Lecture 3: Functions in C++

C++ Function

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

You can pass data, known as parameters, into a function.

There are two types of function:

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

C++ User-defined Function

A user-defined function groups code to perform a specific task and that group of code is given a name (identifier).

When the function is invoked from any part of the program, it all executes the codes defined in the body of the function.

Function Declaration and Definition

Syntax

```
void myFunction() {
  // code to be executed
}
```

Example Explained

- `myFunction()` is the name of the function.
- `void` means that the function does not have a return value.

- inside the function (the body), add code that defines what the function should do.

Note: If a user-defined function, such as myFunction() is declared after the main() function, **an error will occur**. It is because C++ works from **top to bottom;** which means that if the function is not declared above main(), the program is unaware of it:

Example

```
// Function declaration
void myFunction() {
  cout << "I just got executed!";
}

// The main method
int main() {
  myFunction();  // call the function
  return 0;
}
```

Example

```
// Function declaration
void myFunction();

// The main method
int main() {
  myFunction();  // call the function
  return 0;
}

// Function definition
void myFunction() {
  cout << "I just got executed!";
}
```

Call a Function

To call a function, write the function's name followed by two parentheses () and a semicolon ;

In the following example, `myFunction()` is used to print a text (the action), when it is called:

Example Inside `main`, call `myFunction()`:

```
// Create a function
void myFunction() {
  cout << "I just got executed!";
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
```

Parameters

Information can be passed to functions as a parameter. Parameters act as variables inside the function.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma:

Syntax

```
void functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

The following example has a function that takes a `string` called **fname** as parameter. When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
void myFunction(string fname) {
  cout << fname << " Refsnes\n";
}

int main() {
  myFunction("Liam");
  myFunction("Jenny");
```

```
  myFunction("Anja");
  return 0;
}

// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

Default Parameter Value

You can also use a default parameter value, by using the equals sign (=).

If we call the function without an argument, it uses the default value ("Norway"):

Example

```
void myFunction(string country = "Norway") {
  cout << country << "\n";
}

int main() {
  myFunction("Sweden");
  myFunction("India");
  myFunction();
  myFunction("USA");
  return 0;
}

// Sweden
// India
// Norway
// USA
```

Multiple Parameters

Inside the function, you can add as many parameters as you want:

Example

```
void myFunction(string fname, int age) {
  cout << fname << " Refsnes. " << age << " years old. \n";
}

int main() {
```

```
  myFunction("Liam", 3);
  myFunction("Jenny", 14);
  myFunction("Anja", 30);
  return 0;
}

// Liam Refsnes. 3 years old.
// Jenny Refsnes. 14 years old.
// Anja Refsnes. 30 years old.
```

Pass by Reference

You can also pass a **reference** to the function. This can be useful when you need to change the value of the arguments:

Example

```
void swapNums(int &x, int &y) {
  int z = x;
  x = y;
  y = z;
}

int main() {
  int firstNum = 5;
  int secondNum = 3;

  cout << "Before swap: " << "\n";
  cout << firstNum << secondNum << "\n";

  // Call the function, which will change the values of firstNum and
secondNum
  swapNums(firstNum, secondNum);

  cout << "After swap: " << "\n";
  cout << firstNum << secondNum << "\n";

  return 0;
}
```

Return Values

The `void` keyword, used in the previous examples, indicates that the function should not return a value. If you want the function to return a

value, you can use a data type (such as `int`, `string`, etc.) instead of `void`, and use the `return` keyword inside the function:

Example

```
int myFunction(int x) {
  return 5 + x;
}

int main() {
  cout << myFunction(3);
  return 0;
}

// Outputs 8
```

This example returns the sum of a function with **two parameters**:

Example

```
int myFunction(int x, int y) {
  return x + y;
}

int main() {
  cout << myFunction(5, 3);
  return 0;
}

// Outputs 8
```

You can also store the result in a variable:

Example

```
int myFunction(int x, int y) {
  return x + y;
}

int main() {
  int z = myFunction(5, 3);
  cout << z;
  return 0;
}
// Outputs 8
```

Return by Reference

Example:

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

// Global variable
int num;


// Function declaration
int& test();

int main()
{
    test() = 5;

    cout << num;
    return 0;
}

int& test()
{
    return num;
}
```

Output

```
5
```

In program above, the return type of function **test()** is int&. Hence, this function returns a reference of the variable **num**.

