The C program that calls above function and prints cube of a given number is shown below:

Example 8.8: C program to print cube of a number using user defined function

```
#include <stdio.h>
/* Include: Example 8.7: To compute cube of a number  */
```

```
void main()
{
        double          b, a;
        printf("Enter a number :");
        scanf("%lf",&a);

        b = CUBE(a);

        printf("CUBE(%lf) = %lf\n", a, b);
}
```

Tracing



8.3.3 Function to find sum of two numbers

Now, let us design a function to add two numbers. The following mathematical function computes sum of two numbers:

$$\boxed{F(a, b) = a + b}$$

Using the above function, we can write a user defined function as shown below:

Example 8.9: Main program to add two numbers

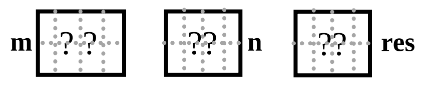
```
double ADD (double  a, double b) // accept two integer  parameters
{
        return    a + b;                  // return sum of two numbers
}
```

The C program that uses the above function to add two numbers is shown below:

Example 8.10: Main program to add two numbers

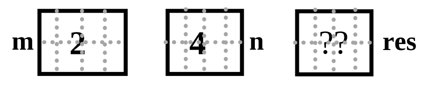
```
#include <stdio.h>

/* Include: Example 8.9: To compute sum of two numbers  */

void main()
{
        double          m, n, res;
```

Tracing

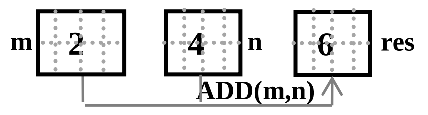


```
        printf("Enter two numbers:");
        scanf("%lf  %lf",&m, &n);
```

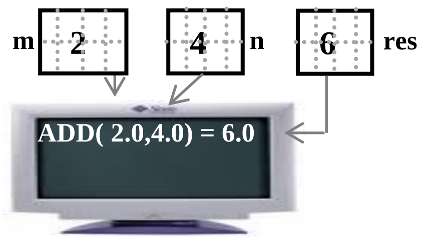
```
Enter two numbers: 2  4
```



```
        res = ADD(m, n);
```



```
        printf("ADD(%lf, %lf) = %lf\n", m, n,res);
}
```



```
ADD( 2.0,4.0) = 6.0
```

Once we know how to write user-defined functions, let us concentrate on various elements of user-defined functions.

8.4 Elements of user-defined functions

In this section, let us see *"What are the elements of user-defined functions?"* The three elements of user-defined functions are shown below:

- **Function definition**
- **Function call**
- **Function declaration**

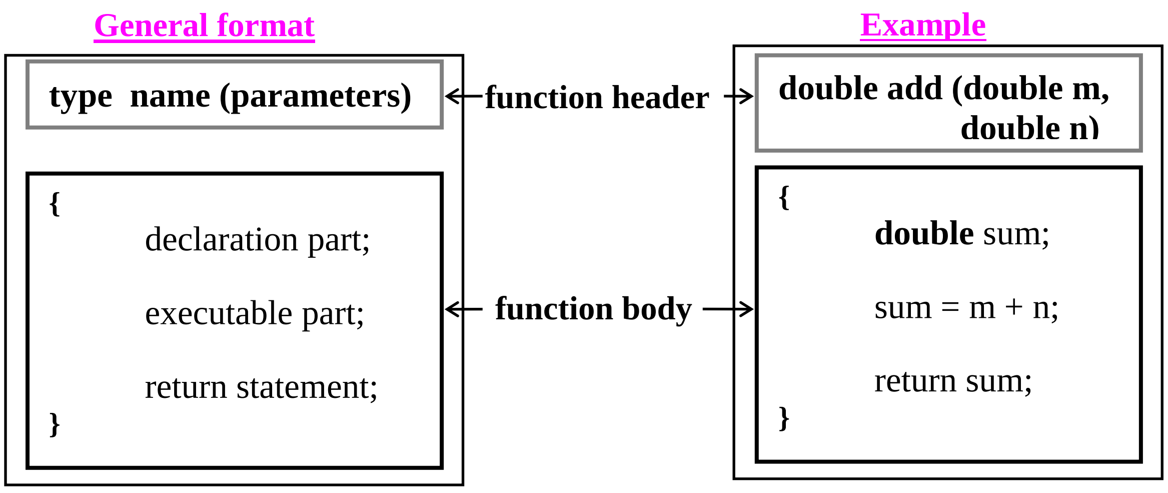
8.4.1 Function definition

This is most important topic and hence, let us discuss in detail. Now, let us see *what is function definition and let us explain in detail.*

Definition: The program module that is written to achieve a specific task is called *function definition.* Each function definition consists of two parts:

- Function header: consisting of data type of the value returned by the function, the name of the function and parameters
- Function body: consisting of set of instructions to do the specific activity.

The general format of function definition along with example is shown below:



```
type  name (parameters)
{
        declaration part;

        executable part;

        return statement;
}
```

```
double add (double m,
            double n)
{
        double sum;

        sum = m + n;

        return sum;
}
```

Function header: The function header consists of three parts:

- type : This is the data type of the value that the function is expected to return. The data type can be **int**, **float**, **double, char, void** etc.
- name : It is the name of the function. It can be any valid identifier. The name of identifier chosen itself should indicate the activity being performed by the function.
- parameters : The parameters are list of variables enclosed within parentheses. *All these variables should be separately declared and each declaration should be separated by comma.*

Function body: The compound statement immediately following the function header is *function body*. The function body consists of following three elements

declaration: All the variables used in the function body should be declared in the declaration part. But, the variables used in the function header should not be declared (they have to be declared only in function header)

executable part: This part contains the statements or instructions that perform the specified activity. In the above example,

```
sum = m + n;
```

is the executable statement.

return: It is a keyword. It is used to return the control to the calling function with/without a value. For example, if a function is not returning any value, use the return keyword as shown below:

```
return;
```

If function is returning a value, use the return keyword as shown below:

```
return value;
```

Example 8.11: Function definition to add two numbers and to return the result

```
int add (int  a, int   b)          | Function header = type + name + parameters
{                                  | Function body =
        int   sum;                 |               declaration part +
                                   |
        sum = a + b;               |               executable part +
                                   |
        return  sum;               |               return statement
}                                  | end of the function body
```

8.4.2 Function declaration/ Function prototype

Now, let us see what is function declaration or function prototype.

Definition: As we normally declare the variables before they are used, the functions also should be declared before they are used. This process of declaring the functions before they are used (or called) is called *function prototype.* The function prototype is also called *function declaration.*

The function declaration or function prototype contains only the function header but terminated with semicolon. It does not contain the body of the function. The function prototype informs the compiler about the following:

- The type of value returned by the function
- The name of the function
- The number of parameters passed to that function
- The type of each parameter in the function header

Now, let us see *what is the syntax for function prototype.* The syntax of function prototype is shown below:

```
         type     fname   (type p1, type p2,…type pn) ;      semicolon is must
           ⇣        ⇣        ⇣     ⇣     ⇣     ⇣
ex 1:    int      add      (int   a,   int   b);
ex 2:    void     add      (int   a,   int   b);
```

where

type	– is type of the value returned from the function such as **void**, **int**, **float**, **double** etc
fname	– is the name of the function
p1, p2,….pn	– are the parameters. All the parameters have to be separately declared and each declaration should be separated by comma.

For example, consider the following prototype:

double sum(**int** a, **int** b);

The above function prototype can also be written as:

double sum(**int** , **int**);

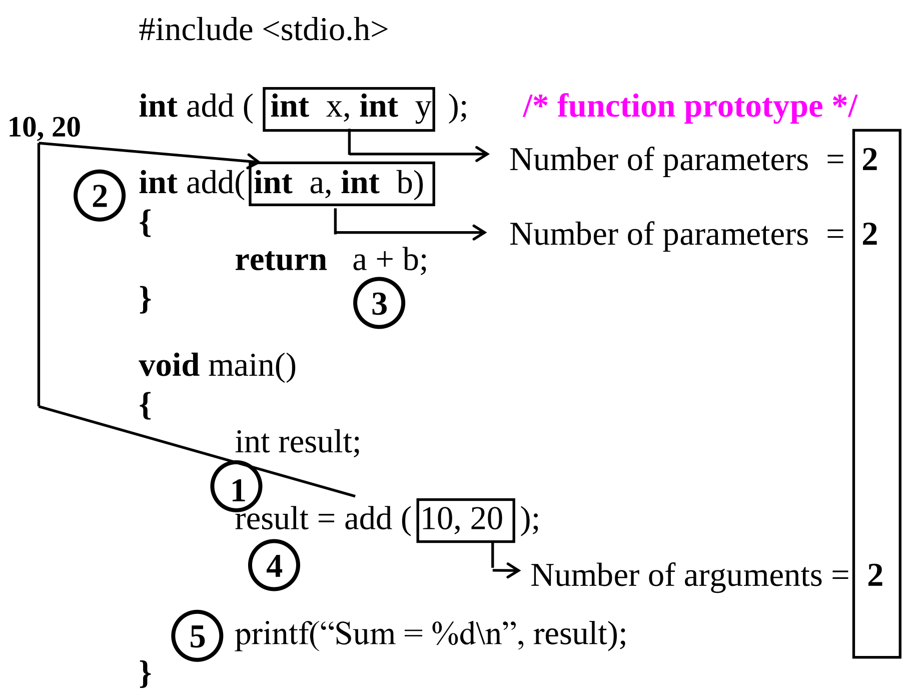
The above prototype informs the compiler that the function “sum()” returns a value of type **double** and accepts two parameters of type **integer**.

