C Functions

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.

Visit standard library functions in C programming to learn more.

User-defined function

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

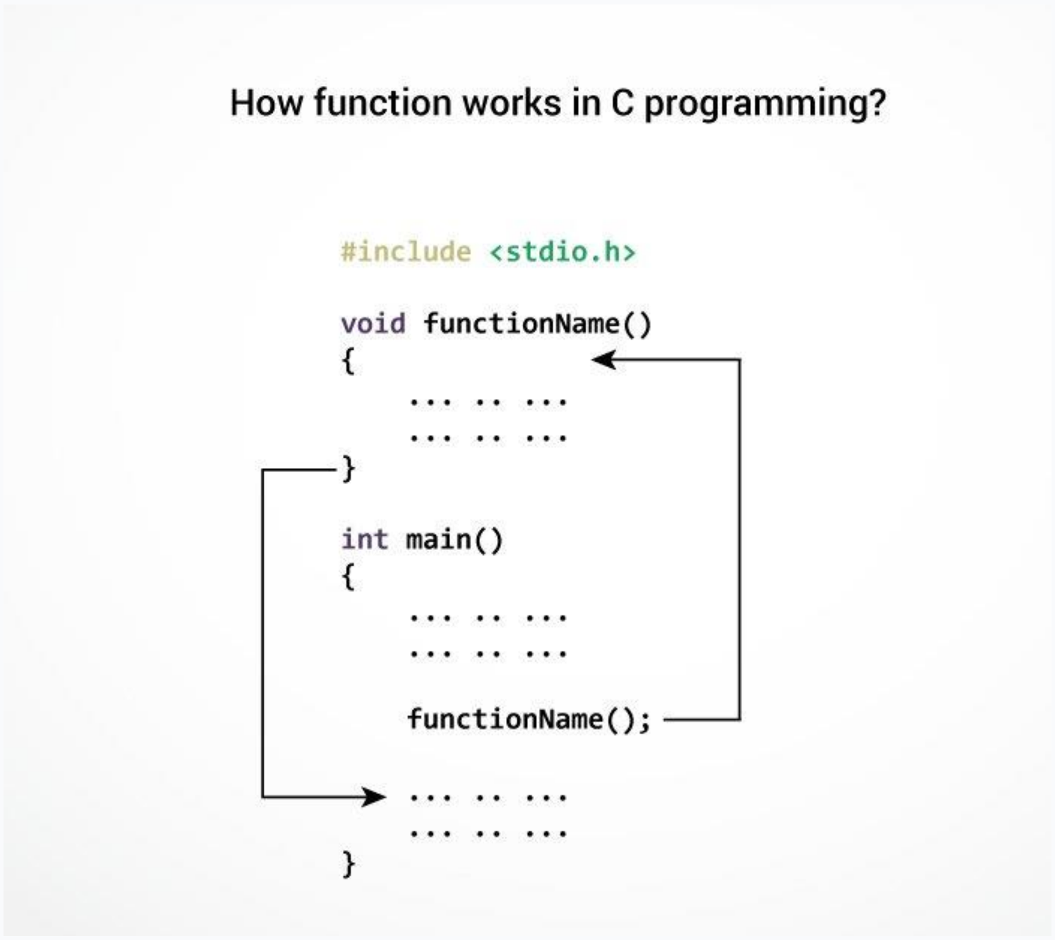
How user-defined function works?

```
#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.



```
#include <stdio.h>

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

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

    functionName();

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

Note, function names are identifiers and should be unique.

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.