The return statement is return num;. Unlike return by value, this statement doesn't return value of num, instead it returns the variable itself (address).

So, when the **variable** is returned, it can be assigned a value as done in test() = 5;

This stores 5 to the variable num, which is displayed onto the screen.

Difference between call by value and call by reference in C++

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

Function Overloading

With **function overloading**, multiple functions can have the same name with different parameters:

Example

```
int myFunction(int x)
float myFunction(float x)
double myFunction(double x, double y)
```

Consider the following example, which have two functions that add numbers of different type:

Example

```
int plusFuncInt(int x, int y) {
  return x + y;
}

double plusFuncDouble(double x, double y) {
  return x + y;
}

int main() {
  int myNum1 = plusFuncInt(8, 5);
  double myNum2 = plusFuncDouble(4.3, 6.26);
  cout << "Int: " << myNum1 << "\n";
  cout << "Double: " << myNum2;
```

```
  return 0;
}
```

Instead of defining two functions that should do the same thing, it is better to overload one.

In the example below, we overload the `plusFunc` function to work for both `int` and `double`:

Example

```
int plusFunc(int x, int y) {
  return x + y;
}

double plusFunc(double x, double y) {
  return x + y;
}

int main() {
  int myNum1 = plusFunc(8, 5);
  double myNum2 = plusFunc(4.3, 6.26);
  cout << "Int: " << myNum1 << "\n";
  cout << "Double: " << myNum2;
  return 0;
}
```

Functions Local and global variables

A scope is a region of the program and broadly speaking there are three places, where variables can be declared −

- Inside a function or a block which is called **local variables,**
- In the definition of function parameters which is called formal parameters.
- Outside of all functions which are called **global variables**.

Local variables can be used only by statements that are inside that function or block of code. Local variables are not known to functions on their own.

Example

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


int main () {
   // Local variable declaration:
   int a, b;
```

```
   int c;
   // actual initialization
   a = 10;
   b = 20;
   c = a + b;

   cout << c;
   return 0;
}
```

Output

This will give the output −

```
30
```

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the lifetime of your program. A global variable can be accessed by any function.

Example

```
#include <iostream>
using namespace std;
// Global variable declaration:
int g;
int main () {
   // Local variable declaration:
   int a, b;

   // actual initialization
   a = 10;
   b = 20;
   g = a + b;

   cout << g;
   return 0;
}
```

Output

This will give the output −

```
30
```

A program can have the same name for local and global variables but the value of a local variable inside a function will take preference. For accessing the global variable with same rame, you'll have to use the scope resolution operator.

Example

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

// Global variable declaration:
int g = 20;
int main () {
   // Local variable declaration:
   int g = 10;

   cout << g;   // Local
   cout << ::g; // Global
   return 0;
}
```

Output

This will give the output −

```
10
20
```

C++ Standard Function

Functions of <cmath> header file.

The C++ <cmath> header file declares a set of functions to perform

mathematical operations.

C++ ceil()

Return ceiling value of number

The ceil() function in C++ returns the smallest possible integer value which is **greater** than or **equal** to the given argument.

ceil() Parameters

The ceil() function takes a single argument whose ceiling value is computed.

