

A decorative vertical border on the left side of the slide, featuring a complex, wavy pattern of thin, overlapping lines in shades of brown, tan, and cream.

FUNCTIONS

CHAPTER # 09

A decorative vertical border on the right side of the slide, featuring a complex, wavy pattern of thin, overlapping lines in shades of brown, tan, and cream.

FUNCTION

- A function is a block of instructions that together performs a specific task .
- A function is the fundamental element of *Modular Programming* and *Top down Programming* approach.
- A function is a block of instructions that is executed when it is called from some other point of the program.

ADVANTAGES OF FUNCTION

- The complexity of entire program can be divided into simple subtasks and function can be written for each subtask.
- Function help to avoid unnecessary repetition of code. Using functions, a single section of code can be used many times in the same program.
- The functions are short, easier to write, understand and debug.
- A function can be shared by other programs by compiling it separately and loading them together.

CLASSIFICATION OF FUNCTIONs

- Built-in functions/ Library functions:

These functions are pre-defined in C library for solving general programming problems. Such as abs(), cos() etc.

- User defined function:

These functions are defined by the programmer for custom requirement Such as addition(), getdata() etc.

PREDEFINED FUNCTIONS OF C LANGUAGE

Common Mathematical Functions

<u>Function</u>	<u>Syntax</u>	<u>Description</u>
abs()	#include <stdlib.h> int abs(int x);	Returns the absolute value of an integer.
cos()	#include <math.h> double cos(double x);	calculates the cosine of a value.
exp()	#include <math.h> double exp(double x);	Calculates the exponential e to the x.
log()	#include <math.h> double log(double x);	Calculates the natural logarithm of x.
pow()	#include <math.h> double pow(double x, double y);	Calculates x to the power of y.
sin()	#include <math.h> double sin(double x);	Calculates the sine of a value.
sqrt()	#include <math.h> double sqrt(double x);	Calculates the positive square root of the number.
tan()	#include <math.h> double tan(double x);	Calculates the tangent of a value.

COMMON CHARACTER FUNCTIONS

Function	Syntax	Description
getch()	#include <conio.h> int getch(void);	Gets character from keyboard, does not echo to screen (does not display).
getche()	#include <conio.h> int getche(void);	Gets character from the keyboard, echoes to screen.
getchar()	#include <stdio.h> int getchar(void);	Gets character from keyboard, echoes to the screen and needs Enter key to be pressed before proceeding.
putch()	#include <conio.h> int putch(int c);	Outputs character to screen.
fgetc()	#include <stdio.h> Int fgetc(FILE *stream);	Gets character from stream(a file).

USER DEFINED FUNCTIONS

- User defined functions are defined by the programmer for custom requirement.

```
#include <stdio.h>

void functionName()
{
    ... ..
    ... ..
}

int main()
{
    ... ..
    ... ..
    functionName();
}

... ..
... ..
}
```

The diagram illustrates the flow of execution between a user-defined function and the main function. It shows two function definitions: `void functionName()` and `int main()`. The `main` function calls `functionName()`. An arrow points from the call site in `main` to the start of the `functionName` function body. Another arrow points from the end of the `functionName` function body back to the point in `main` immediately following the function call, indicating the return of control to the caller.

There are three main actions associated with User Defined Functions handling in C language:

- 1. Function Prototype (Also called Function Declaration)**
- 2. Function Definition.**
- 3. Function Calling.**

first function example:

```
#include<stdio.h>
int addition( int,int);      /* Function Prototype */
int main ()
{
    int z;
    z = addition (5,3);      /* Function Call */
    printf("The result is  %d" ,z);
    getch();
}
int addition (int a, int b)  /* Function Definition */
{
    int r;
    r=a+b;
    return (r);
}
```

The result is 8

An example to add two integers. To perform this task, we have created an user-defined addNumbers().

```
#include <stdio.h>
#include <conio.h>
int addNumbers(int a, int b);      // function prototype
int main()
{
    int n1,n2,sum;
    printf("Enters two numbers: ");
    scanf("%d %d",&n1,&n2);
    sum = addNumbers(n1, n2);      // function call
    printf("sum = %d",sum);
    getch();
}
int addNumbers(int a, int b)      // function definition
{
    int result;
    result = a+b;
    return result;                // return statement
}
```

1. Function Prototype

- The declaration of a function is called Prototype.
- The prototype tells the compiler in advance about some characteristics of a function used in the program.
- There are three main elements of function prototype.

