FAST National University of Computer and Emerging Sciences Peshawar

OOP Lab # 4.1

DEPARTMENT OF COMPUTER SCIENCE

C++ Programming (Functions)

Computer Instructor: Muhammad Abdullah Orakzai

Programming

Prepared By: Muhammad Abdullah Orakzai (Lab Instructor CS)

Contents

Function

- Function is a set of instructions that are designed to perform a specific task.
- A function is a complete and independent program.
- A function is a block of code which only runs when it is called.
- It is executed by the main function to perform its tasks.
- Functions are used to write the code of a large program by dividing it into smaller independent units.
- It avoids the replication of code in the program.

Function...

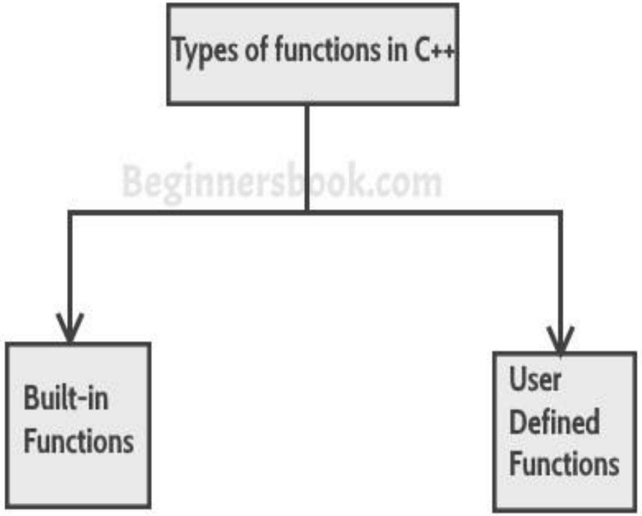
- A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

- You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

- A function is known with various names like a method or a sub-routine or a procedure etc.

Types of Functions

1. Built in Functions or standard library functions
2. User Defined Functions



1) Built in Functions

- The standard library methods are built-in functions in C++ that are readily available for use.
- Built-in functions are also known as library functions. We need not to declare and define these functions as they are already written in the C++ libraries such as iostream, cmath etc. We can directly call them when we need.
- **For example** pow(a,x), sqrt(x), round(x) etc.

2) User Defined Functions

- We can also create functions of our own choice to perform some task. Such functions are called user-defined functions.
- User define functions are the one that programmer write it by himself.

```
void myFunction()
{
    cout<<"welcome to C++ Programming";
}
```

Defining a Function (Syntax)

```
return_type  function_name (parameter(s))
{
  //C++ Statements
}
```

Defining a Function (Syntax)...

Return Type − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Defining a Function (Syntax)…

Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body − The function body contains a collection of statements that define what the function does.

Calling a or invoking function

- Executing the statement(s) of function to perform task is called calling a function.

- Calling a function is called invoking a function.
- Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called.
- To call a function, write the function's name followed by two parentheses () and a semicolon ;

Example:

```
addition();
```

Calling a or invoking function…

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

Example

Inside main, call myFunction():

Calling a or invoking function...

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

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

int main()
{

  myFunction(); // call the function
  return 0;
}

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

Calling a or invoking function...

A function can be called multiple times:

Example

```
void myFunction() {

  cout << "I just got executed!\n";
}

int main() {
  myFunction();
  myFunction();
  myFunction();
  return 0;
}

// I just got executed!
// I just got executed!
// I just got executed!
```

Function Declaration and Definition

A C++ function consist of two parts:

Declaration: the function's name, return type, and parameters (if any). Function declarations is also known prototype of the function.

Definition: the body of the function (code to be executed). The function definition consist of two parts: (1) Declarator (2) Body of the function

void myFunction(int x, int v);

void **myFunction()** { // **declaration**

  // the body of the function (**definition**)

}

Function Declaration and Definition...

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:

```
int main() {
  myFunction();
  return 0;
}

void myFunction() {
  cout << "I just got executed!";
}

// Error
```

Function Declaration and Definition...