8.4.3 Function calls

Now, let us see *what is a function call.*

Definition: Once the function is defined, it has to be called so as to achieve the task. This method of calling a function to achieve a specified task is called *function call.* The function can be called by writing the name of the function and passing the appropriate number of arguments. The number of arguments in function call and number of parameters in function definition must match. Also, the order of arguments in function call and parameters in the function definition must match.

Example 8.12: Program to demonstrate function call



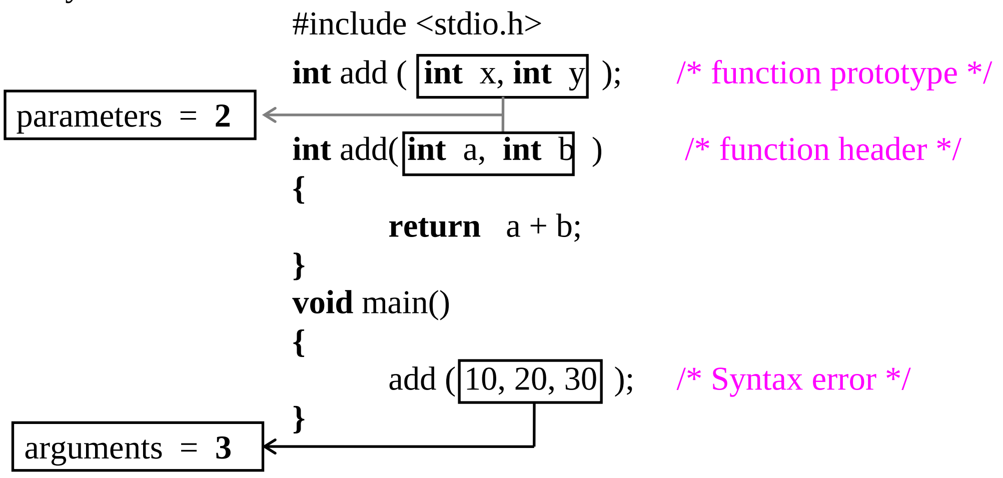
```
#include <stdio.h>

int add ( int  x, int  y );      /* function prototype */

int add(int  a, int  b)
{
        return   a + b;
}

void main()
{
        int result;

        result = add (10, 20);

        printf(“Sum = %d\n”, result);
}
```

Now, observe the following sequence of operations:

1 - The function *add() is called* with two arguments 10 and 20
2 - Control is transferred to the function add() and the values 10 and 20 are received using the variables *a* and *b* (Look at the direction of arrow)
3 - The result is computed by adding 10 and 20
4 - The result is returned to *main()* and will be copied into variable *result*
5 - The result is displayed on the screen.

Now, let us see *what are the rules to be followed while writing function prototype, function definition and during the function call.*

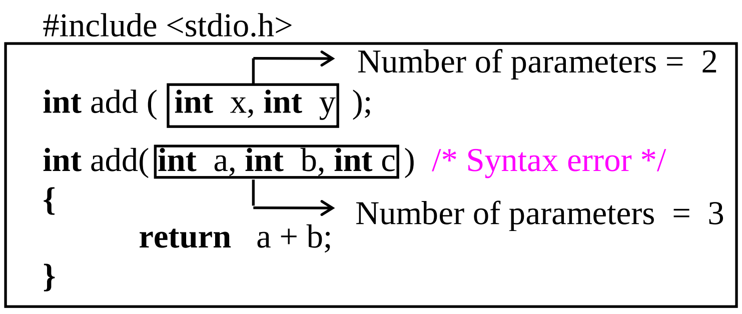
- *The number of arguments in the function call must be equal to the number of parameters in the function prototype and function header.* Otherwise, it results in syntax error as shown below:



```
#include <stdio.h>
int add ( int  x, int  y );      /* function prototype */
int add(int  a,  int  b )        /* function header */
{
        return   a + b;
}
void main()
{
        add (10, 20, 30 );       /* Syntax error */
}
```

Since the number of parameters in the function prototype is different from number of arguments in the function call, it results in syntax error.

- *The number of parameters in the function prototype must be equal to the number of parameters in the function definition.* A function definition that does not match the function prototype results in syntax error as shown below:



```
#include <stdio.h>
int add ( int  x, int  y );
int add(int  a, int  b, int c)  /* Syntax error */
{
        return   a + b;
}
```

- A *void function* does not return a value. So, when a function is called, it can be used only as a statement. It cannot be used in the expression or in printf statement as shown below:

```
void add ( int a,  int b  )
{
          printf("%d", a+b);
}
void  main()
{
          int  sum;

          add(2, 4);              /* OK:Can be used only as statement */

          printf("%d", add(2, 4) );   /* Cannot be  used in printf */

          sum = add(2,4) + 10;        /* Cannot be used in expression */

          print("%d\n", sum);
}
```

- *If function prototype/definition contains void return type, then in the function definition, a value should never be returned*. It results in syntax error, if we return a value as shown below:

```
              #include <stdio.h>

          void      add(int a, int  b)
          {
                 return    a+b;        /* Syntax error */
          }
```

8.5 Formal parameters and actual parameters

Now, let us see *what are formal parameters or dummy parameter.*

Definition: The variables defined in the function header of function definition are called *formal parameters.* All the variables should be separately declared and each declaration must be separated by commas. The formal parameters are also called *dummy parameters.* The formal parameters receive the data from actual parameters.

Now, let us see *what are arguments or actual parameters.*

Definition: The variables that are used when a function is invoked are called *arguments* or *actual parameters.* Using the actual parameters, the data can be transferred to the function. The corresponding formal parameters in the function definition receive them. The actual parameters and formal parameters must match in number and type of data.

For example, the actual parameters and formal parameters are shown in the following program.

```
#include <stdio.h>

double ADD ( double  a, double b )        /* Formal/dummy parameters  */
{
        return    a + b;
}

void main()
{
        double          m, n, res;

        printf("Enter two numbers:");
        scanf("%lf  %lf",&m, &n);

        res = ADD ( m,  n );          /* Actual parameters  or arguments*/

        printf("ADD(%lf, %lf) = %lf\n", m, n,res);
}
```

In the function ADD(), the variables *a* and *b* declared in function header are called *formal parameters.* The variables *m* and *n* used in function main() while calling the function ADD() are called *actual parameters.*

8.6 Categories of functions

Now, let us see how the functions are categorized based on the value returned by the function and parameters accepted. Based on the parameters and return value, the functions are categorized as shown below:

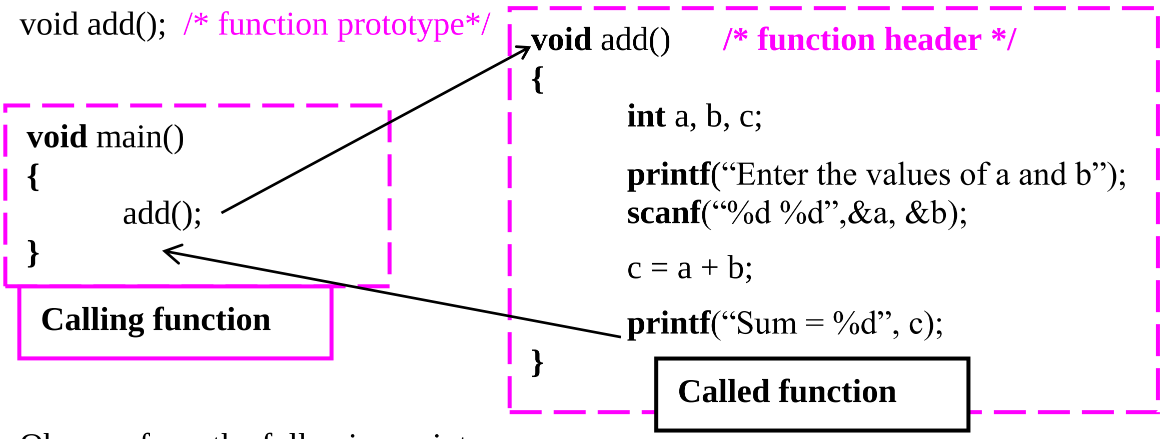
Categories of functions
- Functions with no parameters and no return values
- Functions with no parameters and return values
- Functions with parameters and no return values
- Functions with parameters and return values

8.6.1 Functions with no parameters and no return values

This category is also called *"void functions with no parameters".* In this category, there is no data transfer between the *calling function* and *called function*. So, *calling function* cannot send values and hence *called function* cannot receive the data. For example, consider the program shown below:

Example 8.13: Program showing function call with no parameters & no return value

```
#include <stdio.h>

void add();  /* function prototype*/
```



```
void main()
{
        add();
}

void add()      /* function header */
{
        int a, b, c;

        printf(“Enter the values of a and b”);
        scanf(“%d %d”,&a, &b);

        c = a + b;

        printf(“Sum = %d”, c);
}
```

Observe from the following points:

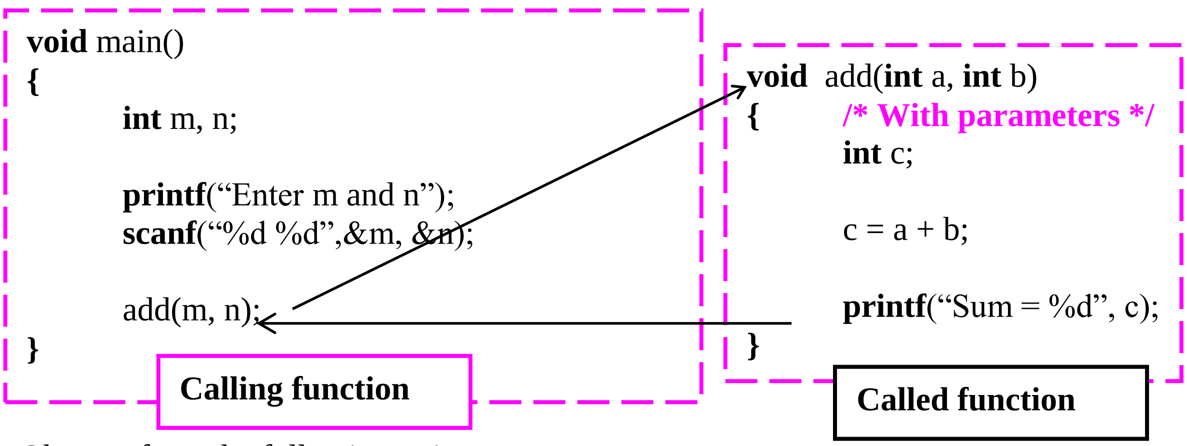
- In the calling function, when the function *add()* is called, no arguments are passed to the function *add()*. So, no parameters are defined in function header.
- When control is transferred to the *called function*, the two values are read, they are added and the result is printed on the monitor.
- When the last statement is executed, control is transferred to the *calling function*.
- In the function main(), when there is no statement to be executed, the program is terminated.

