Functions

(Introduction to modular programming)

- Divide your large program into small modules

- Blocks are also a way to create module – not fully independent module – but you can create isolation upto a certain level. – How we can create blocks in C.
- C is also called block structured language -
- Function can represent one module. Function – supports full isolation.

```
main (){

        // variables are declared

        for(I = 1 to 1000)
                for (J = 1 to 1000)
                        // ....
                        // ....
                        // ....
                        // ....

}
```

```
main (){
    // variables are declared
    for(I = 1 to 1000)
        for (J = 1 to 1000){
            Call a module r = check_relatively_prime (I, J)
            r will be Yes / No.
            Print the pair if r is YES.
        }
}
```

```
check_relatively_prime (I, J){

        // code
        // code
        result ← Yes / No

        Return result

}
```

What is a function in C

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

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

Example of complex program ?

Examples – of problems

Write a program to do the following statistical operations based on users need -

1. Average
2. Standard-deviation
3. Mode
4. Median
5. Sorted order
6. Search a given number

Think of -

How big the program will become ?

The program can be divided into several chunks –

Each chunk can be of huge size

Each chunk is independent

Think of -

Each independent unit can be solved in a single function

Easy understanding and structuring of the code

Modular programing

Reuse

Functions

Types of function

There are two types of function in C programming:

Standard library functions

printf, scanf, …., pow,…. Etc.

User-defined functions

How user-defined function works?

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

   functionName1();
   functionName2();
   … .. …
   … .. …
}
```

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 CPU starts executing the codes inside **functionName**()

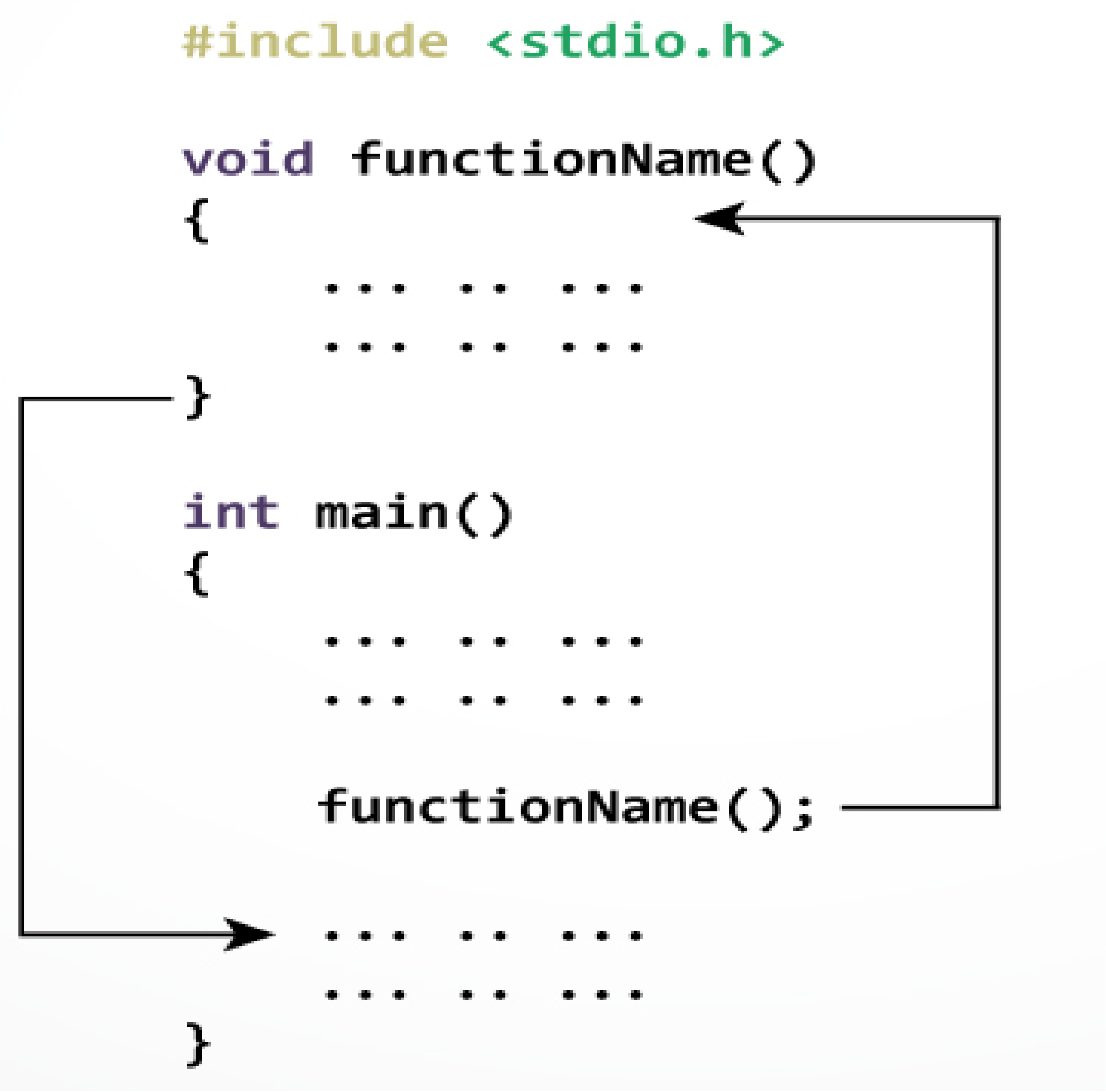
How user-defined function works?

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

int main()
{
    ... .. ...
    ... .. ...
    functionName();
    ... .. ...
    ... .. ...
   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 CPU starts executing the codes inside **functionName**()



```
#include <stdio.h>

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

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

    functionName();

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

Advantages of user-defined function

1. The program will be easier to understand, maintain and debug

2. Reusable codes that can be used in other programs

3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers

Example: User-defined function

Here is an example to add two integers. 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, ...);

Example

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

Name of the function is **addNumbers**()

Return type of the function is **int**

2 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.

- **Example: User-defined function**
- Here is an example to add two integers. An user-defined **addNumbers**().

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

Usage pattern - 1

```
#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
}
```

Usage pattern - 2

```
#include <stdio.h>
int addNumbers(int a, int b)              // function definition
{
   int result;
   result = a+b;
   return result;                         // return statement
}
int main()
{
   int n1,n2,sum;

   printf("Enters two numbers: ");
   scanf("%d %d",&n1,&n2);
   sum = addNumbers(n1, n2);              // function call a = n1, b = n2
   printf("sum = %d",sum);
   return 0;
}
```

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 addNumber it is, adding two numbers and returning it.

Syntax of function definition

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

Example: User-defined function

Here is an example to add two integers. 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
}
```

Passing arguments to a function

Argument refers to the variables passed to the function.

In addNumbers() two variables n1 and n2 are passed during the function call.

Passing arguments to a function

The **parameters** a and b accepts the passed arguments in the function definition.

These arguments are called **formal parameters** of the function.

Example: User-defined function

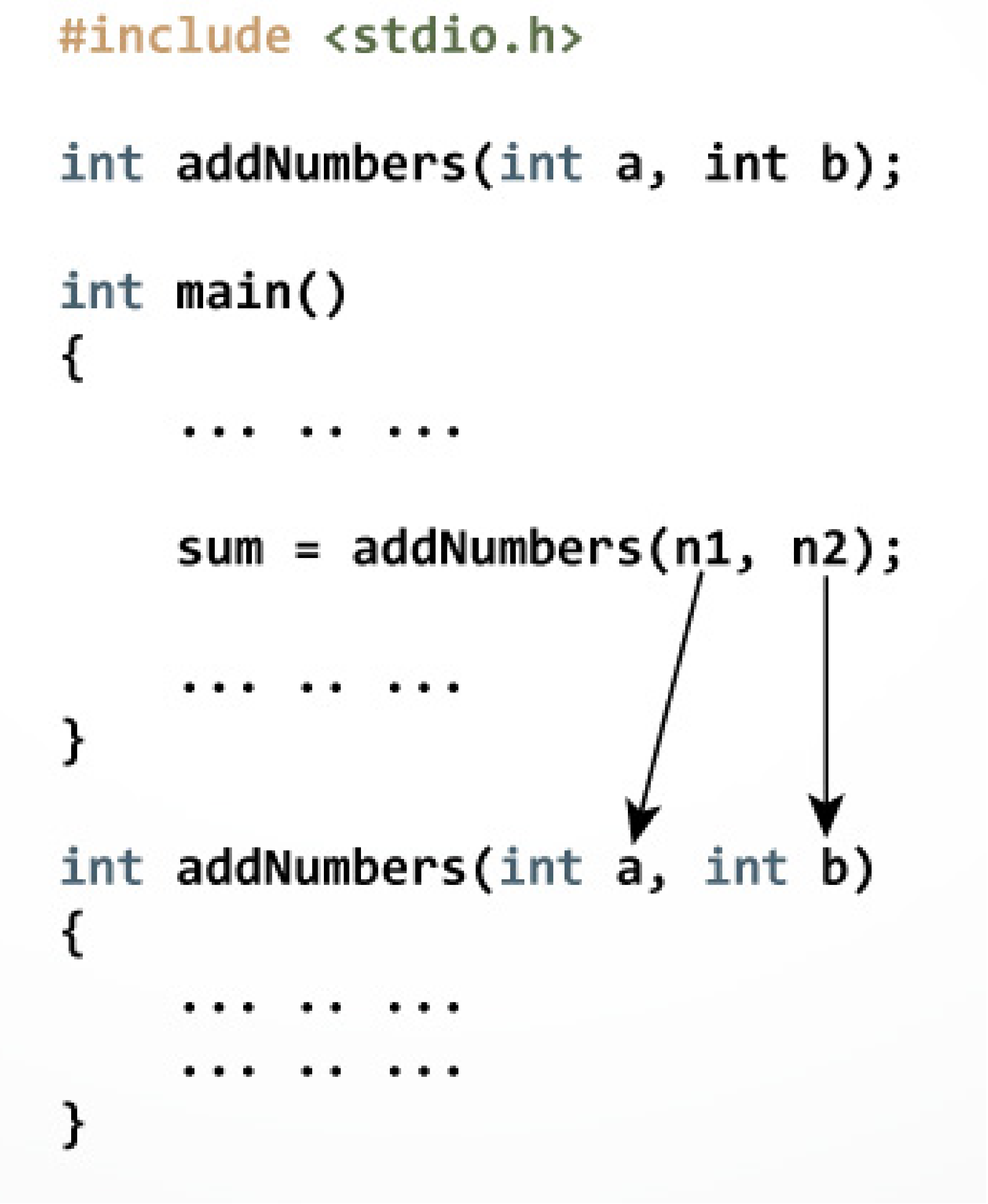
Here is an example to add two integers. An user-defined **addNumbers**().

```
#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
}
```

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

The number of arguments passed to a function and the number of the formal parameters must match, otherwise, the compiler error



```
#include <stdio.h>

int addNumbers(int a, int b);

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

    sum = addNumbers(n1, n2);


    ... .. ...
}

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

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.