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

int main() {
  myFunction();
  return 0;
}

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

```
In function 'int main()':
5:3: error: 'myFunction' was not declared in
this scope
```

Function Declaration and Definition…

- However, it is possible to separate the declaration and the definition of the function - for code optimization.
- You will often see C++ programs that have function declaration above main(), and function definition below main().
- This will make the code better organized and easier to read:

Function Declaration and Definition...

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

// Function declaration
void myFunction();

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

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

Output:
```
I just got executed!
```

Default Parameters

- 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"):

Default Parameters...

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

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

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

Output:

```
Sweden
India
Norway
USA
```

Default Parameters…

A parameter with a default value, is often known as an "**optional parameter**". From the example above, country is an optional parameter and "Norway" is the default value.

Multiple Parameters...

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

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

void myFunction(string fname, int age) {
  cout << fname << "Khalil" << age << " year
s old. \n";
}

int main() {
  myFunction("Abid", 3);
  myFunction("Hassan", 14);
  myFunction("Yasin", 30);
  return 0;
}
```

Output:

```
Abid Khalil 3 years old.
Hassan Khalil 14 years old.
Yasin Khalil 30 years old.
```

Functions different Scenarios

1. Function have no parameters list and return type.
2. Function have no return type but parameter list.
3. Function have both return type and parameter list (Function return values).
4. Function have return type but no parameter list.

1. Function have no parameters list and return type

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

void printStar()
{
    cout<<"**********";
}

int main()
{

  printStar(); // call the function
  return 0;
}
```

2. Function have no return type but parameter list

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

void sum(int x, int y)  // parameters
    {
    int sum=x+y;
    cout<<"Result is: "<<sum;
    }

int main() {

    sum(5,6);      //Arguments
    return 0;

}
```

Output:
Result is: 11

3. Function have both return type and parameter list

Function return values

Function can return only one value.

Return Statement: The return statement is used to return calculated value from function definition to calling function.

Syntax:

```
return x;
```

3. Function have both return type and parameter list…

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

int sum(int x, int y)  //parameters
    {
     return(x+y);
    }

int main() {

    int result=sum(5,6);      //Arguments
    cout<<"Result is: "<<result;
    //cout<<"Result is: "<<sum(5,6);
    return 0;
}
```

Output:
Result is: 11

3. Function have both return type and parameter list…

```
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z<<endl;
}
```

Output:

```
The result is 8
```

4. Function have return type but no parameter list

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

int sum()     {

    int x=50;
    int y=3;
    return(x+y);
    }

int main() {

    int result=sum();
    cout<<"Result is: "<<result;
    //cout<<"Result is: "<<sum();
    return 0;
}
```

Output:

```
Result is: 53
```

Functions User Enters arguments Values

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

int sum(int x, int y)  //parameters
    {
     return(x+y);
    }

int main() {
    int n1, n2;
    cout<<"Enter number 1:";
    cin>>n1;
    cout<<"Enter number 2:";
    cin>>n2;
    int result=sum(n1,n2);      //Arguments
    cout<<"Result is: "<<result;
    //cout<<"Result is: "<<sum(5,6);
    return 0;
}
```

Output:

```
Enter number 1: 3
Enter number 2: 3
Result is: 6
```

How to pass arguments to functions

We can pass arguments to functions by two ways:

1. Passing by values.
2. Passing by references.

1. Pass by Values

Passing arguments in such a way where the function creates copies of the arguments passed to it , it is called passing by value.

When an argument is passed by value to a function, a new variable of the data type of the argument is created and the data is copied into it. The function accesses the value in the newly created variable and the data in the original variable in the calling function is not changed.

