

## Module 4: Functions and Parameters

4.1 Function -Introduction of Function, defining a Function, accessing a Function, Function Prototype, Passing Arguments to a Function, Recursive function

4.2 Storage Classes –Auto, Extern, Static, Register

---

A function is a block of code that performs a specific task.

Suppose, you need to create a program to create a circle and color it. You can create two functions to solve this problem:

- create a **circle** function
- create a **color** function

Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

### Types of function

There are two types of function in C programming:

- Standard library functions
- User-defined functions

### Standard library functions

The standard library functions are built-in functions in C programming.

These functions are defined in header files. For example,

The **printf()** is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the **stdio.h** header file.

Hence, to use the **printf()** function, we need to include the **stdio.h** header file using **#include <stdio.h>** .

The **sqrt()** function calculates the square root of a number. The function is defined in the **math.h** header file.

### User-defined function

You can also create functions as per your need. Such functions created by the user are known as user-defined functions.

## How user-defined function works?

```
#include <stdio.h>
void functionName()
{
    ... ..
    ... ..
}

int main()
{
    ... ..
    ... ..

    functionName();

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

The execution of a C program begins from the **main()** function.

When the compiler encounters **functionName();** , control of the program jumps to void **functionName()**

And, the compiler starts executing the codes inside **functionName()**.

The control of the program jumps back to the **main()** function once code inside the function definition is executed.

## How function works in C programming?

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

```
void functionName()
```

```
{
```

```
    ... ..
```

```
    ... ..
```

```
}
```

```
int main()
```

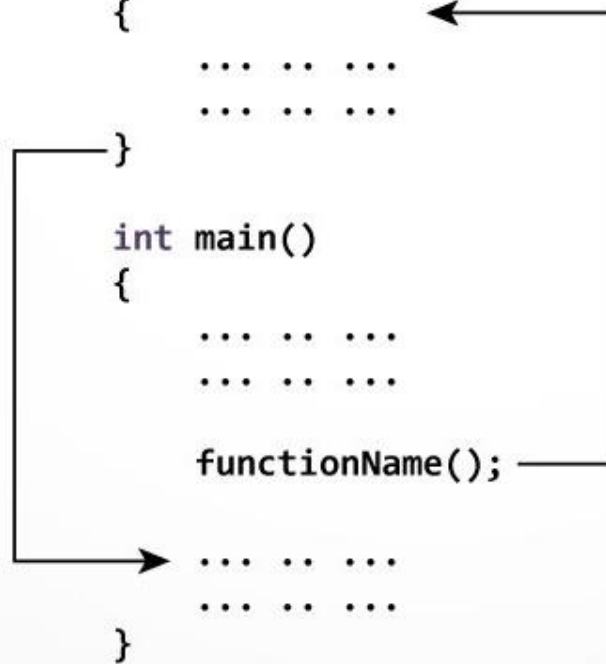
```
{
```

```
    ... ..
```

```
    ... ..
```

```
    functionName();
```

```
}
```



**Note, function names are identifiers and should be unique.**

## C User-defined functions

In this tutorial, you will learn to create user-defined functions in C programming with the help of an example.

A function is a block of code that performs a specific task.

C allows you to define functions according to your need. These functions are known as user-defined functions.

### For example:

Suppose, you need to create a circle and color it depending upon the radius and color. You can create two functions to solve this problem:

- `createCircle()` function
- `color()` function

### Example: User-defined function

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

```
#include <stdio.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);
    return 0;
}

int addNumbers(int a, int b)    // function definition
{
    int result;
    result = a+b;
    return result;              // return statement
}
```

## Function prototype

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.

A function prototype gives information to the compiler that the function may later be used in the program.

### Syntax of function prototype:

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

In the above example, `int addNumbers(int a, int b);` is the function prototype which provides the following information to the compiler:

1. name of the function is **addNumbers()**
2. return type of the function is **int**
3. two arguments of type **int** are passed to the function

The function prototype is not needed if the user-defined function is defined before the **main()** function.

## Calling a function

Control of the program is transferred to the user-defined function by calling it.

Syntax of function call

**functionName(argument1, argument2, ...);**

In the above example, the function call is made using **addNumbers(n1, n2);** statement inside the **main()** function.

## Function definition

Function definition contains the block of code to perform a specific task. In our example, adding two numbers and returning it.

Syntax of function definition

