Laboratory 2: Introduction to Classes in C++

The aims of today's lab are:

- Finish Lab 1 (the Bubble Sort);
- 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.

1 Task -1: 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.

2 Task 0: Using CMake

Same as last week, we will use CMake to make our lifes easier.

3 Task 1: 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 return from Google is: `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. Today, we will demonstrate how bad some C++ code from the Web can be.

Initialisation lists have been around for a very long time, but many people 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 often see:

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

or

```
DummyClass my_instance = 10;
```

Both are equivalent and extremely bad. What will happen in practice is:

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

i.e. two function calls will be performed to 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);
```

In this case, only the constructor (`DummyClass::DummyClass(int)`) will be called. This is more effective and it is better OOP practice. Note that some compilers will automatically interpret

```
DummyClass my_instance = 10;
```

as

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

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.

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

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

You need to identify what is happening. To do this, have a look at the output in the console.

Now, look at the copy constructor. It has an initilisation 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.

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 constructors to have an empty body. Using the list is faster than using the copy `operator=` in the body of the constructor.

4 Task 2: Create your own class

You are going to write a new class called "StringInverter". You need to add 3 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.

For the moment, you can create three empty files. Now, modify CMakeLists.txt to add a new project. Just add:

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

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

```
StringInverter::StringInverter();
StringInverter::StringInverter(const StringInverter& aString);
StringInverter::StringInverter(const char* aString);
StringInverter::StringInverter(const std::string& aString);

void StringInverter::setString(const char* aString);
void StringInverter::setString(const std::string& aString);

const char* StringInverter::getString();
const std::string& StringInverter::getString();

const char* StringInverter::getInvertedString();
```

```
const std::string& StringInverter::getInvertedString();

StringInverter& operator=(const StringInverter& aString);
```

Also add functions overloading `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.