8.6.2 Functions with parameters and no return values

This category is also called "void functions with parameters". In this category, there is data transfer from the *calling function* to *the called function* using parameters. But, there is no data transfer from *called function* to the *calling function.* For example, consider the following program:

Example 8.14: Program showing a function call with parameters and no return value

```
#include <stdio.h>

void  add(int a, int b);   /* function declaration/prototype */
```



```
void main()
{
        int m, n;

        printf("Enter m and n");
        scanf("%d %d",&m, &n);

        add(m, n);
}
```

```
void  add(int a, int b)
{        /* With parameters */
        int c;

        c = a + b;

        printf("Sum = %d", c);
}
```

Observe from the following points:

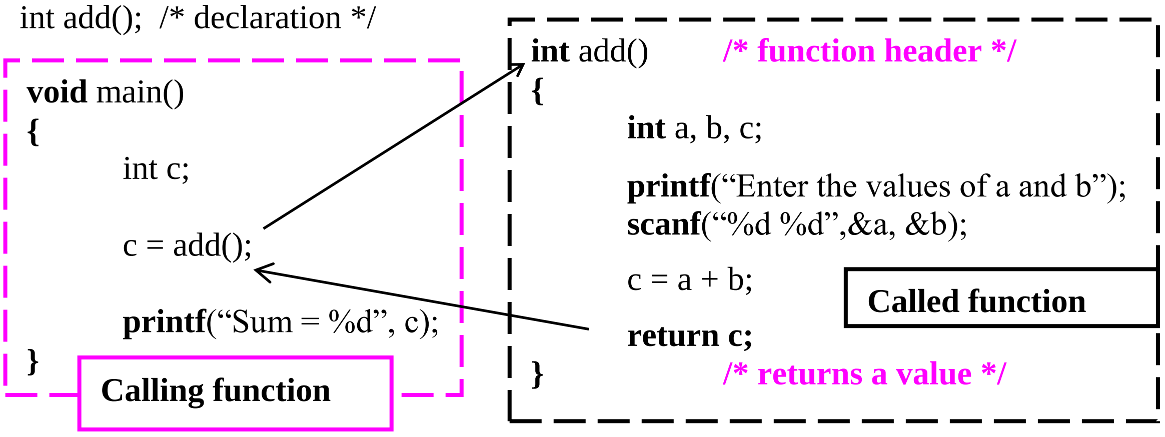
- In the calling function, when the function *add()* is called, two arguments *m* and *n* are passed to the function *add()*. So, two parameters *a* and *b* are defined in function header.
- The values of actual parameters *m* and *n* are copied into formal parameters *a* and *b*.
- The value of *a* and *b* are added and result stored in *c* is displayed on the screen
- When the last statement is executed, control is transferred to the calling function.
- In function main(), when **return** statement is executed, the program is terminated

8.6.3 Functions with no parameters and return values

In this category, there is no data transfer from the *calling function* to the *called function.* But, there is data transfer from *called function* to the *calling function.* When the function returns a value, the *calling function* receives one value from the *called function.* For example, consider the program shown below:

Example 8.15: Program showing function call without parameters, with return value

```
#include <stdio.h>

int add();  /* declaration */
```



```
void main()
{
        int c;

        c = add();

        printf("Sum = %d", c);
}
```

```
int add()        /* function header */
{
        int a, b, c;

        printf("Enter the values of a and b");
        scanf("%d %d",&a, &b);

        c = a + b;

        return c;
}                /* returns a value */
```

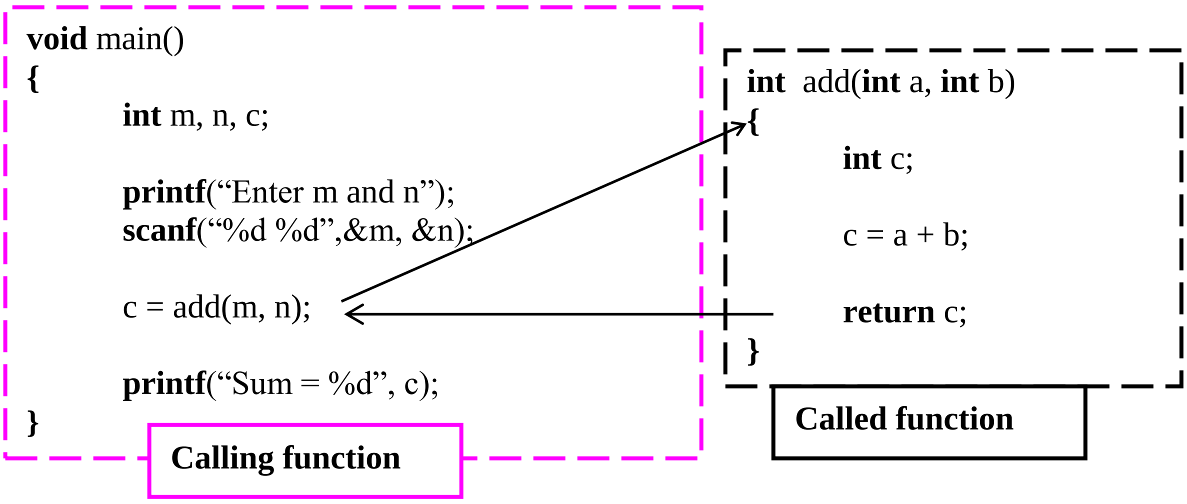
- In the calling function, when the function *add()* is called, no arguments are passed to the function *add()*. So, no parameters are defined in function header.
- When control is transferred to the called function, the two values are read, they are added and the result is stored in *c*.
- When a return statement is executed in the function, the function is terminated immediately and control goes back to the calling function.
- The function call is replaced by the value returned by the return statement and this value is copied into *c* in function main.
- The result is displayed on the screen in function *main* and when return statement is executed, program is terminated.

8.6.4 Functions with parameters and return values

In this category, there is data transfer between the *calling function* and *called function*. When parameters are passed, the *called function* can receive values from the *calling function*. When the function returns a value, the *calling function* can receive a value from the *called function*. For example, consider the program shown below:

Example 8.16: Program showing a function call with parameters & with return value

```
#include <stdio.h>

int  add(int a, int b);  /* declaration */
```



```
void main()
{
        int m, n, c;

        printf("Enter m and n");
        scanf("%d %d",&m, &n);

        c = add(m, n);

        printf("Sum = %d", c);
}
```

Calling function

```
int  add(int a, int b)
{
        int c;

        c = a + b;

        return c;
}
```

Called function

Observe from the following points:

- In the calling function, when the function *add()* is called, two arguments *m* and *n* are passed to the function *add()*. So, two parameters *a* and *b* are defined in function header.

- The values of actual parameters *m* and *n* are copied into formal parameters *a* and *b*.
- The value of *a* and *b* are added and result stored in *c*
- When a return statement is executed in the function, the function is terminated immediately and control goes back to the calling function.
- The function call is replaced by the value returned by the return statement and this value is copied into *c* in function main.
- The result is displayed on the screen in function *main* and when return statement is executed, program is terminated.

8.7 Passing parameters to functions

Now, let us see "What are the different ways of passing parameters to the functions?" There are three ways of passing parameters to the function:

- Pass by value (call by value)
- Pass by reference (call by reference)
- Pass by address

8.7.1 Pass by value (Call by value)

Now, let us see "What is pass by value?" and "Explain with example"

Definition: In pass by value, the values of *actual parameters* are copied into *formal parameters* i.e., formal parameters contain only the copy of actual parameters. So, even if the values of the *formal parameters* are changed in the *called function*, the values of the *actual parameters* in the calling function are not changed.