1)Return Type of function

2)Name of function

3)Arguments

1. Function Prototype

1)Return Type of function:

Tells the compiler that what kind of value function will return, a reserved word “void” is used if function doesn’t return a value.

2)Name of function:

Tells the compiler that what will be the name of function? The rules for naming are identical to the rules for variable declaration.

3)Arguments:

Tell the compiler that how many argument it will receive? And what will the data types? If function will not receive any argument a reserved word “void” will be used.

Syntax of prototype:

data type / **void** Function Name(argument type list / **void**);

OR

returnType functionName(type1 argument1, type2 argument2, ...);

Example:

```
int addNumbers(int a, int b);
```

(A function addNumber will accepts two int type arguments and it will return an int type value to caller.)

2. Function Definition

- The function definition is the function itself. In the definition the program defines the body of the function comprised of programming statements.
- The function definition has two parts:
 1. Function Header
 2. Function Body

2. Function Definition

1. Function Header

The function begins with a header which is exactly the same as the function prototypes except it must not be terminated with semicolon. Header specifies the name of the function and the variable for argument receiving.

2. Function Body

- All statements are written in this part. The statements written in the body are enclosed within curly brackets.
- Example:

```
int getsum(int a,int b,int c)
{
    int x=a+b+c;
    return(x);
}
```


3. Function Calling

A user defined function is called from the main program simply by using the name including the parenthesis which follows the name.

Example:

```
int r=getsum(10,100,200);
```

- A function is a block of instructions that is executed when it is called from some other point of the program.
- The following is its format:

type name (argument1, argument2, ...) statement

- where: type is the type of data returned by the function.
- name is the name by which it will be possible to call the function.
- arguments (as many can be specified as necessary).
- Each argument consists of a type of data followed by a particular name, like in a variable declaration (for example, int x) and which acts within the function like any other variable. They serve to allow passing parameters to the function when this one is called.
- The different parameters are separated by commas.
- Statement is the function's body. It can be a single instruction or a block of instructions. In latter case it must be delimited with key brackets signs {}.

