

Lab 08

Objectives

After performing this lab, students will be able to

- Understand functions and use them in programs

Functions

A **function** is a block of code that performs a specific task. A function is written/created once and can be called again and again as many times as required whenever/wherever needed.

It is a good practice that we should create such programs that use functions to solve the problem. Imagine we need to create a program that creates a circle and then gives some color to it. We can create two functions to solve this problem:

- a function to draw the circle
- a function to color the circle

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

There are two types of function:

1. **User-defined Function:** Created by users/programmers
2. **Standard Library Functions:** Predefined in C++

User-defined Functions

C++ allows the programmers to define their own function. A user-defined function groups code to perform a specific task and that group of code is given a name (identifier). After that, when we invoke/call that function, then all the code inside it is executed.

The syntax to define a function is:

```
returnType functionName (parameter1, parameter2,...) {  
    // function body  
}
```

Here is an example

```
void greet() {  
    cout << "Hello World";  
}
```

Here

- the name of the function is **greet ()**
- the return type of the function is **void**
- the empty parentheses mean it does not have any parameters
- the function body is written inside curly brackets {}

Defining a function

A function definition will be different from the example discussed above depending on the need of the problem being solved. For example, in previous case, we just wanted to print a fixed/hardcoded string “Hello World” to the screen. But a function used to add two numbers and then return that value will be different. Similarly, a function used to check whether a number is even or not will also be different.

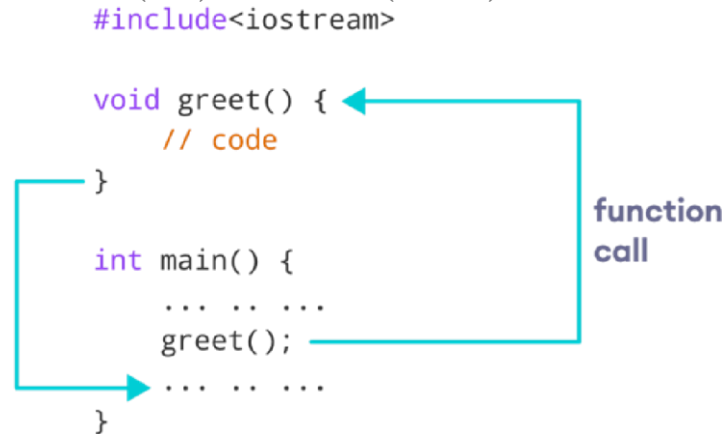
Note: You should define a function before calling it (like above the main () function).

Calling/Invoking a Function

In the example program given above, we have just defined a function named greet (). Until now, its body will not be executed. To actually use a function (like greet () in our case), we need to call/invoke it. Here's how we can call the above greet () function.

```
int main() {  
  
    // calling a function  
    greet();  
  
}
```

When a function is called, whatever is written inside its body will be executed. In this case, **Hello World** will be printed onto the screen.



When a function is called, then the flow of execution jumps to the body of the function being called. And when the body of the function being called is finished, then the remaining code of the caller function (in the example above, main function) resumes. **Example**

```
#include <iostream>

using namespace std;

void test() {
    cout << "This is the body of the function" << endl;
}

int main()
{
    cout << "This line will be executed before the function body" << endl;
    test();
    cout << "This line will be executed after the function body" << endl;

    return 0;
}
```

Output:

```
This line will be executed before the function body
This is the body of the function
This line will be executed after the function body
```

Function Parameters

A parameter is a value that is used when defining a function. A function can be defined with or without parameters depending on the need. For example, in previous case, we did not use any parameter because we just needed to print a fixed/hardcoded string. But for example, we want to create a function that adds two numbers, then it will require two parameters. Similarly, a function used to test if a value is even or not will require one parameter, and so on.

Example:

```
void printNum(int num) {  
    cout << num;  
}
```

Here, the **int** variable **num** is the function parameter. We pass a value to the function parameter while calling the function.

```
int main() {  
    int n = 7;  
  
    // calling the function  
    // n is passed to the function as argument  
    printNum(n);  
  
    return 0;  
}
```