```
return Type functionName(type1 argument1, type2 argument2, ...)  
{  
    //body of the function  
}
```

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

## Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the above example, two variables **n1** and **n2** are passed during the function call.

The parameters **a** and **b** accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.

## How to pass arguments to a function?

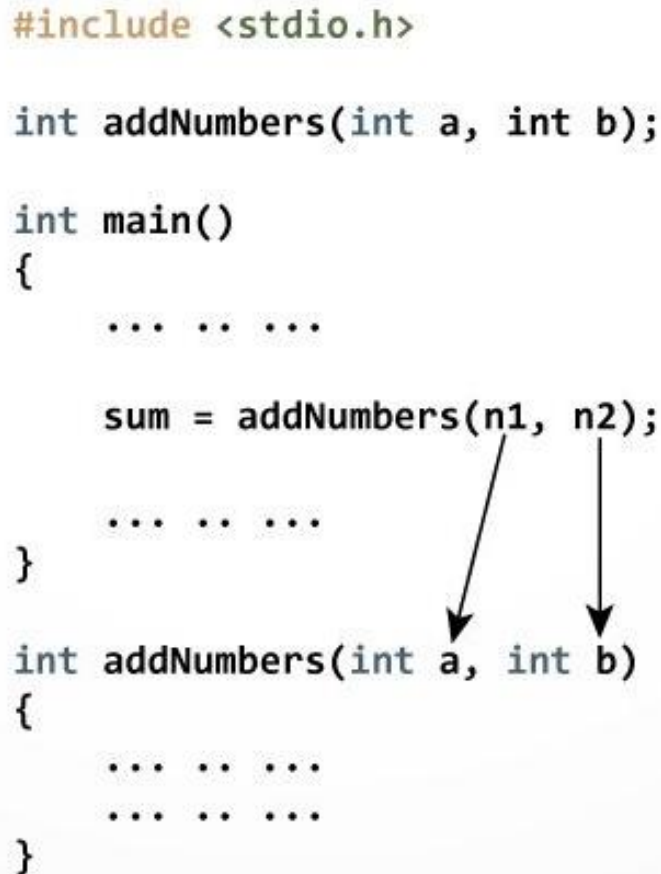
```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);
    ... .. ...
}

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

A diagram illustrating argument passing. In the `main` function, the call `addNumbers(n1, n2)` is shown. Two arrows originate from `n1` and `n2` and point to the parameters `a` and `b` in the function definition `int addNumbers(int a, int b)`.

The type of arguments passed to a function and the formal parameters must match, otherwise, the compiler will throw an error.

If **n1** is of **char** type, **a** also should be of **char** type. If **n2** is of **float** type, variable **b** also should be of **float** type.

A function can also be called without passing an argument.

## Return Statement

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after the return statement.

In the above example, the value of the result variable is returned to the main function. The **sum** variable in the **main()** function is assigned this value.