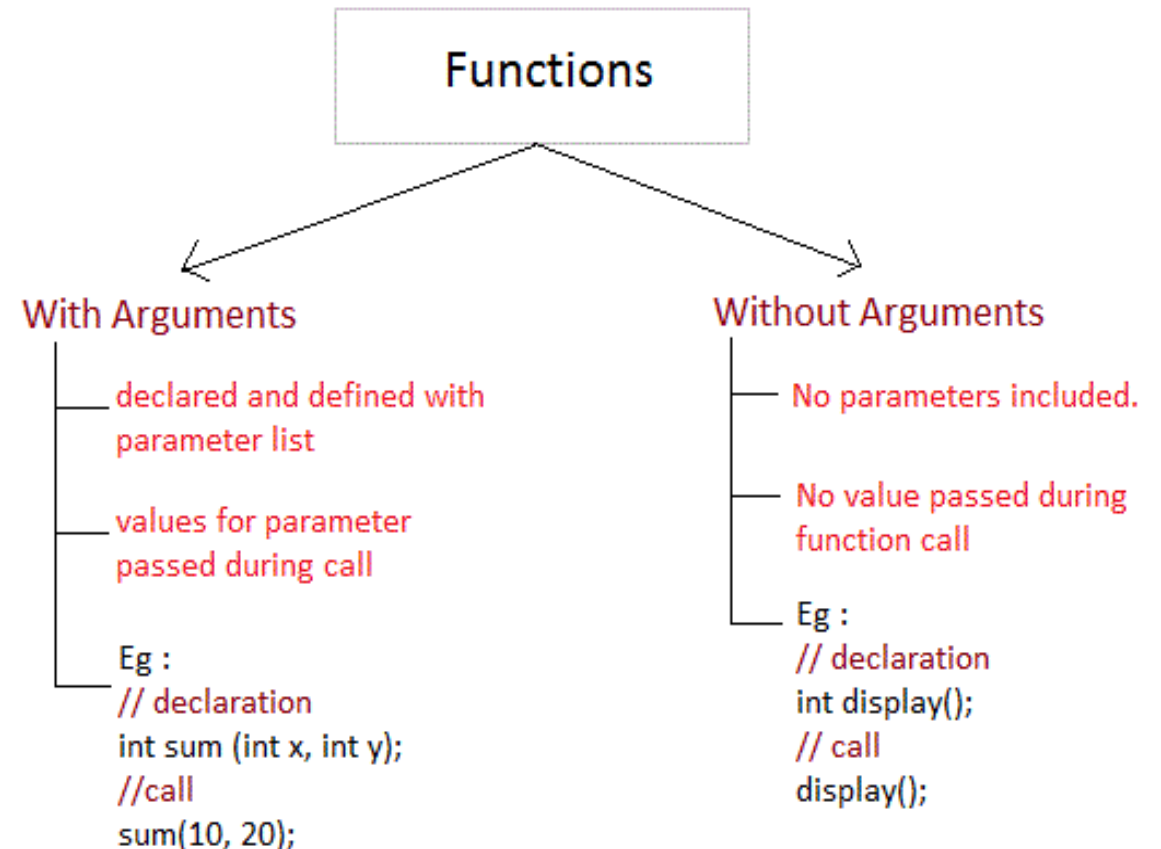
int add(int, int);

- This statement declares a function called add, it has two integer arguments and returns an integer.

Passing Arguments to a function

- Arguments are the values specified during the function call, for which the formal parameters are declared while defining the function.

- It is possible to have a function with parameters but no return type. It is not necessary, that if a function accepts parameter(s), it must return a result too.

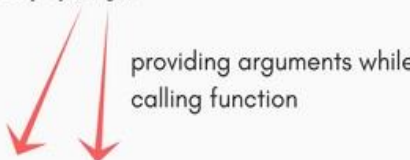


```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ..
    result = multiply(i, j);
    ... ..
}

int multiply(int a, int b)
{
    ... ..
}
```



providing arguments while
calling function

While declaring the function, we have declared two parameters a and b of type int. Therefore, while calling that function, we need to pass two arguments, else we will get compilation error.

And the two arguments passed should be received in the function definition, which means that the function header in the function definition should have the two parameters to hold the argument values.

These received arguments are also known as **formal parameters**.

The name of the variables while declaring, calling and defining a function can be different .

Function Arguments

- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.
- Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
- While calling a function, there are two ways in which arguments can be passed to a function

Local Variables

- Variables that are declared inside a function or block are called local variables.
- They can be used only by statements that are inside that function or block of code.
- Local variables are not known to functions outside their own.

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

```
int main () {
```

```
    /* local variable declaration */
```

```
    int a, b;
```

```
    int c;
```

```
    /* actual initialization */
```

```
    a = 10;
```

```
    b = 20;
```

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

```
    printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
```

```
}
```

Global Variables

- Global variables are defined outside a function, usually on top of the program.
- Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.
- A global variable can be accessed by any function.
- A global variable is available for use throughout your entire program after its declaration.

```
#include <stdio.h>
int g; /* global variable declaration */
int main () {
int a, b; /* local variable declaration */

a = 10; /* actual initialization */
b = 20;
g = a + b;
printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
return 0;
}
```

Functions and Structured Programming

- Structured programming, in which individual program tasks are performed by independent sections of program code.
- "Independent sections of program code" is part of the definition of functions given earlier. Functions and structured programming are closely related.