1. Pass by Values...

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

void sum(int x, int y)
{
    int sum =x+y;
    cout<<"Result is: "<<sum;
}

int main()
{
    int a=5;
    int b=6;
    sum(a,b);
    return 0;
}
```

Output:

```
Result is: 11
```

2. Pass by Reference

The data can also be passed to a function by reference of a variable name and that contains data.

The reference provides the second name (or **alias**) for a variable name.

Alias: Two variables refer to the same thing or entity. Alias are the alternate name for referring to the same thing.

When a variable is passed by reference to a function, no new copy of the variable is created. Only the address of the variable is passed to the function.

2. Pass by Reference

- The original variable is accessed in the function with reference to its second name or alias. Both variables use the same memory location.

- Thus any change in the reference variable also changes the value in the original variable.

- The reference parameters are indicated by an ampersand (&) sign after the data type both in the function prototype and in the function definition.

2. Pass by Reference…

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

void swapNums(int &x, int &y) {

  int z = x;
  x = y;
  y = z;

  }
```

2. Pass by Reference

```
int main() {

  int firstNum = 10;
  int secondNum = 20;

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

  swapNums(firstNum,  secondNum);

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

  return 0;
}
```

Output:

```
Before swap:
10 20
After swap:
20 10
```

2. Pass by Reference

The ampersand sign (&) is also used with the data type of **“x”** and **“y”** variables. The **x** and **y** are the aliases of “`firstNum`” and “`secondNum`” variables respectively.

The memory location of **“x”** and “`firstNum`” is the same and similarly, memory location of **“y”** and “`secondNum`” is same.

Function Overloading

- Functions having same name with different set of parameters (type, order, number) then such kind of method is called overloaded function and this mechanism is called method overloading.
- Method overloading is compile time polymorphism or static binding.
- It increase the readability of the program.

Function Overloading…

Example

```
int myFunction(int x)

float myFunction(float x)

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

Function Overloading...

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;
}
```

Function Overloading…

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:

Function Overloading Example 1

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

Output:

```
Int: 13
Double: 10.56
```

Function Overloading Example 2

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

void sum(int x, int y)  // formal arguments
    {
    cout<<"sum of int is: "<<(x+y)<<endl;
    }
void sum(double x, double y)  // formal arguments
    {
    cout<<"sum of double is: "<<(x+y)<<endl;
    }
void sum(int x, double y)  // formal arguments
    {
    cout<<"sum of int & double is: "<<(x+y)<<endl;
    }
```

Function Overloading…

```
void sum(double x, int y)  // formal arguments
    {
    cout<<"sum of double & int is: "<<(x+y)<<endl;
    }

int main() {
    sum(3,5);
    sum(3.3,5.6);
    sum(3,5.4);
    sum(3.6,5);

  return 0;
}
```

Output:

```
sum of int is: 8
sum of double is: 8.9
sum of int & double is: 8.4
sum of double & int is: 8.6
```

Function with Default Parameters

- The default parameter is a way to set default values for function parameters a value is no passed in (i.e. it is undefined).
- In C++ programming, we can provide default values for function parameters.
- If a function with default arguments is called without passing arguments, then the default parameters are used.
- However, if arguments are passed while calling the function, the default arguments are ignored.

Function with Default Parameters...

```
#include <iostream>
#include <conio.h>
using namespace std;
void repchar(char='*', int=45);
int main()
{
repchar();
repchar('=');
repchar('+', 30);
}
void repchar(char ch, int n)
{
for(int j=0; j<n; j++)
cout << ch;
cout << endl;
}
```

Output:

```
*********************************************
==========================================
===
++++++++++++++++++++++++++++++
```

Tasks

1) Write function in C++ that will calculate table of a number in C++. Number must be passed as argument to function parameters.

2) Write function in C++ that will find factorial of a number. Number must be passed as argument to function parameters.

3) Update your calculator using functions.

References

- https://beginnersbook.com/2017/08/cpp-data-types/
- https://www.geeksforgeeks.org/c-data-types/
- http://www.cplusplus.com/doc/tutorial/basic_io/
- https://www.geeksforgeeks.org/basic-input-output-c/
- https://www.w3schools.com/cpp/default.asp
- https://www.javatpoint.com/cpp-tutorial

THANK YOU