Advantage: The main advantages of pass by value are:
- Makes the functions more self-contained
- Protect them against accidental changes (i.e., even if formal parameters are changed actual parameters are not changed)

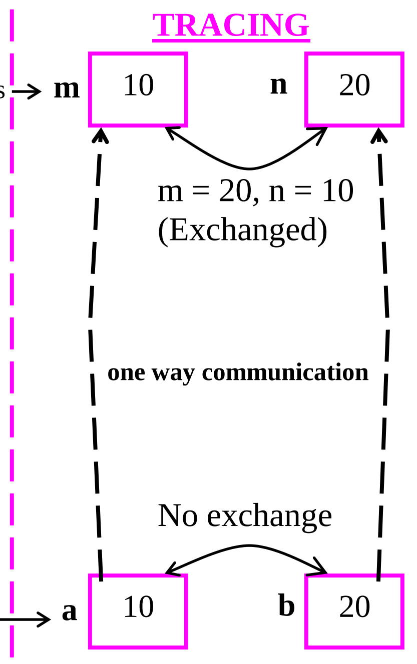
Disadvantage: It does not allow information to be transferred back to calling function via arguments. Thus, pass by value is restricted to one-way transfer of information from calling function to called function.

The concept of pass by value can be explained by considering the following program.

Example 8.17: Program to show the concept of pass by value

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

```
void exchange (int m, int n)  → Formal parameters
{
        int temp;

        temp = m;
        m = n;
        n = temp;
}

void main()
{
        int a, b;

        a = 10, b = 20;

        exchange (a, b);  →  Actual parameters

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



Output

```
a = 10 and b = 20
```

Working The sequence of operations are shown below:

- Execution starts from function *main()* and the variables *a* and *b* are assigned the values 10 and 20 respectively.
- The function exchange() is called with actual parameters *a* and *b* whose values are 10 and 20.
- In the function header of function **exchange(),** the formal parameters *m* and *n* receive the values 10 and 20
- In the function exchange(), the values of *m* and *n* are exchanged.
- But, the values of actual parameters *a* and *b* in function *mian()* have not been exchanged.

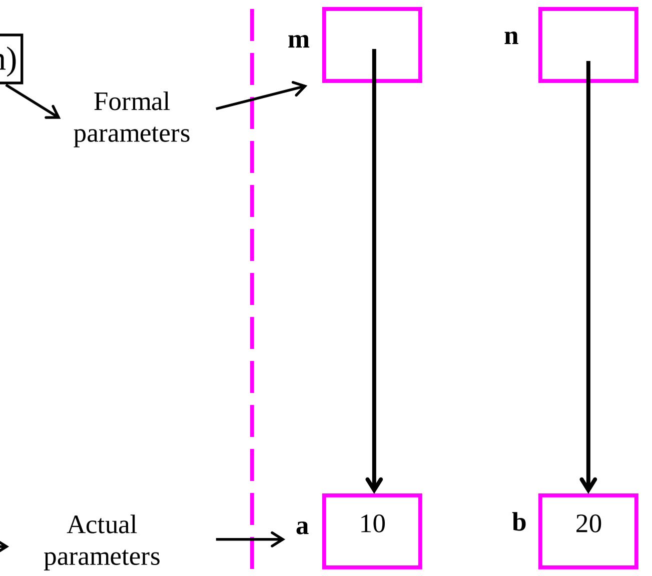
8.7.2 Pass by address

Now, let us see *what is pass by address* and explain with example.

Definition: In pass by address, when a function is called, the addresses of actual parameters are sent. The addresses of *actual parameters* are copied into *formal parameters*. Using these addresses, the values of the *actual parameters* can be changed. This way of changing the actual parameters indirectly using the addresses of actual parameters is called *pass by address.* This concept is explained using the following program:

Example 8.18: Program to show the concept of pass by address

```
#include <stdio.h>

void exchange (int *m, int *n)
{
        int temp;

        temp = *m;
        *m = *n;
        *n = temp;
}
void main()
{
        int a, b;
        a = 10, b = 20;
        exchange (&a, &b);
        printf("a = %d and b = %d\n", a, b);
}
```



Output (After exchange)
a = 20, b = 10

Working The sequence of operations are shown below:

- Execution starts from function *main()* and the variables *a* and *b* are assigned the values 10 and 20 respectively.
- The function exchange() is called by sending the addresses of *a* and *b.*
- In the function header of function **exchange(),** the formal parameters *m* and *n* declared using * operator holds the addresses of *a* and *b.*
- In the function exchange(), using *m and *n we can access the values of *a* and *b* and they are exchanged.
- When control comes to the calling function, the values of *a* and *b* have been exchanged.

8.7.3 Pass by reference (call by reference)

Pass by reference is not supported in C language. It is supported in C++ language.

8.8 Advantages of Functions

The various advantages of using functions are shown below:

- **Reusability and Reduction of code size**: The existing functions can be re-used as building blocks to create new programs. This results in reduced program size. These functions can be used any number of times.

- **Readability of the program can be increased.** Programs can be written easily and we can keep track of what each function is doing.
- **Modular programming approach**: A large program is divided into smaller sub programs so that each sub program performs a specific task. This approach makes the program development more manageable.
- **Easier debugging** Using modular approach, localizing, locating and isolating a faulty function is much easier.
- **Build library**: The functions that are used repeatedly can be generalized, tested and kept in a library for future use. This reduces program development time and coding time.
- **Function sharing**: A function can be shared by many programmers.

8.9 Other user defined functions

In this section, let us design user defined functions for various types of problems. Now, let us discuss each of the problem and see how to write functions one by one.

8.9.1 Implementing right rotation using right shift

In this section, let us design and develop a C function ***RightShift****(x ,n)* that takes two integers *x* and *n* as input and returns value of the integer *x* rotated to the right by *n* positions. Assume the integers are unsigned. Write a C program that invokes this function with different values for *x* and *n* and tabulate the results with suitable headings.

We know that all decimal numbers are converted into binary and the numbers are stored in memory in binary form. So, given a binary number, we can check whether the number is even or odd as shown below:

Decimal numbers	Binary
0	0000 000**0**
2	0000 001**0**
4	0000 010**0**
6	0000 011**0**
8	0000 100**0**
10	0000 101**0**
12	0000 110**0**
14	0000 111**0**

Even numbers

Decimal numbers	Binary
1	0000 001**1**
3	0000 001**1**
5	0000 010**1**
7	0000 011**1**
9	0000 100**1**
11	0000 101**1**
13	0000 110**1**
15	0000 111**1**

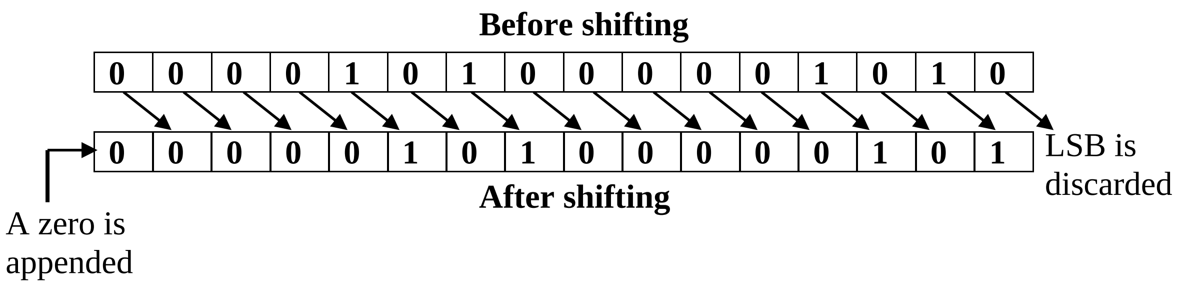
Odd numbers

Note: All even binary numbers ends with 0 All odd numbers ends with 1

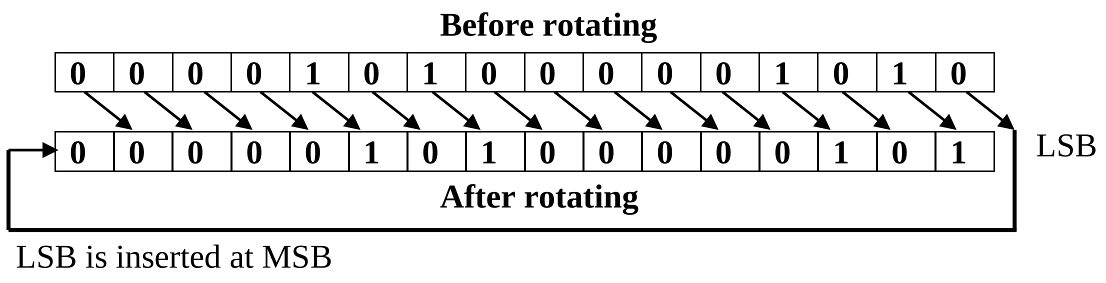
Design: Let us assume size of an integer is 2 bytes i.e., 16 bits. The maximum unsigned integer using 16-bits is 65535 (Refer section 2.8.1) for more details.

Rotate right: Rotating a number by 1-bit position is nothing but shifting all bits towards right by 1 bit position and moving LSB bit to MSB bit. The rotation can be done for a number as shown below:

Right shift by 1-bit position for even number: Consider the following figure that shows the contents of memory before shifting and after shifting towards right by 1-bit position.



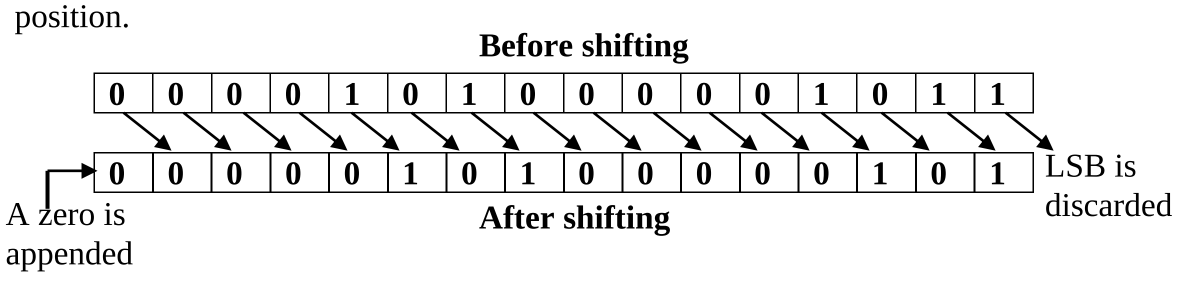
Right rotate by 1-bit position for even number: This is exactly same as above. But, instead of discarding LSB bit, it is moved to the MSB as shown in figure below:



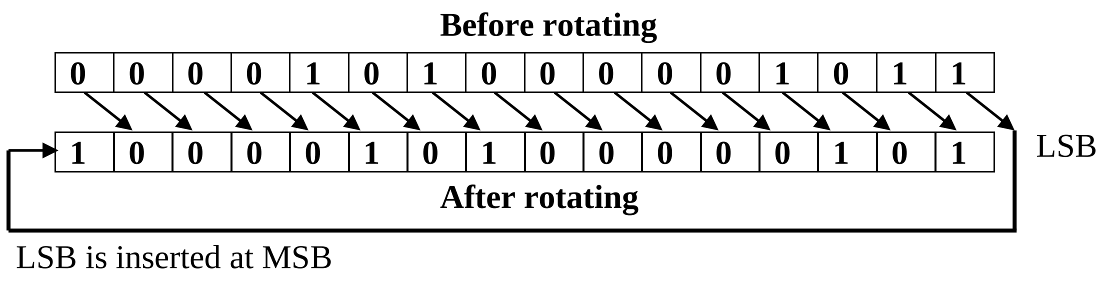
Observe from above figures that, the result obtained by right shift and right rotate for a given even number is same. So, right rotation of an even number say *x* is nothing but right shift *x* by 1. The code for this situation can be written as shown below:

```
if (x % 2 == 0) x = x >> 1;
```

Right shift by 1-bit position for odd number: Consider the following figure that shows the contents of memory before shifting and after shifting towards right by 1-bit position.



Right rotate by 1-bit position for odd number: This is exactly same as above. But, instead of discarding LSB bit, it is moved to the MSB as shown in figure below:



Observe from above figures that, the result obtained by right shift and right rotate for a given odd number is different. Since a 1 in LSB is moved to MSB which is 15th bit position, its value is increased by 2^{15} = 32768. So, right rotation of an odd number say *x* is nothing but right shift *x* by 1 and add 32768. The code for this can be written as shown below:

```
                if (x % 2 == 1) x = x >> 1, x += 32768;                // 16-bit machine
```

So, the code for right rotating a number by 1 bit-position can be written as shown below:

```
                if (x % 2 == 0)
                        x = x >> 1;
                else
                        x = x >> 1, x += 32768;
```

But, for rotating a number by n-bit positions, the above statement has to be executed for *n* times. The code for this can be written as shown below:

```
        for (i = 1; i <= n; i++)
        {
                if (x % 2 == 0)
                        x = x >> 1;
                else
                        x = x >> 1, x += 32768;
        }
```

The complete C function to rotate unsigned *x* by n-bits towards right is shown below:

Example 8.19: C function to right rotate *x* by n-bit positions

```
unsigned  int  right_rotate (unsigned  int  x, int  n)
{
        int     i;
        for (i = 1; i <= n; i++)
        {
                if (x % 2 == 0)
                        x = x >> 1;
                else
                        x = x >> 1, x += 32768;      // 16-bit machine: 2^15 = 32768
        }

        return x;       /* x rotated right n times */
}
```

Now, the complete program can be written as shown below:

Example 8.20: C program to right rotate *x* by n-bit positions

```
#include <stdio.h>

/* Include: Example 8.19: C function to right rotate x by n-bit positions */

void main()
{
        unsigned        int     x;      /* Given unsigned integer to be rotated */

        unsigned        int     res;    /* Rotated unsigned result */

        int                     n;      /* Number of bit-positions to be rotated */

        int                     m;      /* Number of unsigned integers to rotate */

        int                     i;

        printf("Enter number of unsigned integers to rotate\n");
        scanf("%d", &m);

        for (i = 1; i <=m; i++)
        {
                printf("(unsigned no., no. of rotations):");
                scanf("%u, %d", &x, &n);

                result = right_rotate(x, n);

                printf("Right Rotate(%u, %d) = %u\n", x, n, result);
        }
}
```

Output:

```
Enter number of unsigned integers to rotate: 3

unsigned no., no. of rotations: 8, 4
Right Roate(8, 4) = 32768

unsigned no., no. of rotations: 32768, 15
Right Roate(32768, 15) = 1

unsigned no., no. of rotations: 65535, 5
Right Roate(8, 5) = 65535
```

8.9.2 Check for prime number

Before we check whether a number is *prime* or not we should know the answer for the question "*What is a prime number?*"

Definition: A number which is divisible by 1 and itself is a *prime number.* For example, numbers such as 1, 2, 3, 5, 7, 11 etc., are not divisible by any numbers other than 1 and themselves. So, they are all prime numbers.

But, the numbers such as 4, 6, 8, 16 etc., are divisibly by 2 and hence they are *non-prime numbers*. The number 9 is divisibly by 3 and hence it is also *non-prime number*. Now, let us see "*How to check whether a given number is prime or not?*"

Design: Consider the following two points before we check for a prime number:

♦ *A number m cannot be divided by a number which is greater than m/2.* For example, consider the number 16. This number cannot be divisible by any number which is greater than 16/2. That is, it ***cannot*** be divisible by 9, 10, 11, 12, 13, 14 and 15 which are all greater than 16/2.

♦ *A number m can be divided by 2, 3, 4,5……..m/2.* For example, the number 16, can be divided by 2, 4, 8.

So, to check whether given number *m* is prime or not, it is sufficient to divide *m* by the numbers 2, 3, 4,….*m/2* one after the other. During this process, if the remainder is zero, the number *m* is not prime This can be done using the statement:

```
if ( m % i == 0)  return 0;       // for i = 2, 3, 4,…..m/2
```

The above statement has to be repeatedly executed for each value of *i* = 2, 3, 4…..m/2. The equivalent statement can be written as:

```
                    for ( i = 2;  i <=  m/2,  i++)
                    {
                            if ( m % i == 0)   return 0;   /* Number is not prime */
                    }
```

But, even after dividing *m* by all the numbers from 2 to *m/2*, if the remainder is still not zero, the number *m* is prime and return the value 1. The partial code can be written as:

```
                    return 1;        /* Number is prime */
```

Now, using the above partial code let us identify the elements of the function one by one:

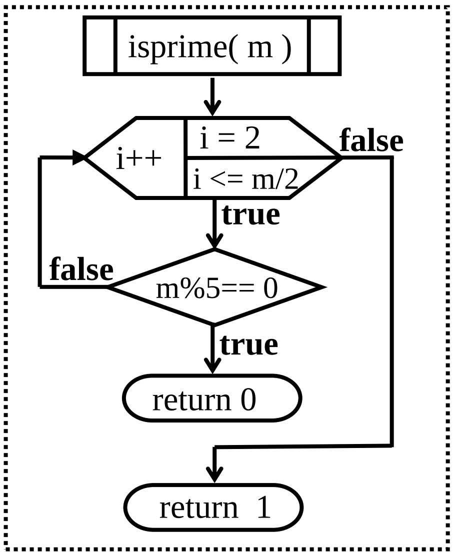
- **<u>Return type:</u>** The function should return either 1 or 0 which is an integer value. So, the return type must be int.
- **<u>Name of the function:</u>** Since the function is used to check whether the given number is prime or not, the name of the function is *isprime().*
- **<u>Parameter:</u>** Given *m*, we have to check whether the number is prime or not. So, *m* is the parameter to be passed to the function.
- **<u>Local variable:</u>** Other than *m* all other variables should be declared as local variables. So, *i* should be used as local variable

Now, the complete function can be written as shown below:

Example 8.21: C function that returns 1 if number is prime. Otherwise, it returns 0

```
int  isprime (int  m)
{
        int   i;

        for ( i = 2;  i <=  m/2;  i++)
        {
                if ( m % i == 0)
                {
                        /* Number is not prime */
                        return 0;
                }
        }

        return 1;        /* Number is prime */
}
```

FLOWCHART



The complete program to check whether the given number is prime or not can be written as shown below:

Example 8.22: C program that prints whether the given number is prime or not

```
#include <stdio.h>

/* Include: Example 8.21:  Function to check for prime or not */

void main()
{
        int   n;

        printf("Enter a +ve integer :");
        scanf("%d", &n);

        if ( isprime(n) )
                printf ("%d is prime\n", n);
        else
                printf ("%d is not prime\n", n);
}
```

TRACING

Input

```
Enter integer        Enter integer
10                   7
```

Output

```
                     7 is prime

10 not prime
```

8.9.3 Prime numbers within range n1 and n2

Design: Let us generate numbers between 10 and 30 as shown below:

i = 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30

After generating each of the above number, we have to check whether the number is prime or not. If it is prime display the prime number:

```
 i =  10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30
          ↓       ↓               ↓       ↓               ↓                           ↓
Primes:  11      13              17      19              23                          29
```

This can be achieved using the following code:

```
for (i = n1;  i <= n2; i++)
{
        if (isprime(i)) printf ("%d  ", i);
}
```

The C program to generate prime numbers within range n1 and n2 is shown below:

Example 8.23: C program that prints prime numbers in the range n1 and n2

```
#include <stdio.h>
/* Include: Example 8.21: Function to check for prime or not */
void main()
{
        int  i,  n1,   n2;

        printf("Enter the range :");
        scanf("%d %d", &n1, &n2);

        printf("Primes between %d to %d\n", n1, n2);
        for (i = n1; i <= n2; i++)
        {
                if ( isprime(i) )  printf("%d   ", i);
        }
}
```

TRACING

Input

```
Enter the range: 10  30
```

Output

```
Primes between 10 to 30

11   13   17   19   23    29
```

8.9.4 Function to compute factorial of a number

The design procedure to compute factorial of a number is already discussed in previous chapter. (Refer section 7.3.11. for details). For convenience, the code is repeated once again as shown below:

```
prod = 1;

for (i = 1;  i <= n;  i++)
{
        prod *=  i;
}
```

The result is available in the variable *prod* and it has to be returned to the calling function. This can be done using the statement:

```
return prod;
```

Now, using the above partial code let us identify the elements of the function one by one:

- **Return type:** The function should return factorial of a number stored in *prod* which is an integer value. So, the return type must be int.
- **Name of the function:** Since the function is used to compute factorial of a number, the name of the function is *fact().*

- **Parameter:** Given *n*, we can compute factorial of *n*. So, *n* is the parameter to be passed to the function.
- **Local variable:** Other than *n* all other variables should be declared as local variables. So, *i* and *prod* should be used as local variable

Now, the complete function can be written as shown below:

Example 8.24: C function that returns factorial of a number.

