Object-oriented Programming Laboratory
Lab 01
Introduction to C++
(Switching from C to C++)
Version 1.0

Welcome to the C++ tutorials! The tutorials in this section are aimed primarily at beginning level programmers, including those who have little to no prior programming experience. Intermediate level programmers will probably also find plenty of tips and tricks that may be of use in improving their programming skills.

Probably the best way to start learning a programming language is by writing a program. Therefore, here is our first program:

```
// my first program in C++

#include <iostream>
using namespace std;

int main ()
{
  cout << "Hello World!";
  return 0;
}
```

```
// output
Hello World!
```

It is one of the simplest programs that can be written in C++, but it already contains the fundamental components that every C++ program has. We are going to look line by line at the code we have just written:

```
#include <iostream>
```

The directive #include <iostream>tells the preprocessor to include the iostream standard file. This specific file (iostream) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

```
using namespace std;
```

All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name *std*. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library, and in fact it will be included in most of the source codes included in these tutorials.

```
cout << "Hello World!";
```

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program.

cout is the name of the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (*cout*, which usually corresponds to the screen).

cout is declared in the *iostream* standard file within the *std* namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code.

```
return 0;
```

The return statement causes the main function to finish. return may be followed by a return code (in our example is followed by the return code with a value of zero). A return code of *0* for the *main* function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

Fundamental data types and keywords

Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

Name	Description	Size*	Range*
Char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int(short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character.	2 *or* 4 bytes	1 wide character

The standard reserved keywords for C++ are:

```
asm, auto, bool, break, case, catch, char, class, const, const_cast,
continue, default, delete, do, double, dynamic_cast, else, enum,
explicit, export, extern, false, float, for, friend, goto, if, inline,
int, long, mutable, namespace, new, operator, private, protected,
public, register, reinterpret_cast, return, short, signed, sizeof,
static, static_cast, struct, switch, template, this, throw, true, try,
typedef, typeid, typename, union, unsigned, using, virtual, void,
volatile, wchar_t, while
```

Introduction to strings

The C++ language library provides support for strings through the standard string class. This is not a fundamental type, but it behaves in a similar way as fundamental types do in its most basic usage. A first difference with fundamental data types is that in order to declare and use objects (variables) of this type we need to include an additional header file in our source code: <string> and have access to the std namespace;

```
// my first string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring;
  mystring = "This is the initial string
content";
  cout << mystring << endl;
  mystring = "This is a different string
content";
  cout << mystring << endl;
  return 0;
}
```

```
// output

This is the initial string content
This is a different string content
```

Here endl is used for inserting a newline after printing the string.

Standard Input (cin) and Output (cout)

By default, the standard output of a program is the screen, and the C++ stream object defined to access it is cout. Already we discussed about cout. Now let us see the use of cin;

```
// i/o example

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

int main ()
{
  int i;
  cout << "Please enter an integer value: ";
  cin >> i;
  cout << "The value you entered is " << i;
  cout << " and its double is " << i*2 <<
".\n";
    string mystr;
  cout << "What's your name? ";
  getline (cin, mystr);
  cout << "Hello " << mystr << ".\n";

  return 0;
}
```

```
// output

Please enter an integer value:
702
The value you entered is 702
and its double is 14
```

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the cin stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream.

Operators in C++

All the C operators are valid in C++ also. In addition C++ introduces some new operators. The operators are:

Name	Description
<<	Insertion operator
>>	Extraction operator
::	Scope resolution operator
::*	Pointer-to-member declator
->*	Pointer-to-member operator
.*	Pointer-to-member operator
delete	Memory release operator
endl	Line feed operator
New	Memory allocation operator
Setw	Field width operator

Scope resolution operator ::

In C the global version of a variable cannot be accessed from within the inner block if there is a variable with the same name. C++ resolves this problem by introducing a new operator :: called scope resolution operator. This can be used to uncover the hidden variable. The following program illustrates the use of :: operator.

```
// :: operator example

#include <iostream>
using namespace std;
int m = 10; //global m
int main ()
{
    int m = 20; //here m is local to main
    {
      int k = m;
      int m = 30; // here m is local to inner
block

      cout<<"we are in inner block\n";
      cout<<"k = "<<k<<endl;
      cout<<"m = "<<m<<endl;
      cout<<"::m = "<<::m<<endl;
    }
    cout<<"\n we are in outer block\n";
    cout<<"m = "<<m<<endl;
    cout<<"::m = "<<::m<<endl;
    return 0;
}
```

```
// output

we are in inner block
k = 20
m = 30
::m = 10

 we are in outer block
m = 20
::m = 10
```

In the above program it is to be noted that ::m will always refer to the global m. The major application of the scope resolution operator is in the classes which will discussed later on other labs.