The Advantages of Structured Programming

- It's easier to write a structured program, because complex programming problems are broken into a number of smaller, simpler tasks.
- Each task is performed by a function in which code and variables are isolated from the rest of the program.
- We can progress more quickly by dealing with these relatively simple tasks one at a time.
- It's easier to debug a structured program. If your program has a bug (something that causes it to work improperly), a structured design makes it easy to isolate the problem to a specific section of code (a specific function).

The Advantages of Structured Programming

- A related advantage of structured programming is the time you can save. If you write a function to perform a certain task in one program, you can quickly and easily use it in another program that needs to execute the same task.
- Even if the new program needs to accomplish a slightly different task, we'll often find that modifying a function you created earlier is easier than writing a new one from scratch. Consider how much you've used the two functions `printf()` and `scanf()` even though you probably haven't seen the code they contain.
- If your functions have been created to perform a single task, using them in other programs is much easier.

Type of User-defined Functions in C

There can be 4 different types of user-defined functions, they are:

- Function with no arguments and no return value
- Function with no arguments and a return value
- Function with arguments and no return value
- Function with arguments and a return value

1.Function with no arguments and no return value

Such functions can either be used to display information or they are completely dependent on user inputs.

Below is an example of a function, which takes 2 numbers as input from user, and display which is the greater number.


```
#include<stdio.h>
#include<conio.h>
void greatNum( ); // function declaration
```

```
int main()
{ greatNum(); // function call
  getch();
}
```

```
void greatNum( ) // function definition
{
  int i, j;
  printf("Enter 2 numbers that you want to compare...");
  scanf("%d%d",&i,&j);
  if(i> j)
  {
    printf("The greater number is: %d",i);
  }
  else {
    printf("The greater number is: %d", j);
  }
}
```

2.Function with no arguments and a return value

We have modified the above example to make the function `greatNum()` return the number which is greater amongst the 2 input numbers.

```
#include<stdio.h>
#include<conio.h>
int greatNum( ); // function declaration

int main()
{
int result;
result=greatNum(); // function call
printf("The greater number is:
%d", result);
getch();
}
```

```
int greatNum( ) // function definition
{
int i, j,greaterNum;
printf("Enter 2 numbers to
compare");
scanf("%d%d",&i,&j);
if(i> j)
{
greaterNum=i;
}
else {
greaterNum= j;
} // returning the result return greaterNum
}
```

3.Function with arguments and no return value

We are using the same function as example again and again, to demonstrate that to solve a problem there can be many different ways.

This time, we have modified the above example to make the function greatNum() take two int values as arguments, but it will not be returning anything.


```
include<stdio.h>
#include<conio.h>
void greatNum(int a,int b); // function
                             declaration

int main()
{
int i, j;
printf("Enter 2 numbers that you
want to compare...");
scanf("%d%d",&i,&j);
greatNum(i, j); // function call
getch();
}
```

```
void greatNum(int x,int y) // function
                             definition
{
if(x > y)
{
printf("The greater number is: %d", x);
}
else
{
printf("The greater number is: %d", y);
}
}
```

4.Function with arguments and a return value

This is the best type, as this makes the function completely independent of inputs and outputs, and only the logic is defined inside the function body.

```
#include<stdio.h>
#include<conio.h>
int greatNum(int a,int b); // function
                             declaration

int main( )
{
    int i, j, result;
    printf("Enter 2 numbers that you
        want to compare...");
    scanf("%d%d",&i,&j);
    result=greatNum(i, j); // function call
    printf("The greater number is: %d",
        result);
    getch();
}
```

```
int greatNum(int x,int y) //
    function
    definition
{
    if(x > y)
    {
        return x;
    }else
    {
        return y;
    }
}
```

Program to find average of 3 numbers using Function in C

```
#include<stdio.h>
#include<conio.h>
void avg (int a, int b, int c);
void main()
{
    int n1,n2,n3;
        clrscr();
    printf("Enter three numbers:");
    scanf("%d %d %d",&n1,&n2,&n3);
    avg(n1,n2,n3);
    getch();
}
```

```
void avg (int a, int b, int c)
{
    float average;
        average=(a+b+c)/3.0;
        printf("Average=%f",average);
}
```


Program to calculate the area of a triangle. $a=(b \times h)/2$

```
#include<stdio.h>
#include<conio.h>
int Area(int l,int b);
int Area(int l,int b)
{
    int a;
    a=l*b;
    return a;
}
```

```
int main()
{
    int length,breath,A;
    printf("Enter length and breath of the Rectangle \n");
    scanf("%d%d",&length,&breath);
    A=Area(length,breath);
    printf("Area of the Rectangle is: %d",A);
    getch();
}
```

Write a program that read integer from keyboard in main program and pass it as an argument to a function where it check whether the no is prime or composite.

Recursive function

- A function that calls itself is known as a recursive function. And, this technique is known as recursion.

How does recursion work?

```
void recurse()
{
    ... ..
    recurse();
    ... ..
}

int main()
{
    ... ..
    recurse();
    ... ..
}
```

The diagram illustrates the flow of recursive calls. A line from the `recurse();` statement in the `main()` function extends to the right and then upwards to a double-headed arrow pointing to the `recurse()` function definition. Another line from the `recurse();` statement inside the `recurse()` function extends to the right and then upwards to a double-headed arrow pointing to the `recurse()` function definition. The text "recursive call" is placed between these two arrows.

- The recursion continues until some condition is met to prevent it.
- To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call, and other doesn't.