C User-defined functions

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

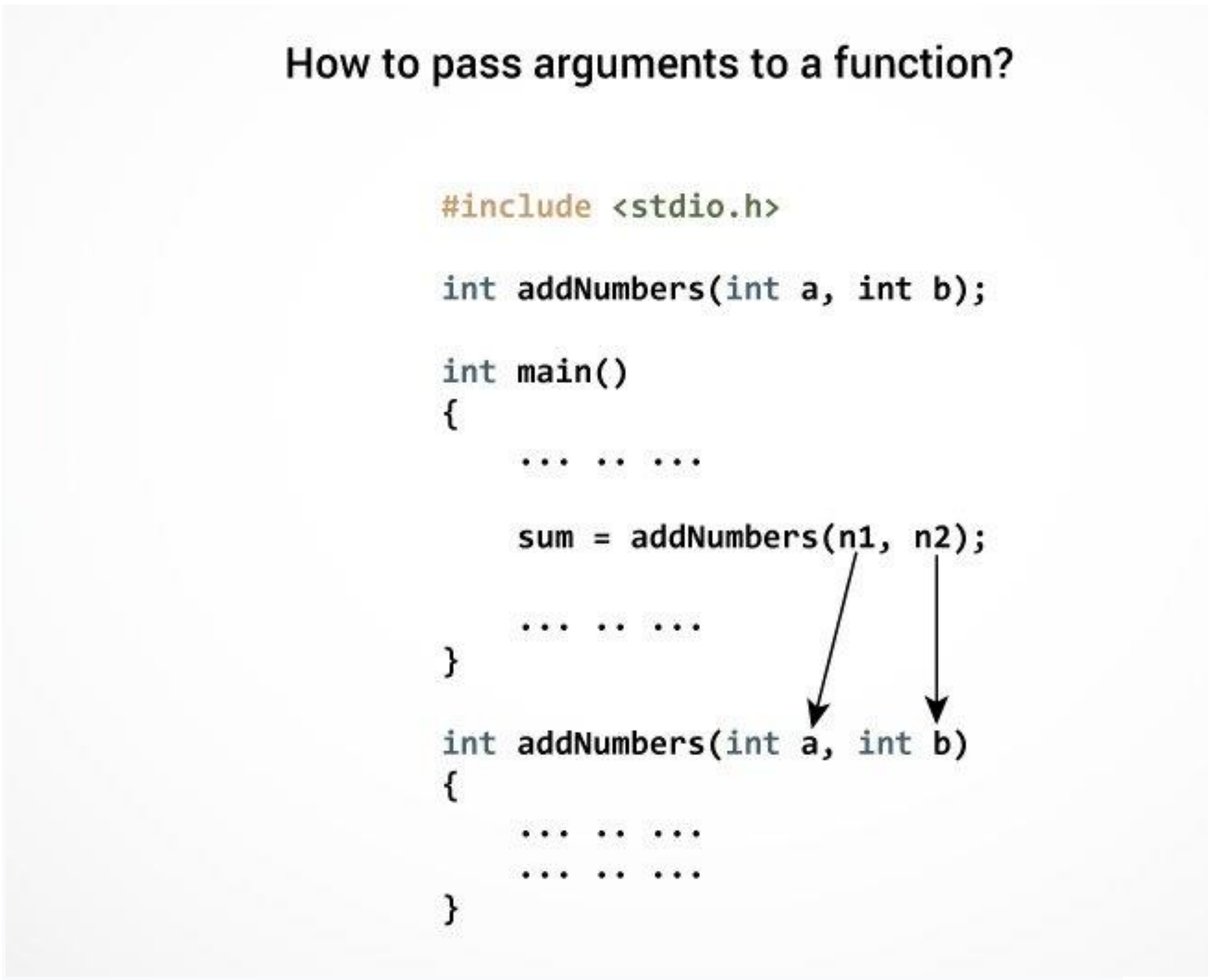
```
returnType functionName(type1 argument1, type2 argument2, ...)
{
    //body of the function
}
```

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 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.



```
#include <stdio.h>

int addNumbers(int a, int b);

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

    sum = addNumbers(n1, n2);

    ... .. ...
}

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

Passing Argument to Function

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.

Different ways of passing Function Arguments

.

While calling a function, there are two ways in which arguments can be passed to a function –

Sr.No.	Call Type & Description
1	Call by value This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	Call by reference This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

call by value

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming uses *call by value* to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

```
/* function definition to swap the values */
void swap(int x, int y) {

   int temp;

   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */

   return;
}
```

Now, let us call the function **swap()** by passing actual values as in the following example –

```
#include <stdio.h>

/* function declaration */
void swap(int x, int y);

int main () {
   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values */
   swap(a, b);

   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );

   return 0;
}
void swap(int x, int y) {

   int temp;

   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */

   return;
}
```

Let us put the above code in a single C file, compile and execute it, it will produce the following result −

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 100
After swap, value of b : 200
```

It shows that there are no changes in the values, though they had been changed inside the function.

call by reference

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

```
/* function definition to swap the values */
void swap(int *x, int *y) {

   int temp;
   temp = *x;    /* save the value at address x */
   *x = *y;      /* put y into x */
   *y = temp;    /* put temp into y */

   return;
}
```

Let us now call the function **swap()** by passing values by reference as in the following example −

Live Demo

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values */
   swap(&a, &b);

   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );

   return 0;
}
void swap(int *x, int *y) {

   int temp;

   temp = *x; /* save the value of x */
   *x = *y;    /* put y into x */
   *y = temp; /* put temp into y */

   return;
}
```

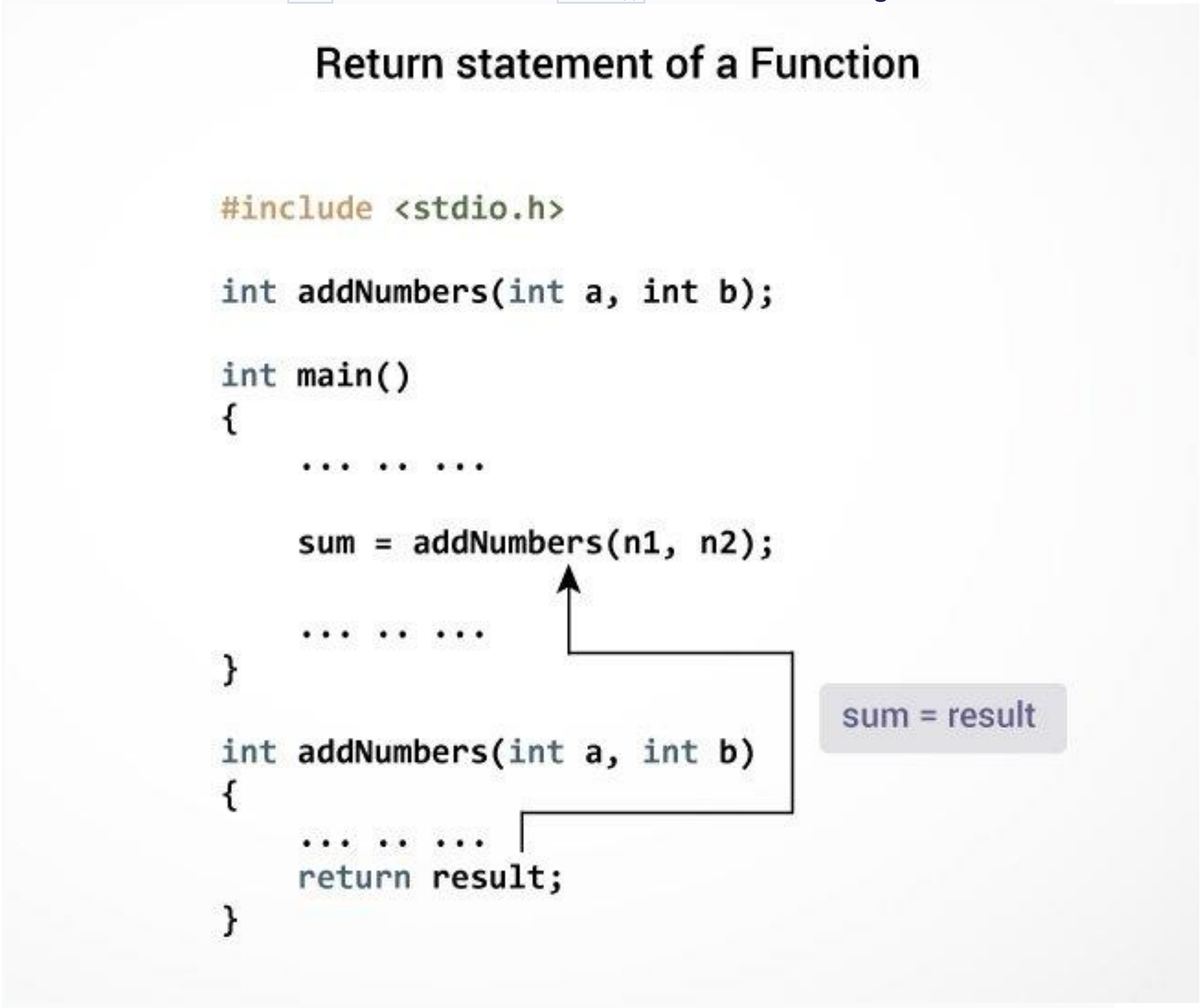
Let us put the above code in a single C file, compile and execute it, to produce the following result −

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

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.