Memory management operators (new and delete)

C use malloc(), calloc() and free() to allocate and free memory dynamically. C++ also supports these functions. It also defines two unary operators new and delete that perform the task of allocating and freeing memory in a better and easier way. Let us see a simple example;

```
// new delete operator example

#include <iostream>
using namespace std;

int main ()
{
    int *p = new int;
    float *q = new float(7.5);
    *p = 25;
    cout<<"*p = "<<*p<<endl;
    cout<<"*q = "<<*q<<endl;

    delete p;
    delete q;
    return 0;
}
```

```
// output

*p = 25
*q = 7.5
```

Here two pointer variable *p and *q are holding the address of the allocated memory of int and float. Also we can initialize the allocated memory with a value at the time of allocation like float memory allocation. Finally we use delete operator to delete the allocated memory with the help of pointer p and q.

Functions in C++

C++ supports all the function definition rules defined in C. In addition C++ has some special function definition rules. Some of these are discuss below and other will be discussed alter on other labs.

Inline functions

Every time we call a function it takes a lot of extra time for jumping to the function, saving registers, pushing arguments to the stack and returning to the calling function. For a function with small definition in its body and calling huge number of times causes the program to take a lot of time. To eliminate the cost of calls to small functions, C++ proposes a new feature called *inline function.* The compiler replaces the inline function call with corresponding function code. An example is given here;

```
// inline function example

#include <iostream>
using namespace std;

inline float mul(float x, float y){

    return(x*y);
}

inline float div(float x, float y){

    return(x/y);
}

int main ()
{
    float a = 12.345;
    float b = 9.82;

    cout<<mul(a,b)<<endl;
    cout<<div(a,b)<<endl;

    return 0;
}
```

```
// output

121.228
1.25713
```

Default argument in function

In C++, we can call a function with less number of values rather than the number of arguments in its parameter. But to do this, we must need to set a default value for the argument at the time function prototype declaration to which we do not want to pass value. The following example illustrates this;

```
// default argument function example

#include <iostream>
using namespace std;

float mul(float x, float y=2){

    return(x*y);
}

int main ()
{
    float a = 12.345;
    float b = 9.82;

    cout<<mul(a,b)<<endl;
    cout<<mul(a)<<endl;
    return 0;
}
```

```
// output

121.228
24.69
```

Here in the above program first mul() function called with two value a, b and assigned to parameter x, y respectively. Then the second mul() function called with single value a and assigned to parameter x and the other parameter y assigns its default value.

Function overloading

Overloading refers to the use of same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. The following simple example illustrates the use of function overloading;

```
// function overloading example

#include <iostream>
using namespace std;

int volume(int x){

    return(x*x*x);
}
double volume(double r, int h){

    return(3.14519*r*r*h);
}
long volume(long l, int b, int h){

    return(l*b*h);
}
int main ()
{
    float a = 12.345;
    float b = 9.82;

    cout<<volume(10)<<endl;
    cout<<volume(2.5,8)<<endl;
    cout<<volume(100,75,15)<<endl;
    return 0;
}
```

```
// output

1000
157.26
112500
```

Here we create three function with same name volume() but with different number of arguments. When we call volume() with one of the number of arguments, the compiler only calls that volume() which exactly match the number of arguments.

C structures revisited

Structure is unique feature in C language. Structures provide a method for packing together data of different types. It is a user-defined data type with a template that serves to define its data properties. Once the structure type has been defined, we can create variable of that type using declarations that are similar to the built-in type declarations. Let us a simple program that remids the use of structure;

```
// structure example

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

struct student{
```

```
// output

John
111
89
```

```
    char name[20];
    int roll;
    int mark;
};
int main ()
{

    struct student a;
    strcpy(a.name,"John");
    a.roll=111;
    a.mark=89;
    cout<<a.name<<endl;
    cout<<a.roll<<endl;
    cout<<a.mark<<endl;

    return 0;
}
```

Here a structure type student is created and a structure variable a is declared of type struct student and assign values to its different properties.

Limitations of C structures

For the above structure student if we declare three variable a, b, and c and assign values to a and b. Then if we write a statement like c = a + b, will produce an error. Because C does not permit addition or subtraction of complex variables.

Another important limitation of C structures is that they do not permit data hiding. Structure members can be directly accessed by the structure variables by any function anywhere in their scope. In other words structure members are public members.