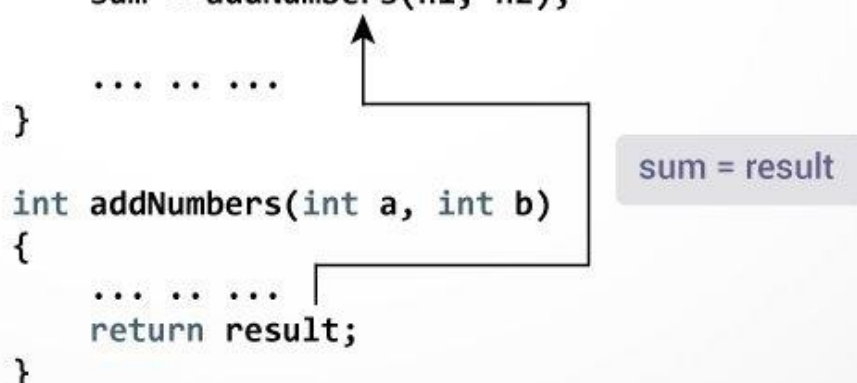
### Return statement of a Function

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ..
    sum = addNumbers(n1, n2);
    ... ..
}

int addNumbers(int a, int b)
{
    ... ..
    return result;
}
```



sum = result



## Syntax of return statement

```
return (expression);
```

For example,

```
return a;
```

```
return (a+b);
```

The type of value returned from the function and the return type specified in the function prototype and function definition must match.

## Types of User-defined Functions in C Programming

In this tutorial, you will learn about different approaches you can take to solve the same problem using functions.

These 4 programs below check whether the integer entered by the user is a prime number or not.

The output of all these programs below is the same, and we have created a user-defined function in each example. However, the approach we have taken in each example is different.

### Example 1: No Argument Passed and No Return Value

```
#include <stdio.h>

void checkPrimeNumber();

int main() {
    checkPrimeNumber(); // argument is not passed
    return 0;
}

// return type is void meaning doesn't return any value
void checkPrimeNumber() {
    int n, i, flag = 0;

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

    // 0 and 1 are not prime numbers
    if (n == 0 || n == 1)
        flag = 1;

    for(i = 2; i <= n/2; ++i) {
        if(n%i == 0) {
```

```

        flag = 1;
        break;
    }
}

if (flag == 1)
    printf("%d is not a prime number.", n);
else
    printf("%d is a prime number.", n);
}

```

The **checkPrimeNumber()** function takes input from the user, checks whether it is a prime number or not, and displays it on the screen.

The empty parentheses in **checkPrimeNumber();** inside the **main()** function indicates that no argument is passed to the function.

The return type of the function is void. Hence, no value is returned from the function.

## Example 2: No Arguments Passed But Returns a Value

```

#include <stdio.h>
int getInteger();

int main() {

    int n, i, flag = 0;

    // no argument is passed
    n = getInteger();

    // 0 and 1 are not prime numbers
    if (n == 0 || n == 1)
        flag = 1;

    for(i = 2; i <= n/2; ++i) {
        if(n%i == 0){
            flag = 1;
            break;
        }
    }
}

```

```

}

if (flag == 1)
    printf("%d is not a prime number.", n);
else
    printf("%d is a prime number.", n);

return 0;
}

// returns integer entered by the user
int getInteger() {
    int n;

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

    return n;
}

```

The empty parentheses in the **n = getInteger();** statement indicates that no argument is passed to the function. And, the value returned from the function is assigned to **n**.

Here, the **getInteger()** function takes input from the user and returns it. The code to check whether a number is prime or not is inside the **main()** function.

### Example 3: Argument Passed But No Return Value

```

#include <stdio.h>
void checkPrimeAndDisplay(int n);

int main() {

    int n;

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

```

```

// n is passed to the function
checkPrimeAndDisplay(n);

return 0;
}

// return type is void meaning doesn't return any value
void checkPrimeAndDisplay(int n) {
    int i, flag = 0;

    // 0 and 1 are not prime numbers
    if (n == 0 || n == 1)
        flag = 1;

    for(i = 2; i <= n/2; ++i) {
        if(n%i == 0){
            flag = 1;
            break;
        }
    }

    if(flag == 1)
        printf("%d is not a prime number.",n);
    else
        printf("%d is a prime number.", n);
}

```

The integer value entered by the user is passed to the **checkPrimeAndDisplay()** function.

Here, the **checkPrimeAndDisplay()** function checks whether the argument passed is a prime number or not and displays the appropriate message.

## Example 4: Argument Passed and Returns a Value