Example: User-defined function

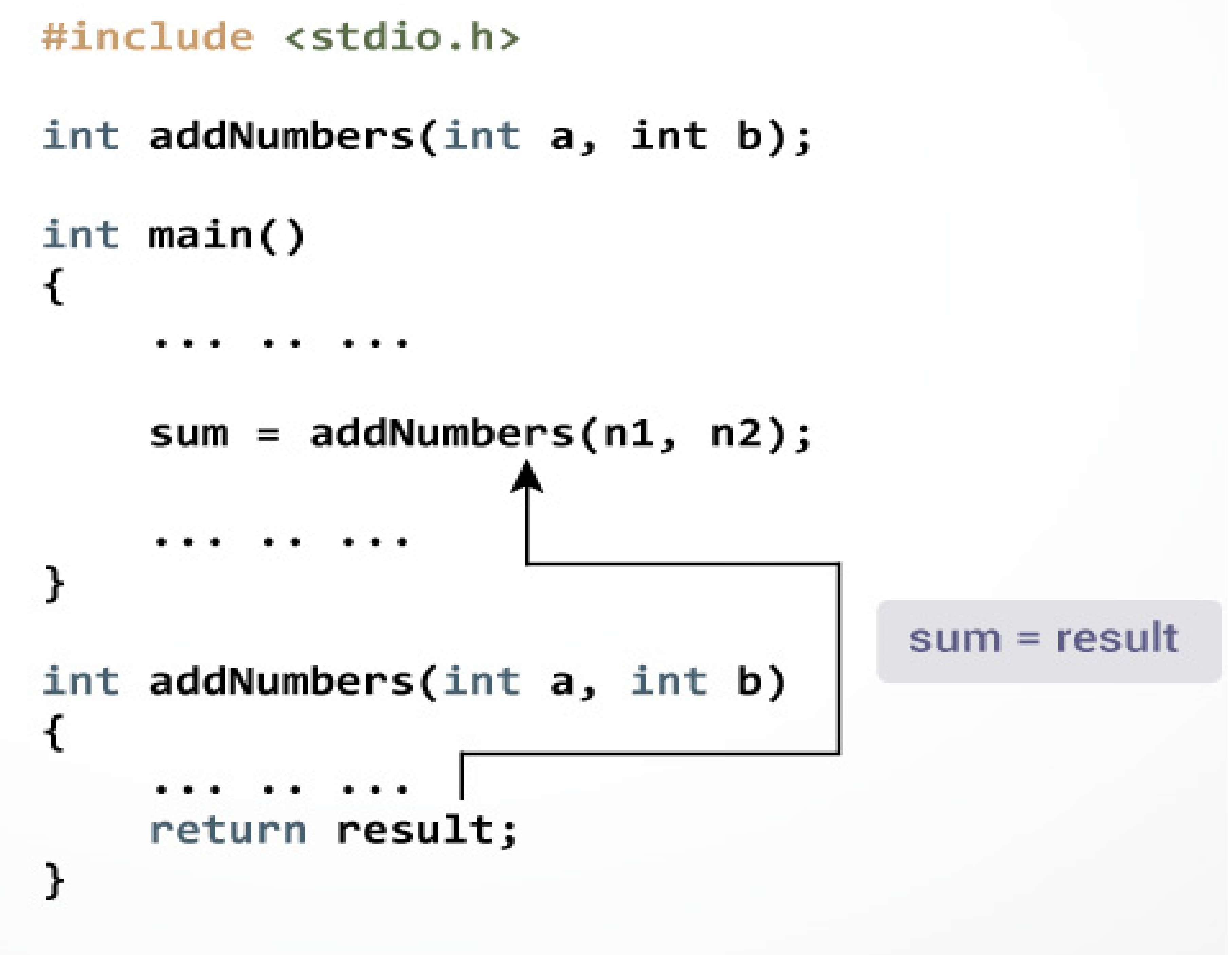
Here is an example to add two integers. An user-defined **addNumbers**().

```
#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
}
```

Return Statement

- The value of the result variable is returned to the main function.
- The sum variable in the main() function is assigned this value.



```
#include <stdio.h>

int addNumbers(int a, int b);

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

    sum = addNumbers(n1, n2);

    ... .. ...
}

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

Return Statement

Syntax of return statement

return (expression);

For example,

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

Return Statement

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

Different possible ways – through examples

Write a program using a function to check - whether the integer entered by the user is a prime number or not

No arguments 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);

    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0)
        {
            flag = 1;
        }
    }
    if (flag == 1)
        printf("%d is not a prime number.", n);
    else
        printf("%d is a prime number.", n);
}
```

No arguments passed and no return value

Explanation

- 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**(); statement 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.

No arguments passed but a return value

```
int getInteger();

int main()
{
    int n, i, flag = 0;

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

    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;
}
```

No arguments passed but a return value

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

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

    return n;
}
```

No arguments passed but a return value

Explanation

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

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.

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;

    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);
}
```

Explanation

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

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

Argument passed and a return 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;
}
```

Argument passed and a return value

Argument passed and a return value

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

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

    return 0;
}
```

More modularity

```
#include <stdio.h>
int checkPrime(int);
int getInteger();
void printResult(int);

int main()
{
        int n, flag;

        n = getInteger();

        flag = checkPrime(n);

        printResult(flag);
}
```

More modularity

```
#include <stdio.h>

int getInteger(void){
        // Use scanf…
}

int checkPrime(int p){
        // Use for loop and return the flag …
}

void printResults(int f){
        // Use printf statements ….
}
```

Example

```
#include <stdio.h>
int checkPrime(int);
int getInteger();
void printResult(int);

int main()
{
        int n, flag, a[100], b[5][10];

        n = getInteger();

        flag = checkPrime(n, a, b);

        printResult(flag);
}
```

More modularity

```
#include <stdio.h>
int checkPrime(int);
int getIntegers();
void printResult(int);

int main()
{
        int n, flag, a[100];

        getIntegers(a);

        n = checkPrime(a);

        printResult(n);
}
```

More modularity

```
#include <stdio.h>

int getIntegers( int a[]){
        a[0]. ….
        // Use scanf…
}

int checkPrime(int n[]){
        // Use for loop and return the flag …
}

void printResults(int f){
        // Use printf statements ….
}
```

Global variable

```
#include <stdio.h>
int checkPrime(int);
int getIntegers();
void printResult(int);
int n, flag, a[100];
int main()
{
        getIntegers();

        checkPrime();

        printResult();
}
```

Global Variables

```
#include <stdio.h>

void getIntegers(void){
        a[0]. ….
        // Use scanf…
}

void checkPrime(void){
        // Use for loop and return the flag …
}

void printResults(void){
        // Use printf statements ….
}
```

Why function - ?
Re-use of the same code

```
int main(){

        ……
        ……
        //declarations../..
        ….
        average computation,          // Based on numbers given by the user
        use the average value for some other operation….
        // Some new numbers are generated…
        average computation, of some different set of numbers // Based on the new numbers…
        …..
        average computation, of some different set of numbers // Based on the new numbers…
}
```

Why function - ?
Re-use of the same code

```
int main(){

            ……
            ……
            //declarations../..
            ….
            avg = compute_average(inputs..)
            use the average value for some other operation….
            // Some new numbers are generated…
            avg1 = compute_average(inputs_1..)
            …..
            avg2 = compute_average(inputs_2…)
}
```

A function can call another function

```
main(){

        int primes[100];

        int number = get_primes(x,y,s,primes);

        // x and y are the upper bound and the lower bound
        // s is the step , ie x, x+s, x+2s, x+3s … these are to be checked
        // if s = 1, its all numbers

}
```

A function can call another function

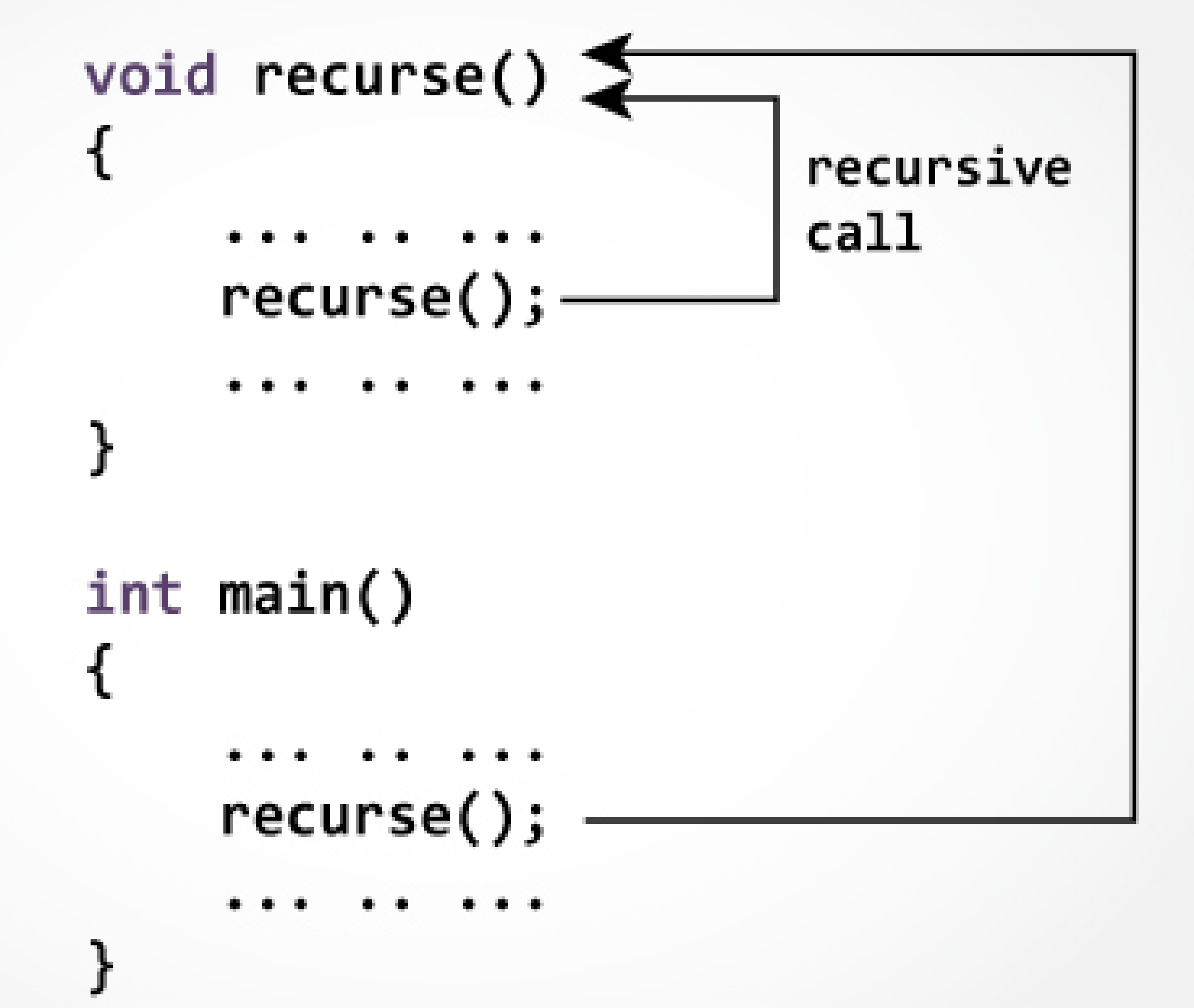
```
int get_primes(int x, int y, int s, int primes[]){

        int i;
        int count = 0;
        for(i=x;    i<=y;       x=x+s){
                flag = 0;
                // Check i is prime or not;
                flag = check_prime(i);
                if (flag==1) primes[count++] = i;
        }

}
```

A function can call another function

```
int check_prime(int x){

        …….
        …….
        return flag;
}
```

Recursion

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



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

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

Simplest possible example

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

When does it stop ?

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;
}
```

Example: Sum of Natural Numbers Using Recursion

Define sum using recursion

RSum (N) = RSum(N-1) + N;
= RSUm(N-2) + N-1 + N ..
…
= Rsum(1) + 2 + 3 + … N
= Rsum(0) + 1 ….
= 0 + 1+ 2 + 3….
Answer…

Define factorial using recursion

Rfact (N) = Rfact(N-1) * N;

```
int sum(int n) {
  if (n != 0)
    // sum() function calls itself
    return n + sum(n-1);
  else
    return n;
}
```

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

Explanation

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.

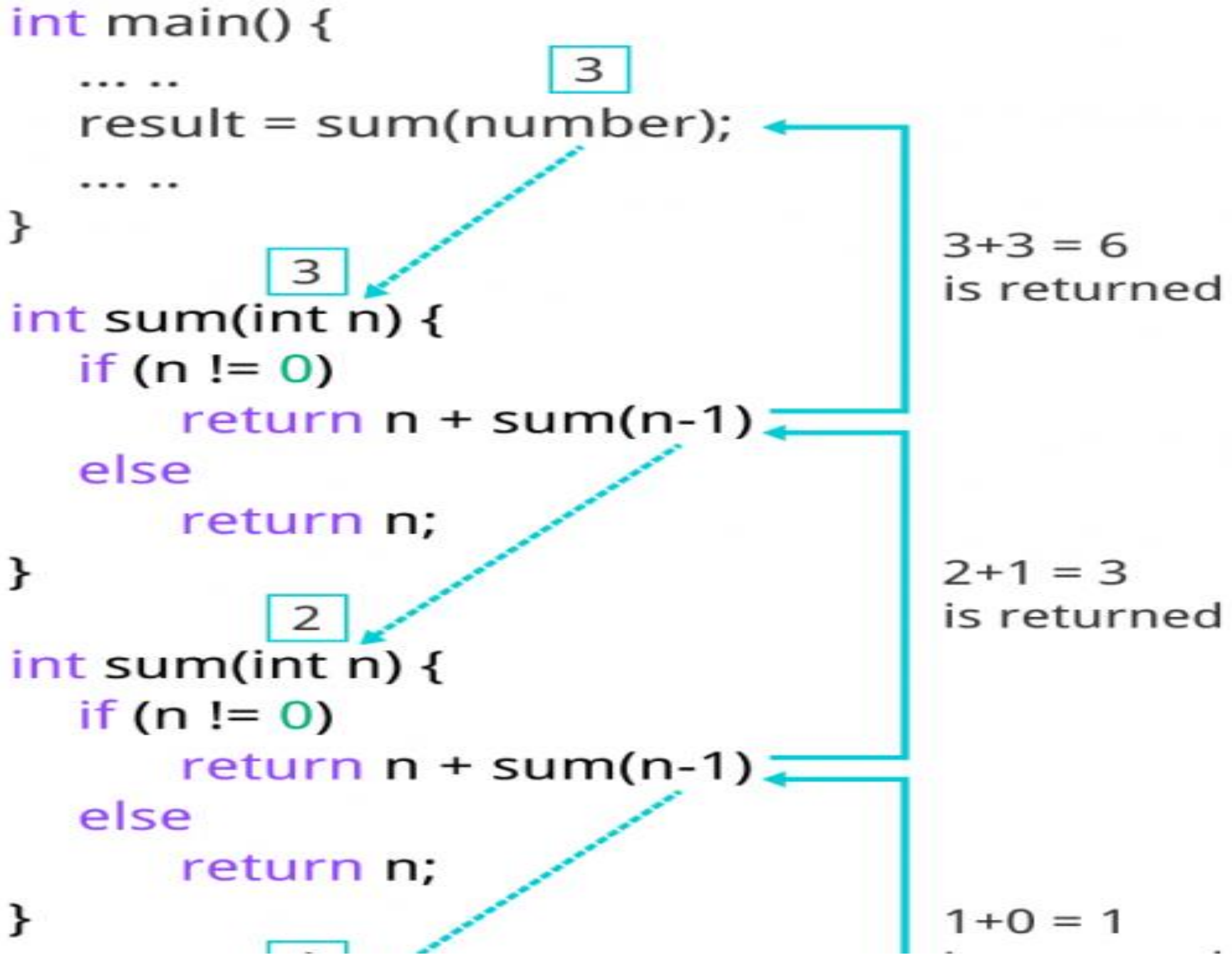
During the next function call, 2 is passed to the sum() function.

This process continues until n is equal to 0.

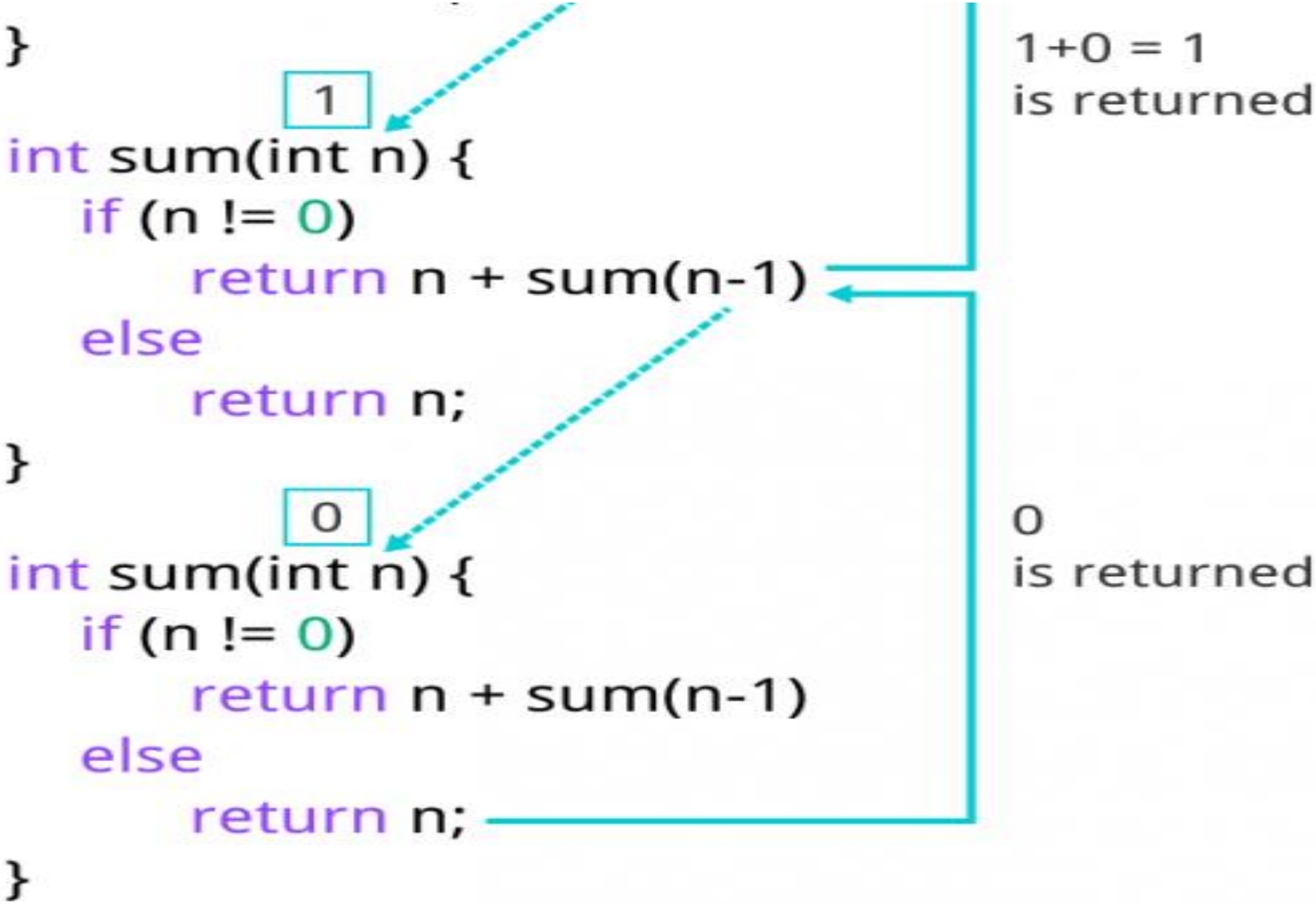
Explanation

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.