```
int fact (int n)
{
        int prod, i;

        prod = 1;

        for (i = 1;  i <= n;  i++)
        {
                prod *=  i;
        }

        return prod;
}
```

The above function can be called as shown below:

Example 8.25: C program that prints factorial of a given number

```
#include <stdio.h>

/* Include: Example 8.24: To find factorial of a number */

void main()
{
        int   n, res;

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

        res = fact(n);

        printf("FACT(%d) = %d\n", n, res);
}
```

TRACING

```
Enter the value of n
5

res = 120

FACT(5) = 120
```

8.9.5 Computing power i.e., x^n

Design: We have used the following statement to compute factorial of a number in the previous section:

```
prod = prod * i;
```

The above statement is repeatedly executed for each of value of *i* ranging from 1 to n and repeated *n* times. If we replace, *i* by *x*, we are multiplying *x* repeatedly *n* times getting x^n. So, the function shown in section 8.9.4 can be written as it is but by replacing the statement:

```
prod *= i;
```

with the statement:

```
prod *= x;
```

In that case, we need to pass one more parameter *x* which is of type **int.** Now, the complete function to find x^n can be written as shown below:

Example 8.26: C function that to compute x^n

```
int power (int x, int n)
{
        int prod, i;

        prod = 1;

        for (i = 1;  i <= n;  i++)
        {
                prod *=  x;
        }

        return prod;
}
```

The C program that calls above function to find x^n can be written as shown below:

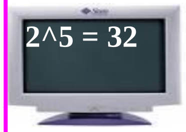
Example 8.27: C program that prints x^n

```
#include <stdio.h>

/* Include: Example 8.26: To find xn */
```

```
void main()
{
        int  x,  n, res;

        printf ("Enter base and exponent\n");
        scanf ("%d %d", &x, &n);

        res = power(x, n)

        printf("%d^%d= %d\n", x, n, res);
}
```

Output

```
Enter the value of n
2   5

res = 2 * 2 * 2 * 2 * 2 =32
```



8.9.6 Sum of terms with alternate + and – terms

Let us design the program to find the sum of the following series

1 – 2 + 3 – 4 + 5 – 6 ….n

Design: Before adding any of the terms, *sum* will be zero. So, we initialize *sum* to 0. Observe that odd numbers have to be added and can be written as:

```
        if (i % 2 == 1)
                sum = sum + i;          // for odd value of i
```

Otherwise, all even numbers have to be subtracted and can be written as

```
                sum = sum – i;          //for even value of i
```

The above two statements can be combined together using *if-else* as shown below:

```
if (i % 2 == 1)                 i = 1, 2, 3, 4,……..n
        sum = sum + i;              // add odd value of i
else
        sum = sum – i;              // subtract even value of i
```

So, the complete code can be written as shown below:

```
sum = 0;

for (i = 1;  i <= n;  i++)
{
        if (i % 2 == 1)
                sum = sum + i;
        else
                sum = sum – i;
}
```

Now, the result is available in the variable *sum* and can be returned to the calling function using the statement:

```
        return  sum;
```

Now, the various function elements are identified as shown below:

- **Return type:** Since *sum* contains integer value, return type will be **int**
- **Name of function:** Since, we are adding numbers, name of the function is **add_numbers()**
- **Parameter :** The value of *n* is given and hence *n* is passed as a parameter.
- **Local variables:** Other than *n*, all other variables used must be defined inside the function. So, variables *i* and *sum* should be declared inside the function as shown below:

Example 8.28: C function to add the series: 1 – 2 + 3 – 4 + 5 – 6 ………n

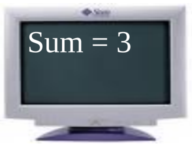
```
int  add_numbers (int  n)
{
        int   i, sum;

        sum = 0;

        /* find sum of 1 – 2 + 3 – 4 + 5 – 6 ………n */
        for ( i = 1;  i <= n; i++)
        {
                if ( i % 2 == 1)
                        sum = sum + i;
                else
                        sum = sum – i;
        }

        return sum;
}
```

The main function that uses the above function is shown below:

Example 8.29: C program to add the series 1 – 2 + 3 – 4 + 5 – 6 ………n

```
#include <stdio.h>
/* Include: Example  8.28:  Function to add 1 – 2 + 3 – 4 + 5 – 6 ………n */

void main()
{
        int  n, sum;

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

        sum  = add_number (n);

        printf("Sum = %d\n", sum);
}
```

Input
```
Enter the value of n
5

sum = 1 – 2 + 3 – 4 + 5 =3
```


```
Sum = 3
```

8.9.7 Taylor's Series to compute e^x

The program to compute e^x is already designed and it is given in section 7.3.16, example 7.26. Look at the shaded portion in that example. Using that code, the function to compute e^x can be written as shown below:

- **Return Type:** The result is available in variable *sum* which is of type *float*. So, the return type is **float**
- **Name of the function:** Since we compute e^x, the name of the function be **exp()**
- **Paramters:** To compute e^x, we require only *x* and *n* where *x* is of type **float** and *n* is of type **int**.
- **Local variables:** All other variables such as *term*, *sum* and *i* are used as local variables.
- **Body of the function:** The shaded portion can be the body of the function.

Now, the complete function can be written as shown below:

Example 8.30: C function to compute e^x

```
float exp(float x, int n)
{
        float term, sum;
        int    i;

        term = 1;                          /* generate the first term */
        sum = term;                        /* and initialize sum with first term */

        for (i = 1;  i <= n;  i++)
        {
                term = term*x / i;         /* Generate the next term */
                sum = sum + term;          /* Add the term to the partial result */
        }

        return sum;                        /* return e^x */
}
```

The main program that invokes above function to compute e^x can be written as shown below:

Example 8.31: C program to compute e^x

```
#include <stdio.h>
/* Include: Example 8.30: To compute ex  */
void main()
{
        int     n, i;
        float   x, sum, term;

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

        sum = exp (x, n);             /* compute ex */

        printf("Result = %f\n", sum);  /* Print ex */
}
```

TRACING

```
Enter the value of x and n
1  10

Result = 2.71
```

8.9.8 Taylor's series to compute sine value

The program to compute sine of an angle is already designed and it is given in section 7.3.17, example 7.27. Look at the shaded portion in that example. Using that code, the function to compute sine of a given angle (degrees) can be written as shown below:

- **Return Type:** The result is available in variable *sum* which is of type *float*. So, the return type is **float**
- **Name of the function:** Since we compute sine value, the name of the function be **my_sin()**
- **Paramters:** To compute sine value, we require only *degree* and *n* where *degree* is of type **float** and *n* is of type **int**.
- **Local variables:** All other variables such as *term, sum, i* and *x* are used as local variables.
- **Body of the function:** The shaded portion can be the body of the function.

Now, the complete function can be written as shown below:

Example 8.32: C function to compute sine value

```
float my_sin(float degree, int n)
{
        float   x, term, sum;
        int     i;

        x = degree*3.1416/180;

        term = x;
        sum = term;

        for (i = 3;  i <= n;  i+=2)
        {
                term = -term*x*x / ( (i-1)*i );
                sum = sum + term;
        }

        return sum;
}
```

The main program that uses the above function to compute sine value is shown below:

Example 8.33: C program to compute sine value using Taylor's series

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

/* Include: Example 8.32: To compute sine value  */

void main()
{
        int     n, i;
        float   x, degree, sum, term;

        printf (“Enter degree, number of terms\n”);
        scanf (“%f %d”, &degree, &n);

        sum = my_sin(degree, n);

        printf(“MYSIN(%f)= %f\n”, degree, sum);

        printf(“SIN(%f) = %f\n”, degree, sin(x));
}
```

```
Output1                      Output 2                     Output 3
Enter degree and terms       Enter degree & terms         Enter degree, terms
90  10                       45    10                     30    10
mysine(90) = 1.00011         mysine(45) = 0.7071          mysine(30) = 0.51
sine(90) = 1.0               sine(45) = 0.707             sine(30) = 0.50
```

8.9.9 Taylor's series to compute co-sine value

The program to compute cosine of an angle is already designed and it is given in section 7.3.18, example 7.28. Look at the shaded portion in that example. Using that code, the function to compute cosine of a given angle (degrees) can be written as shown below:

- **Return Type:** The result is available in variable *sum* which is of type *float*. So, the return type is **float**
- **Name of the function:** Since we compute cosine value, the name of the function be **my_cos()**
- **Paramters:** To compute sine value, we require only *degree* and *n* where *degree* is of type **float** and *n* is of type **int**.
- **Local variables:** All other variables such as *term, sum, i* and *x* are used as local variables.
- **Body of the function:** The shaded portion can be the body of the function.