Example 1: ceil() function for double, float and long double types

```
#include <iostream>
#include <cmath>

using namespace std;
```

```
int main()
{
    double x = 10.25, result;
    result = ceil(x);
    cout << "Ceil of " << x << " = " << result << endl;
    return 0;
}
```

Output

```
Ceil of 10.25 = 11
```

C++ floor()

The floor() function in C++ returns the largest possible integer value which is **less** than or **equal** to the given argument.

floor() Parameters

The floor() function takes a single argument whose floor value is computed.

floor() Return value

The floor() function returns the largest possible integer value which is **less than** or **equal** to the given argument.

Example 1: How floor() works in C++?

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    double x = 10.25, result;
    result = floor(x);
    cout << "Floor of " << x << " = " << result << endl;
```

```
    x = -34.251;
    result = floor(x);
    cout << "Floor of " << x << " = " << result << endl;

    x = 0.71;
    result = floor(x);
    cout << "Floor of " << x << " = " << result << endl;

    return 0;
}
```

Output:

```
Floor of 10.25 = 10
Floor of -34.251 = -35
Floor of 0.71 = 0
```

Example 2: floor() function for integral types

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    int x = 15;
    double result;
    result = floor(x);
    cout << "Floor of " << x << " = " << result << endl;

    return 0;
}
```

Output:

```
Floor of 15 = 15
```

C++ round()

Returns integral value nearest to argument

The round() function in C++ returns the integral value that is nearest to the argument, with halfway cases rounded away from zero.

round() Parameters

The round() function takes a single argument value to round.

Example 1: How round() works in C++?

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    double x = 11.16, result;
    result = round(x);
    cout << "round(" << x << ") = " << result << endl;

    x = 13.87;
    result = round(x);
    cout << "round(" << x << ") = " << result << endl;

    x = 50.5;
    result = round(x);
    cout << "round(" << x << ") = " << result << endl;

    x = -11.16;
    result = round(x);
    cout << "round(" << x << ") = " << result << endl;

    x = -13.87;
    result = round(x);
    cout << "round(" << x << ") = " << result << endl;

    x = -50.5;
    result = round(x);
    cout << "round(" << x
```

```
round(11.16) = 11
round(13.87) = 14
round(50.5) = 51
round(-11.16) = -11
round(-13.87) = -14
round(-50.5) = -51
```

C++ trunc()

Truncates the decimal part of a number

Truncates a double value after the decimal point and gives the integer part as the result. The return value and the arguments are of the type double.

trunc() Parameters

The trunc() function takes a single argument whose trunc value is to be computed.

Example 1: How trunc() works in C++?

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    double x = 10.25, result;
    result = trunc(x);
    cout << "trunc(" << x << ") = " << result << endl;

    x = -34.251;
    result = trunc(x);
    cout << "trunc(" << x << ") = " << result << endl;

    return 0;
}
```

```
trunc(10.25) = 10
```

```
trunc(-34.251) = -34
```

Output:

value	round	floor	ceil	trunc
2.3	2.0	2.0	3.0	2.0
3.8	4.0	3.0	4.0	3.0
5.5	6.0	5.0	6.0	5.0
-2.3	-2.0	-3.0	-2.0	-2.0
-3.8	-4.0	-4.0	-3.0	-3.0
-5.5	-6.0	-6.0	-5.0	-5.0

C++ exp()

returns exponential (e) raised to a number

The exp() function in C++ returns the exponential.

exp() Parameters

The exp() function takes a single mandatory argument and can be any value i.e. negative, positive or zero.

exp() Return value

The exp() function returns the value in the range of [0, ∞].

Example 1: How exp() function works in C++?

```
#include <iostream>
#include <cmath>

using namespace std;
```

```
int main()
{
	double x = 2.19, result;

	result = exp(x);
	cout << "exp(x) = " << result << endl;

	return 0;
}
```

```
exp(x) = 8.93521
```

C++ sqrt()

Computes Square Root of a Number

The sqrt() function in C++ returns the square root of a number.

sqrt() Parameters

The sqrt() function takes a single non-negative argument.

If negative argument is passed to sqrt() function, domain error occurs.

Example 1: How sqrt() works in C++?

