

Lecture 3_

March 8, 2021

Today Topics

Functions

Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in C++.

A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

type name (parameter1, parameter2, ...) { statements }

```
[2]: #include <iostream>
using namespace std;
```

```
[3]: int addition (int a, int b){

    int r;
    r = a+b;
    return r;
}
```

```
[ ]:
```

```
[6]: int main(){

    int z;

    z = addition(5,3);
    cout<<"The result is "<<z;
    return 0;

}
main();
```

The result is 8

[]:

A C++ program always begins its execution by the main function.

The parameters and arguments have a clear correspondence. Within the main function we called to addition passing two values: 5 and 3, that correspond to the int a and int b parameters declared for function addition.

[]:

```
[6]: // function example 2
#include <iostream>
using namespace std;
```

```
[7]: int subtraction (int a, int b)
{
    int r;
    r=a-b;
    return (r);
}
```

```
[9]: int main ()
{

    int x=5, y=3, z;
    z = subtraction (7,2);

    cout << "The first result is " << z << '\n';
    cout << "The second result is " << subtraction (7,2) << '\n';
    cout << "The third result is " << subtraction (x,y) << '\n';

    z= 4 + subtraction (x,y);
    cout << "The fourth result is " << z << '\n';
    return 0;
}
main();
```

```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

[]:

Functions with no type. The use of void

If you remember the syntax of a function declaration:

type name (argument1, argument2 ...) statement

you will see that the declaration begins with a type, that is the type of the function itself (i.e., the type of the datum that will be returned by the function with the return statement). But what if we want to return no value?

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value. In this case we should use the void type specifier for the function. This is a special specifier that indicates absence of type.

```
[10]: // void function example
#include <iostream>
using namespace std;
```

```
[11]: void printmessage ()
{
    cout << "I'm a function!";
}
```

```
[12]: int main ()
{
    printmessage ();

    return 0;
}
main();
```

I'm a function!

```
[13]: void printmessage(void){
    cout<<"I am a function";
}
```

```
[ ]:
```

```
[ ]:
```

Arguments passed by value and by reference

Until now, in all the functions we have seen, the arguments passed to the functions have been passed by value. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function addition using the following code:

This way, when the function addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have

any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

```
[9]: // example:3 passing parameters by reference
#include<iostream>
using namespace std;
```

```
[10]: void duplicate (int& a, int& b, int& c)
{
    a*=2;
    b*=2;
    c*=2;
}
```

```
[14]: int main ()
{
    int x=2, y=3, z=7;
    duplicate (x, y, z);
    cout << "x=" << x << ", y=" << y << ", z=" << z;

    duplicate (x, y, z);
    cout << "\nx=" << x << ", y=" << y << ", z=" << z;

    return 0;
}
main();
```

x=4, y=6, z=14

x=8, y=12, z=28

```
[ ]:
```

```
[18]: //example 4: swap numbers
#include<iostream>
using namespace std;
```

```
[15]: void swapNums( int &x, int &y){
    int z = x;
    x = y;
    y = z;
}
```

```
[16]: int main() {
```

```

int firstNum = 10;
int secondNum = 20;

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

// Call the function, which will change the values of firstNum and secondNum
swapNums(firstNum, secondNum);
cout << "After swap: " << "\n";
cout << firstNum << "\t" << secondNum << "\n";

return 0;
}

main();

```

Before swap:

10 20

After swap:

20 10

[]:

Default values in parameters

When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:

```

[17]: // default values in functions
#include <iostream>
using namespace std;

```

```

[18]: int divide (int a, int b=2)
{
    int r;
    r=a/b;
    return (r);
}

```

```

[19]: int main ()
{

```

```
    cout << divide (12);  
    cout << endl;  
    cout << divide (20,4);  
    return 0;  
}  
main();
```

6

5

Overloaded functions.

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. For example:

```
[20]: // overloaded function  
#include <iostream>  
using namespace std;
```

```
[21]: int operate (int a, int b)  
{  
    return (a*b);  
}
```

```
[22]: float operate (float a, float b)  
{  
    return (a/b);  
}
```

```
[23]: int main ()  
{  
    int x=5,y=2;  
    float n=5.0,m=2.0;  
  
    cout << operate (x,y);  
    cout << "\n";  
  
    cout << operate (n,m);  
    cout << "\n";  
  
    return 0;  
}  
main();
```

10
2.5

In this case we have defined two functions with the same name, operate, but one of them accepts two parameters of type int and the other one accepts them of type float. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two ints as its arguments it calls to the function that has two int parameters in its prototype and if it is called with two floats it will call to the one which has two float parameters in its prototype.

```
[24]: // overloaded function
#include <iostream>
using namespace std;
```

```
[25]: int plusFuncInt(int x, int y) {
    return x + y;
}
```

```
[26]: double plusFuncDouble(double x, double y) {
    return x + y;
}
```

```
[27]: int main() {

    int myNum1 = plusFuncInt(8, 5);

    double myNum2 = plusFuncDouble(4.3, 6.26);

    cout << "Int: " << myNum1 << "\n";

    cout << "Double: " << myNum2;

    return 0;
}
main();
```

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

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

```
[28]: int plusFunc(int x, int y) {
    return x + y;
}
```

```
}
```

```
[29]: double plusFunc(double x, double y) {  
    return x + y;  
}
```

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

Int: 13

Double: 10.56

inline functions

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because execution time of small function is less than the switching time.

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small. The syntax for defining the function inline is:

```
inline return-type function-name(parameters)  
{ // function code }
```

```
[42]: // example 4: inline function  
#include <iostream>  
using namespace std;
```

```
[43]: inline int cube(int s)
{
    return s*s*s;
}
```

```
[44]: int main()
{
    cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
}
main();
```

The cube of 3 is: 27

[]:

Recursivity

Recursivity is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or calculate the factorial of numbers. For example, to obtain the factorial of a number (n!) the mathematical formula would be:

$$n! = n * (n-1) * (n-2) * (n-3) \dots * 1$$

```
[31]: // factorial calculator
#include <iostream>
using namespace std;
```

```
[32]: long factorial (long a)
{
    if (a > 1)
        return (a * factorial (a-1));
    else
        return (1);
}
```

```
[33]: int main ()
{
    long number;
    cout << "Please type a number: ";

    cin >> number;

    cout << number << "! = " << factorial (number);
    return 0;
}
```

```
main();
```

Please type a number: 5

5! = 120

[]:

Declaring functions

Until now, we have defined all of the functions before the first appearance of calls to them in the source code. These calls were generally in function main which we have always left at the end of the source code. If you try to repeat some of the examples of functions described so far, but placing the function main before any of the other functions that were called from within it, you will most likely obtain compiling errors. The reason is that to be able to call a function it must have been declared in some earlier point of the code, like we have done in all our examples.

[69]:

[]: