Laboratory 2: Introduction to Classes in C++

The aims of today's lab are:

- Finish Lab 1;
- Understand the constructor in C++ and using initialisation lists;
- Get familiar with the implementation of (basic) classes;
- Overwrite some operators (=, << and >>);
- Use `std::string` instead of `char*`.

Next week, we will use the STL library and write our own template class.

Task 0: Last week

Make sure you finish your Bubble Sort implementation! You can find a good example of Bubble Sort execution on Wikipedia: `https://en.wikipedia.org/wiki/Bubble_sort` There is an animated GIF file.

Task 1: Using CMake

Same as last week, we will use CMake to make our lifes easier. So create your MS Visual Studio solutions, Xcode projects, or Unix Makefiles using CMake.

1 Task 2: Illustrating the usefulness of initialisation lists

For this task, you are given three C++ files:

1. `include/Integer.h`, the header file defining a class to handle integer numbers;
2. `src/Integer.cpp`, the source file that implements the code;
3. `src/TestInteger.cpp`, a test program to try the class.

We want to adopt good C++ practice. There are plenty of tutorials, forums, etc. on the Web that deal with how to write simple C++ classes. I googled 'How to write a simple class in C++?'. The first item returned from Google was: `http://stackoverflow.com/questions/865922/how-to-write-a-simple-class-in-c` The answers to the question on StackOverflow are mostly correct, but not perfect. Most people did not use initialisation lists.

Initialisation lists have been around for a very long time, but many people still ignore them despite their advantages. Initialisation lists are used in constructors to set the initial values of class members in an efficient manner. On the web, we may often see:

```
DummyClass my_instance;
my_instance = 10;
```

This is not very good. What will happen in practice?

1. One call of the default constructor (`DummyClass::DummyClass()`);
2. One call of the copy operator (`DummyClass::operator=(int)`)

i.e. two function calls will be performed to declare and initialise the object. It is obviously not very good if we consider how many times we may have to create objects in a large program. Instead, we should use:

```
DummyClass my_instance(10);
```

or

```
DummyClass my_instance = 10;
```

Both are equivalent and only the constructor (`DummyClass::DummyClass(int)`) will be called. This is more effective and it is better OOP practice.

Have a look at the code an execute it. 3 instances of Integer have been created and data has been printed in the standard output. Look at the different values of `m_data` that have been printed and try to understand what happened.

For Task 1, you only have to modify `Integer.cpp`. The methods that you need to modify are:

```
Integer::Integer()
Integer::Integer(int anInteger)
```

But first, look below at the code of the copy constructor. It has an initialisation list. See the ':' character at the end of Line 2 and look at Line 4 to see how to set the initial value of `m_data`.

```
1  //-----------------------------------------
2  Integer::Integer(const Integer& anInteger):
3  //-----------------------------------------
4          m_data(anInteger.m_data)
5  //-----------------------------------------
6  {
7      std::cout << "IN:␣Integer::Integer(const␣Integer&␣anInteger)" << std::endl;
8
9      std::cout << "\t" << m_data << std::endl;
10
11     std::cout << "OUT:␣Integer::Integer(const␣Integer&␣anInteger)" << std::endl;
12 }
```

Add an initialisation list to the other constructors. (see `src/Integer.cpp` for further instructions)

The code of the initialisation list is execute first, at the instanciation of the objet. Then the body of the constructor is executed. It is not unusual for C++ constructors to have an empty body. Using the list is faster than using the copy `operator=` in the body of the constructor.

Task 3: Create your own class

You are going to write a new class called "StringInverter". You need 3 extra files:

1. `include/StringInverter.h`, the header file defining a class to handle integer numbers;
2. `src/StringInverter.cpp`, the source file that implements the code;
3. `src/TestStringInverter.cpp`, a test program to try the class.

Now, modify CMakeLists.txt to add a new project. Just add:

```
add_executable(Task2
        include/StringInverter.h
        src/TestStringInverter.cpp
        src/StringInverter.cpp)
```

Now you can start implementing the class. It should include the public methods as follows:

```
StringInverter();
StringInverter(const StringInverter& aString);
StringInverter(const char* aString);
StringInverter(const std::string& aString);
~StringInverter();
void setString(const char* aString);
void setString(const std::string& aString);
const std::string& getString() const;
const std::string& getInvertedString() const;
StringInverter& operator=(const StringInverter& aString);
StringInverter& operator=(const char* aString);
StringInverter& operator=(const std::string& aString);
bool operator==(const StringInverter& aString) const;
bool operator==(const char* aString) const;
bool operator==(const std::string& aString) const;
bool operator!=(const StringInverter& aString) const;
bool operator!=(const char* aString) const;
bool operator!=(const std::string& aString) const;
```

It should also include the following private method:

```
void invertString();
```

Also add friend functions `operator<<` and `operator>>`:

```
std::ostream& operator<<(std::ostream& aStream, const StringInverter& aString);
std::istream& operator>>(std::istream& aStream, StringInverter& aString);
```

In `TestStringInverter.cpp`, you have to test each method and each function.

include/StringInverter.h