```
#include <stdio.h>
int checkPrimeNumber(int n);

int main() {

    int n, flag;

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

    // n is passed to the checkPrimeNumber() function
    // the returned value is assigned to the flag variable
    flag = checkPrimeNumber(n);

    if(flag == 1)
        printf("%d is not a prime number",n);
    else
        printf("%d is a prime number",n);

    return 0;
}

// int is returned from the function
int checkPrimeNumber(int n) {

    // 0 and 1 are not prime numbers
    if (n == 0 || n == 1)
        return 1;

    int i;

    for(i=2; i <= n/2; ++i) {
        if(n%i == 0)
            return 1;
    }

    return 0;
}
```

The input from the user is passed to the **checkPrimeNumber()** function.

The **checkPrimeNumber()** function checks whether the passed argument is prime or not.

If the passed argument is a prime number, the function returns **0**. If the passed argument is a non-prime number, the function returns **1**. The return value is assigned to the flag variable.

Depending on whether flag is **0** or **1**, an appropriate message is printed from the **main()** function.

## Which approach is better?

Well, it depends on the problem you are trying to solve. In this case, passing an argument and returning a value from the function (example 4) is better.

A function should perform a specific task. The **checkPrimeNumber()** function doesn't take input from the user nor it displays the appropriate message. It only checks whether a number is prime or not.

## Recursion in C

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

### How recursion works?

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

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

### How does recursion work?

```
void recurse() ←  
{  
    ... ..  
    recurse(); — recursive call  
    ... ..  
}  
  
int main()  
{  
    ... ..  
    recurse(); —  
    ... ..  
}
```



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

```
#include <stdio.h>
int sum(int n);
int main() {
    int number, result;

    printf("Enter a positive integer: ");
    scanf("%d", &number);

    result = sum(number);