Now, the complete function can be written as shown below:

Example 8.34: C function to compute cosine value

```
float my_cos(float degree, int n)
{
        float   x, term, sum;
        int     i;

        x = degree*3.1416/180;

        term = 1;
        sum = term;

        for (i = 2;  i <= n;  i+=2)
        {
                term = -term*x*x / ( (i-1)*i );
                sum = sum + term;
        }

        return sum;
}
```

The main program that uses the above function to compute cosine value is shown below:

Example 8.35: C program to compute cosine value using Taylor's series

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

/* Include: Example 8.34: To compute cosine value  */

void main()
{
        int     n, i;
        float   x, degree, sum, term;

        printf ("Enter degree, number of terms\n");
        scanf ("%f %d", &degree, &n);

        sum = my_cos(degree, n);

        printf("mycos(%f)= %f\n", degree, sum);
        printf("cos(%f) = %f\n", degree, cos(x));
}
```

Output1	Output 2	Output 3
Enter degree and terms	Enter degree & terms	Enter degree, terms
90 10	45 10	30 10
mycos(90) = 0.00011	mycos(45) = 0.7071	mycos(30) = 0.8667
cos(90) = 0.0	cos(45) = 0.707	cos(30) = 0.866

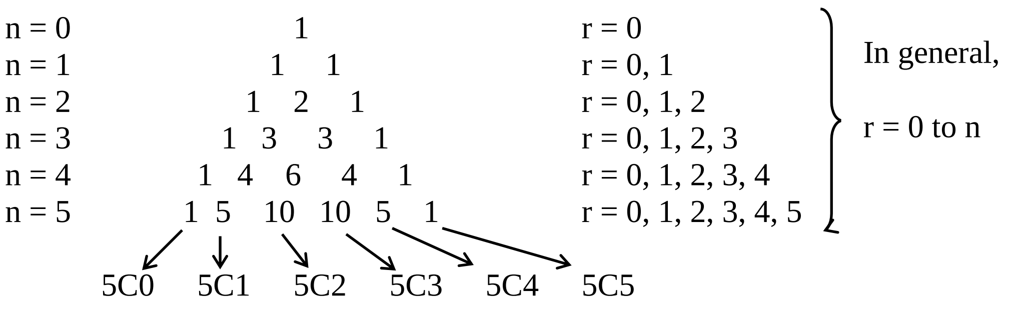
8.9.10 Pascals triangle

By calculating binomial co-efficients it is possible to obtain Pascal's triangle. The Pascals triangle is shown below:

```
           1                    line 1 has one element
         1   1                  line 2 has two elements
       1   2   1                line 3 has three elements
     1   3   3   1              line 4 has four elements
   1   4   6   4   1            line 5 has five elements
 1   5  10  10   5   1          line 6 has six elements
```

Design: Observe that each number in the above Pascal's triangle is a binomial co-efficient and can be obtained as shown below:

Number of rows is given by m = 6


n = 0 1 r = 0
n = 1 1 1 r = 0, 1
n = 2 1 2 1 r = 0, 1, 2
n = 3 1 3 3 1 r = 0, 1, 2, 3
n = 4 1 4 6 4 1 r = 0, 1, 2, 3, 4
n = 5 1 5 10 10 5 1 r = 0, 1, 2, 3, 4, 5
5C0 5C1 5C2 5C3 5C4 5C5
In general,
r = 0 to n

Note: Observe that the first number is 1 and last number is also 1. But, any other number is obtained by adding the two numbers just above it. Observe that each number is abinomial coefficient can be calculated and printed using the following statement:

```
                printf("%d  ", fact(n) / ( fact(n – r) * fact(r)));
```

The above statement should be repeated for each value of *r* ranging from 0 to *n* and the code can be written as shown below:

```
        for (r = 0; r <= k; r++)
                printf("%d  ", fact(n) / ( fact(n – r) * fact(r)));
```

The above loop has to be executed for each *n* = 0 to m – 1. Now, the code can be written as shown below:

```
        for (n = 0; n < m; n++)
        {
                for (r = 0; r <= n; r++)
                        printf("%d  ", fact(n) / ( fact(n – r) * fact(r)));
        }
```

Just before executing the above code and printing appropriate binomial coefficients, it is better to display appropriate spaces. Also, after printing the binomial coefficients, we display *new line* character so that next subsequent numbers are displayed in the next line. Now, the final program can be written as shown below:

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

```
/* Function to find factorial of n */
int fact (int n)
{
          int i, prod = 1;

          for (i = 1; i <= n; i++) prod = prod * i;

          return prod;
}

void main()
{
          int       m, n, r, i;

          printf("Enter the number of rows: ");
          scanf("%d", &m);

          for (n = 0; n < m; n++)
          {
                    for (i = 0; i <= m - n; i++) printf(" ");  // print spaces in the beginning

                    for (r = 0 ; r <= n; r++)                   //compute & print binomial coeffint
                              printf("%d ",fact(n)/(fact(r)*fact(n-r)));

                    printf("\n");                              // Move the cursor to the next line
          }
}
```

8.10 Function prototype

Now, let us see *"What is function prototype/declaration?"*

Definition: As we normally declare the variables before they are used, the functions also should be declared before they are used. This process of declaring the functions before they are used (or called) is called *function prototype.* The function prototype is also called *function declaration.* It is same as function header but terminated with semicolon. It does not contain the body of the function. For example,

```
Ex: int fact(int n);          // It indicates that the function fact accepts an integer
                              // parameter and returns an integer
```

Now, let us see *"What information is contained in function prototype?"* The function prototype contains the following information in a single line terminated by semicolon:

- The type of value returned by the function
- The name of the function
- The number of parameters passed to that function
- The type of each parameter

Ex: **float** atof(char s[]); // It indicates that the function *atof* accepts a character
// string as the parameter and returns an integer value

Now, let us see *"What is the purpose of function prototype?"* When we use certain C compilers we may omit the function prototype. But, it is always better to use and it is strongly recommended to use function prototype because of following reasons:

- Some library functions (e.g. sqrt) do not work correctly if we do not include the header files which contains the function prototypes for various functions.
- If the function prototype is omitted, the compiler does less type checking on a function call. The main idea of including function prototype is to ensure strong type checking of a function call. For example, if function expects to receive one formal parameter of type *double*, the compiler should make sure that the main program sends such a value; If answer returned by the function is of type **double**, the compiler should make sure that the main program expects that.
- Some user-defined functions may not work correctly if function prototypes are omitted.
- Some mistakes in parameter matching will not be caught if function prototype is omitted.

Note: Observe the following points:
- Whether we like it or not, it is better to place the function prototype in the beginning of the program.
- The information specified in the function prototype should match with the information provided in function header.

8.11 SCOPE Rules

Before knowing the meaning of word "scope", note the following points:

- A block is zero or more statements enclosed within braces i.e., inside '{' and '}'.
- The function body is also a block, since function's body is enclosed within braces.
- The body of the function or a block has two sections namely: declaration section and executable section. In declaration section, variables are declared and execution section contains the statements to be executed.
- The global area of a program consists of all statements that are outside functions.

Based on where objects such as variables and functions are declared, the scope and lifetime of objects change. Now, let us *"Define scope"*

Definition: Scope of an object is defined as the region or a boundary of the program in which an object is visible. Objects can be variables or function prototypes.

The two types of scopes are:

- Global scope
- Local scope

Example 8.39: Sample program showing global scope and local scope

```
#include <stdio.h>

   int add(int  a, int  b);        /* Function prototypes */
   int sub(int  a, int  b);         Global scope

   int  f, g;                      /* global variables */

void main()                          Local scope
{
          int sum, difference;

          sum         = add(10, 20);
          difference  = sub(20, 10);

          printf("Sum = %d\n", sum);
          printf("Difference = %d\n", difference);
}

int add(int a, int b)                  Local scope
{
          int  d;

          d = a + b;

          return d;
}

int sub(int a, int b)                    Local scope
{
          int c;

          c = a - b;

          return c;
}
```

- **Global scope:** The objects that are defined outside a block have global scope i.e., any object defined in global area of a program is visible from its definition until the end of the program. For example, the variables declared before all the functions or function prototypes are visible everywhere in the program and they have global scope.
- **Local scope:** The objects that are defined inside a block have local scope. They exist only from the point of their declaration until the end of the block. They are not visible outside the block.

The following objects have global scope in the program given in example 8.39:

- The two function prototypes:

  ```
  int add(int  a, int  b);
  int sub(int  a, int  b);
  ```

  have global scope. Any function can access these two global functions.
- The two variables *f* and *g* declared in global area also have global scope. They can be accessed by all the functions. These variables need not be defined in any of the function.

In the previous program, the following objects have local scope:

- In function main(), the variables *sum* and *difference* have local scope. They are not visible outside the function *main()* i.e., they cannot be accessed or used outside the function main().
- In function add(), the variable *d* has local scope. It is not visible outside the function add() i.e., *d* cannot be accessed or used outside the function add().
- In function sub(), the variable *c* has local scope. It is not visible outside the function sub() i.e., *c* cannot be accessed or used outside the function sub().

Now, let us see *what is the lifespan of a variable.*

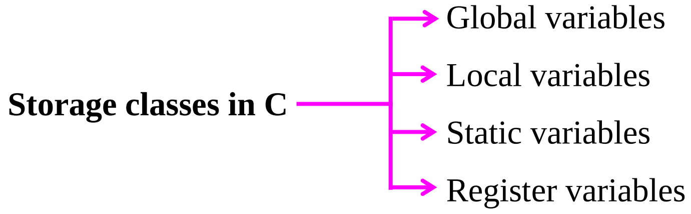
Definition: The lifespan of a variable is defined as the period during which a variable is active during execution of a program. For example,

- The lifespan of a global variable is the lifespan of the program i.e., as long as the program is executing, global variables exist and once the program is terminated, global variables are destroyed.
- The life span of local variables is the life span of a function i.e., they are created whenever there is a call to the function and they are destroyed when control goes out of the function.

8.12 Storage classes

The various storage classes have different scope and lifetime. Now, let us *see what are various storage classes available in C.*

The various storage classes in C language are classified as shown below:



8.12.1 Global variable

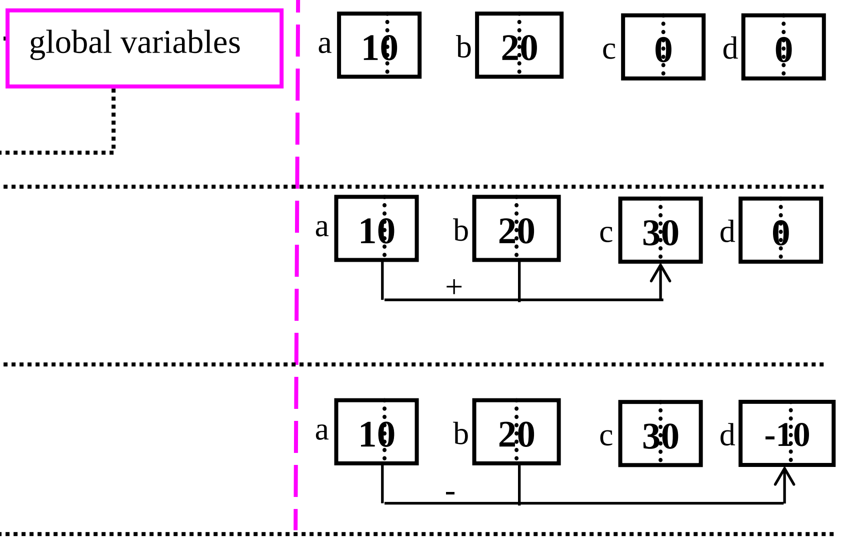
Now, let us see *what are global variables and explain by giving examples.*

Definition: *Global variables* are the variables which are defined before all functions in global area of the program. The global variables have the following features:

- These variables are declared before the functions.
- Memory is allocated for these variables only once and all the allocated memory is initialized to zero.
- These variables can be accessed by any function and are alive and active throughout the program.
- Any function can use a global variable and change its value.
- Memory is de-allocated once the execution of the program is over.

Example 8.40: Program to demonstrate the use of global variables.

```
#include <stdio.h>

int a = 10;
int b = 20;
int c, d;

void add()
{
        c = a + b;
}

void subtract()
{
        d = a – b;
}
```



```
void main()                                  |
{                                            |
        add();                               |   Output
                                             |   a = 10b = 20
        printf("a = %d\nb= %d\n", a, b);     |   a + b = 30
        printf("a + b = %d\n", c);           |
                                             |
        subtract();                          |
        printf("a - b = %d\n",d);            |   a – b = -10
}                                            |
```

Note: When program is terminated, memory allocated for *a*, *b*, *c*, *d* is de-allocated.

Note: The global variables are initialized to 0's by default. In the above program segment, the variables *c* and *d* are initialized to 0's.

In this program, the variables *a*, *b*, *c* and *d* are defined before all the functions. So, these variables can be accessed and modified by any function without passing them as function arguments. We should remember that global variables are visible only from the point of declaration to the end of the program.

Quality tip The number of global variables should be very minimum. If we find ourselves using many global variables, we are probably writing code that will be difficult to maintain and extend. As a thumb rule, "No more than two global variables for every thousand lines of code are allowed" as an industry standard.

8.12.2 Local variable (Automatic variable)

Now, let us see *what are local variables and explain by giving examples.*

Definition: *Local variables* are the variables which are defined within a function. These variables are also called *automatic variables.* All variables must be defined before the first executable in the function. Whenever control enters into the function, memory is allocated for the local variables and whenever control goes out of the function, memory will be de-allocated. That is the reason, the local variables cannot be accessed by any other function. The local variables have the following features:

- These variables cannot be accessed by any function and are alive and active within the function.
- The scope of these variables is limited only to the function in which they are declared and cannot be accessed outside the function.
- No other function can use a local variable and change its value.

Note: A local variable is accessible only within the function or block in which it is defined.

Example 8.41: Program to demonstrate local variables

```
#include <stdio.h>

void add(int a, int b)
{
        int c;

        c = a + b;

        printf("Result = %d\n", c);
}

void main()
{
        int a = 10,  b = 20;

        add(a,b);
}
```

The local variables used in above program are:

- The variable *c* used in function add() is a local variable as it is defined only in that function. If we use the variable *c* outside the function an error "undefined symbol c" is displayed by the compiler.

- The variables *a* and *b* in function main() are also local variables, as they are defined only in function main().

8.12.3 Static variable

Now, let us see *what are static variables and explain by giving examples.*

Definition: The variables that are declared using the keyword **static** are called static variables. The static variables can be declared outside the function or within the function. They have the characteristics of both local and global variables. The declaration of a static variable should begin with the keyword **static**.

For example,

```
static int a, b;
static float c;
```

The static variables have the following features:

- Memory is allocated for these variables only once and memory contents are not initialized to zero.
- If these variables are defined inside a function, they cannot be accessed by any function. But, they are alive throughout the execution of the program. The scope of these variables is limited only to the function in which they are declared and cannot be accessed outside the function.
- No other function can use a static variable and change its value if it is declared inside a function.
- Memory is de-allocated for static variables when the execution of the program stops.

Note: The static variable have some features of local variable and some features of global variable.

Example 8.42: Program to demonstrate static variables

```
#include <stdio.h>

void display()
{
        static int i = 0;

        i++;

        printf("%d\n ",i);
}

void main()
{
        display();
        display();
        display();
        display();
        display();
}
```

Note: The value of *i* is retained even though control goes out of the function. But, it cannot be accessed by other functions.

Output

```
1
2
3
4
5
```

Note: Each time control goes into the function, the previous value of *i* is used and it is updated. The updated value of *i* is used in the next subsequent call of the function.

8.12.4 Register variable

First, let us define a register.

Definition: Registers are storage units inside the microprocessor. When the microprocessor is doing various operations, the data can be temporarily stored in these registers. Since the registers are part of microprocessor, the data stored in these registers can be accessed very fast.

Now, let us see *what are register variables and explain by giving an example.*

Definition: Any variable declared with the qualifier **register** is called a register variable. This declaration instructs the compiler that the variable under use is to be stored in one of the registers but, not in main memory. Register accessing is much faster compared to the memory access and hence, frequently accessed variables such as loop variables (discussed in later sections) are usually stored in register variables. This leads to faster execution of the program. If the register is not free, the compiler can ignore the request and the memory is allocated as it is done for other variables. The register declaration is shown below:

```
register int x;
```

This declaration is allowed only for the local (automatic) variables and to the formal parameters. So, all register variables are automatic. The registers are allocated for these variables as the control enters into the function and registers are freed immediately once the control comes out of the function. Their definitions are valid only to the function in which they are defined.

Now, let us see *what are the differences between local and global variables.*

Global variables	**Local variables**
1. These variables are declared outside all the functions in the global area of the program.	1. These variables are declared inside the function
2. Global variables are alive and holds the values as long as the program is being executed.	2. Local variables are alive when control enters into the function during execution and are destroyed when the control comes out of the function.

3. The global variables are accessible throughout the program	3. The local variables are accessible only in the function inside which they have been created.
4. Global variables are always initialized to zero	4. Local variables are not initialized automatically in some compilers.

Exercises:

1. What is a function? What are the different types of functions?
2. What are library functions? What are the advantages and disadvantages of library functions?
3. What are User Defined Functions (UDFs)?
4. What are the advantages of functions? or What is the need for user defined functions?
5. What is function definition? Explain in detail
6. What are formal parameters? What are actual parameters? What is the difference between them?
7. What is function declaration or function prototype? Explain with example
8. What are the various categories of functions?
9. What are the different ways of passing parameters to the functions? Explain with example
10. How the functions are categorized based on the value returned by the function and parameters accepted?
11. What are the different ways of passing parameters to the functions
12. Design and develop a C function RightShift(x ,n) that takes two integers x and n as input and returns value of the integer x rotated to the right by n positions. Assume the integers are unsigned. Write a C program that invokes this function with different values for x and n and tabulate the results with suitable headings.
13. Write a function to check whether a number is prime or not.
14. Write a function to generate n prime numbers within the range of n1 and n2
15. Write a function to compute factorial of a number
16. Write a function for computing power i.e., x^n
17. Write a function to compute e^x using Taylor's Series
18. Write a function to compute sine value using Taylor's Series
19. Write a function to compute co-sine value using Taylor's Series
20. What is function prototype/declaration
21. Define scope. Explain with example
22. What are various storage classes available in C?
23. What are global and local variables and explain by giving examples
24. What are static variables and explain by giving examples