```
#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 of Function

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.

C User-defined Function Types

Types of User-defined Functions

1. Function with no arguments and no return value
2. Function with no arguments and a return value
3. Function with arguments and no return value
4. Function with arguments and no return value

Below, we will discuss about all these types, along with examples. Consider a situation in which you have to check greater number. This problem is solved below by making user-defined function in 4 different ways as mentioned above.

Function with no arguments and no return value

When we make any function with no arguments and no return value, it neither receives any data from the calling function nor returns a value. In other words, we can say the calling function does not get any value from the called function.

Such functions neither take any `argument` nor `return` any value.

Such functions can either be used to display information because they are completely dependent on user inputs.
In this type of function each function is independent.

```
#include <stdio.h>
int main(){
  greatNum();    // argument is not passed
  return 0;
}
  // return type is void meaning doesn't return any value
void greatNum(){
  int i, j;
  printf("Enter two integer: ");
  scanf("%d %d",&i ,&j);
  if(i > j){
    printf("The greater number is : %d", i);
  }
  else {
    printf("The greater number is : %d", j);
  }
}
```

Output :

```
Enter two integer : 12 14
The greater number is : 14
```

Here, The `greatNum()` function takes two integer inputs from the user, and checks which one is greater .

The empty parenthesis in `greatNum();` statement inside the `main()` function indicates that no argument is passed to the function.

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

Function with no arguments and a return value

This type of functions, arguments are passed through the calling function to called function but the called function returns value.

Such type of function are independent.

```
#include <stdio.h>
int main(){
  int result;
  result = greatNum();  //Function called and argument is not passed
  printf("The greater number is %d", result);
  return 0;
}
  // return type is int meaning it return some value
int greatNum(){
  int i, j, greater_number;
  printf("Enter two integer: ");
  scanf("%d %d",&i ,&j);
  if(i > j){
  greater_number = i;
  }
  else {
  greater_number = j;
  }
  return greater_number;
}
```

Output :

```
Enter two integer: 12 11
The greater number is 12
```

Here, The empty parenthesis in the `result = greatNum();` statement indicates that no argument is passed to the function. And, the value returned from the function is assigned to `result`.

The `greatNum()` function takes two input from the user and returns it, Then check which one is greater.

The return type of the `greatNum` is `int`. Hence, it returns a value `greater_number`.

Function with arguments and no return value

This type of function, arguments are passed through the calling function to called function but does not return value.

Such type of function practically dependent on each other.

```
#include <stdio.h>
int main(){
  int i, j;
  printf("Enter two integers :");
  scanf("%d %d",&i ,&j);
  greatNum(i, j);  //Function called and argument is passed
  return 0;
}
  // return type is void meaning it return no value
void greatNum(int x,int y){
  if(x > y){
    printf("The greater number is %d", x);
  }
  else {
    printf("The greater number is %d", y);
  }
}
```

Output :

```
Enter two integers : 45 54
The greater number is 54
```

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

Here, the `greatNum()` function checks which one is greater.

The return type of the `greatNum()` is `void`, Hence, no value is returned from the function.

Function with arguments and a return value

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

Both functions are dependent on each other.

```
#include <stdio.h>
int greatNum(int i, int j);
int main(){
  int i, j, result;
  printf("Enter two integers :");
  scanf("%d %d",&i ,&j);
  result = greatNum(i, j);
  printf("The greater number is %d", result);
//Function called and argument is passed
  return 0;
}
  // return type is void meaning it return no value
void greatNum(int x, int y){
  if(x > y){
    return x;
  }
  else {
    return y;
  }
}
```

Output :

```
Enter two integer: 57 61
The greater number is 61
```

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

Here, the `greatNum()` function checks which one is greater.

The return type of the `greatNum()` is `int`. Hence, it returns a value.

****