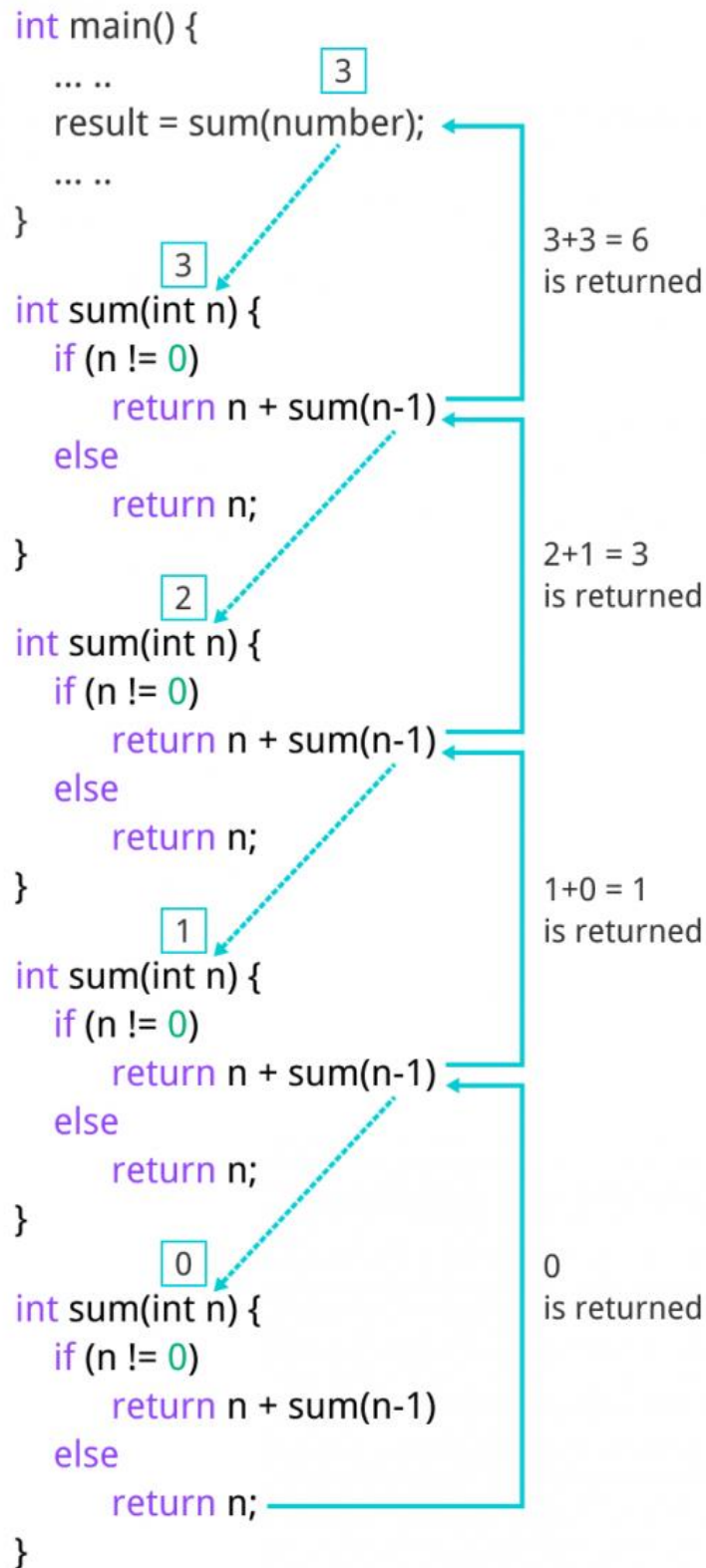
    printf("sum = %d", result);
    return 0;
}
int sum(int n) {
    if (n != 0)
        // sum() function calls itself
        return n + sum(n-1);
    else
        return n;
}
```

### Output

```
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.



## Advantages and Disadvantages of Recursion

Recursion makes program elegant. However, if performance is vital, use loops instead as recursion is usually much slower.

That being said, recursion is an important concept. It is frequently used in data structure and algorithms. For example, it is common to use recursion in problems such as tree traversal.

## Storage Classes in C

Every variable in C programming has two properties: type and storage class.

Type refers to the data type of a variable. And, storage class determines the scope, visibility and lifetime of a variable.

There are 4 types of storage class:

1. automatic
2. external
3. static
4. register

### Local Variable

The variables declared inside a block are automatic or local variables. The local variables exist only inside the block in which it is declared.

Let's take an example.

```
#include <stdio.h>

int main(void) {

    for (int i = 0; i < 5; ++i) {
        printf("C programming");
    }

    // Error: i is not declared at this point
    printf("%d", i);
    return 0;
}
```

When you run the above program, you will get an error undeclared identifier **i**. It's because **i** is declared inside the for loop block. Outside of the block, it's undeclared.

Let's take another example.

**Let's take another example.**

```
int main() {  
    int n1; // n1 is a local variable to main()  
}
```

```
void func() {  
    int n2; // n2 is a local variable to func()  
}
```

In the above example, **n1** is local to **main()** and **n2** is local to **func()**. This means you cannot access the **n1** variable inside **func()** as it only exists inside **main()**. Similarly, you cannot access the **n2** variable inside **main()** as it only exists inside **func()**.

## Global Variable

Variables that are declared outside of all functions are known as external or global variables. They are accessible from any function inside the program.

### Example 1: Global Variable

```
#include <stdio.h>  
void display();  
  
int n = 5; // global variable  
  
int main()  
{  
    ++n;  
    display();  
    return 0;  
}  
  
void display()  
{
```

```
    ++n;  
    printf("n = %d", n);  
}
```

## Output

n = 7

Suppose, a global variable is declared in **file1**. If you try to use that variable in a different file **file2**, the compiler will complain. To solve this problem, keyword **extern** is used in **file2** to indicate that the external variable is declared in another file.