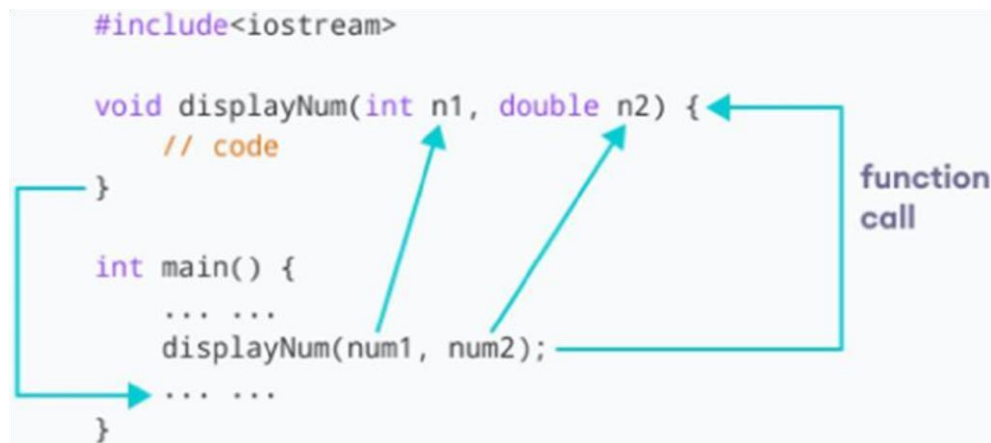
Output:

The number 7 will be printed

Example 2: A function with two parameters

```
void displayNum(int n1, float n2) {  
    cout << "The int number is " << n1;  
    cout << "The double number is " << n2;  
}  
  
int main() {  
  
    int num1 = 5;  
    double num2 = 5.5;  
  
    // calling the function  
    displayNum(num1, num2);  
  
    return 0;  
}
```

In the above program, we have used a function that has one **int** parameter and one **double** parameter. We then pass **num1** and **num2** as arguments. These values are stored by the function parameters **n1** and **n2**, respectively.



Note: The type of the arguments passed while calling the function **must** match with the corresponding parameters defined in the function declaration.

The value passed to a function parameter while calling it is also called as an **argument**.

The return statement

In the above programs, we have used **void** in the function definition. For example,

```
void displayNumber() {  
    // code  
}
```

This means the function is not returning any value.

It is also possible to return a value from a function. For this, we need to specify the **returnType** of the function during function declaration/definition. Then, the **return** statement can be used to return a value from a function.

For example,

```
int add (int a, int b) {  
    return (a + b);  
}
```

Here, we have the data type **int** instead of **void**. This means that the function returns an **int** value. The code **return (a + b);** returns the sum of the two parameters as the function value. The **return** statement denotes that the function has ended. Any code after **return** inside the function is not executed.

Example: Adding two numbers and returning their sum

```
int add(int a, int b) {  
    return (a + b);  
}  
  
int main() {  
  
    int sum;  
  
    // calling the function and storing  
    // the returned value in sum  
    sum = add(100, 78);  
  
    cout << "100 + 78 = " << sum << endl;  
  
    return 0;  
}
```

Output:

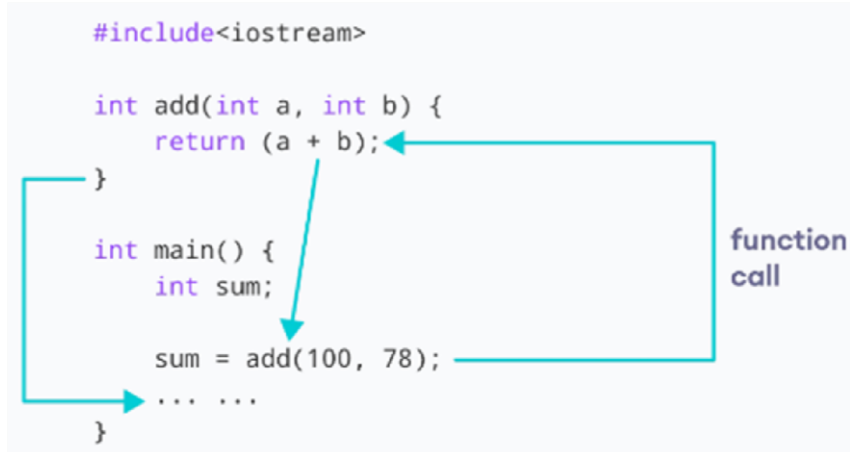
```
100 + 78 = 178
```

In the above program, the **add ()** function is used to find the sum of two numbers. We pass two **int** literals 100 and 78 while calling the function. We store the returned value of the function in the variable **sum**, and then we print it. Notice that **sum** is a variable of **int** type. This is because the return value of **add ()** is of **int** type.

```
#include<iostream>

int add(int a, int b) {
    return (a + b);
}

int main() {
    int sum;
    sum = add(100, 78);
    ... ..
}
```



The diagram illustrates the function call process. A blue arrow points from the `add(100, 78);` line in the `main()` function to the `return (a + b);` line in the `add()` function. Another blue arrow points from the `return` statement back to the `sum =` assignment in `main()`, indicating the return value is stored in `sum`. A bracket on the right side of the diagram, labeled "function call", spans the `add()` function definition and the `add(100, 78);` call.

