C++ Programming

For **BCA 4th Semester**

Lecture 3

[Function, Call by Value Call by Reference, Inline Function, Const. Reference, Function Overloading]

Compiled
By
Subhadip Mukherjee
Dept. of Computer Science and BCA
Kharagpur College,
Kharagpur 721305

Function in C++

Functions allow to structure programs in segments of code to perform individual tasks.

In C++, a function is a group of statements that is given a name, and which can be called from some point of the program. The most common syntax to define a function is:

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

Where:
- `type` is the type of the value returned by the function.
- `name` is the identifier by which the function can be called.
- `parameters` (as many as needed): Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma.
- `statements` is the function's body. It is a block of statements surrounded by braces { } that specify what the function actually does.

Let's have a look at an example:

```
// 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;
}
```

```
The result is 8
```

In the example above, `main` begins by declaring the variable `z` of type `int`, and right after that, it performs the first function call: it calls `addition`. The call to a function follows a structure very similar to its declaration. In the example above, the call to `addition` can be compared to its definition just a few lines earlier:

```
int addition (int a, int b)
                   ↑      ↑
z = addition (  5  ,   3  );
```

Recursive Function

Recursivity is the property that functions have to be called by themselves. It is useful for some tasks, such as sorting elements, or calculating the factorial of numbers. For example, in order to obtain the factorial of a number (`n!`) the mathematical formula would be:

```
n! = n * (n-1) * (n-2) * (n-3) ... * 1
```

More concretely, `5!` (factorial of 5) would be:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

And a recursive function to calculate this in C++ could be:

```cpp
// factorial calculator
#include <iostream>
using namespace std;

long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return 1;
}

int main ()
{
  long number = 9;
  cout << number << "! = " <<
factorial (number);
  return 0;
}
```

```
9! = 362880
```

Notice how in function factorial we included a call to itself, but only if the argument passed was greater than 1, since, otherwise, the function would perform an infinite recursive loop, in which once it arrived to 0, it would continue multiplying by all the negative numbers (probably provoking a stack overflow at some point during runtime).

Call by value

In call by value, original value can not be changed or modified. In call by value, when you passed value to the function it is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only but it not change the value of variable inside the caller function such as main().

```
#include<iostream.h>
#include<conio.h>

void swap(int a, int b)
{
        int temp;
        temp=a;
        a=b;
        b=temp;
}

void main()
{
	int a=100, b=200;
	clrscr();
	swap(a, b);  // passing value to function
	cout<<"Value of a"<<a;
	cout<<"Value of b"<<b;
	getch();
}
```

Output

```
Value of a: 200
Value of b: 100
```

Call by reference

In call by reference, original value is changed or modified because we pass reference (address). Here, address of the value is passed in the function, so actual and formal arguments shares the same address space. Hence, any value changed inside the function, is reflected inside as well as outside the function.

```
#include<iostream.h>
#include<conio.h>

void swap(int *a, int *b)
{
	int temp;
	temp=*a;
	*a=*b;
	*b=temp;
}

void main()
{
	int a=100, b=200;
	clrscr();
	swap(&a, &b);  // passing value to function
	cout<<"Value of a"<<a;
	cout<<"Value of b"<<b;
	getch();
}
```

Inline functions

Calling a function generally causes a certain overhead (stacking arguments, jumps, etc...), and thus for very short functions, it may be more efficient to simply insert the code of the function where it is called, instead of performing the process of formally calling a function.

Preceding a function declaration with the `inline` specifier informs the compiler that inline expansion is preferred over the usual function call mechanism for a specific function. This does not change at all the behavior of a function, but is merely used to suggest the compiler that the code generated by the function body shall be inserted at each point the function is called, instead of being invoked with a regular function call.

For example, the concatenate function above may be declared inline as:

```
inline string concatenate (const string& a, const string& b)
{
  return a+b;
}
```

This informs the compiler that when `concatenate` is called, the program prefers the function to be expanded inline, instead of performing a regular call. `inline` is only specified in the function declaration, not when it is called.

Note that most compilers already optimize code to generate inline functions when they see an opportunity to improve efficiency, even if not explicitly marked with the `inline` specifier. Therefore, this specifier merely indicates the compiler that inline is preferred for this function, although the compiler is free to not inline it, and optimize otherwise. In C++, optimization is a task delegated to the compiler, which is free to generate any code for as long as the resulting behavior is the one specified by the code.

Efficiency considerations and const references

Calling a function with parameters taken by value causes copies of the values to be made. This is a relatively inexpensive operation for fundamental types such as `int`, but if the parameter is of a large compound type, it may result on certain overhead. For example, consider the following function:

```
string concatenate (string a, string b)
{
  return a+b;
}
```

This function takes two strings as parameters (by value), and returns the result of concatenating them. By passing the arguments by value, the function forces `a` and `b` to be copies of the arguments passed to the function when it is called. And if these are long strings, it may mean copying large quantities of data just for the function call.

But this copy can be avoided altogether if both parameters are made *references*:

```cpp
string concatenate (string& a, string& b)
{
  return a+b;
}
```

Arguments by reference do not require a copy. The function operates directly on (aliases of) the strings passed as arguments, and, at most, it might mean the transfer of certain pointers to the function. In this regard, the version of `concatenate` taking references is more efficient than the version taking values, since it does not need to copy expensive-to-copy strings.

On the flip side, functions with reference parameters are generally perceived as functions that modify the arguments passed, because that is why reference parameters are actually for.

The solution is for the function to guarantee that its reference parameters are not going to be modified by this function. This can be done by qualifying the parameters as constant:

```cpp
string concatenate (const string& a, const string& b)
{
  return a+b;
}
```

By qualifying them as `const`, the function is forbidden to modify the values of neither `a` nor `b`, but can actually access their values as references (aliases of the arguments), without having to make actual copies of the strings.

Therefore, `const` references provide functionality similar to passing arguments by value, but with an increased efficiency for parameters of large types. That is why they are extremely popular in C++ for arguments of compound types. Note though, that for most fundamental types, there is no noticeable difference in efficiency, and in some cases, const references may even be less efficient!

Function Overloading

In C++, two different functions can have the same name if their parameters are different; either because they have a different number of parameters, or because any of their parameters are of a different type. For example:

```cpp
// overloading functions
#include <iostream>
using namespace std;

int operate (int a, int b)
{
  return (a*b);
}

double operate (double a, double b)
{
  return (a/b);
}

int main ()
{
  int x=5,y=2;
  double n=5.0,m=2.0;
  cout << operate (x,y) << '\n';
  cout << operate (n,m) << '\n';
```

```
10
2.5
```

```
  return 0;
}
```

In this example, there are two functions called `operate`, but one of them has two parameters of type `int`, while the other has them of type `double`. 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 `int` arguments, it calls to the function that has two `int` parameters, and if it is called with two `double`s, it calls the one with two `double`s.