NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Programming Fundamentals

LAB # 03
Functions and Recursion

Functions:

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Types of functions in C programming

Depending on whether a function is defined by the user or already included in C compilers, there are two types of functions in C programming

There are two types of functions in C programming:

- Standard library functions
- User defined functions

Standard library functions

The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc.

These functions are defined in the header file. When you include the header file, these functions are available for use. 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 `"stdio.h"` header file.

There are other numerous library functions defined under "stdio.h", such as scanf(), fprintf(), getchar() etc. Once you include "stdio.h" in your program, all these functions are available for use.

User-defined functions

As mentioned earlier, C allows programmers to define functions. Such functions created by the user are called user-defined functions.

Depending upon the complexity and requirement of the program, you can create as many user-defined functions as you want.

Benefits of Using Functions

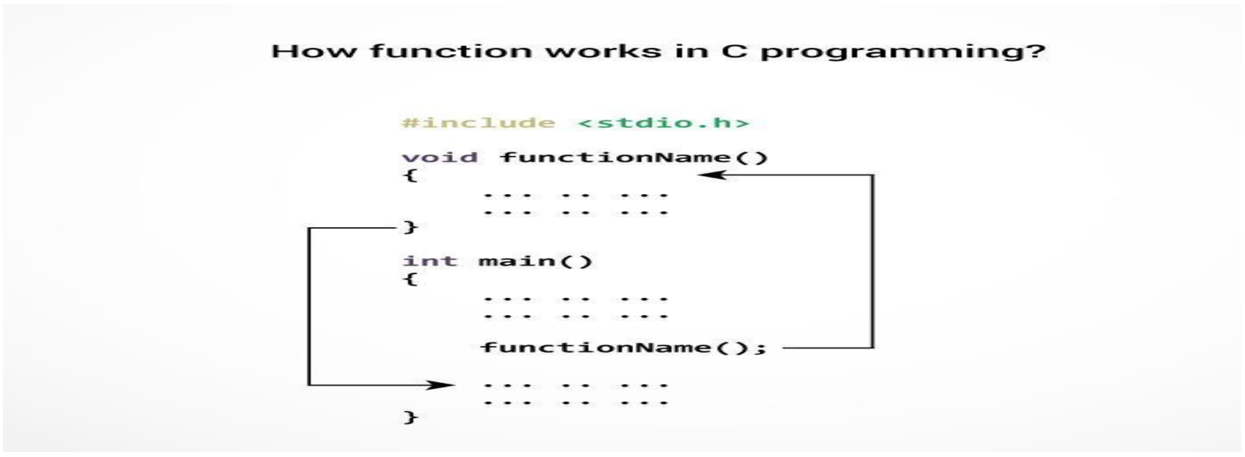
1. It provides modularity to your program's structure.

2. It makes your code reusable. You just have to call the function by its name to use it, wherever required.

3. In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.

4. It makes the program more readable and easy to understand.

How user-defined function works?



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

Example: User-defined function

Here is a example to add two integers. To perform this task, a user-defined function addNumbers() is defined.

```
#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 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, function call is made using `addNumbers(n1,n2);` statement inside the `main()`.

Function definition

Function definition contains the block of code to perform a specific task i.e. in this case, adding two numbers and returning it.

Syntax of function definition

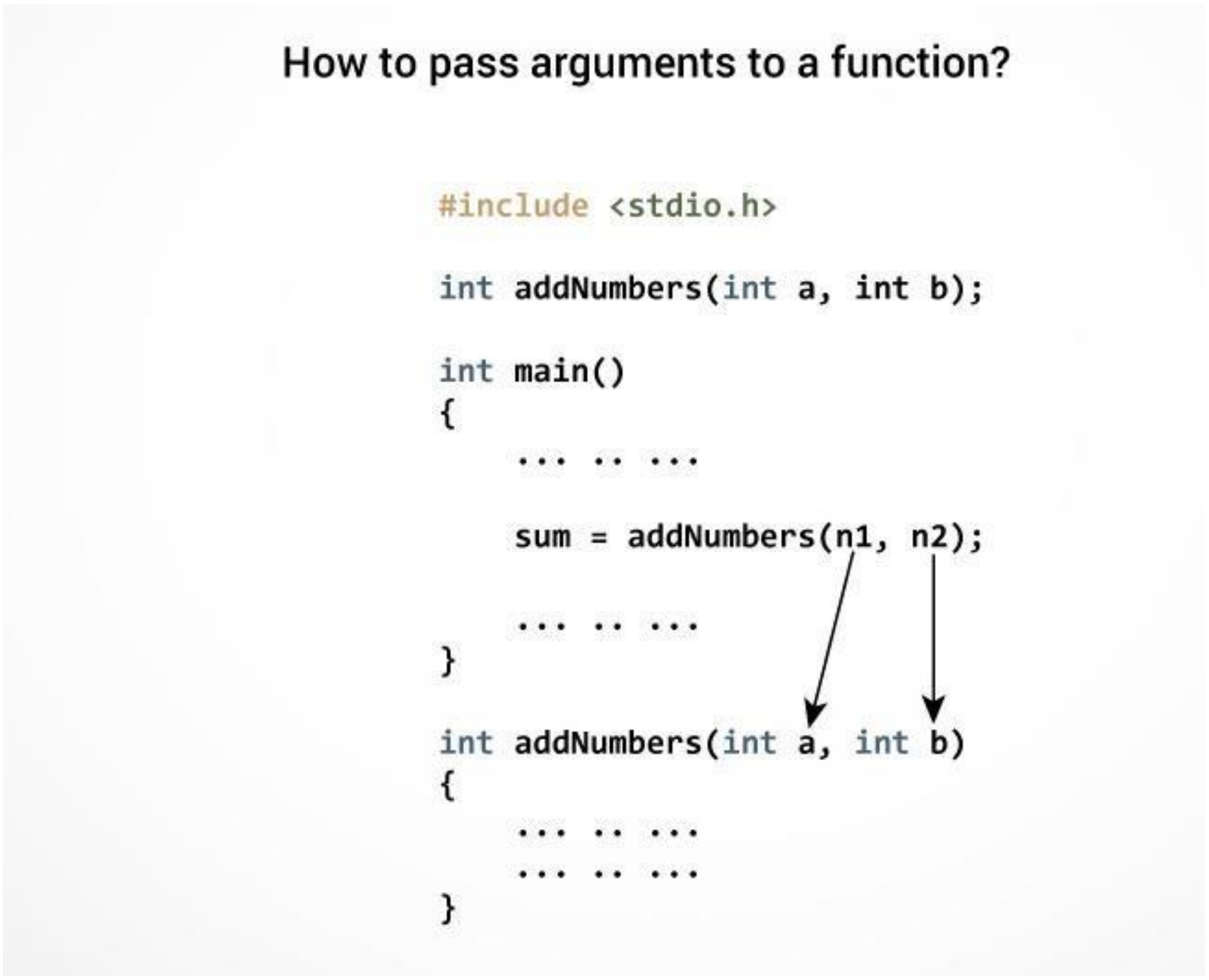
```
returnType functionName(type1 argument1, type2 argument2, ...)

{

    //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 function call.

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



```
#include <stdio.h>

int addNumbers(int a, int b);

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

    sum = addNumbers(n1, n2);

    ... .. ...
}

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

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

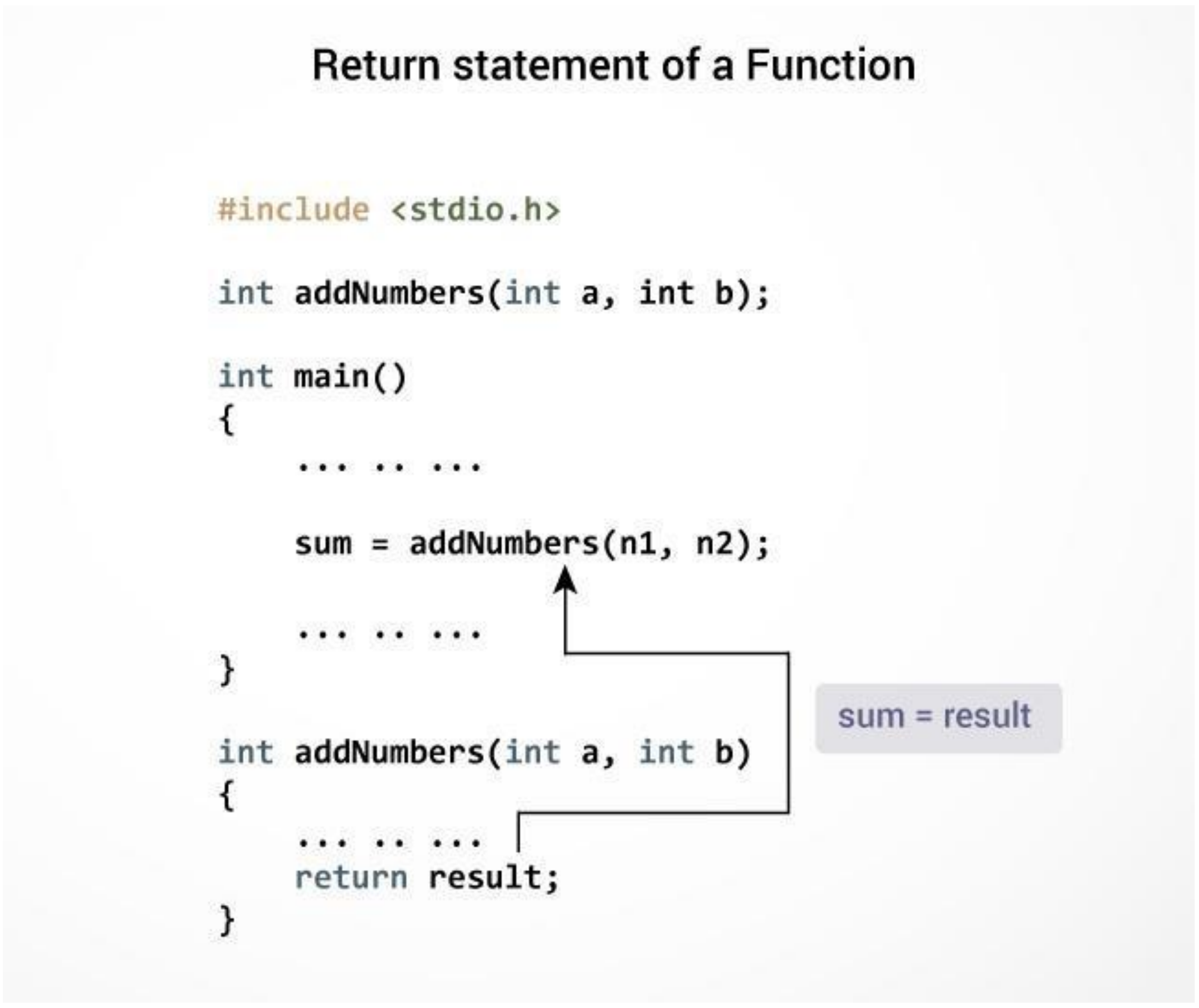
If `n1` is of char type, `a` also should be of char type. If `n2` is of float type, variable `b` also should be of float type.

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 return statement.

In the above example, the value of variable `result` is returned to the variable `sum` in the `main()` function.



```
#include <stdio.h>

int addNumbers(int a, int b);

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

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    return 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 function prototype and function definition must match.

Recursion:

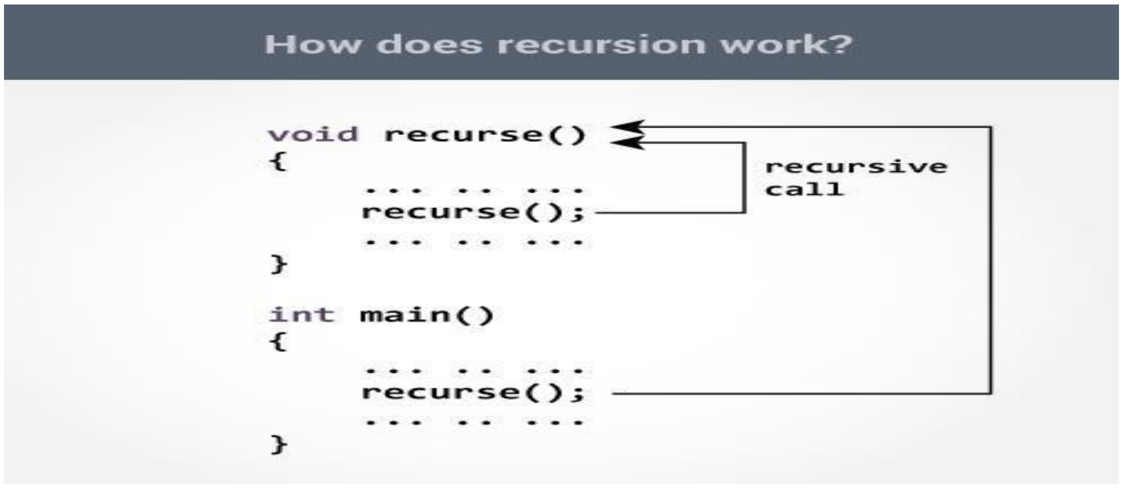
Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion() {

   recursion(); /* function calls itself */

}


int main() {

   recursion();

}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

How recursion works?



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

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.