Function Prototype

In C++, the code of function definition should be before the function call. However, if we want to define a function after the function call, we need to use the function prototype (also called **function declaration**). For example,

```
// function prototype
void add(int, int);

int main() {
    // calling the function before declaration.
    add(5, 3);
    return 0;
}

// function definition
void add(int a, int b) {
    cout << (a + b);
}
```

In the example shown above, the function prototype is:

```
void add(int, int);
```

This provides the compiler with information about the function name and its parameters. That's why we can use the code to call a function before the function has been defined.

The syntax of a function prototype is:

```
returnType functionName(dataType1, dataType2, ...);
```

Example:


```
// function prototype
int add(int, int);

int main() {
    int sum;

    // calling the function and storing
    // the returned value in sum
    sum = add(100, 78);

    cout << "100 + 78 = " << sum << endl;

    return 0;
}

// function definition
int add(int a, int b) {
    return (a + b);
}
```

Output:

```
100 + 78 = 178
```

Library Functions

Library functions are the built-in functions in C++ programming. Programmers can use library functions by invoking the functions directly; they do not need to write the functions by themselves. Some of the common library functions in C++ are **sqrt ()**, **abs ()**, **isdigit ()**, etc.

In order to use library functions, we usually need to include the header file in which these library functions are defined. For instance, in order to use mathematical functions such as **sqrt()** and **abs ()**, we need to include the header file **cmath**.

Example:

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    double number, squareRoot;

    number = 25.0;

    // sqrt() is a library function to calculate the square root
    squareRoot = sqrt(number);

    cout << "Square root of " << number << " = " << squareRoot;

    return 0;
}
```

Output:

```
Square root of 25 = 5
```

Function Overloading

In C++, two functions can have the same name if the number and/or type of arguments passed is different. These functions having the same name, but different arguments are known as **overloaded functions**. For example:

```
// same name different arguments
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

Here, all 4 functions are overloaded functions. Notice that the return types of all these 4 functions are not the same. Overloaded functions may or may not have different return types but they **must** have different arguments. For example,

```
// Error code
int test(int a) { }
double test(int b){ }
```

Here, both functions have the same name, the same type of argument, and the same number of arguments. Hence, the compiler will throw an error.

Example: Function overloading using different types of parameters

```
// function with float type parameter
float absolute(float var){
    if (var < 0.0)
        var = -var;
    return var;
}

// function with int type parameter
int absolute(int var) {
    if (var < 0)
        var = -var;
    return var;
}

int main() {

    // call function with int type parameter
    cout << "Absolute value of -5 = " << absolute(-5) << endl;

    // call function with float type parameter
    cout << "Absolute value of 5.5 = " << absolute(5.5f) << endl;
    return 0;
}
```

Output:

```
Absolute value of -5 = 5
Absolute value of 5.5 = 5.5
```

In this program, we overload the **absolute ()** function. Based on the type of parameter passed during the function call, the corresponding function is called, like shown below.

```
float absolute(float var) {  
    // code  
}  
  
int absolute(int var) {  
    // code  
}  
  
int main() {  
    absolute(-5);  
    absolute(5.5f);  
    ...  
}
```

The diagram illustrates function overloading. Two teal arrows originate from the `absolute` calls in the `main` function. One arrow points from `absolute(-5);` to the `int absolute(int var)` function definition. The other arrow points from `absolute(5.5f);` to the `float absolute(float var)` function definition, demonstrating how the compiler selects the appropriate function based on the argument type.

Example 2: Function overloading using different number of parameters

```
#include <iostream>
using namespace std;

// function with 2 parameters
void display(int var1, double var2) {
    cout << "Integer number: " << var1;
    cout << " and double number: " << var2 << endl;
}

// function with double type single parameter
void display(double var) {
    cout << "Double number: " << var << endl;
}

// function with int type single parameter
void display(int var) {
    cout << "Integer number: " << var << endl;
}

int main() {

    int a = 5;
    double b = 5.5;

    // call function with int type parameter
    display(a);

    // call function with double type parameter
    display(b);

    // call function with 2 parameters
    display(a, b);

    return 0;
}
```

Output:

```
Integer number: 5
Double number: 5.5
Integer number: 5 and double number: 5.5
```

Here, the **display ()** function is called three times with different arguments. Depending on the number and type of arguments passed, the corresponding **display ()** function is called.

```
void display(int var1, double var2) {  
    // code  
}  
  
void display(double var) {  
    // code  
}  
  
void display(int var) {  
    // code  
}  
  
int main() {  
    int a = 5;  
    double b = 5.5;  
  
    display(a);  
    display(b);  
    display(a, b);  
  
    ... ..  
}
```

The diagram illustrates function calls from the `main()` function to three overloaded `display()` functions. Red arrows originate from the function calls in `main()` and point to the corresponding function definitions. The arrow from `display(a)` points to `void display(int var)`. The arrow from `display(b)` points to `void display(double var)`. The arrow from `display(a, b)` points to `void display(int var1, double var2)`.