Example: Sum of Natural Numbers Using Recursion

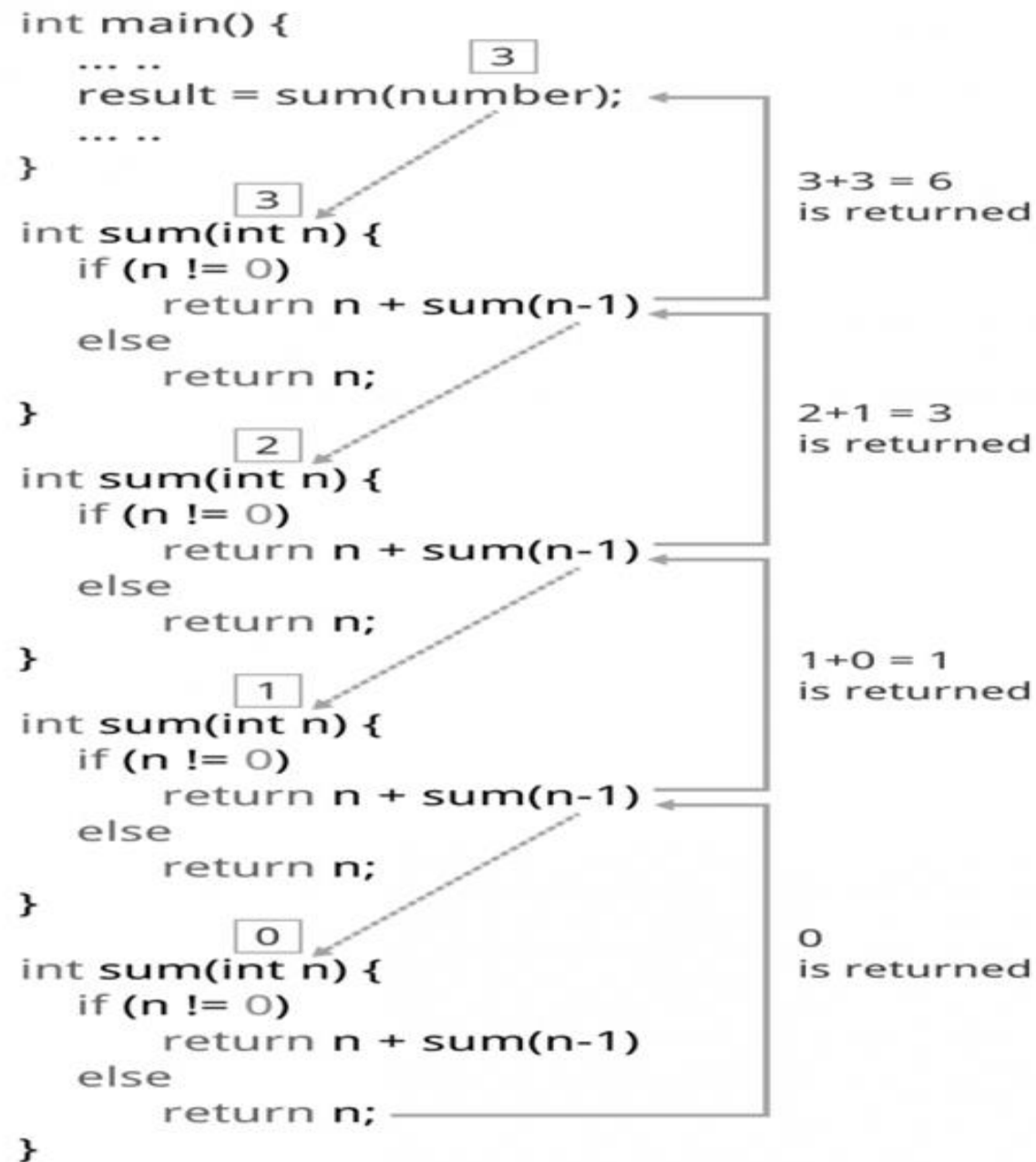
```
1  #include <stdio.h>
2  int sum(int n);
3
4  int main() {
5      int number, result;
6
7      printf("Enter a positive integer: ");
8      scanf("%d", &number);
9
10     result = sum(number);
11
12     printf("sum = %d", result);
13     return 0;
14 }
15
16 int sum(int n) {
17     if (n != 0)
18         // sum() function calls itself
19         return n + sum(n-1);
20     else
21         return n;
22 }
```

```
Enter a positive integer: 3
sum = 6
```

Initially, the sum() is called from the main() function with number passed as an argument.

Suppose, the value of n inside sum() is 3 initially. During the next function call, 2 is passed to the sum() function. This process continues until n is equal to 0.


When n is equal to 0, the if condition fails and the else part is executed returning the sum of integers ultimately to the main() function.



Factorial with recursion

1/12

A Call Stack Calculating 4!


$$4 * 3! = 4!$$

We want to calculate 4!

Functions and Structured Programming

By using functions in your C programs, you can practice *structured programming*, in which individual program tasks are performed by independent sections of program code. "Independent sections of program code" sounds just like part of the definition of functions given earlier, doesn't it? Functions and structured programming are closely related.

The Advantages of Structured Programming

There are two important reasons of structured programming :

1. It's easier to write a structured program, because complex programming problems are broken into a number of smaller, simpler tasks. Each task is performed by a function in which code and variables are isolated from the rest of the program.
2. It's easier to debug a structured program. If your program has a *bug* (something that causes it to work improperly), a structured design makes it easy to isolate the problem to a specific section of code (a specific function).

The Advantages of Structured Programming

- A related advantage of structured programming is the time you can save.

If you write a function to perform a certain task in one program, you can quickly and easily use it in another program that needs to execute the same task.

Even if the new program needs to accomplish a slightly different task, you'll often find that modifying a function you created earlier is easier than writing a new one from scratch.

Consider how much you've used the two functions `printf()` and `scanf()` even though you probably haven't seen the code they contain.

If your functions have been created to perform a single task, using them in other programs is much easier.