Factorial(5)=5*factorial(4)
Factorial(4)=4*factorial(3)
Factorial(3)=3*factorial(2)

Factorial (2) = 2*factorial (1)
Factorial (1) = 1
n!=1 (for n=0)
n!=n*(n-1)! (For n>0)

Example: Sum of Natural Numbers Using Recursion:

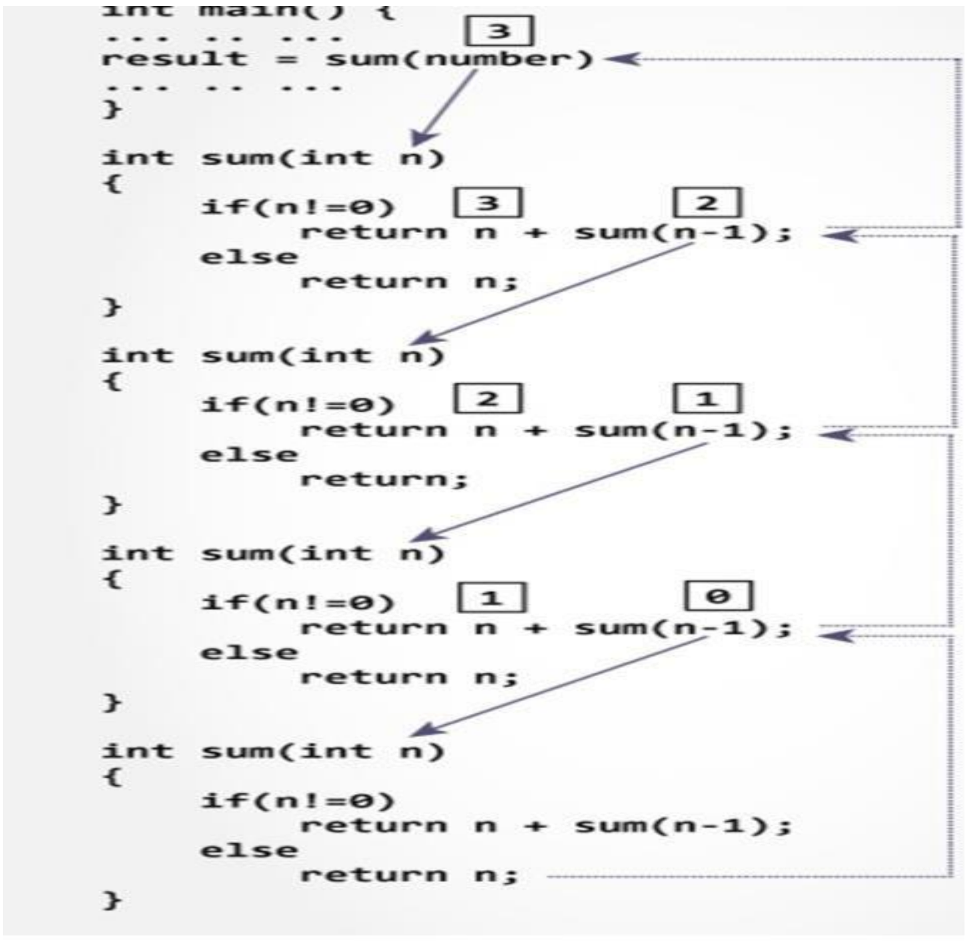
```c
#include <stdio.h>
int sum(int n);

int main()
{
    int number, result;

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

    result = sum(number);

    printf("sum=%d", result);
}

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

output:

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

Initially, the sum() is called from the main() function with number passed as an argument.Suppose, the value of num is 3 initially. During next function call, 2 is passed to the sum()function. This process continues until num is equal to 0.When num is equal to 0, the if condition fails and the else part is executed returning the sum of integers to the main() function.

How recursion works step by step:



```
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;
}

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;
}
```

Advantages and Disadvantages of Recursion

Recursion makes program elegant and cleaner. All algorithms can be defined recursively which makes it easier to visualize and prove.

If the speed of the program is vital then, you should avoid using recursion. Recursions use more memory and are generally slow. Instead, you can use loop.

Example:

Number Factorial:

The following example calculates the factorial of a given number using a recursive function

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

```
unsigned long long int factorial(unsigned int i) {


   if(i <= 1) {

      return 1;

   }

   return i * factorial(i - 1);

}


int  main() {

   int i = 12;

   printf("Factorial of %d is %d\n", i, factorial(i));

   return 0;

}
```

When the above code is compiled and executed, it produces the following result –

```
Factorial of 12 is 479001600
```

Fibonacci Series:

The following example generates the Fibonacci series for a given number using a recursive function –

```
#include <stdio.h>


int fibonacci(int i) {


   if(i == 0) {

      return 0;

   }


   if(i == 1) {

      return 1;

   }

   return fibonacci(i-1) + fibonacci(i-2);

}


int  main() {


   int i;


   for (i = 0; i < 10; i++) {

      printf("%d\t\n", fibonacci(i));

   }


   return 0;

}
```

When the above code is compiled and executed, it produces the following result −

```
0
1
1
2
3
5
8
13
21
34
```