Note: In C++, many standard library functions are overloaded. For example, the **sqrt ()** function can take **double**, **float**, **int**, etc. as parameters. This is possible because the **sqrt ()** function is overloaded in C++.

Exercises

1. Tasks related to functions

- a. Just like the sample program discussed in the class that was performing the addition of two numbers, create four (04) separate functions, each of which gets two integer values as parameters and returns the subtraction, multiplication, division, and the modulus (remainder) of those two values, respectively.

Then call these functions in the `main ()` function and print the values returned by them.

Note: Each user-defined function should only return a value after performing the respective arithmetic operation. The remaining logic, which prints the results, should be put in the `main ()` function.

- b. Create a function named **square** that takes one integer number as parameter and returns the square of that number. Then, call that function in the main () by passing any value, and display the result.
- c. Create a function that takes two integer values as parameters and returns the greater value of those two. Then, call that function in the main () and display the result.

Note: We are assuming that the values of parameters will be unique (the caller will NOT give the same value for both parameters)

- d. Create a function that takes one integer value as a parameter and then return the word **Even** (if that number is an even number) or **Odd** (if the value is odd). After that, call that function in the main () and display the result.
- e. Create a function that computes the percentage of a student. The function should take two values (**obtained marks** and **total marks**) as parameters and return the percentage (that can be a value with a decimal point). Then, call that function in the main () by passing two values to it, and display the result.
- f. Create a function that takes the percentage (a value that might contain value with decimal point) as a parameter and returns the grade (A, B, C, or F). The rules for calculating the grade are:
 - i. If the percentage is between 80 and 100 (inclusive), the grade will be **A**
 - ii. If the percentage is between 70 and 79, the grade should be **B**
 - iii. If the percentage is between 60 and 69, the function should return the grade **C**
 - iv. If the percentage is less than 60, the grade will be **F**

Then, call that function in the main () by passing the percentage and show/display the result/grade returned by the function.

Note: The value of percentage passed to this function should be the one returned by the function created in the **previous** task (the **f** part above this task). In other words, first, call the function created in the **previous** task, save the result returned by that function in a variable, and then pass that variable as the argument to the function created in **this** task.

- g. Create a function that takes an integer value as a parameter and returns a **boolean** value (true or false) depending on whether the parameter value is even or not. If that value is **even**, the function should return **true**; otherwise, it should return **false**.

In the end, call that function in the main () function by passing a value taken as input from the user, and depending on the returned value, print whether the number is even or not. For example,

Sample Output 1

```
Enter the number to check if it is even or not: 23
The number is NOT even
```

Sample Output 2

```
Enter the number to check if it is even or not: 140
The number is even
```

Note: The user-defined function should only return a boolean value (**true** or **false**). The remaining logic, which prints the respective message whether a number is even or not, should be put in the main () function.

- h. Create a user-defined function that takes on parameter **range** and returns the sum of all positive integers up to that **range**. For example,

Sample Output 1

```
Enter the range up to which you want to find the sum: 4
The sum of all 4 positive integers is 10
```

Sample Output 2

```
Enter the range up to which you want to find the sum: 10
The sum of all 10 positive integers is 55
```

Note: Again, the user-defined function should only return the **sum**. The remaining logic, which prints the result, should be put in the main () function.

- i. Create a function named **printTable** that gets one integer value as a parameter and prints its table of that number (from 1 to 10). Then, in the main function, get a value as input from the user and call the function by passing that input value as an argument. As shown below:


```
Enter any number to print its table: 4
4x1 = 4
4x2 = 8
4x3 = 12
4x4 = 16
4x5 = 20
4x6 = 24
4x7 = 28
4x8 = 32
4x9 = 36
4x10 = 40
```

However, this printing operation should be performed inside the user-defined function.

Then, create another function with the same name **printTable** but it gets two integer values (**num** and **range**) as parameters and prints the table of **num** up to the **range** value (in the previous task, the code was printing a table up to 10; thus, the range value was always 10 but here it should be equal to the **range** parameter).

Then, in the main function, get two values as input from the user and call the function by passing those input values as arguments. For example:

Sample Output 1:

```
Enter the number to print its table: 12
Enter the final range value: 10
12x1 = 12
12x2 = 24
12x3 = 36
12x4 = 48
12x5 = 60
12x6 = 72
12x7 = 84
12x8 = 96
12x9 = 108
12x10 = 120
```

Sample Output 2:

```
Enter the number to print its table: 3
Enter the final range value: 6
3x1 = 3
3x2 = 6
3x3 = 9
3x4 = 12
3x5 = 15
3x6 = 18
```

Hint/Note: Here, you will use the concept of **function overloading**.