## Register Variable

The **register** keyword is used to declare register variables. Register variables were supposed to be faster than local variables.

However, modern compilers are very good at code optimization, and there is a rare chance that using register variables will make your program faster.

Unless you are working on embedded systems where you know how to optimize code for the given application, there is no use of register variables.

## Static Variable

A static variable is declared by using the static keyword. For example;

```
static int i;
```

The value of a static variable persists until the end of the program.

## Example 2: Static Variable

```
#include <stdio.h>  
void display();
```

```
int main()
```

```
{
    display();
    display();
}
void display()
{
    static int c = 1;
    c += 5;
    printf("%d ",c);
}
```

## Output

**6 11**

During the first function call, the value of **c** is initialized to **1**. Its value is increased by **5**. Now, the value of **c** is **6**, which is printed on the screen.

During the second function call, **c** is not initialized to **1** again. It's because **c** is a static variable. The value **c** is increased by **5**. Now, its value will be **11**, which is printed on the screen.

## SAMPLE PROGRAMS

### 1. Prime Numbers Between Two Integers

```
#include <stdio.h>

int checkPrimeNumber(int n);

int main() {
    int n1, n2, i, flag;
    printf("Enter two positive integers: ");
    scanf("%d %d", &n1, &n2);
    // swap n1 and n2 if n1 > n2
    if (n1 > n2) {
        n1 = n1 + n2;
        n2 = n1 - n2;
        n1 = n1 - n2;
    }
    printf("Prime numbers between %d and %d are: ", n1, n2);
    for (i = n1 + 1; i < n2; ++i) {
        // flag will be equal to 1 if i is prime
        flag = checkPrimeNumber(i);
        if (flag == 1) {
            printf("%d ", i);
        }
    }
    return 0;
}
```



```
// user-defined function to check prime number
int checkPrimeNumber(int n) {
    int j, flag = 1;

    for (j = 2; j <= n / 2; ++j) {

        if (n % j == 0) {
            flag = 0;
            break;
        }
    }

    return flag;
}
```

**Output**

**Enter two positive integers: 12**

**30**

**Prime numbers between 12 and 30 are: 13 17 19 23 29**

## 2. C Program to Check Prime or Armstrong Number Using User-defined Function

```
#include <math.h>
#include <stdio.h>
int checkPrimeNumber(int n);
int checkArmstrongNumber(int n);
int main() {
    int n, flag;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    // check prime number
    flag = checkPrimeNumber(n);
    if (flag == 1)
        printf("%d is a prime number.\n", n);
    else
        printf("%d is not a prime number.\n", n);
    // check Armstrong number
    flag = checkArmstrongNumber(n);
    if (flag == 1)
        printf("%d is an Armstrong number.", n);
    else
        printf("%d is not an Armstrong number.", n);
    return 0;
}
```

```

// function to check prime number
int checkPrimeNumber(int n) {
    int i, flag = 1, squareRoot;
    // computing the square root
    squareRoot = sqrt(n);
    for (i = 2; i <= squareRoot; ++i) {
        // condition for non-prime number
        if (n % i == 0) {
            flag = 0;
            break;
        }
    }
    return flag;
}

// function to check Armstrong number
int checkArmstrongNumber(int num) {
    int originalNum, remainder, n = 0, flag;
    double result = 0.0;
    // store the number of digits of num in n
    for (originalNum = num; originalNum != 0; ++n) {
        originalNum /= 10;
    }
    for (originalNum = num; originalNum != 0; originalNum /= 10) {
        remainder = originalNum % 10;

```

```
// store the sum of the power of individual digits in result
result += pow(remainder, n);
}
// condition for Armstrong number
if (round(result) == num)
    flag = 1;
else
    flag = 0;
return flag;
}
```

## **Output**

**Enter a positive integer: 407**

**407 is not a prime number.**

**407 is an Armstrong number.**

### 3. Find the sum of natural numbers using recursion

```
#include <stdio.h>

int addNumbers(int n);

int main() {
    int num;

    printf("Enter a positive integer: ");
    scanf("%d", &num);
    printf("Sum = %d", addNumbers(num));
    return 0;
}

int addNumbers(int n) {
    if (n != 0)
        return n + addNumbers(n - 1);
    else
        return n;
}
```

#### Output

Enter a positive integer: 20

Sum = 210

#### 4. Calculate the factorial of a number using recursion

```
#include<stdio.h>

long int multiplyNumbers(int n);

int main() {
    int n;
    printf("Enter a positive integer: ");
    scanf("%d",&n);
    printf("Factorial of %d = %ld", n, multiplyNumbers(n));
    return 0;
}

long int multiplyNumbers(int n) {
    if (n>=1)
        return n*multiplyNumbers(n-1);
    else
        return 1;
}
```

#### Output

Enter a positive integer: 6

Factorial of 6 = 720

## 5. Find G.C.D using recursion

```
#include <stdio.h>

int hcf(int n1, int n2);

int main() {
    int n1, n2;

    printf("Enter two positive integers: ");
    scanf("%d %d", &n1, &n2);
    printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1, n2));
    return 0;
}

int hcf(int n1, int n2) {
    if (n2 != 0)
        return hcf(n2, n1 % n2);
    else
        return n1;
}
```

### Output

Enter two positive integers: 366

60

G.C.D of 366 and 60 is 6.

## 6. Reverse a sentence using recursion

```
#include <stdio.h>

void reverseSentence();

int main() {
    printf("Enter a sentence: ");
    reverseSentence();
    return 0;
}

void reverseSentence() {
    char c;
    scanf("%c", &c);
    if (c != '\n') {
        reverseSentence();
        printf("%c", c);
    }
}
```

### Output

Enter a sentence: margorp emosewa

awesome program



## 7. Calculate the power of a number using recursion

```
#include <stdio.h>

int power(int n1, int n2);

int main() {
    int base, a, result;
    printf("Enter base number: ");
    scanf("%d", &base);
    printf("Enter power number(positive integer): ");
    scanf("%d", &a);
    result = power(base, a);
    printf("%d^%d = %d", base, a, result);
    return 0;
}

int power(int base, int a) {
    if (a != 0)
        return (base * power(base, a - 1));
    else
        return 1;
}
```

### Output

Enter base number: 3

Enter power number(positive integer): 4

3<sup>4</sup> = 81

## 8. Convert a binary number to decimal and vice-versa

```
// convert binary to decimal
```

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

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

```
// function prototype
```

```
int convert(long long);
```

```
int main() {
```

```
    long long n;
```

```
    printf("Enter a binary number: ");
```

```
    scanf("%lld", &n);
```

```
    printf("%lld in binary = %d in decimal", n, convert(n));
```

```
    return 0;
```

```
}
```

```
// function definition
```

```
int convert(long long n) {
```

```
    int dec = 0, i = 0, rem;
```

```
    while (n!=0) {
```

```
        rem = n % 10;
```

```
        n /= 10;
```

```
        dec += rem * pow(2, i);
```

```
        ++i;
```

```
}  
return dec;  
}
```

## Output

**Enter a binary number: 1101**

**1101 in binary = 13 in decimal**

```
// convert decimal to binary
```

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

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

```
long long convert(int);
```

```
int main() {
```

```
    int n, bin;
```

```
    printf("Enter a decimal number: ");
```

```
    scanf("%d", &n);
```

```
    bin = convert(n);
```

```
    printf("%d in decimal = %lld in binary", n, bin);
```

```
    return 0;
```

```
}
```

```
long long convert(int n) {
```

```
    long long bin = 0;
```

```
    int rem, i = 1;
```

```
    while (n!=0) {
```

```
        rem = n % 2;
```

```
n /= 2;

bin += rem * i;

i *= 10;

}

return bin;

}
```