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

int main()
{
	double x = 10.25, result;
	result = sqrt(x);
	cout << "Square root of " << x << " is " << result << endl;

	return 0;
}
```

```
Square root of 10.25 is 3.20156
```

C++ fmax()

returns largest among two arguments passed

The fmax() function in C++ takes two arguments and returns the largest among them. If one of the argument is NaN, the other argument is returned.

fmax() Parameters

x: The first argument of fmax().

y: The second argument of fmax().

Example 1: fmax() function for arguments of same type

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    double x = -2.05, y = NAN, result;

    result = fmax(x, y);
    cout << "fmax(x, y) = " << result << endl;

    return 0;
}
```

Example 2: fmax() function for arguments of different types

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    double x = 56.13, result;
    int y = 89;

    result = fmax(x, y);
    cout << "fmax(x, y) = " << result << endl;

    return 0;
}
```

When you run the program, the output will be:

```
fmax(x, y) = 89
```

C++ fmin()

returns smallest among two given arguments

The fmin() function in C++ takes two arguments and returns the smallest among them. If one of the argument is NaN, the other argument is returned.

C++ fabs()

returns absolute value of argument

The fabs() function in C++ returns the absolute value of the argument.

fabs() Parameters

The fabs() function takes a single argument, x whose absolute value is returned.

Example 1: How fabs() function works in C++?

```
#include <iostream>
#include <cmath>
```

```
using namespace std;

int main()
{
    double x = -10.25, result;

    result = fabs(x);
    cout << "fabs(" << x << ") = |" << x << "| = " << result <<
endl;

    return 0;
}
```

```
fabs(-10.25) = |-10.25| = 10.25
```

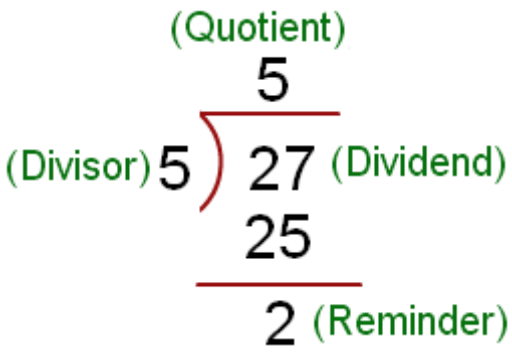
C++ remainder()

Returns remainder of x/y

The remainder() function in C++ computes the floating point remainder of numerator/denominator (rounded to nearest).

remainder() Parameters

- x - The value of numerator.
- y - The value of denominator.



Example 1: How remainder() works in C++?

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    double x = 7.5, y = 2.1;
    double result = remainder(x, y);
    cout << "Remainder of " << x << "/" << y << " = " << result <<
endl;

    x = -17.50, y=2.0;
    result = remainder(x, y);
    cout << "Remainder of " << x << "/" << y << " = " << result <<
endl;

    y=0;
    result = remainder(x, y);
    cout << "Remainder of " << x << "/" << y << " = " << result <<
endl;

    return 0;
}
```

```
Remainder of 7.5/2.1 = -0.9
Remainder of -17.5/2 = 0.5
Remainder of -17.5/0 = -nan
```

Example 2: remainder() function for arguments of different types

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    int x = 5;
    double y = 2.13, result;
```

```
    result = remainder(x, y);
    cout << "Remainder of " << x << "/" << y << " = " << result <<
endl;

    return 0;
}
```

```
Remainder of 5/2.13 = 0.74
```

C++ pow()

Computes Power a Number

The pow() function computes a base number raised to the power of exponent number.

pow() Parameters

The pow() function takes two arguments:

- base - the base value
- exponent - exponent of the base

Example 1: How pow() works in C++?

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

int main ()
{
       double base, exponent, result;

       base = 3.4;
       exponent = 4.4;
       result = pow(base, exponent);
```

```
        cout << base << "^" << exponent << " = " << result;

        return 0;
}
```

output

```
3.4^4.4 = 218.025
```