```
int main() {
   ... ..
   result = sum(number);
   ... ..
}

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

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



```
}

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

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

- Problem (N) = Problem (N/2) + Problem (N/2)

= 2* Problem (N/2)

Base Condition ??

Factorial(N) = ….

Factorial (N) = N* Factorial (N-1);

Base Condition ??

Reverse a sentence using recursion

```c
#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);
    }
}
```

```
Enter a sentence:
margorp emosewa
awesome program
```

Explanation

This program first prints Enter a sentence:

Then, immediately reverseSentence() is called.

Explanation

This function stores the first letter entered by the user in c.

If the variable is any character other than \n (enter character), reverseSentence() is called again.

Explanation

This process goes on until the user enters \n.

When the user finally enters \n, the last reverseSentence() function prints the last character. It is because of printf("%c", c);. Then it returns to the second last reverseSentence().

Explanation

This process goes on and the final output will be the reversed sentence.

C program to calculate the power using recursion

C Program to Find Factorial of a Number Using Recursion

C Program to Find G.C.D Using Recursion

Factorial (n)

= n*(n-1)*(n-2)…. 3.2.1.0

= n * Factorial (n-1)

Base conditions

Any one –

Factorial (1) = 1

Factorial (0) = 1

Fib (n) =

Fib (n-1) +
Fib (n-2)

Base conditions

Fib (0) = 0

Fib (1) = 1