### **Output**

**Enter a decimal number: 13**

**13 in decimal = 1101 in binary**

## 9. Convert an octal Number to decimal and vice-versa

### //Program to Convert Decimal to Octal

```
#include <stdio.h>
#include <math.h>
int convertDecimalToOctal(int decimalNumber);
int main()
{
    int decimalNumber;
    printf("Enter a decimal number: ");
    scanf("%d", &decimalNumber);
    printf("%d in decimal = %d in octal", decimalNumber,
convertDecimalToOctal(decimalNumber));
    return 0;
}
int convertDecimalToOctal(int decimalNumber)
{
    int octalNumber = 0, i = 1;

    while (decimalNumber != 0)
    {
        octalNumber += (decimalNumber % 8) * i;
        decimalNumber /= 8;
        i *= 10;
    }
    return octalNumber;
```

```
}
```

## Output

Enter a decimal number: 78

78 in decimal = 116 in octal

## //Program to Convert Octal to Decimal

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

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

```
long long convertOctalToDecimal(int octalNumber);
```

```
int main()
```

```
{
```

```
    int octalNumber;
```

```
    printf("Enter an octal number: ");
```

```
    scanf("%d", &octalNumber);
```

```
    printf("%d in octal = %lld in decimal", octalNumber,  
convertOctalToDecimal(octalNumber));
```

```
    return 0;
```

```
}
```

```
long long convertOctalToDecimal(int octalNumber)
```

```
{
```

```
    int decimalNumber = 0, i = 0;
```

```
    while(octalNumber != 0)
```

```
    {
```

```
        decimalNumber += (octalNumber%10) * pow(8,i);
```



```
    ++i;
    octalNumber/=10;
}

i = 1;

return decimalNumber;
}
```

## Output

Enter an octal number: 116

116 in octal = 78 in decimal

## 10. Convert a binary number to octal and vice-versa

### // Program to Convert Binary to Octal

```
#include <math.h>
#include <stdio.h>
int convert(long long bin);
int main() {
    long long bin;
    printf("Enter a binary number: ");
    scanf("%lld", &bin);
    printf("%lld in binary = %d in octal", bin, convert(bin));
    return 0;
}

int convert(long long bin) {
    int oct = 0, dec = 0, i = 0;

    // converting binary to decimal
    while (bin != 0) {
        dec += (bin % 10) * pow(2, i);
        ++i;
        bin /= 10;
    }
    i = 1;

    // converting to decimal to octal
```

```
while (dec != 0) {  
    oct += (dec % 8) * i;  
    dec /= 8;  
    i *= 10;  
}  
return oct;  
}
```

## **Output**

**Enter a binary number: 101001**

**101001 in binary = 51 in octal**

## // Program to Convert Octal to Binary

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

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

```
long long convert(int oct);
```

```
int main() {
```

```
    int oct;
```

```
    printf("Enter an octal number: ");
```

```
    scanf("%d", &oct);
```

```
    printf("%d in octal = %lld in binary", oct, convert(oct));
```

```
    return 0;
```

```
}
```

```
long long convert(int oct) {
```

```
    int dec = 0, i = 0;
```

```
    long long bin = 0;
```

```
    // converting octal to decimal
```

```
    while (oct != 0) {
```

```
        dec += (oct % 10) * pow(8, i);
```

```
        ++i;
```

```
        oct /= 10;
```

```
    }
```

```
    i = 1;
```

```
    // converting decimal to binary
```

```
while (dec != 0) {  
    bin += (dec % 2) * i;  
    dec /= 2;  
    i *= 10;  
}  
return bin;  
}
```

### **Output**

**Enter an octal number: 67**

**67 in octal = 110111 in binary**